The Use of Domain Ontologies for the Virtual Scenes Management

CRENGUTA BOGDAN DORIN MIRCEA POPOVICI
Faculty of Mathematics and Computer Science
Ovidius University
124 Mamaia Blvd., Constanta, 900527
ROMANIA
cbogdan@univ-ovidius.ro dmpopovici@univ-ovidius.ro www.cerva.ro

Abstract: - In this paper, an object-oriented software system for the management of virtual scenes is presented. This system, called OntSceneBuilder, uses a domain ontology in order to obtain at least three benefits: accuracy, ease of content reuse and management of virtual scenes. The accuracy is ensured by default, since any ontology provides a precise specification of the concepts and their relations of a domain. Each concept is associated with 2D and 3D resources and a virtual artifact. The graph of the virtual artifacts forms a virtual scene of a virtual exposition. The paper also presents some models of the system development process, mainly realized during the analysis and design activities. Our aim was to analyze the OntSceneBuilder from the functional and interactional viewpoints and to create its software use case diagram. Furthermore, each software use case was designed from the structural and dynamical viewpoints. At the same time we also constructed the system software architecture. Some classes of the software architecture manage the concepts of the domain ontology associated with the topic chosen by user. The system has been experimented in the realization of virtual scenes of a virtual historic exposition.

Key-Words: - Virtual scene, domain ontology, concept, virtual artifact, software analysis, software architecture, virtual historic exposition

1 Introduction

The present study introduces the OntSceneBuilder software system, the models obtained during its developing process, and it concentrates on the analysis and the design of the system. OntSceneBuilder is an intensive object-oriented software system that uses a domain ontology to construct virtual scenes with the help of the user.

For users, the benefits of using a domain ontology are at least three: accuracy, ease of content reuse and management of virtual scenes. The accuracy is ensured by default, since a domain ontology is a precise and formal specification of the knowledge of the domain. The specification contains the concepts and their relations which the user can use to construct the virtual scenes and manage their content. The ontology also acts as a compulsion factor in the process of the scenes content management. For example, the user cannot choose the Color concept if he/she did not add to the scene the concept by which the first concept depends on.

OntSceneBuilder can be used with any domain ontology. We tested the system functioning using the ontology of the Roman artifacts found in the Tomis fortress [1]. The ontology is based on the top-level ontology DOLCE [2] and its modules, such as D&S [3], Temporal Relations and so on.

The main objective was the creation of the virtual scenes of a virtual exposition of the Tomis fortress. The virtual exposition shows virtual artifacts of the Roman exhibits found at the National History and Archaeology Museum of Constanta, Romania. The exhibits have been ontologically classified in categories such as objects, constructions, decorative elements, clothing parts and their accessories, activities, and so on.

And least but not last, OntSceneBuilder permits the virtual reality (VR) expert to easily manage content of a virtual scene using interactive graphical interfaces. The user can also move the virtual artifacts of a scene through drag-and-drop actions. Therefore, the VR expert can construct his/her scenes without having to be an ontology or domain expert.

The paper is organized as follows: sections 2 and 3 present some of the models created during the analysis and design of the OntSceneBuilder system. The models have been created using a standard language called the Unified Modeling Language (UML). This language allows the creation either of the graphic models of the information or of the software system [4]. Section 4 focuses on the implementation and testing aspects. Section 5 covers related work. Finally, section 6 concludes and presents our future work plans.

2 Software Analysis

The objective of the OntSceneBuilder is to provide an easy-to-use content management tool that allows users to add, edit, view and save virtual scenes.

2.1 Software actors

A software actor is a role "played" by one or more individuals (person, team, or organization) or even another software application in its interactions with our system. The role is characterized by a set of properties and actions which each individual in this role can exhibit or "play" in the given context [5].

In the case of the OntSceneBuilder application, we identified two software actors: user and time.

The user interacts with the system for that the later to provide the following functionalities: a) it creates a scene based on the concepts of the domain ontology used by the system; b) it modifies a saved scene, by changing the artifacts properties, or the concepts resources used or deleting virtual artifacts from scene. Modification of the topological properties of the virtual artifacts can also be made from the 3D environment. By default, when a user asks for removing of a virtual artifact, the associated resource and concept are removed by the system from the used lists and the virtual artifact from the scene. However, the system does not delete on the disk the resource associated to the virtual artifact and the concept from the domain ontology by which belongs to; c) it shows and saves an opened scene.

Another software actor of our application is time, since at each five minutes the system saves the current scene.

2.2 Software use case diagram

The software use case diagram belongs to the functional model created during the object-oriented analysis activity of a software system. Besides the software use case diagram, a functional model contains the application requirements, the description of the software use cases and their activity diagrams [5].

The software use case diagram is formed by the software actors and uses cases and their relations. For example, the OntSceneBuilder application has the software use case diagram presented in Fig. 1.

Each use case was informally described using a structured text-based template. For example, the description of the "Create scene" and "Save scene as XML file" use cases is presented in Table 1.

