Master of Science in Computer Science

Staffordshire University UK, TEI of Larissa GR

Designing and Implementing the CROP Reference Architecture for Learning Objects: The CROP Editor

Master Thesis

Dimoklis Despotakis

Advisor: Prof. C. Hartonas

January 2009

Contents

CHAPTER 1 – INTRODUCTION

The evolution of technology and growth of knowledge over various scientific and non scientific fields of interest leads to the need of knowledge transfer and distribution using more effective solutions that can overcome the disadvantages of media in printed form. Actual factors that affect the quality of this procedure, the knowledge capture, formatting, distribution, presentation and testing, introduce the scope that *Learning Content Management Systems* (LCMS) intend to address.

Moreover, these systems have to manage information not in a raw data format, but in a structured way. At this point, we slightly introduce the notion of Learning Object, which consists in general a specific format of knowledge data. More on this are presented in Chapter 2.

This dissertation is a part of a more wide project, that involves the development of a Modeling Framework (MF) for the design, implementation and deployment of Learning Objects, based on the notions of Concept, Resource, Order, Product and Service. Currently, the *CROPS MF* is under development and its basic aims are to:

- Provide a reference architecture for Learning Objects introducing a compositional design pattern for learning objectives. This work has been done and is presented in [1].
- Provide a tool for CROP Learning Objects development. This part of the project consists the objective of this dissertation. The scope of this work is to design and implement an editor for the CROP Learning Objects, the CROP Editor. The main aspect of the Editor is to support all the features that derive from the CROP Reference Architecture at [1] and facilitate the production and editing of Learning Objects.
- Provide a tool for Learning Object Metadata creation, including the possibility to be compatible and support existing tools (IEEE

LOM v.1.0, CanCore, UKLOMCore). This is a work in progress that is elaborated by Professor C. Hartonas.

- Develop a new proposal for managing learning content, based on a particular species of Semantic Web Services, which are called *Learning Services in the CROPS MF*. This part of the project is subdivided into two main aspects:
    - Provide an account of adaptivity of learning objects as an emerging property of learning object and learning service interaction. A preliminary work has been done by Tom Katsaros in his dissertation at [7].
    - Provide a specification of Learning Services, by extending some currently predominant framework (such as OWL-S or WSMO). There is work in progress relating to this issue as part of the PhD dissertation work by Maria Tsiakmaki [8].

This report can be divided into two main Sections: the *Background and State of the Art* and the *Requirement Analysis and Implementation.* The first section includes a brief review on the Learning Objects and the Learning Object Metadata Standard (LOM Standard), the IMS Implementation Guide to Learning Design and the SCORM Specification in Chapter 2, while it focuses on the CROP Reference Architecture analysis in Chapter 3 and the CROP Learning Object development stages in Chapter 4.

The second section cites the Requirements Analysis and the design decisions in Chapter 5, the CROP Editor implementation in Chapter 6 and a development example at runtime in Chapter 7.

Chapter 8 concludes this thesis citing the evaluation of the implemented features and their advantages over the Learning Object editing process, while addresses open issues to discuss and future work and improvements that could be done.

The actual product of this thesis is a software application written in Java code, whereof a code segment of the application· the CROP Core package· is attached as an Appendix to this report.

CHAPTER 2 – LEARNING OBJECTS AND LEARNING OBJECT METADATA

This Chapter introduces the *Background and State of The Art* section of this report and covers in brief the Learning Object and Learning Object Metadata theoretical background, while it presents the Instructional Management Systems (IMS) Implementation Guide and the SCRORM specification.

2.1 Learning Objects

Many definitions have been given to describe the notion of Learning Object (LO). A general approach to depict this concept can be that a Learning Object is a component that encapsulates data sources in a structural way to support knowledge extraction and distribution. Moreover, there is no unison between the attempts to define a Learning Object and the concept by itself tends to be vaguely described.

The Institute of Electrical and Electronics Engineers (IEEE) defines a Learning Object as "*any entity, digital or non-digital, that may be used for learning, education or training*" at [9]. David A. Wiley at [10] cites the LO definition as "*any digital resource that can be reused to support learning*".

Changing the scope that the above two definitions are focused, R. McGreal at [3], argues that a definition that contains words as "any", "everything" etc. is not a definition because there is nothing that can distinguish the object under description from other similar. He defines

LO's as "*a digital file (image, movie, etc.) intended to be used for pedagogical purposes, which includes, either internally or via association, suggestions on the appropriate context in which to use the object*".

The CROP Reference Architecture [1, 2] generally encapsulates this definition and slightly introduces a more tight and specific role to LOs that derives from the architecture itself while enriches the definition, becoming more accurate and expositive. In Chapter 3 the CROP Learning Object is fully described and the standard of the actual internal structure that CROP aims to put forward is analyzed.

2.2 Learning Object Metadata (LOM)

2.2.1 LOM Standard Definition

As it is documented in [9] the IEEE Learning Object Meta-data (LOM) Standard specifies and documents the appropriate format and attributes that learning resources must adopt in order to be fully descriptive and well presented. To achieve this purpose the IEEE LOM Standard defines a set of metadata elements that can be used to describe learning resources. Such elements are:

1. Names

2. Definitions

3. Data types

4. Field lengths

IEEE LOM Standard defines a conceptual model for the metadata and an XML binding also, so it comprises a multi-part standard. It includes guidelines of how documents must be formatted and how applications can exchange information and manipulate them.

2.2.2 The LOM Document Structure and Data Schema

The LOM documents follow a tree structure that is hierarchically organized. So, the "root" is the first element of hierarchy. Additional elements the so called sub-elements below the root are called "branches" if they also have sub-elements or "leaves if they are terminal nodes. An example document hierarchy taken by [11] is shown at Figure 2.1 below.

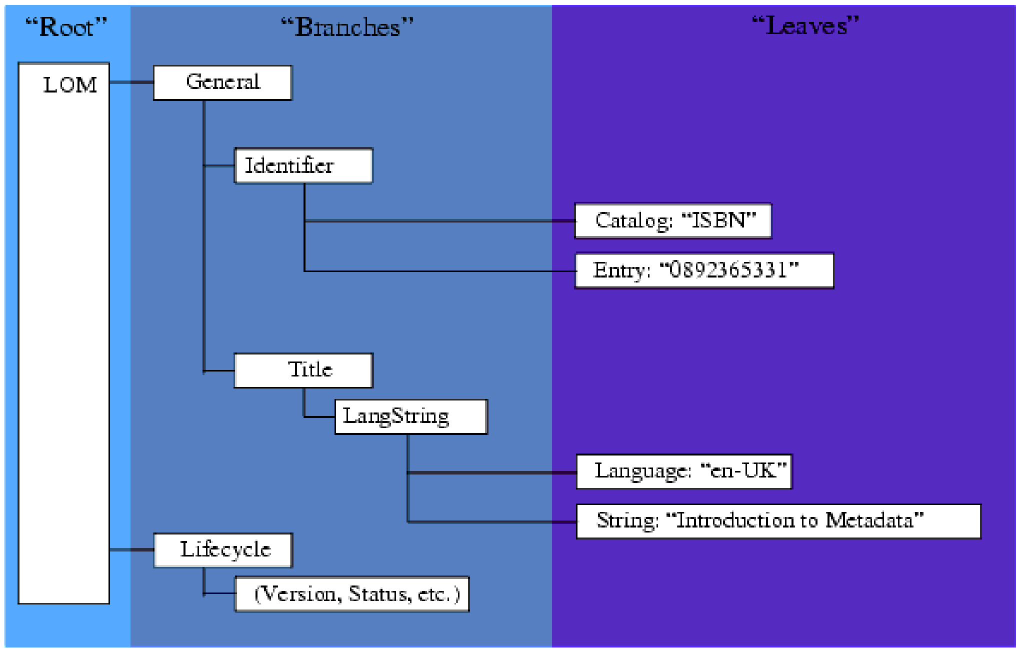


Figure 2.1 Root to leaf "tree view" of metadata [11].

The LOM Data Schema lists all the metadata elements in tabular format. General, Life Cycle, Meta-Metadata, Technical, Educational, Rights, Relation, Annotation, and Classification are the nine main categories of element types that LOM defines using again a tree representation with branches and leaves guide to several different types (Figure 2.2). Each leaf element in the LOM conceptual data schema has a data type and a value space that defines the encoding of the data for that element. Those spaces and data types are listed below:

1. Repertoire of ISO/IEC 10646-1:2000 Value Space (Unicode 3.0.1 characters, glyphs etc.)
2. LanguageID Value Space (Language definition like En- UK)
3. MIME Types Value Space (encoding of digital format of a resource)
4. vCard Value Space (encodes information similar to that found on a business card)
5. CharacterString Data type
6. LangString Data type
7. Vocabulary Data type (limited choice of words or phrases)
8. DateTime Data type
9. Duration Data type (Duration and a Description like P3M for three months period)