Table 1: The informal structured description of the "Create scene" use case

Name	Create scene
Software actor	User
Trigger event	User asks to create a scene.
Preconditions	The system has to function.
Postconditions	The system has shown the created scene
Main flow:	
User	*System*
1. He/she asks to create a scene.	2. Displays a list of topics.
3. Chooses a topic	4. Shows the list of concepts that belong to chosen topic.
5. Chooses a concept that is used by scene[1]	6. Displays a frame that contains the tree structure of the files system of the current computer.
7. Chooses the resource associated to the concept selected previously.	8. Registers the choice. 9. Shows the resource. 10. Requires user to enter values of the geometry properties: location, orientation and scale of the virtual artifact associated with the resource selected previously.
11. Provides values demanded	12. Registers the property values of the virtual artifact.
13. Requires the scene construction	14. Creates a scene by composing virtual artifacts. 15. Shows the scene.
Name	Save scene as XML file
Software actor	User
Trigger Event	User asks to save a scene.
Preconditions	The system has showed a scene.
Postconditions	The system has saved the scene.
Main flow:	
User	*System*
1. He/she asks to save a scene.	2. Displays a frame that contains the tree structure of the files system of the current computer.
3. Chooses a new path and/or a new name of the scene [A1].	4. Saves the scene.
	5. Deletes the previous saving.
Alternative flow:	
[A1] There is a file with the same name at the current path: 1. The system shows an attention message 2. The system replaces the old file with the new one.	

[1] The steps 5-11 repeat for each concept that is graphically represented by a virtual artifact in scene.

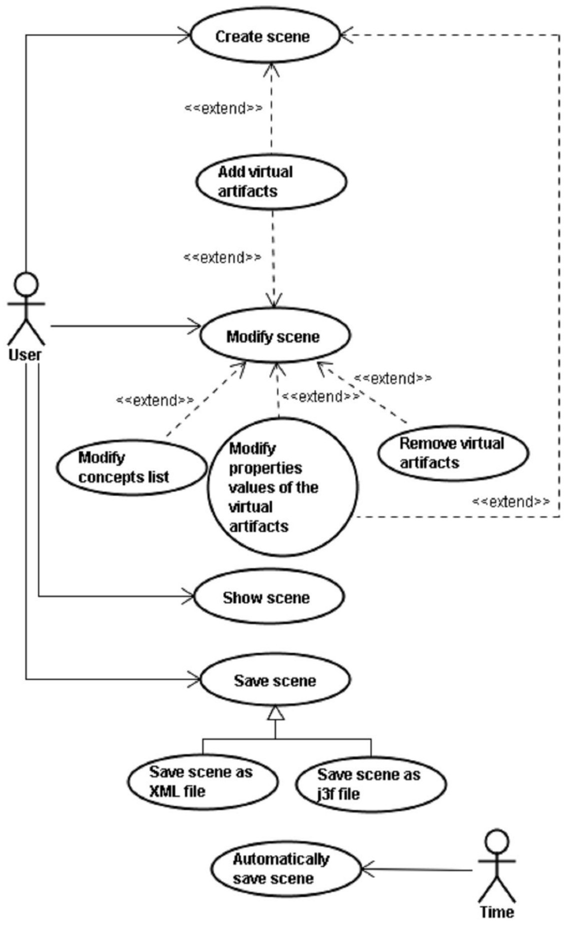


Fig. 1: The use case diagram of the OntSceneBuilder system

3 Software Design

The object-oriented analysis has focused on learning to "do the right thing"; that is, the understanding some of the goals for the OntSceneBuilder application, and related rules and constraints. By contrast, the design work will stress "do the thing right" [5]; that is, skillfully designing a solution to satisfy the system objective. The heart of this solution is creation of the system software architecture.

3.1 Software architecture

The software architecture of the OntSceneBuilder system has been done in two steps. The first step consisted of the construction of the application architecture that is a layered architecture that partitions the application into three layers (Fig. 2):

- the presentation layer is formed by GUI components which are shown to users as frames;
- the logic layer contains components independent by the ontology used and fulfill the most part of the system functionalities depicted in Fig. 1, and
- the persistence management layer contains components consisted of the manager classes which are described in detail in the next subsection. Briefly, these classes deal with the resources (ontology and 2D/3D files) used by system.

The relations between layers are dependencies between classes of a layer to the next lower layer. We have also dependency relations between the presentation and logic layers of our system and the ARéViJava toolkit used (see the next section).

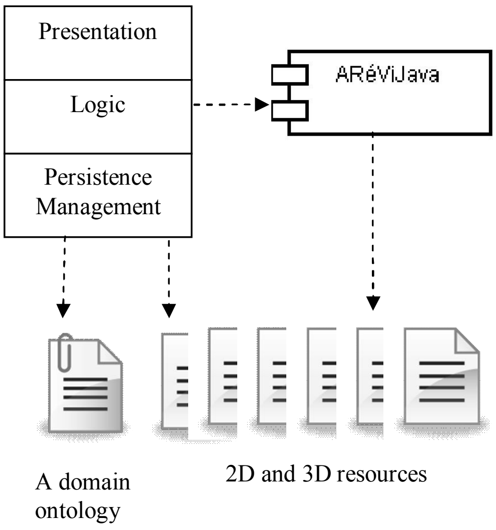


Fig. 2: The layered application architecture of the OntSceneBuilder system

In the second step, we continued the design of the software architecture detailing the application architecture (Fig. 2) following the software use cases and applying the Model-View-Controller (MVC) architectural pattern [6].

According to this pattern, the objects are classified in three categories: models, views, and controllers. The classification criterion is given by the responsibilities of the objects from each category. The view objects are objects with which user stakeholders interact directly, such as frames, forms, panels, and so on. The view objects form a part of the presentation layer of the application architecture of our system (Fig. 2).

The model objects eventually contain persistent information managed by system. Many such objects come from the business objects of the domain model of the information system where the software system will operate. In our case, the domain model was substituted by the domain ontology used. This is why, we introduced the Concept and Ontology classes into the software architecture.

Another kind of model objects is constituted by the composite objects. These contain model objects created at the beginning or during the system execution. The idea is that if there are many objects of the same class and they have to be used during the system execution, we temporarily put them in a composite object that is linked using aggregation by the object parts. In addition, a composite object has the responsibilities to create and change the states of objects it contains. In this way, we also fulfill the Creator pattern [7].