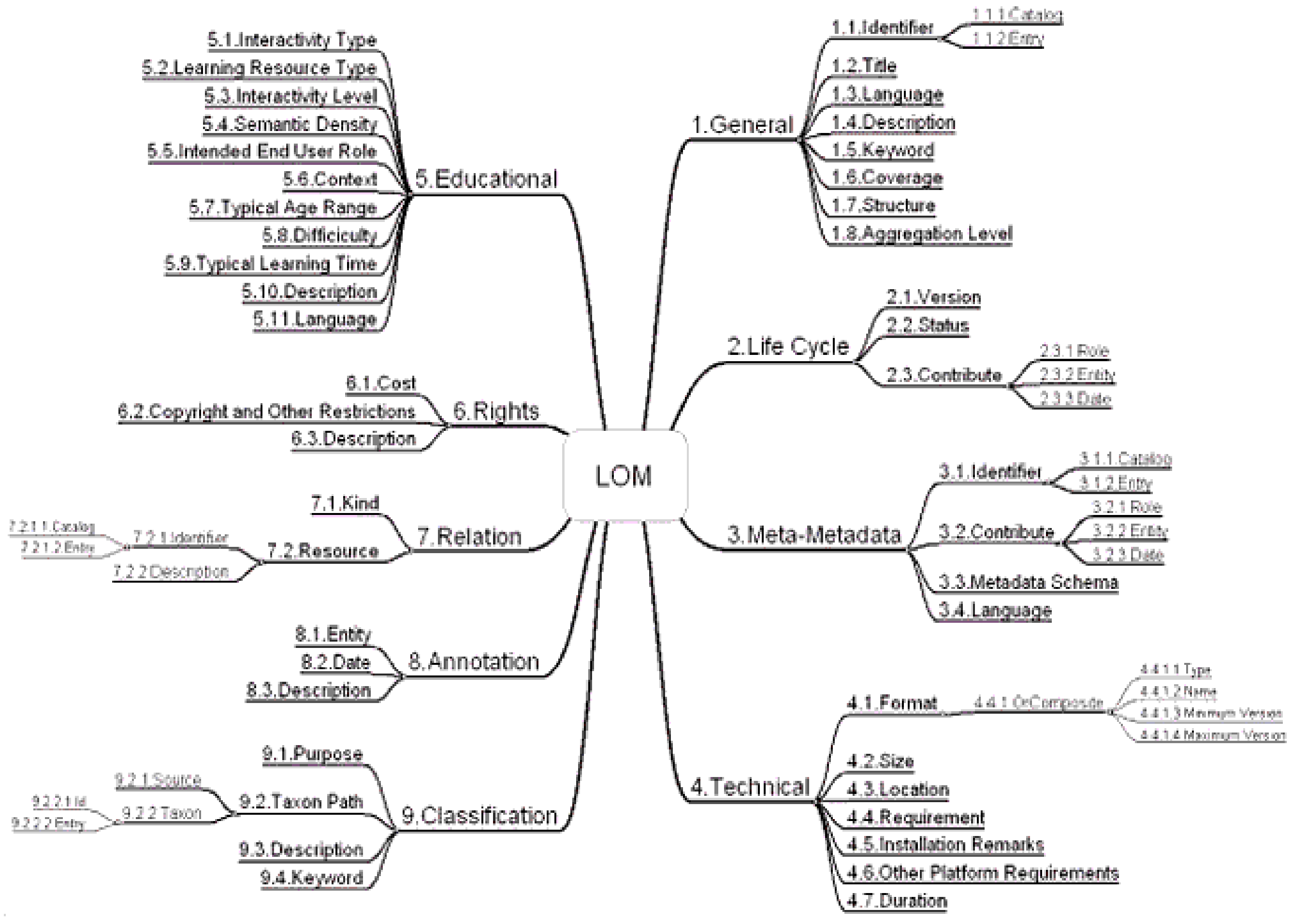


Figure 2.2: The elements and structure of the LOM conceptual data schema [9].

2.3 Learning Object Development Standards

2.3.1 The IMS Implementation Guide

An Instructional Management Systems (IMS) Implementation Guide is defined at [14] that addresses issues for developing a framework to create and support pedagogical material. The main objectives are to define specifications for managing learning units focusing on:

1. Resources
2. Instructions for learning activities
3. Templates for structured interactions
4. Conceptual models (e.g., problem-based learning)
5. Learning goals, objectives and outcomes
6. Assessment tools and strategies

Main goals that these specifications must cover as they presented at [14] are listed below:

1. Different learning methods must be defined
2. Enable repeatable, effective, and efficient units of learning
3. Interchange of learning resources
4. Different types of knowledge delivery
5. Usability
6. Support the reuse or repurposing of the framework and components of a unit of learning
7. Amplification of specifications and standards
8. Accessibility (internationalization)
9. Support multiple learners and multiple roles in a learning activity, reflecting learning experiences that are collaborative or group-based
10. Support reporting and performance analysis

2.3.2 Content Packaging: The SCORM Specification

The Shareable Content Object Reference Model available at [15] illustrates a specification and a model description derived from the work at the Advanced Distributed Learning (ADL) Initiative. SCORM introduces guidelines to develop learning content, implementing its labeling, storage mechanism and presentation over distributed learning.

The basic principles that SCORM adopts for constructing LO's in e learning environments, or else the "*ilities*" as they cited at ADL, contain the following:

1. Accessibility: Locate and gain access to instructional components (say
2. LO's) from distance and share them to many other locations.
3. Adaptability: Adjustment of learning material according to individual needs.
4. Affordability: The ability to increase efficiency and productivity by reducing the time and costs involved in delivering instruction.
5. Durability: The ability to acquaint instructional components following technology evolution and new specifications with minor cost of redesign and recoding process.
6. Interoperability: Strong compatibility between different platforms and environments (say UNIX or MS Windows environments can use and manage the same learning component).
7. Reusability: High level utilization between different applications.

ADL adopts another concept for SCORM, the "*Web based assumption*" which concerns the collaboration of Web standards to efficiently increase instructional material management, access and reusability. This came out from the fact that Web and internetworking are rapidly expanding, Web based learning technology specifications are not stabilized yet and its content can be delivered using any medium (say optical medium disks). Combining the above requirements with this assumption, ADL produced the following principles for SCORM operation:

1. A Web based Learning Management System (LMS) must be able to manage learning content that is authored using different tools on different platforms and use that content to exchange data.
2. Different Web based LMS's must have the ability to use the same content and be able to exchange data with it.
3. Multiple Web based LMS's can share the same content source.

CHAPTER 3 – THE CROP REFERENCE ARCHITECTURE

In this Chapter I describe the CROP Reference Architecture for Learning Objects [1, 2]. I cite the internal structure that the CROP tends to standardize for constructing LOs and I explain in details each structural component, its theoretical background and its technical presence inside an LO. Moreover, I give an example of implementing an LO using the CROP Architecture to depict the features and facilitations that CROP introduces when is applied.

This architecture is based on the notions of **C**oncept, **R**esource, **O**rder and **P**roduct and its aim is *"to put forward a standard of internal structure that a learning object is to abide by, without making any commitment to particular educational/teaching theories, styles or preferences…"* as pointed out in [1].

The basic structural component of a CROP LO is its ontology, whereof the notion of *Concept* is derived, meaning a structured class of concepts that shapes the learning material that an LO introduces. CROP introduces a novel hierarchy model between this concept instances and separates this from the subsumption hierarchy that ontology has, using the *Subclass-of* axiom defined in the Ontology Web Language (OWL) [12]. The hierarchy that CROP introduces is a learning hierarchy between concepts for a specific learning subject, which is constructed using the *hasPrerequisite* relation. Thus, a concept has another concept as a learning prerequisite. This relation can be visualized as a directed edge, where the source anchor is bounded to the prerequisite concept and the target anchor is bounded to the objective concept. Hence, this ontology, i.e the content ontology of an LO acquires a cohesive, rooted and directed graph structure as Figure 3.1 shows.

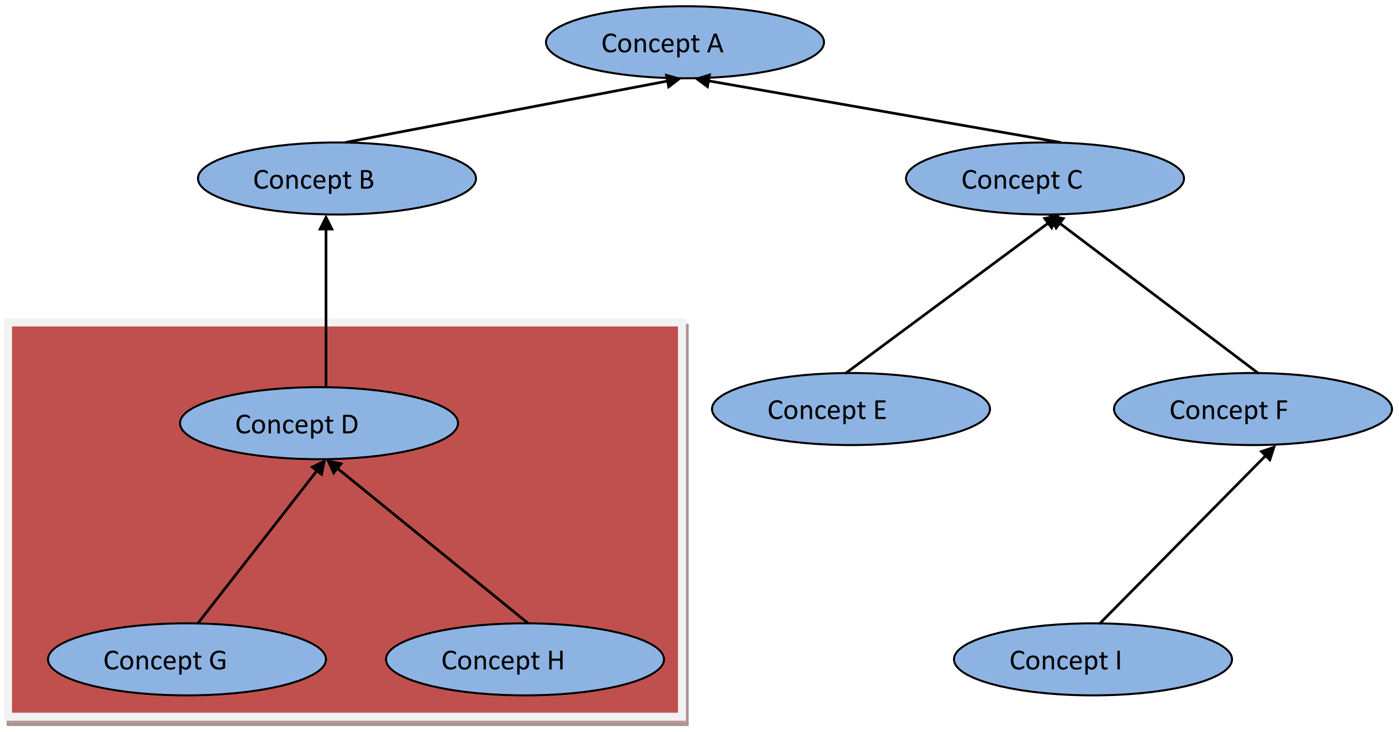


Figure 3.1 Learning Object Content Ontology. A directed edge defines a prerequisite relation.

Considering the above graph as the content ontology representation for an LO, *Concept A* consists the *Target Concept* of the learning object, i.e. its learning objective, the top concept that the LO intends to teach to the user - learner.

Moreover, using the same graph as an example, one or more sub-graphs can be extracted including those concepts that are more tightly interrelated. This sub-graph consist the graph for the content ontology of another LO with *Target Concept* the *D* concept in our graph (red colored area in Figure 3.1). Here is introduced the compositional model that CROP adopts meaning that an LO may contain another LO according to the designer's needs (Figure 3.2).