Furthermore, the controller objects might send requests to the composite object that solves them. These objects are useful especially when they have to be unique during the system execution. In these cases, we apply the Singleton pattern [6] to the composite objects. For instance, the ConceptsList class contains a collection of objects of the Concept class. We observe in Fig. 3 the ConceptsList class fulfils the Singleton pattern containing three elements:

- a single and private constructor (the operation create());
- a private and static variable named instance of the type ConceptsList, i.e. the class which belongs, and
- a public and static operation called getInstance() that returns (using the variable instance) a reference to the unique object of the ConceptsList class.

There are also other classes that fulfill the Singleton pattern such as Ontology and TopicsList (Fig. 3).

The last category of the model objects is constituted by manager objects. These objects have the responsibility to manage the operations with the ontology or files. The manager objects deal with loading and saving the individuals from/in ontology.

Finally, the controller objects have the responsibility to manage the logical flow and the events produced by users in their interactions with view objects. For instance, the part of the software architecture that contains the design classes used in the “Create scene” use case execution is shown in Fig. 3.

In order to obtain a quality and modular software architecture we applied the design principles of low coupling, high cohesion, and assignment of responsibilities. These principles are fulfilled if we use the general responsibilities assignment patterns (shortly, GRASP) like Information Expert, Creator, Low Coupling, High Cohesion, Controller, and Polymorphism [7].

The behavioral aspect of the software architecture is shown by sequence diagrams. They present the interactions between the user and system in terms of objects and messages sent between them. For example, Fig. 4 and Fig. 5 depict the sequence diagrams of the “Create scene” and “Save scene as XML file” software use cases.

Most of the messages are synchronous, such as those from Fig. 4, but also there are asynchronous messages, such as some messages of the sequence diagram of the “Automatically save scene” use case.

Moreover, OntSceneBuilder integrates a distributed virtual reality toolkit called the ARéViJava [8] (Fig. 6(b)). For example, the ARéViJava and ControlPanel classes of Fig. 3 are part of the ARéViJava toolkit. The first class is a graphical user interface that contains a 3D canvas used to render the virtual scene. The second one permits users to load inside the virtual scene objects and dynamically attach to them specific behaviors.

In essence, ARéViJava is a Java3D based open-source API, used for rendering of the dynamic scene, that is adaptive to different configurations, ranging from desktops to 3D stereoscopic immersion systems and integrating a wide sort of interaction devices from mouse and joystick to space-mouse and WiiMote. It implements a reactive agent-based architecture that permits us to build immersive and interactive virtual spaces as were defined in [9].

4 Implementation

For the implementation of the OntSceneBuilder system, we used the Java programming language (Java Developing Kit 1.6 version).

The ontology was written using the languages Web Ontology Language (OWL) and Semantic Web Rule Language (SWRL) with the assistance of the Protégé tool [10] (version 3.4).

To work with our ontology, we used the Protégé-OWL API. Nevertheless, other object oriented software libraries for the management of OWL ontologies could be used, such as O3L [11, 12], which derives from OWLET [13], an object-oriented environment that maps an OWL ontology into a Java representation.

The Protégé-OWL API is an open-source Java library for OWL and RDF [14]. For example, the concepts tree of Fig. 6(a) shows a part of the ontology taxonomy that contains the concepts that belong to the domain ontology, but do not to the imported ontologies, such as DOLCE and D&S. Using the Protégé-OWL API we read all the RDFS/OWL classes of the ontology and filtered them taking only the concepts and their superclasses by which we need them.

Moreover, due to the ARéViJava framework facilities, once the virtual scene is created, the user may adopt different navigation and/or interaction metaphors (e.g. hover, drive, fly, orbit, spacemouse in Fig. 6(b)) in order to obtain better views of the virtual environment and to adjust its topology by means of direct 3D interaction on the virtual artifacts.

The system functioning has been tested to construct virtual scenes of a virtual historic exposition. In this case, the museum custodian “plays” the role of user in our system. With the assistance of the system, the custodian creates his/her own virtual scenes of a historic exposition based on a domain ontology and 2D/3D files incorporated in system (Fig. 6(c)).

5 Related Work

The Intelligent 3D Visualization Platform - I3DVP [15] is a framework for the enhancing virtual scenes with semantic

information and for performing reasoning by inference on content and the semantics of the scenes. The framework uses two kinds of OWL ontologies: a graphic ontology (called OntologyX3D) which describes graphics and virtual reality concepts of 3D models and their animation and is based on VRML and X3D standard; and domain ontologies. The used domain ontology is mapped to OntologyX3D through 15 relationships described by the classes of another intermediate ontology. The virtual scenes created by I3DVP are organized as graphs of OWL individuals (i.e. instances of OWL classes) of the OntologyX3D. Instead, in our system, a scene is organized as a graph of virtual artifacts, where each virtual artifact is associated with a 3D resource which in turn is related to a concept from the domain ontology used.
Moreover, I3DVP permits users to construct and manipulate virtual scene through sets of rules that allow inference-based decision making [15].

The HANNAH framework [16] is based on an ontology converted to a database to generate visual graph representations of 3D layouts. The OntSceneBuilder does not use a database, but files to store ontologies, the topics list, and so on.

The MUG application [17] allows users to author the structural, behavioural and functional knowledge about a design in a 3D virtual environment of CAD system. Knowledge is described by ontologies written in DAML language [18]. The ontologies are used for synchronous communication and interaction of designers, annotation and saving designs in DAML files. The application is useful for the collaborative creation of conceptual designs for devices.