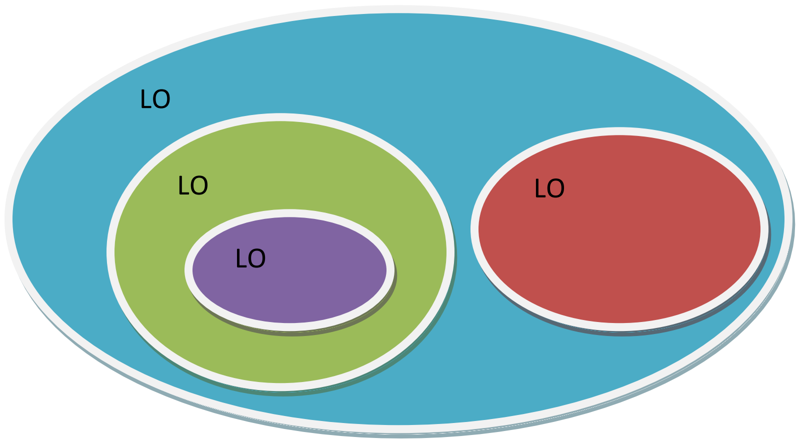


Figure 3.2 Compositional model

Next, I present the internal structure of a CROP Learning Object, explaining each structural component and their associations. Figure 3.3 shows the Model Diagram designed with UML [4].

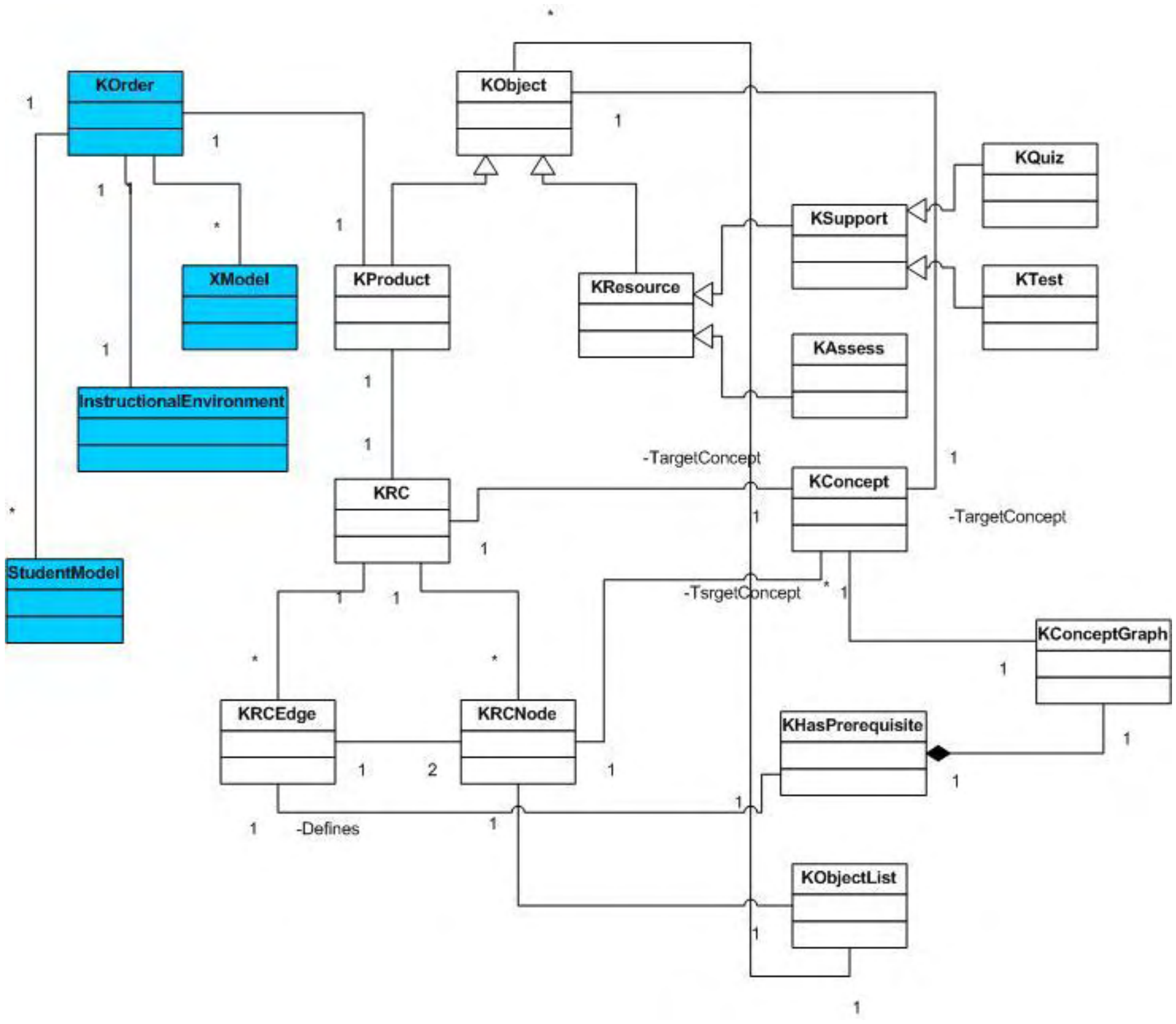


Figure 3.3 The CROP Reference Architecture Model Diagram.

This diagram presents the Model Diagram in UML for the CROP Reference Architecture. Starting from the bases of the architecture, the **KConcept (KnowledgeConcept)** class in the diagram comprises all the concept instances in the content ontology of the Learning Object, hence, all the nodes in the **KConceptGraph** class. Every KConcept instance is learning objective. The **KHasPrerequisite** class comprises all the applications of the *hasPrerequisite* relation (property) discussed previously, between the KConcept instances. A KHasPrerequisite instance is depicted with a single-directed edge in the KConceptGraph of content ontology meaning that the KConcept instance being in the target anchor of the edge has as learning prerequisite the KConcept instance being at the source anchor of the edge. This dependence relation between concepts can be depicted from the context analysis of learning objective at it is cited in [1]. A KConcept instance in the content ontology graph of the learning object can be either a GroundConcept meaning that it has no prerequisite concepts, a *leaf* node in the tree structure, or an internal node concept.

A KConcept instance may be the TargetKConcept for **KObject** instance. As it is shown in Figure 3.3, using the abstraction of the KObject class, its instances can be either a **KProduct** or a **KResource.** The existence of a KConcept instance does not necessarily means that has to be bounded at KObject instance. Here is introduced the *hasTargetConcept* property in the CROP ontology between KConcept and KObject instances.

The KProduct class in Figure 3.3 is the actual Learning Object. These terms have the same meaning. A KResource instance (Knowledge Resource) can be of two kinds: either to support learning (**KSupport** resource), or to assess learning (**KAssess** resource). A KSupport instance is associated with digital resource file (image, document, video, web page etc.) via an appropriate URI and a KAssess resource instance can be of two kinds in our implementation: either a **KQuiz** (a question multiple choice styled answer) or a **KTest** (sequence of KQuiz objects)**.** Adopting the derived from the KObject inheritance of classes-objects each KSupport o KAssess instance is associated with a TargetKConcept.

Moreover, a KProduct is fully described by its content ontology that defines its **KRC** graph. The KRC graph has a "1-1" correlation with the KConceptGraph or a sub-graph of it. A **KRCNode** emerges from a KConcept instance having this as its TargetKConcept. Similarly, a **KRCEdge** emerges from a KHasPrerequisite edge instance in the KConceptGraph.

A KRCNode instance, maintains a KObjectList that comprises its *node type*. Each KObject instance in the KObjectList has the same KConcept as its TargetKConcept, similar to the KRCNode. Thus, a KProduct (Learning Object) can encapsulate another Learning Object, which results the compositionality of the CROP architecture.

A KRCNode in the KRC graph is actually a teaching act. It is a learning objective that has to be taught, where a KRCEdge consist a teaching step. The sequence and the conditions under which a KRCNode will be taught, arises from the **KOrder** of the KProduct under discussion. The KOrder class contains instances of execution models, **XModel** instances associated with **StudentModels** running under an **Instructional Environment**. An XModel instance is also a graph structure that contains the KRC graph of the KProduct plus other type of nodes and edges that will be discussed later in Chapter 6 where the implementation of the CROP Editor takes place. A StudentModel is a class that actually describes the profile of a learner according to the succeeded assessment he took or his learning preferred style.

Every KObject instance in the CROP Architecture is associated with a LOM document. The LOM structure as it is briefly described in Chapter 2 and can be found at [9] is engineered to describe, present and maintain information about a Learning Object. The CROP Reference Architecture [1, 2], adopts this structure and cites a LOM implementation reengineered as ontology. This is the LOM Ontology available at [13]. The association between KObject instances and LOM elements takes place using the URI where the LOM document is stored.

Figure 3.4 below shows an example KProduct (a Learning Object) that contains two additional LO's in its content KRC graph. This KProduct has as learning objective (target concept) the "Computer" concept and it is concisely expanded to cover the example's purpose.