6 Conclusions and Future Work

This paper presented the OntSceneBuilder system and a part of its development process. The system uses a domain ontology to construct the virtual scenes of a virtual exposition. Therefore, the system software architecture contains classes which manage the domain ontology chosen by a user. The quality of the software architecture is ensured by the MVC architectural pattern and the design patterns applied: Information Expert, Creator, Low Coupling, High Cohesion and so on.

Our work to improve the current OntSceneBuilder system will include reasoning for modifying of the scenes. For the time being, OntSceneBuilder allows users to modify values of the virtual artifacts properties and the resources of the related concepts.

OntSceneBuilder also uses single domain ontology, but we will take into account the problem that the designer chooses to use two or many domain ontologies. In this case, we will need a mapping of the imported ontologies. There already are some methods to realize this mapping, such as that proposed in [19]. The method uses the information flow theory to make a semantically relation between two ontologies using a family of infomorphisms between concepts and their instances.

Further work will also consider the ontological description of the processes, activities, and agent states that will be incorporated into the virtual scenes as ARéViJava behavior objects.

Acknowledgements

This work is partially funded within the "Using virtual reality in 3D Multi-modal reconstruction of historical sites" (TOMIS) project, no: 11-041/2007, by the National Centre of Programs Management, PNCDI-2 – Partnerships program.

References:

[1] C. Bogdan, Domain Ontology of the Roman Artifacts found in the Tomis Fortress, *Knowledge Engineering. Principles and Techniques*. Selected Papers, editors Militon Frenţiu and Horia F. Pop, Presa Universitara Clujeană, Romania, 2009, pp. 117-123.

[2] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari, *WonderWeb Deliverable D18. Ontology Library*. IST Project 2001-33052 WonderWeb: Ontology Infrastructure for the Semantic Web, 2003

[3] A. Gangemi, P. Mika, Understanding the Semantic Web through Descriptions and Situations, *Proceedings of the International Conference ODBASE03*, Italy, Springer, 2003, pp. 689-706.

[4] OMG, *Unified Modelling Language Superstructure*, version 2.0, ptc/03-0802, 2003

[5] R. S. Pressman, *Software Engineering. A Practitioner's Approach*, McGraw-Hill Publishing Company, 2000

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, 1994

[7] C. Larman, *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall, 2004

[8] P. Reignier, F. Harrouet, S. Morvan, J. Tisseau, T. Duval, ARéVi: A Virtual Reality Multiagent Platform, *Lectures Notes in Computer Science*, Vol. 1434, 1998, http://www.cerv.fr/fr/activites/AReVi.php, accessed 10 september 2009

[9] D. M. Popovici, L. D. Serbanati, F. Harrouet, The Virtual Environment-Another Approach. *Proceedings of the 11-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* 2003 in cooperation with Eurographics (WSCG'2003), Plzen, Czech Republic, 2003, pp. 109-112.

[10] J. Gennari, M. Musen, R. Fergerson, W. Grosso, M. Crubzy, H. Eriksson, N. Noy, S. Tu, The evolution of Protégé-2000: An environment for knowledge-based systems development, *International Journal of Human-Computer Studies*, 58(1), 2003, pp. 89-123.

[11] Poggi, A., O3L: An OWL Object-Oriented Library for the Realization of Ontology Based Applications. In Proc. of the 13th WSEAS International Conference on COMPUTERS, pp. 340-345, Rhodes, Greece, 2009

[12] Poggi, A., Developing Ontology Based Applications with O3L, *WSEAS Transactions on Computers*, 8(8): 1286-1295, 2009

[13] Poggi, A., OWLET: An Object-Oriented Environment for OWL Ontology Management. In Proc. of the 11th WSEAS International Conference on COMPUTERS, pp. 44-49, Agios Nikolaos, Greece, 2007.

[14] Protégé-OWL API, link: http://protege.stanford.edu/plugins/owl/api/, accessed 4 september 2009

[15] E. Kalogerakis, S. Christodoulakis, N. Moumoutzis, Coupling Ontologies with Graphics Content for Knowledge Driven Visualization, *Proceedings of the IEEE Virtual Reality Conference* (VR'06), 2006, pp. 43-50.

[16] K. Einsfeld, A. Achim Ebert, J. Wölle, Modified Virtual Reality for Intuitive Semantic Information Visualization, *Proceedings of the 12th International Conference Information Visualization*, 2008, pp. 515-520.

[17] D. C. Cera, W. C. Regli, I. Braude, Y. Shapirstein, C. V. Foster, A Collaborative 3D Environment for Authoring Design Semantics, *Drexel University Technical Report DU-MCS-01-06*, 2001, pp. 1-16.

[18] DAML Language, http://www.daml.org/language/, accessed 8 december 2009

[19] H. Liu, H. Bao, J. Feng, Instance Assisted Ontology Alignment for Digital Museums, *Proceedings of the 7th WSEAS International Conference on Simulation, Modelling and Optimization*, Biejing, China, 2007, pp. 79-84.

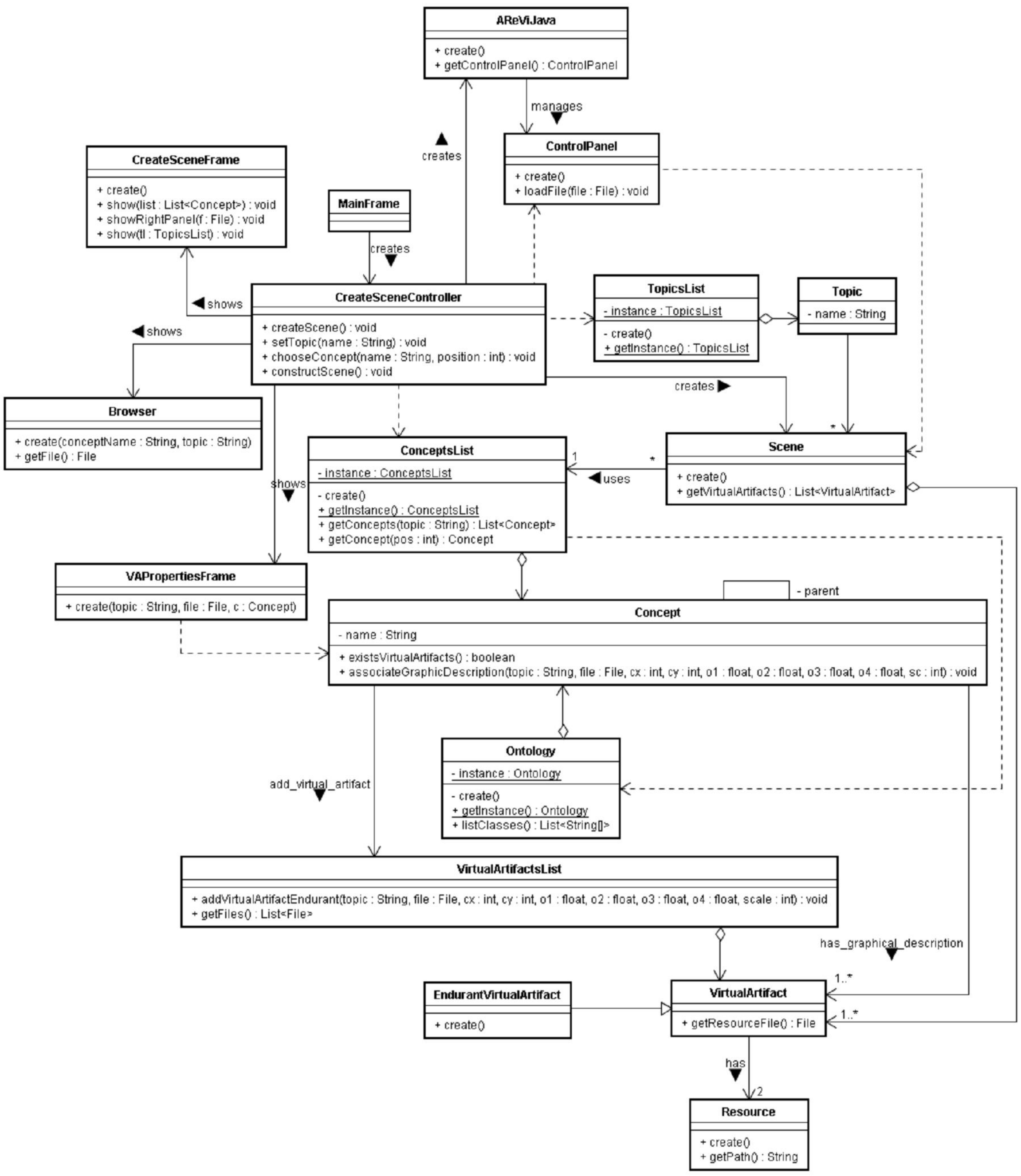
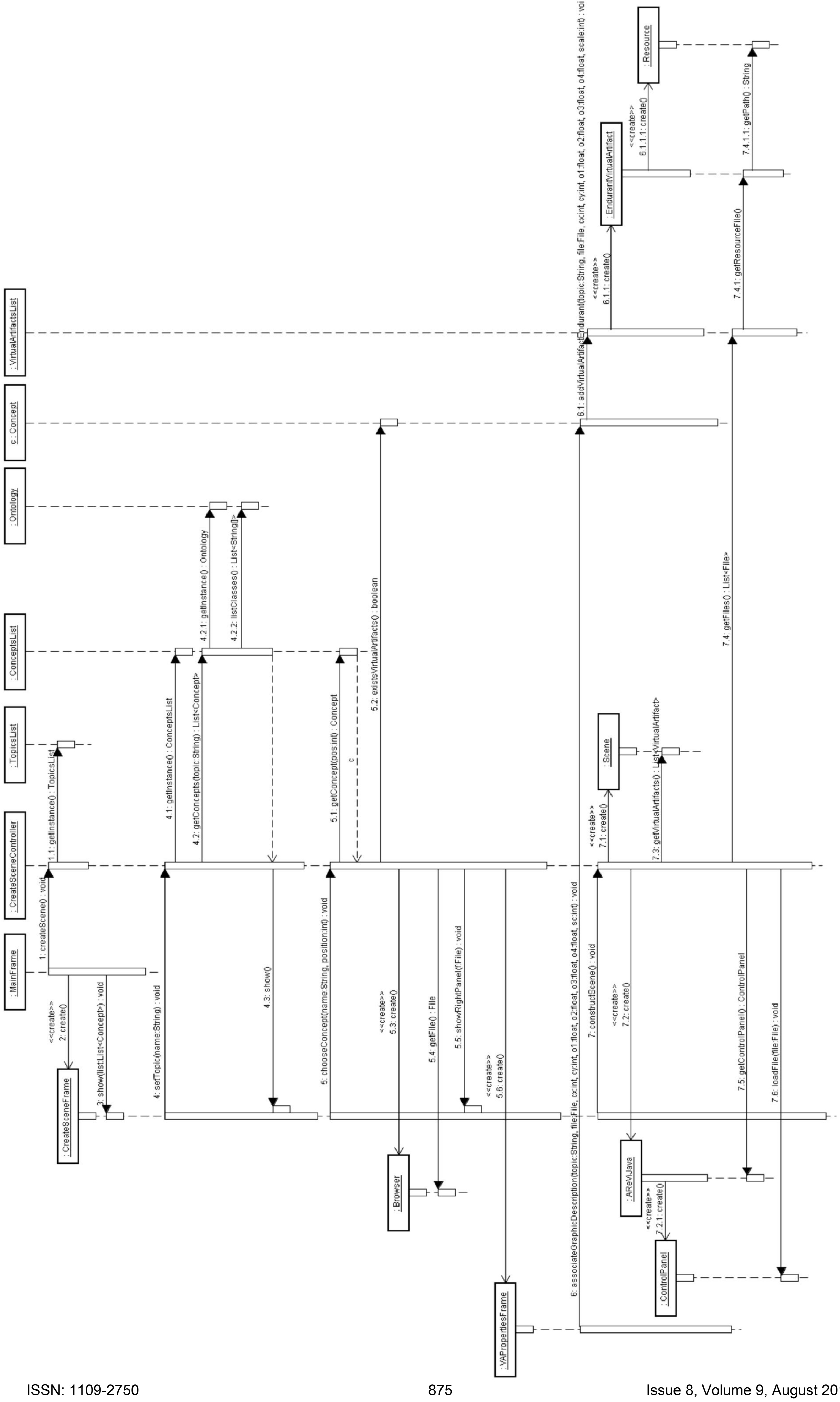