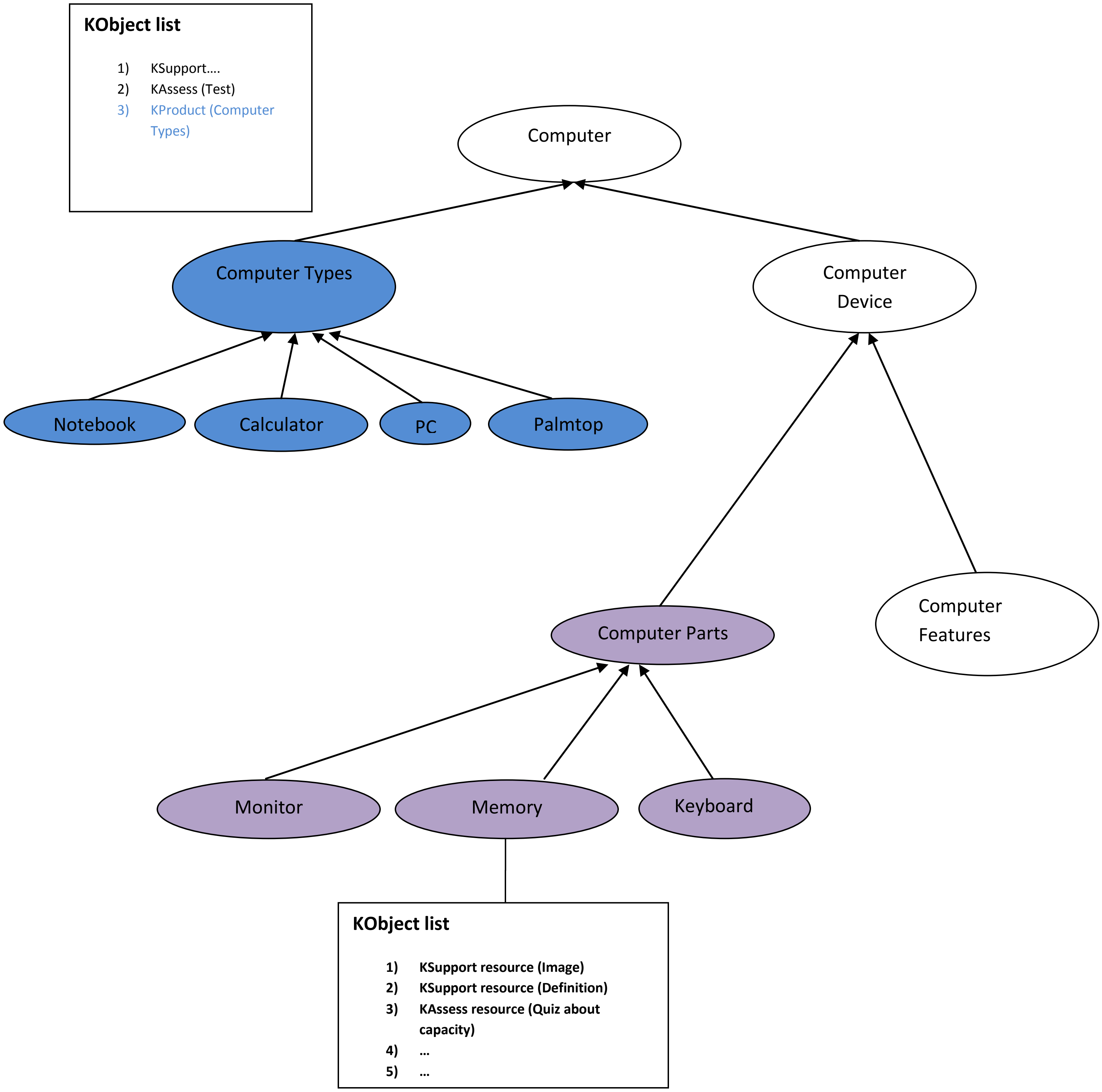


Figure 3.4 A CROP example for the "*Computer*" learning concept

CHAPTER 4 – DESIGNING CROP LEARNING OBJECTS

4.1 The Domain Ontology (KConcept) Development Stage

This stage of development CROP Learning Objects contains the procedure where the learning objectives (i.e. the concepts) are to be extracted and defined. In CROP a concept is a KConcept instance in the CROP ontology. These instances are concerned to be the learning objectives for a specific learning subject and inductively they conclude the knowledge space for the learning subject/objective. As it is cited in [1, 2] each KConcept instance can be handled as an atomic value and can be segmented into several other instances. This is actually a recursive process until the designer of the Learning Object closes to the *GroundConcept* instances of the Learning Object content ontology. Definition of KConcept instances presupposes a knowledge base existence where the learning subject is fully described or attached as a sub − component. The notion of knowledge base is very important at this stage of development it is considered to be fully descriptive for a learning subject. This actually is a basic quality factor for the consistency and completeness the Learning Object under development tends to cover according to its learning objective.

CROP architecture adopts the Ontology knowledge base tool and takes advantage of the Web Ontology Language (OWL/RDF) format where information data are structured and documented via the appropriate axioms, rules and properties [10]. Using the subsumption hierarchy of OWL Classes which is defined with the *SubClassOf* axiom extracts the desired concept instances from the actual class names. This subgroup of classes that are defined in the ontology is the *Domain Ontology* of the Learning Object under construction.

Moreover, the KConcept instances conclude the set of nodes in the KConcept Graph, i.e. the content ontology, discussed in Chapter 3. The set of edges in the graph is formed with the application of the *KHasPrerequisite* between KConcept instances.

4.2 The KResource Development Stage

As it is described in Chapter 3, a KResource instance concerns the development of an object that can either support (KSupport) or assess learning (KAssess) for a specific objective (a KConcept instance). So, this stage contains the decision of the Learning Object designer between KSupport or KAssess implementation and associates this object with a KConcept instance that is to be the *TargetKConcept*.

In KSupport case, this KObject instance is to be associated with a digital resource file via its Uniform Resource Identifier (URI) or Locator (URL). This file can be off different types: an image file, web page, a video or sound file etc…

In KAssess case the designer can implement either a KQuiz or KTest KObject instance.

In both cases a LOM document association is also needed. CROP architecture handles KResource objects as *"..atomic Knowledge Products in the sense that they lack any internal structure"* [1].

More on these types of KObject instance are discussed in Chapters 5 and 6.

4.3 The KProduct Development Stage

This stage concerns the development of the KProduct object that is the single main component type of the Learning Object. Developing a KProduct includes the following steps:

1. Select a KConcept instance as the TargetKConcept from the KConceptGraph.
2. Update TargetKConcept dependencies recursively using the KHasPrerequisite property defined in the content ontology (KConceptGraph).

3. Steps 1, 2 conclude the KRC graph construction.
4. Populate each KRC node with its KObject list by selecting appropriate KResource or KProduct instances.

4.4 The KOrder Development Stage

After the designer of the LO has decided the structure of KRC Graph and has populated KRC nodes with KObject instances (KResource and KProduct), he is able to implement several execution models, XModel instances, for the KProduct. As CROP defines in [1], a KRC node is considered as a *teaching act* and a KRC edge is considered as a *teaching step*. An XModel can either implemented as a default traversing of the KRC graph or a custom created execution graph. More on XModel construction are discussed in Chapters 5 and 6.

CHAPTER 5 – DESIGN AND REQUIREMENT ANALYSIS FOR THE CROP EDITOR

The objective of this dissertation is to design and develop an editor that facilitates the design and construction of Learning Objects, following the CROP Reference Architecture [1, 2]. The design face of the editor consists a milestone, so to detect, analyze and specify the requirement that has to cover.

A large amount of existing knowledge over the CROP Reference Architecture and the perspectives that the Editor must support were provided to me from my advisor in this thesis, Professor C. Hartonas. The analysis over the software product specification and the design decisions that are presented further in this report consist the outcome of discussions and suggestions that took place during the elaboration.

Software requirements keep track of user needs and the actual functionality that a software project has to provide. In our case, the CROP Editor under construction has to be analyzed first by defining the users. Then we will adapt the functional and non – functional requirements for each user group. Despite we separate users and their requirements may be different is several cases, we have to optimize the analysis so that each software requirement has the best collaboration factor with other even in different user groups.

This Chapter introduces the second section of this report· the *Requirement Analysis and Implementation of the CROP Editor·* First, I define the runtime requirements (functional and non - functional), i.e. what the user expects to see and how he interacts with the system. Then, I define in details technical requirements and design decisions that have to be taken introducing the Use Cases for the CROP Editor.

5.1 Runtime Requirements Analysis

5.1.1 Users Definition

The CROP Learning Object Architecture comprises a theoretical and a technical approach for designing and implementing educational material. CROP supports knowledge representation and definition in such way that a user can interact as a teacher (Learning Object constructor) or as a student (learner – knowledge receiver – Learning Object executor). This is defined in [1], in such way that a knowledge resource (a KResource object) can either assess or support knowledge representation. So to proceed we have to consider the following admissions for Learning Object (LO) usability:

I. An LO must be able to be constructed.
II. An LO must be able to be executed.

Note that, these admissions do not yet take in consideration the actual actors of our system. The complete set of actors is presented in the technical part of requirement analysis. Users are a subset of actors in our implementation. More details will be presented in the UML Modeling parts of this Chapter. I separate the UML Use Case diagrams for the requirement analysis from the decided implementation features that the system actually performs.

Figure 5.1 shows a draft representation of the above.

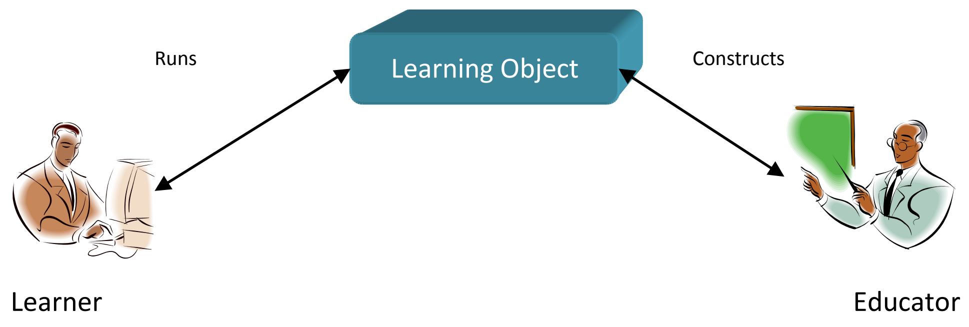


Figure 5.1

The physical presence users are defined to be two: i) the Educator that behaves and interacts as the Learning Object constructor and ii) the Learner as the Learning Object Executor.

Next, in Section 5.1.2 I present the functional requirements for each defined user and in Section 5.1.3 the non – functional. These Sections cite the requirements that the system has to support at runtime, not the actual features that have to be implemented or the decisions that we will take to do so. Section 5.2 cites the technical requirements that meet the runtime user needs. Section 5.3 maps requirements analysis to the CROP Editor introducing the Domain Model and a draft architectural approach introducing CROP Editor Components and their implementation purpose.

It is very important to conclude about the separation and actually the distinction between runtime i.e. executing requirements and the technical requirements of the CROP Editor. First, this is done to clarify the difference between "*what I want the system to support*" and "*how the system supports my needs – otherwise what tools I am able to use due to my needs*". Second, separating the technical requirements allows the reader to slightly proceed to the actual implementation facts and the decisions about the tools and technologies used and presented in Chapter 6.

5.1.2 CROP Editor – Functional Runtime Requirements

5.1.2.1 User: Educator – Constructor

1. Define, construct and refine (reedit) the knowledge repository using personal experience (custom construction) or existing knowledge bases. Hence, he must be able to define his own concepts of knowledge that have to be taught (learning objectives) or use a knowledge base already defined by someone else. The knowledge surface actually consists the domain ontology that describes what has to be taught in the Learning Object under construction implementing its content ontology (the KConcept Graph). The domain ontology can be extracted in two ways: using a reference Ontology (preexisting or newly created) or defining individual knowledge concepts (for the KConcept repository discussed in Chapter 3).
2. Select, produce and edit knowledge resources, i.e. KResource instances. These knowledge resources are digital files of various types. The system must support images, rich text documents, video etc. stored locally or attached to a website.
3. Introduce teaching – learning strategies to define execution models for the Learning Object to present knowledge derived from its resources (KOrder instances).
4. Assess or support knowledge material.
5. Design and implement an object (i.e. a Learning Object) to wrap the above.
6. Describe this object in order to be accessible and searchable with a universal standard (the LOM standard).
7. Group similar objects to construct a domain project.
8. Reedit the final product and update each constructive fragment.