AReViJava
+ create()
+ getControlPanel() : ControlPanel
manages
ControlPanel
+ create()
+ loadFile(file : File) : void
creates
CreateSceneFrame
+ create()
+ show(list : List<Concept>) : void
+ showRightPanel(f : File) : void
+ show(tl : TopicsList) : void
MainFrame
creates
shows
shows
CreateSceneController
+ createScene() : void
+ setTopic(name : String) : void
+ chooseConcept(name : String, position : int) : void
+ constructScene() : void
TopicsList
- instance : TopicsList
- create()
+ getInstance() : TopicsList
Topic
- name : String
creates
Browser
+ create(conceptName : String, topic : String)
+ getFile() : File
ConceptsList
- instance : ConceptsList
- create()
+ getInstance() : ConceptsList
+ getConcepts(topic : String) : List<Concept>
+ getConcept(pos : int) : Concept
1
uses
*
Scene
+ create()
+ getVirtualArtifacts() : List<VirtualArtifact>
*
shows
VAPropertiesFrame
+ create(topic : String, file : File, c : Concept)
- parent
Concept
- name : String
+ existsVirtualArtifacts() : boolean
+ associateGraphicDescription(topic : String, file : File, cx : int, cy : int, o1 : float, o2 : float, o3 : float, o4 : float, sc : int) : void
Ontology
- instance : Ontology
- create()
+ getInstance() : Ontology
+ listClasses() : List<String[]>
add_virtual_artifact
VirtualArtifactsList
+ addVirtualArtifactEndurant(topic : String, file : File, cx : int, cy : int, o1 : float, o2 : float, o3 : float, o4 : float, scale : int) : void
+ getFiles() : List<File>
has_graphical_description
1..*
EndurantVirtualArtifact
+ create()
VirtualArtifact
+ getResourceFile() : File
1..*
has
2
Resource
+ create()
+ getPath() : String

Fig. 3 : The design class diagram of the “Create scene” software use case


: MainFrame
: CreateSceneController
: TopicsList
: ConceptsList
: Ontology
c : Concept
: VirtualArtifactsList
: CreateSceneFrame
: Browser
: VAPropertiesFrame
: Scene
: AReVJava
: ControlPanel
: EndurantVirtualArtifact
: Resource
1: createScene() : void
1.1: getInstance() : TopicsList
<<create>>
2: create()
3: show(listList<Concept>) : void
4: setTopic(name:String) : void
4.1: getInstance() : ConceptsList
4.2: getConcepts(topic:String) : List<Concept>
4.2.1: getInstance() : Ontology
4.2.2: listClasses() : List<String[]>
4.3: show()
5: chooseConcept(name:String, position:int) : void
5.1: getConcept(pos:int) : Concept
c
5.2: existsVirtualArtifacts() : boolean
<<create>>
5.3: create()
5.4: getFile() : File
5.5: showRightPanel(f:File) : void
<<create>>
5.6: create()
6: associateGraphicDescription(topic:String, file:File, cx:int, cy:int, o1:float, o2:float, o3:float, o4:float, sc:int) : void
6.1: addVirtualArtifactEndurant(topic:String, file:File, cx:int, cy:int, o1:float, o2:float, o3:float, o4:float, scale:int) : voi
<<create>>
6.1.1: create()
<<create>>
6.1.1.1: create()
7: constructScene() : void
<<create>>
7.1: create()
<<create>>
7.2: create()
<<create>>
7.2.1: create()
7.3: getVirtualArtifacts() : List<VirtualArtifact>
7.4: getFiles() : List<File>
7.4.1: getResourceFile()
7.4.1.1: getPath() : String
7.5: getControlPanel() : ControlPanel
7.6: loadFile(file:File) : void
Fig. 4: The sequence diagram of the “Create scene” use case