5.1.2.2 User: Learner

1. Acquire a Learning Object.
2. Run available implementations of its execution model.
3. Test level of knowledge derived from the Learning Object by launching any assessment resources (KQuiz, KTest) associated and gain result/score feedback.

Figure 5.2 shows the UML Use Case diagram for the functional requirements.

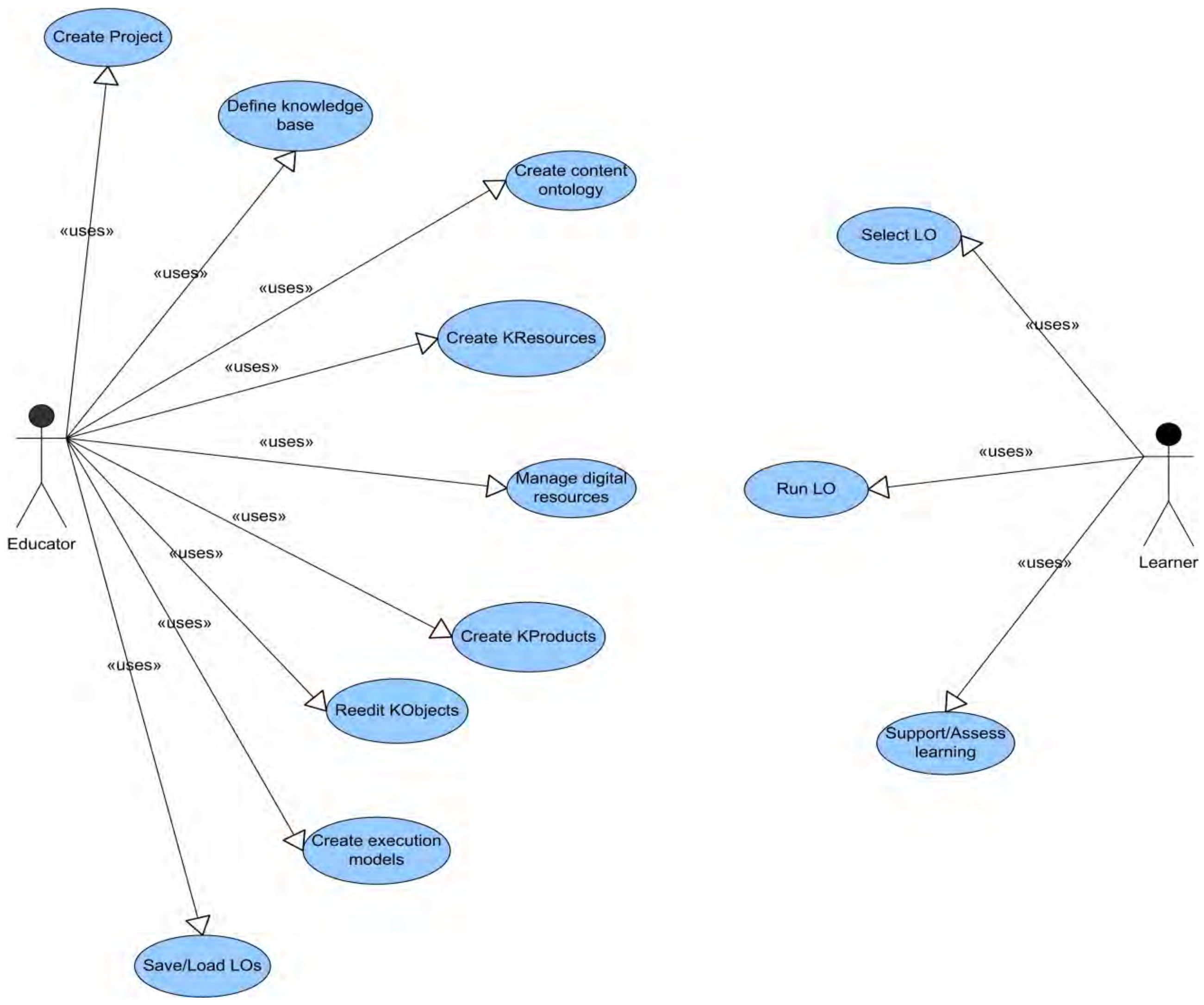


Figure 5.2 Use Case Diagram for runtime requirements of the CROP Editor

5.1.3 CROP Editor – Non Functional Runtime Requirements

We have to face the desired system as a solution of several parts, each responsible for a specific purpose. Before proceeding to technical requirements specification that will actually serve our purposes for designing and implementing CROP Learning Object Architecture Editor, we have to closely review and define the non – functional requirements and system qualities. There is a variety of literature that can be examined [4, 5, 6]. In general, they attend to optimize software quality factors that emerge from requirements analysis and user objectives. In [6], the system quality is defined to be "*the degree of match between the product requirements (stated or otherwise) and the actual product. It is defined from the point of view of the user's perception, expectation and goals or need*". So, the non – functional requirements that a system must support and we map to the CROP Editor Application while executed, include the above:

- Usability: The ease of use and interaction with the CROP Editor in our case. More especially the Graphical User Interface (GUI) interaction efficiency capturing how productive the use can be using metrics such eye focus and graphical component action factor, memorability and learnability.

- Supportability: For example, the types of resource files supported. I mention "Supportability" in technical requirements also with different meaning.

- Reliability: Actions warranty. The proof of choice otherwise. An educator has to be sure that the teaching model he constructs will be taught as it should be.

- Availability and interoperability: Embeddable in different OS architectures and accessible without special requirements. These two are both general aims for every project implementation, considering the technological variety. The user has to be sure that he will manage his work everywhere. This is also a technical issue and will be mentioned later too.

- Performance and response time to user actions: Say, loading a web resource or running a lecture.

- Configurability: Set the system as desired by configuring the workspace directory or the "Look and Feel" appearance.

- In addition to usability mentioned first, the simplicity has to be in a high level, too. The actual user in many cases will not be a computer expert.

5.2 Technical Requirements Analysis

From the technical view, I present the features of the CROP Editor by specifying in details how the runtime requirements are to be implemented so the user has the desired functionality available. In this face, I merge functional and non – functional requirements in order to give a total preview of the editor’s aspects.

Below I present those requirements, characterizing them with the properties of: type (functional or non - functional), the runtime aspect that have to cover and the user that is enrolled.

Abbreviations: F (Functional), NF (Non - Functional), E (Educator), L (Learner).

1. **Runtime aspect:** Define the Domain Ontology

Technical aspects:

The user must be able to:

I. Instantiate KConcepts extracted from Ontology files (reference Ontology, existing or custom created) using an appropriate handler or defined by self choice.
II. Relate KConcepts implementing the “*has Prerequisite*” property.

Type: F

User(s) enrolled: E

2. **Runtime aspect:** Manage resources, support learning

Technical aspect:

I. Choose between different types of digital resource files (image, text, video, audio etc.)
II. Choose local or web stored resources
III. Provide an editor to construct text documents with rich format including images, tables etc...
IV. Associate resource files with KResources.
V. Attach support KResources to KRC Nodes.

Type: F

User(s) enrolled: E

3. **Runtime aspect:** Assess learning

Technical aspect:

I. Provide editors for assessment resources i.e. a KQuiz editor and a KTest Editor and associate derived products with KResources of assessment type.
II. Attach assessment resources to KRC nodes.
III. Provide a mechanism for user interaction, an appropriate player.
IV. Provide feedback for user results.

Type: F
User(s) enrolled: E, L

4. **Runtime aspect:** Produce different types of execution models

Technical aspect:
Define different traversal paths for the KRC to implement several types of KOrder.
Type: F
User(s) enrolled: E, L

5. **Runtime aspect:** Describe a Learning Object to support distribution and search ability with a universal standard for digital resources. Associate the LOM document with the KObject.

Technical aspect: Provide a LOM Editor
Type: F
User(s) enrolled: E

6. **Runtime aspect:** Produce a Learning Object

Technical aspect:

I. Associate a Learning Object with a LOM entity
II. Associate a Learning Object with a KRC entity
III. Associate a Learning Object with execution models (KOrders)

Type: F
User(s) enrolled: E

7. **Runtime aspect:** Group Learning Objects to introduce a knowledge domain

Technical aspect: Define a project structure with available Learning Objects, Resources and LOM entities.
Type: F
User(s) enrolled: E

8. **Runtime aspect:** Save, load, update (reedit) Learning Objects
Technical aspect: Provide load/save mechanisms
Type: F
User(s) enrolled: E

9. **Runtime aspect:** Supportability
Technical aspect: Dual meaning:
I. Support different kind of resources and internetworking as mentioned before and
II. Support guidance via a help file (updateable due to the designer needs).
Type: F, NF
User(s) enrolled: E, L

10. **Runtime aspect:** High level usability
Technical aspect: Ergonomic and productive user interface.
Type: F, NF
User(s) enrolled: E, L

11. **Runtime aspect:** Reliability
Technical aspect: Construct a Learning Object that is stable and execution valid.
Type: F, NF
User(s) enrolled: E, L

12. **Runtime aspect:** Availability and interoperability
Technical aspect: Running in most frequent used Operating Systems (Microsoft Windows, Linux)
Type: NF
User(s) enrolled: E, L

13. **Runtime aspect:** Configurability and Performance
Technical aspect: Set preferences such:
I. Workspace directory to store resources and Learning Objects
II. Graphical appearance
III. Incorporate external tolls for resource editing.
IV. Perform actions with quick system responses.
Type: NF
User(s) enrolled: E, L

5.3 Mapping Requirements Analysis to CROP Editor Software

In this Section I map the previous defined requirements to the CROP Editor Software under construction. The concepts of KConcept, KRC, KNode, KEdge, KObject, KResource, KProduct and KOrder are already discussed in Chapter 3.

Two different diagrams are presented here, the CROP Editor Domain Model and an Architecture diagram that shows the basic structural components. The detailed implementation of the CROP Editor is presented in Chapter 6 with all the necessary diagrams describing the actual CROP Editor features.

First, I introduce the Domain Model of the system that will help us proceed to a more detailed definition. The CROP Editor Domain Model (Figure 5.3) shows the noteworthy domain concepts of the system as they derived from the requirements analysis made previously.