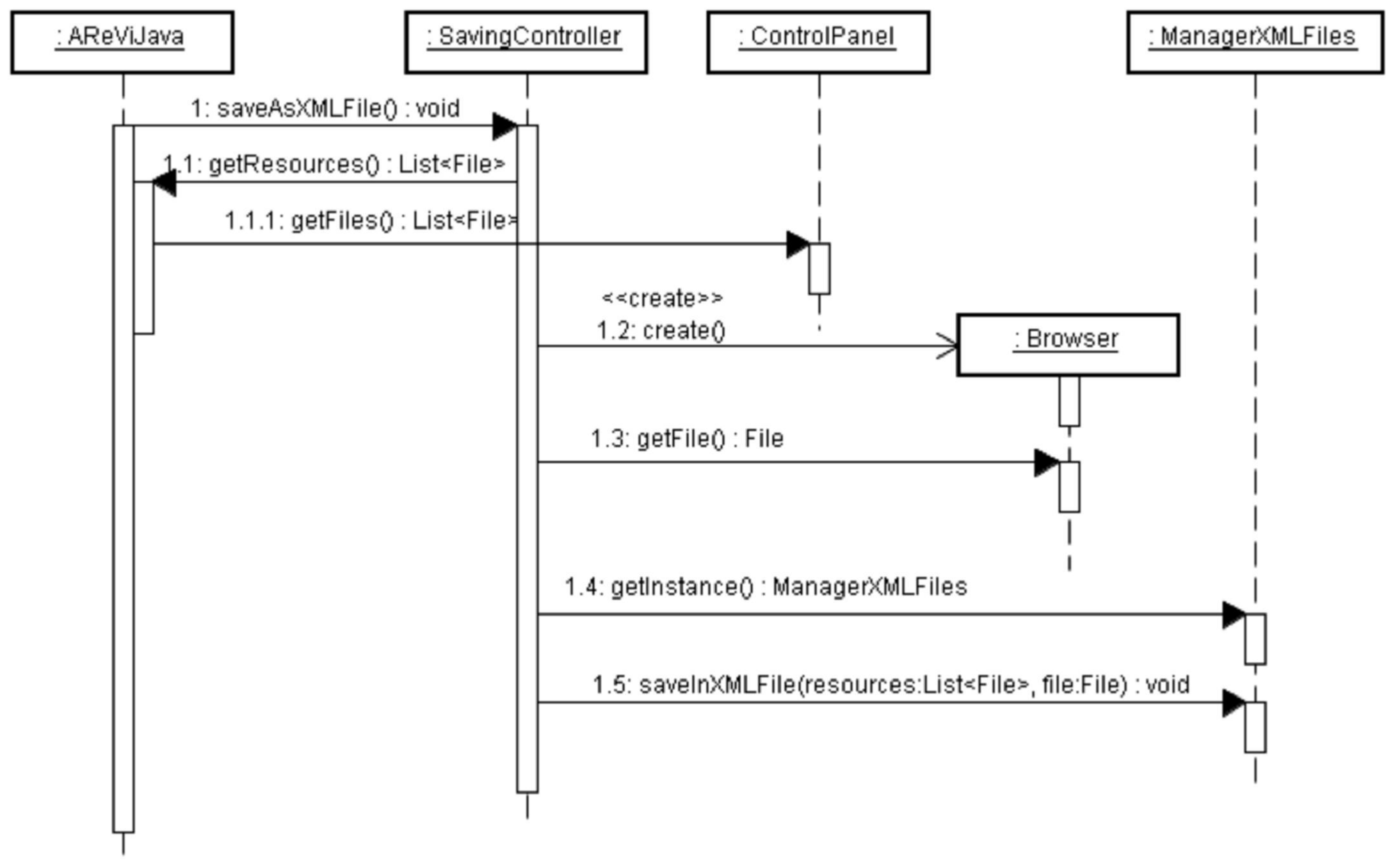
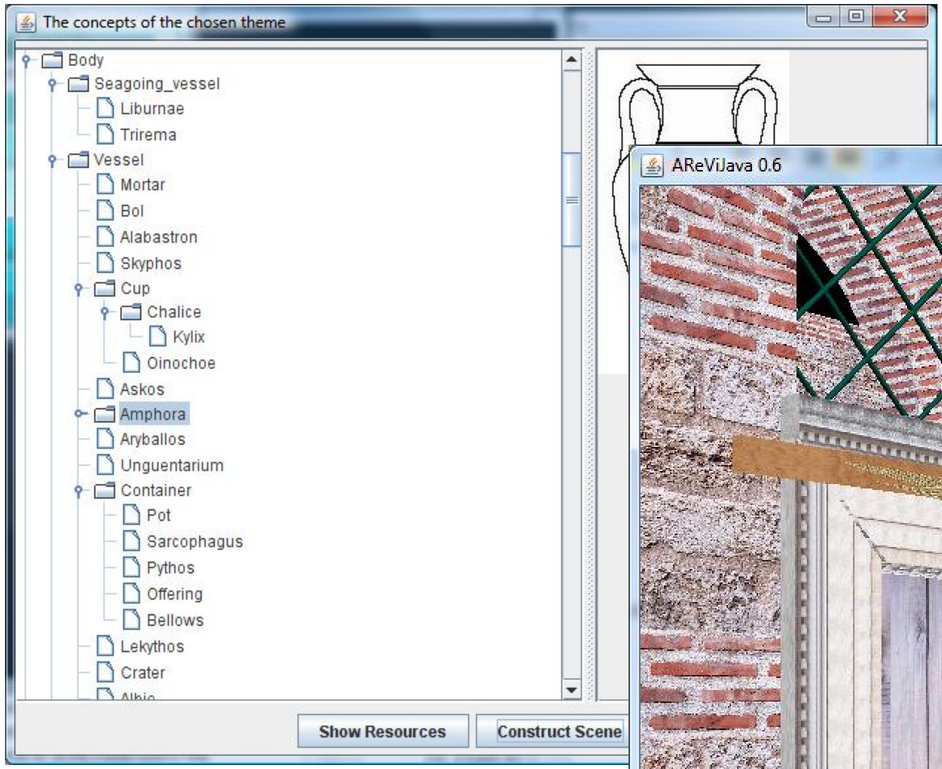
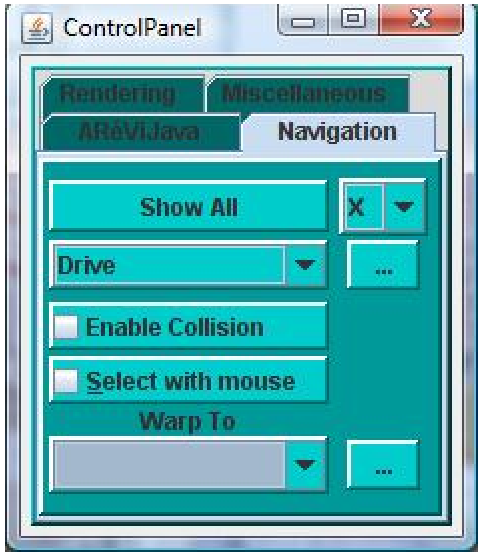