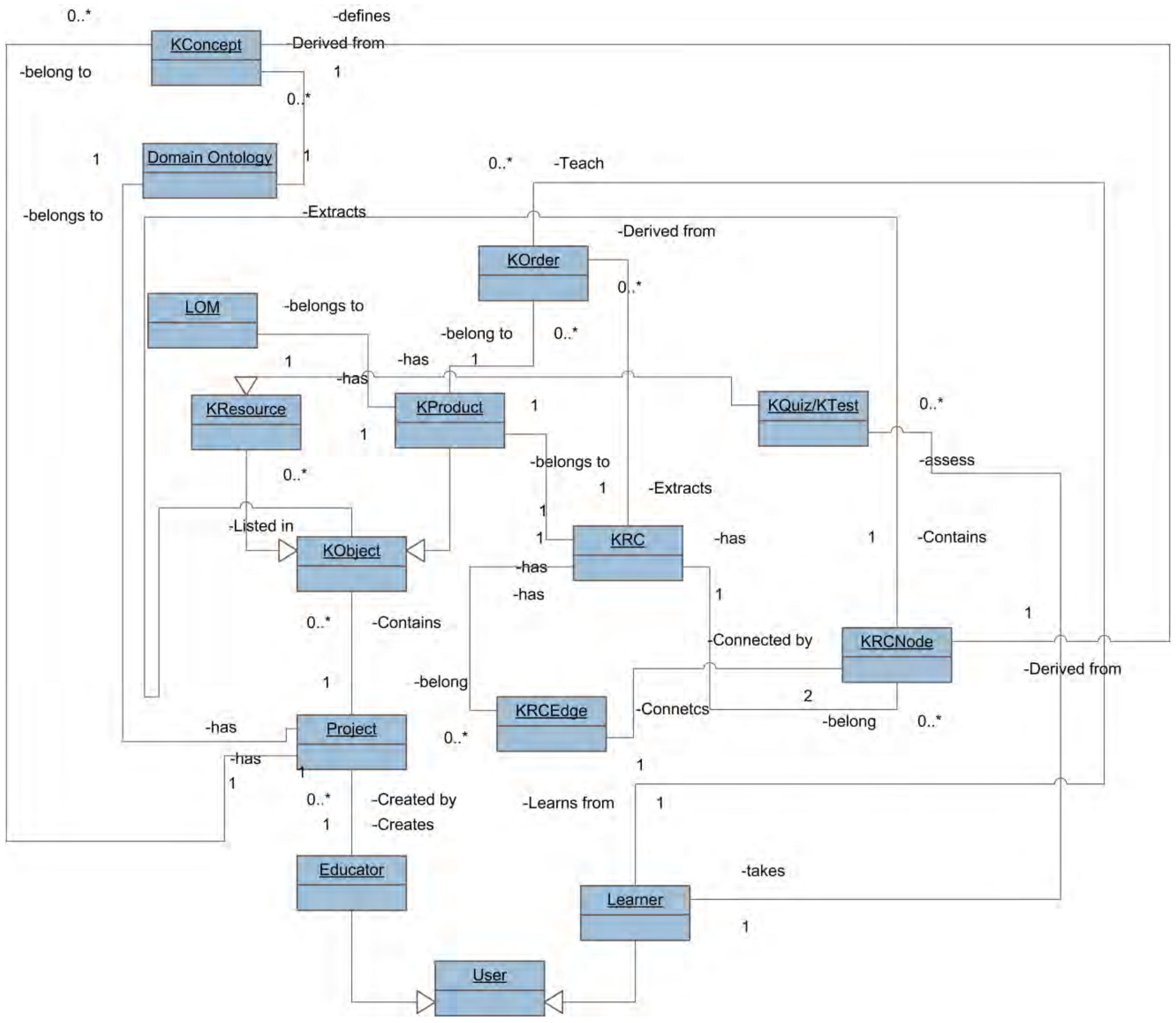


Figure 5.3 CROP Editor Domain Model

A constructor of Learning Objects, using the CROP Editor must be able to:

- Create a new Project, with save and load capability using an appropriate solution file for the project.
- Select and define a workspace to store project files. The project directory file system structure is defined in Chapter 6.

- Select digital resource files that support knowledge and implement KResource objects to associate to KRC nodes.
    - Select different types of files:
        - ✓ Image, video, textual documents (presentations, rich text formatted, spreadsheets etc.), sound, xml etc.
        - ✓ Select locally stored or web attached resource files.

- Create resource files using the implemented resource Editor:
    - Text documents with rich text format, image, tables and graphics support.

- Handle ontology files:
    - View and navigate through an existing ontology.
    - Create new ontology with basic edit functionality.
    - Extract concept instances for the KConcept repository.

- Create CROP Learning Objects (KProducts) with save, load and edit capabilities:
    - Use an ontology handler to extract concepts from reference ontology and define KConcept instances.
    - Produce domain ontology for the project by defining KConcepts and relations between them through the "*has Prerequisite*" property to construct the concept graph.
    - Construct the KRC for the Learning Object using the domain ontology.
    - Associate resource files and/or other Learning Objects with a KRCNode to support or assess learning for the current underlying concept.
    - Create, save, load and edit LOM and associate it with a KObject (KResource or KProduct).
    - Support assessment of learning material contained in the Learning Object by constructing KQuizes as assessment resource.
    - Produce different KOrder execution models that derive from different KRC traversal paths.
    - Run execution models.

- Running a Learning Object:
    - Support knowledge: Preview learning resources using a universal resource type player.
    - Assess knowledge: run interactive quizzes and tests.

Considering the CROP Editor as a basic system we can segment its architecture to several subsystems that have to be implemented and map runtime and technical requirements to software components. These subsystems include:

1. A Project Editor/Manager:
    - Using this structural component the user can create a new project, load an existing one and save a newly updated project.
    - He can define a name and secure its uniqueness for the name property.
    - He can select where the projects are to be stored by configuring the workspace entry and actually a dedicated file system path for CROP Editor's projects.
    - Navigate through the project entities (resource files, Learning Objects, reference and domain ontology etc.) handling a stably structured project directory.

2. A Resource file Handler/Editor
    - This component supports several features for the CROP Editor providing many user options for handling knowledge material.
    - Allows the user to navigate through different types of files including: images, text documents, web pages, media files etc.
    - Supports web attached files to be previewed when an internet connection exists.
    - Can import locally stored files to the project solution directory in dedicated folder for resource files.
    - Edit text documents with rich formatted text including images, tables and graphs.
    - Interact with external available editors.

3. A KResource Editor
    - The KResource Editor component allows the user to develop, save, load and update KResources.
    - Provides an input field for the "*Name*" property
    - Associates resource files with KResource objects using the physical path URI.

- Defines the target concept (KConcept) of the domain ontology.
- Associates the object with a LOM instance
- Provides an input field for a description text.

4. A KQuiz Editor
    - The KQuiz Editor component allows the user to create, save, load and update KQuiz instances.
    - Provides an input field for the "*Name*" property.
    - Provides an input method for the "Question" string.
    - Provides an input method for "Possible Answers". The answer sheet can be of two types:
        - ✓ True or False
        - ✓ Multiple choices: single or multiple correct choices.
    - Provides an input method for the "Right Answer".
    - Defines the target concept (KConcept) of the domain ontology.
    - Associates the object with a LOM instance
    - Provides an input field for a description text.

5. A KTest Editor
    - The KTest Editor allows the user to define sequences of KQuiz objects.
    - Supports save, load and update functionality.
    - Defines the target concept (KConcept) of the domain ontology.
    - Associates the object with a LOM instance
    - Provides an input field for a description text.

6. An Ontology Handler/Editor
    - The Ontology Handler component provides basic navigation functionality for existing ontology files locally stored or web attached.
    - Allows the user to produce custom reference ontology file, define class names and assert "*subclass of*" axioms, save it locally to a dedicated ontology folder inside the project root directory, reload it and update it.
    - The navigation functionality involves the class hierarchy retrieval and actually the expansion of the "*subclass of*" axiom.

7. A KConcept Editor
   - This component allows the user to define KConcept instances and relations.
   - A KConcept instance can be derived from a class name presented in the Ontology navigation tree or can be defined by the user providing a custom name of his choice.
   - Allows the user do define relations between KConcept instances implementing the "*has Prerequisite*" property for CROP Learning Objects.
   - Every KConcept instance includes the concept name, the reference ontology URI (null if it is user defined) and a vector of user defined relations with other instances.

8. A KRC Editor/Handler
   - The KRC Editor component involves the development of a KRC instance for a specific Learning Object (KProduct).
   - Uses the KConcept repository to define KRC node instances and the KConcept tree hierarchy for the "*has Prerequisite*" relation to define KRC edges.
   - It also has a constructive role for the KConcept tree hierarchy: when adding a new KRC edge (meaning that the underlying "*Prerequisite*" relation does no exist from the fact it does not yet appears in the KRC) it implements a property constructor that relates two KConcepts with the "*has Prerequisite*" property.
   - It is responsible to populate KRC nodes with KObject instances, i.e. KResources, KQuiz, KTest and KProduct instances.

9. A LOM Editor
   - The LOM Editor component is responsible for creation, save, load and update functionality of LOM entities.
   - This tool can be external, while only the file URI association is needed.

10. A KOrder Editor
   - The KOrder Editor component allows the educator to implement different traversal paths of the KRC instance inside the KProduct (Learning Object) under construction.
   - It provides save, load and editing functionality.
   - More on KOrder construction and the actual algorithms are discussed in Chapter 6.

11. A Player for the Learning Object
    - This component essentially reduces to four aspects:
        I. A resource file viewer
            - ✓ Appropriate for different resource file types (2)
        II. A KOrder player and
            - ✓ Interactive traversal of KRC nodes playing each of attached instances in the KObject list using the resource player/viewer.
        III. A KQuiz player
            - ✓ Encapsulates a KQuiz object.
            - ✓ Provides a mechanism to return feedback for the results.
        IV. A KTest player
            - ✓ Encapsulates a KTest object.
            - ✓ Provides a mechanism to return feedback for the results.
    - Supports user interaction.

A milestone decision designing the CROP Editor is to select an embedded structural components architecture (built -in) or external subsystem collaborators. The best tactic when designing and implementing software is to embed any subsystem in order to increase portability, interoperability and availability. The "*all in one*" choice lets the user to interact independently from the configuration that a system that hosts the software application has. The LOM Editor and a basic functionality Resource Editor can be embedded in the CROP Editor. This does not eliminate the user to interact with external editors for LOM or textual document files. More on this are presented in Chapter 7 where the implementation tools review exists.

Figure 5.4 shows an architectural approach for the CROP Editor modeling the system and its subsystems.

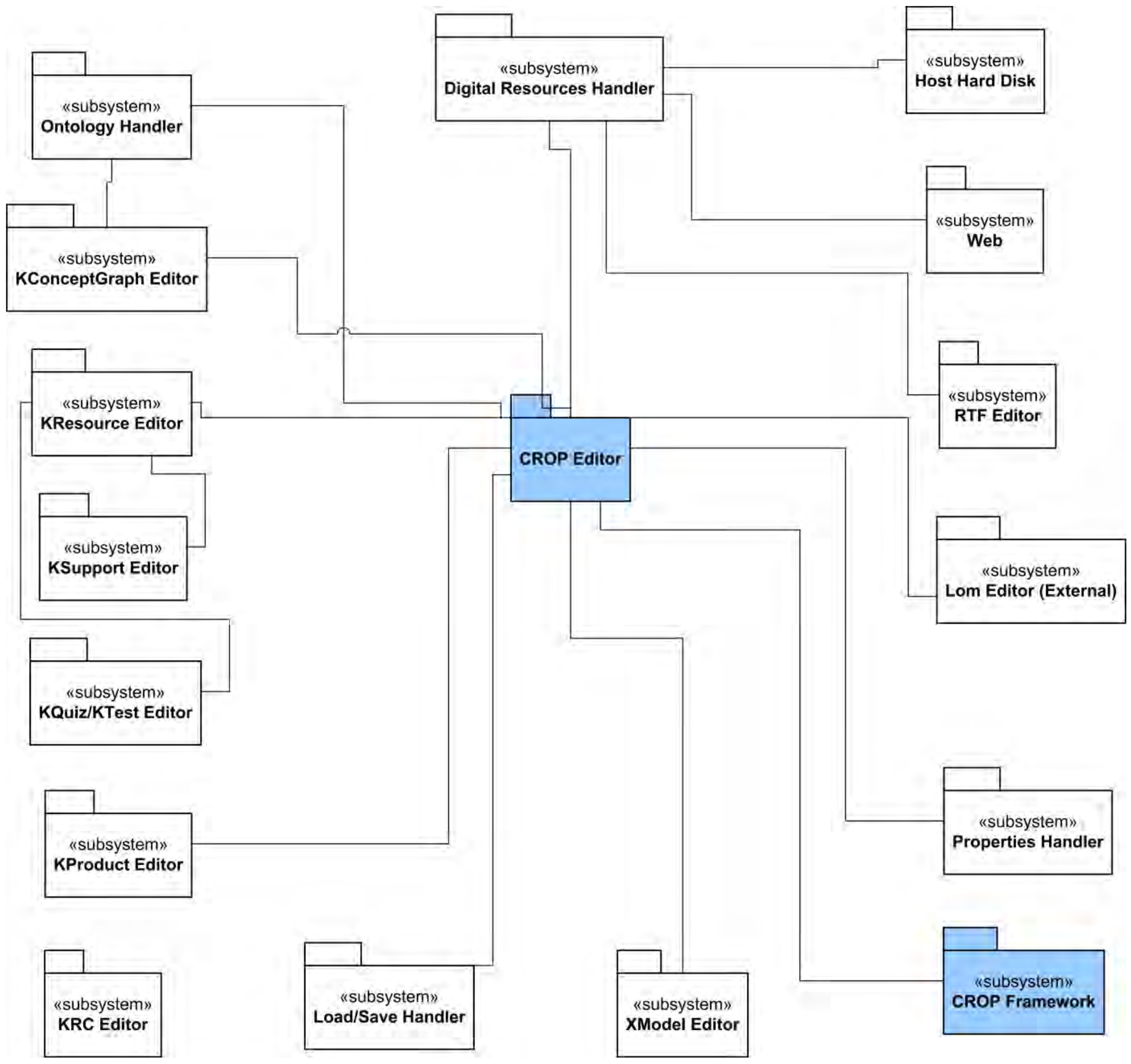


Figure 5.4 CROP Editor Architecture

CHAPTER 6 – CROP EDITOR IMPLEMENTATION

This Chapter contains the implementation report for the CROP Editor. I cite all the technologies used for development and I map the design decisions over the actual program functionality. An extended section of this Chapter concerns the Class Diagrams that are properly presented following the construction sequence. Additional components used are also discussed in details and I present their functionality and their collaboration with other software structures in the Editor. Most of the classes I have implemented are examined here briefly, while more details can be found at the programming notes that exist in the source code. All CROP Editor features are presented here, while the last Section contains the application runtime prerequisites and the setup process.

6.1 Development Tools

CROP Editor is developed using the Java programming language available at [14] and Java Development Kit (JDK) version 5 available at [15] for the Java framework kernel. The application is build using a Java Integrated Development Environment (IDE), the NetBeans IDE version 6.1 available at [16]. Java is a very flexible Object Oriented programming language that supports our needs due to the extensibility and modifiability that provides. Both tools are free to use and modify.

6.2 Software Architecture

Figure 6.1 shows the architecture of the CROP Editor application. This architecture presentation focuses on the software packages that are implemented and categorized in such way to achieve code independence and avoid duplication. Each package presented here is a actually concerned as a subsystem that collaborates in the CROP Editor. More details for each subsystem are given in the Class Diagram section of this Chapter.

- **cropedit** package is the main structural component system that consists the CROP Editor implementation and encapsulates all the subsystems.
- **core** package contains the implementation of the CROP Reference Architecture.
- **Customization** package contains classes that support CROP Editor configuration.
- **Help** package contains classes for the help file of the application.
- **KPlayer** package is the implementation of the KProduct player. This subsystem is in a preliminary development stage and is discussed later where this report concludes for the CROP Editor application.
- **ReferenceOntology** package supports handling methods for the ontology that is used as an input of the KConcept (content ontology) graph of an LO.
- **Resources** package contains classes and methods for resource files handling that are used for KSupport object instances.
- **Tools** package contains classes necessary for file handling, String and Images management and Save/Load mechanism for the CROP.
- **UI (User Interface)** contains the implementation for the GUI of the CROP Editor. This System is tightly connected with all the other subsystems.

- **CROP** package is the implementation component for the CROP project.
- **javax, org, edu and chrriis** packages are also discussed later in the Section where I present various technologies integration.

ArchitecureCD.jpeg (6.1)

6.3 Class Diagrams

In this Section I present the Class Diagrams for each subsystem giving details of the implementation and the purpose of each individual class.

6.3.1 cropedit Class Diagram

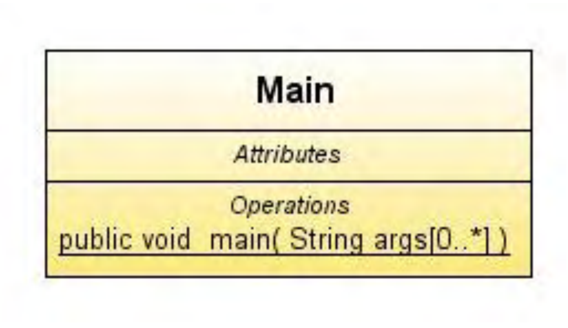


Figure 6.2 cropedit class Diagram

Figure 6.2 shows the Class Diagram of the cropedit package. It contains a single class, the *Main* class, which is responsible for starting the application.

6.3.2 core Class Diagram

Figure 6.3 shows the Class Diagram of the core package. It contains classes that implement the CROP Reference Architecture, and is actually the kernel framework for the CROP Editor.

coreCD.jpeg (6.3)

- ***KConcept*** class: is the implementation of the KConcept notion in CROP. It contains a single attribute, the *Name* attribute, which depicts the name of the KConcept instance.
- ***KHasPrerequisite*** class: is the KHasPrerequisite implementation that appears in CROP. It has two (2) attributes: *TargetKConcept* and *PrerequisiteKConcept* that are the target and source anchors of the edge which depicts the dependence relation between two KConcept instances.
- ***KConceptGraph, KConceptGraphManager*** classes*:* the KConceptGraph class describes the KConceptGraph (i.e. the content ontology) of an LO. Three (3) attributes are defined here: *CROPOwner* which is the CROP project that the graph belongs to, *KConcepts* which is a vector of KConcept instances (nodes in the graph) and *KHasPrerequisites* which is a vector of KHasPrerequisite instances (edges in the graph). The KConceptGraphManager class contains methods to manage the graph structure.
- ***KObject*** class: this class is an abstract implementation of the KObject in CROP.
- ***KResource, KSupport*** and ***KAssess*** classes: the KResource class also depicts the KResource notion in CROP. Moreover, it extends the KObject class and defines five (5) attributes: *Name* is the name for the KResource instance, *TargetKConcept* is the learning objective of this instance which appears in the KConcept graph structure, *ReferenceOntologyURI* is the URI that points to the ontology whereof the KConcept instance was extracted, *LOMURI* is the string that points the URI where the LOM document for this instance is stored and *Description* is a simple text to describe this instances which actually lacks any format. Extending the KResource class, two classes are derived: the KSupport class and KAsses class. KSupport class implements the KSupport notion in CROP and additionally contains one (1) more attribute, the *ResourceFileURI* attribute, which points to the location where the digital resource file is stored. KAssess class is extended by two classes: the *KQuiz* class which is the implementation of the KQuiz notion in CROP and contains a

Question attribute for the question citation, a vector of *Answer* instances (implement a choice in the quiz) and the *SuccessResponse* and *FailureResponse* strings· and the KTest class that defines a vector of KQuiz instances and *SuccessResponse, FailureResponse* correspondingly.

- ***KProduct*** *c*lass: KProduct calls is the implementation of the KProduct notion CROP. Extends the KObject class and defines eight (8) attributes: *Name* is the name for the KProduct (i.e. the Learning Object) instance, *TargetKConcept* is the learning objective of this instance which appears in the KConcept graph structure, *CROPOwner* which is the CROP project that the graph belongs to, *ReferenceOntologyURI* is the URI that points to the ontology whereof the TargetKConcept instance was extracted, *LOMURI* is the string that points the URI where the LOM document for this instance is stored, *Description* is a simple text to describe the instance which actually lacks any format, *KRC* corresponds to the KRC class in CROP (next paragraph) and *KOrder* for the execution models of this KProduct instance (next Subsection).
- ***KRCNode**, **KRCEdge***, ***KRC*** and ***KRCManager*** classes: KRCNode class corresponds to the *teaching act* which is defined in CROP. It is the base for the execution of a KProduct instance and moreover for its KOrder described in the next Section. KRCNode defines six (6) attributes: *Name* attribute for the KRCNode which is constructed by the *TargetKConcept* attribute name plus the *"_node"* application, *ProductOwner* which is the KProduct instance where it belongs to and three (3) vectors of KSupport (*KSupportObjects*), KAssess (*KAssessObjects*) and KProduct (*SubProducts*) instances that correspond to the KObjectList for this KRCNode instance. The KRCEdge class corresponds to the *teaching step* that is defined in CROP and visualizes the dependence relation between two (2) KRCNode instances (i.e. KConcept and KHasPrerequisite respectively). The KRC class is the actual graph that is constructed by a vector of KRCNode instances (graph nodes) and a vector of KRCEdge instances (graph edges). In addition to the structural vectors of nodes and

edges two (2) more attributes are defines here: the *ProductOwner* (i.e. the KProduct instance where it belongs to) and *TargetKConcept* which is the root node in the graph, say the learning target for this KProduct. Correspondingly to the KConceptGraphManager, KRCManager contains methods to handle the graph structure (parsing and editing features).