: AReViJava
: SavingController
: ControlPanel
: ManagerXMLFiles
: Browser
1: saveAsXMLFile() : void
1.1: getResources() : List<File>
1.1.1: getFiles() : List<File>
<<create>>
1.2: create()
1.3: getFile() : File
1.4: getInstance() : ManagerXMLFiles
1.5: saveInXMLFile(resources:List<File>, file:File) : void

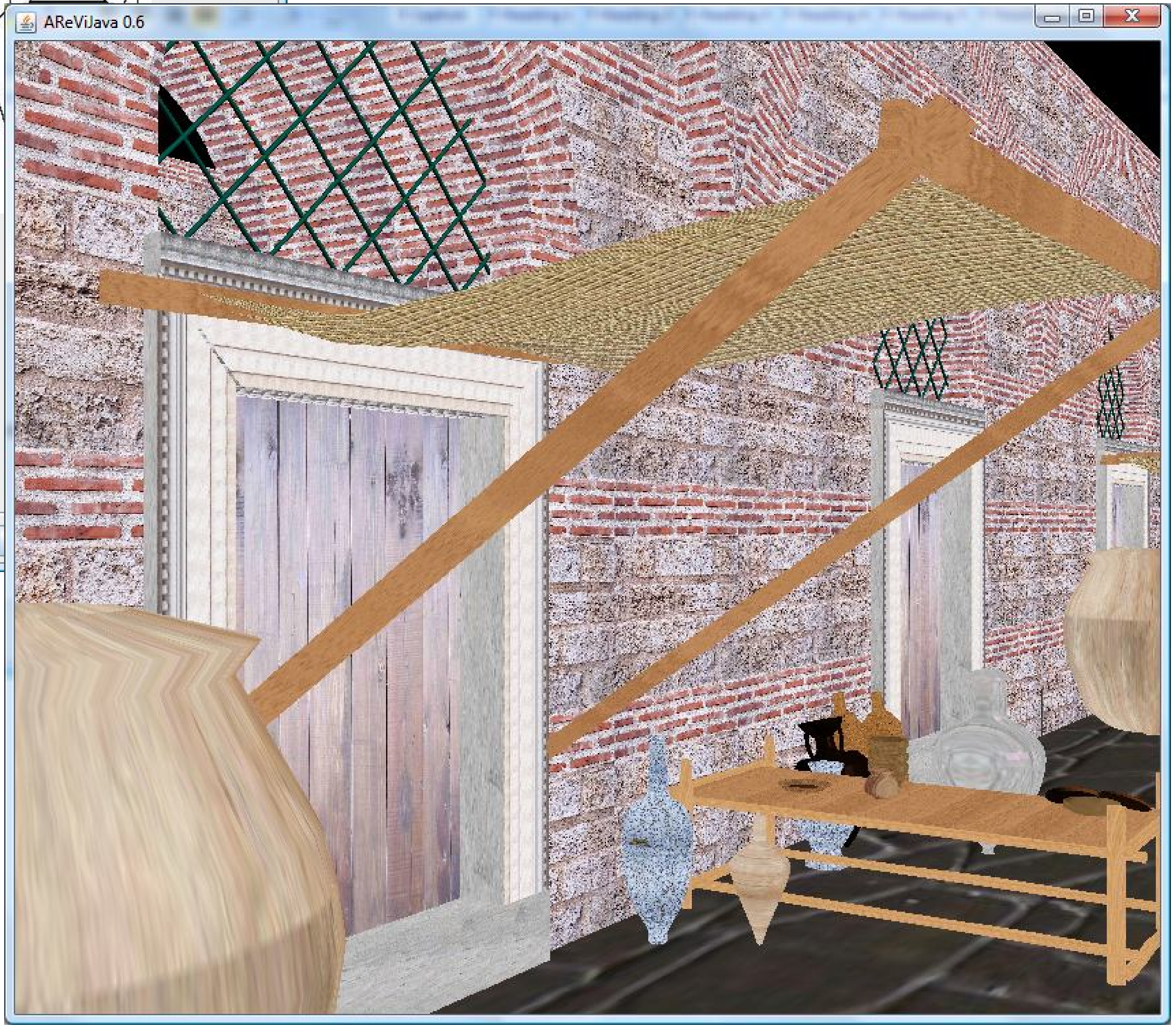
Fig. 5: The sequence diagram of the “Save scene as XML file” use case



(a) The choice of concepts from the ontology taxonomy and displaying their 2D resources



(b) The Navigation tabbed pane of the ControlPanel frame of the ARéViJava toolkit



(c) Displaying of a virtual scene

Fig. 6: OntSceneBuilder system execution