6.3.3 KOrder sub- package Class Diagram

In this Subsection I present the KOrder package and I separate it from the *core* package because of its complexity. Figure 6.4 shows the Class Diagram for the KOrder class which corresponds to the KOrder notion in CROP. In fact, the KOrder class can be certified as a repository of *XModel* (execution models) instances for the KProduct under discussion, attached with a *StudentModel* and an *Instructional Environment*.

- ***KOrder*** and ***KOrderManager*** classes: KOrder class has two (2) attributes: a vector of XModel instances (*XModels*) and the *ProductOwner* which is the owner for this KOrder instance. The KOrderManager class contains methods to handle the vector (repository) of XModel instances in the KOrder.

korderCD.jpg (6.4)

- ***XModel, XModelManager*** classes and ***XModelType*** enumerator: the XModel class is the implementation of the *execution model* motion as it is defined at CROP. This class contains five (5) attributes: *OrderOwner* is the KOrder where this XModel instance belongs to· *Name* is the name for the instance· *Description* is a simple unformatted text variable to describe the purpose of this execution model· *Type* is a variable which takes value (element) from the value space that is defined in the *XModelType* enumerator and it can one of the: *Custom* (for custom made execution models), *DefaultHorizontal* or *DefaultVertical* (for a default construction)· and the *XGraph* attribute which is the execution graph (next paragraph). The XModelManager class contains methods for parsing and structure handling of the XModel.
- ***XGraph*** and ***XGraphManager*** classes: the XGraph class is the implementation of the execution graph for a specific XModel instance. An XGraph can be of two (2): a custom made or default correspondingly to the XModelType. In both cases the XGraph contains a sub-graph which is actually the KRC Graph of the KProduct under discussion. In case of *custom* XModelType, the execution graph contains not only the sub-Graph that is derived from the KRC bit also a variety of other node types (XNode instances· presented in next paragraph) and a variety of edge types (XEdge instances· presented also in a next paragraph). In the case of a *default* construction, the XGraph contains only the mapping of the KRC graph to appropriate XNode and XEdge instances that are presented next. So, the XGraph class defines three (3) attributes: a vector of XNode instances (*XNodes*), a vector of XEdge instances (*XEdges*) and the *ModelOwner* attribute that is the XModel owner of this execution graph. Talking about the default construction of an XGraph, this stands from the fact that the graph contains only the mapping of the KRC Graph. The two (2) types of default construction (*Horizontal* or *Vertical*) concern the traversing path of the

graph at runtime, i.e. when the KProduct is executed using the specific XModel. Figure 6.5 shows the two types of traversing. The XGraphManager class contains methods for parsing and structure handling of the input Graph.

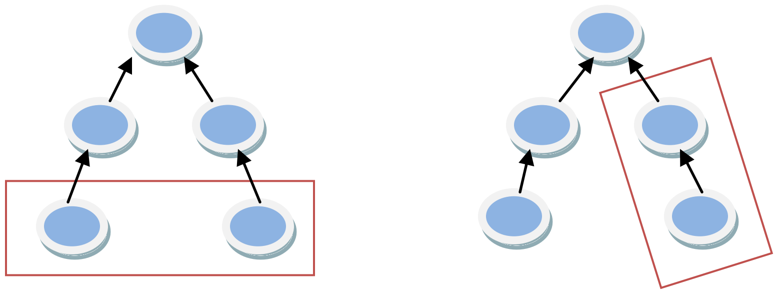


Figure 6.5 Default horizontal and vertical traversing of an XGraph.

- ***XNode*** class: the XNode class is the implementation of a node in the execution graph. Two (2) attributes are defined here: *Name* of the instance and *GraphOwner* the XGraph that this node belongs to. This is an abstract class and is inherited by various types of XNode instances. These types are:
    - ***XDialogues*** class: is a class implementation to present a dialogues node and extends the XControlNode class. This class contains two (2) attributes: *IntroText* which is a simple unformatted text and *IntroURI* which is a string URI that points to a digital file. Both attributes are to inform the user (learner) for what is to be expected next at runtime or to introduce some teaching act. An XDialogues instance stands a node where some teaching steps are about to take place.
    - ***XStartNode, XEndNode*** classes: present the start and the end of the graph. These classes extend the XDialogues

class and they appear with different image/shape in the execution graph. An XStartNode instance has a *single step* while lacks any.

- ***XDialogueNode*** class: this class has the exact meaning as any XDialogues instance but differs in such way that many dialogue steps (via the *XStepEdge* implementation which is discussed later) can be associated with it.
- ***XGroupNode, XSeqGroupNode, XParGroupNode*** and ***XEndGroupNode*** classes: an XGroupNode instance depicts the grouping of node instances. This is an abstract implementation and is inherited by: the XSeqGroupNode that stands for a sequential group of nodes, the XParGroupNode that is a parallel grouping of nodes and the XEndGroupNode that present the end of grouping in both *sequence* and *parallel* cases. Figure 6.6 shows an example grouping segment.

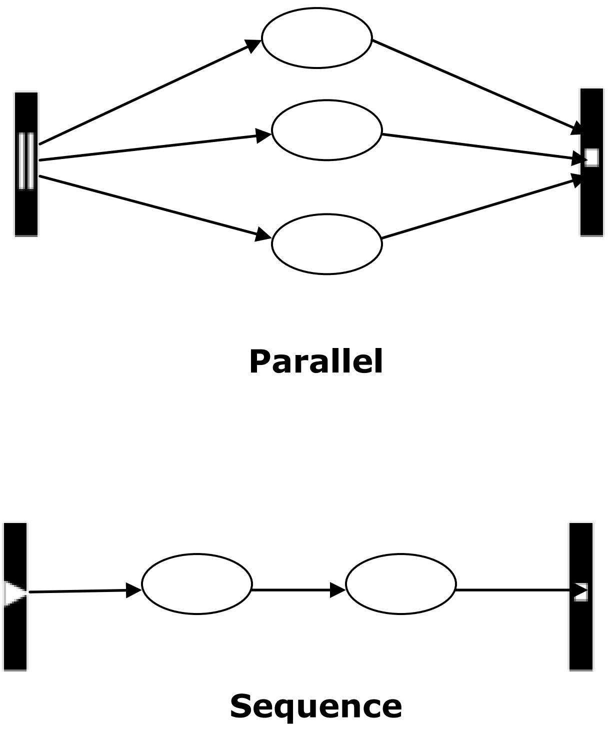


Figure 6.6 Sequence and Parallel group node implementation

- ***XConceptNode*** class: this type of Node stands for representing a KRCNode as an executable component. In the XGraph of the given XModel one node exists for each KRCNode in the KRC Graph. An XConceptNode instance is unique in an XGraph and has three (3) more attributes: the *ComponentKRCNode* which is the input KRCNode, the *ComponentGraph* attribute which is recursively an XGraph that is structured with the KObjectList elements (the node type) of the KRCNode. These graph apparently from the XNode types mentioned here, contains *XResourceNode* and *XsubProductNode* instances. The set of XConceptNode instances with the set of *XDependenceEdge* instances (discussed in next subsection) illustrate the *XDependenceGraph* for the given XModel. The third (3rd) attribute is an instance of the *XConceptNodeRuntimePerformance* class, which contains a set of Properties defined in the *PerformanceProperty* enumerator and stores/checks the runtime performance of the student – learner during the KProduct execution.
- ***XIFNode*** class: extends the XControlNode class and implements an automated decision mechanism. This class contains three (3) more attributes: a vector of conditions· *XIFNodeCondition* instances that are constructed using a property element from the value space that is defined in *PerformanceProperty* enumerator, a *PerformanceOperator* value from the value space defined in this enumerator and a value (*double* variable)·and two (2) nodes (XNode instances in the XGraph), one for the *True* evaluation of the condition sequence and one for the *False*. Table 6.7 lists the available performance properties and performance operators.

PerformanceProperty	**PerformanceOperator**
QuizSum	Grater
TestSum	Equal
QuizTaken	Less
TestTaken	Not
QuizSuccess	LessEqual
TestSuccess	GreaterEqual
OverAllPerformance	
CurrentNodePerformance	
PreviousNodePerformance	

Table 6.7 Performance properties and operators

- ***XResourceNode*** and ***XsubProductNode*** classes: These classes represent the executable XNode instances for a KResource or a KProduct respectively. These XNode types appear inside the XGraph of an XConceptNode and are produced using the KObjectList of the KRCNode that is input to the XConceptNode instance. Due to designer's selection not all the KResources or SubProducts of the KRCNode must be implemented in XConceptNode graph structure.

▪ ***XEdge*** class: this class is the implementation of an edge in the XGraph (XModel graph or XConceptNode graph). Two (2) attributes are defined: the *SourceXNode and the TargetXNode* which are represent the source anchor and the target anchor of the edge respectively. XEdge class is extended by four (4) more types:
  - ***XDependenceEdge*** class: maps a KRCEdge in the XGraph. As already said, the set of XDependenceEdge instances with XConceptNode instances consist the XDependenceGraph (sub-Graph) of an XModel graph.
  - ***XStepEdge*** class: this class depicts a user (learner) choice when the current node type is an XStartNode or

XDialogueNode instance. This edge defines one (1) more attribute· the *StepText attribute* which is a short string to describe the choice.
- ***XFalseEdge*** and ***XTrueEdge*** classes: These classes implement the *False* and *True* condition evaluation when in an XIFNode instance correspondingly.

- ***StudentModel*** class: this class is not yet implemented. The aim of this implementation is to collaborate with the KOrder object in an instructional environment. More on this are discussed at Chapter 8 where this report concludes.

6.3.4 *CROP* Class Diagram

Figure 6.8 show the class diagram for the CROP package. Three (3) classes are defined here:

- ***CROP*** class: This class represents a CROP Editor Project. There is variety of attributes defined here including the project *Name, DiskPath, Creation* and *Modification Dates* , *ReferenceOntologyURI* string, names of the folders to store data and vectors of objects (KResources and KProducts) implemented for this project.
- ***CROPManager*** class: this class includes management methods for a given CROP project. It supports project creation, load and save mechanism and parsing of KObject instances implemented in this project.
- ***SaveOWLFactory*** class: this calls is to implement a save mechanism for the CROP project as an OWL onology. This feature is discussed in the conclusion Chapter 8 and maps the requirements for the CROP Ontology.

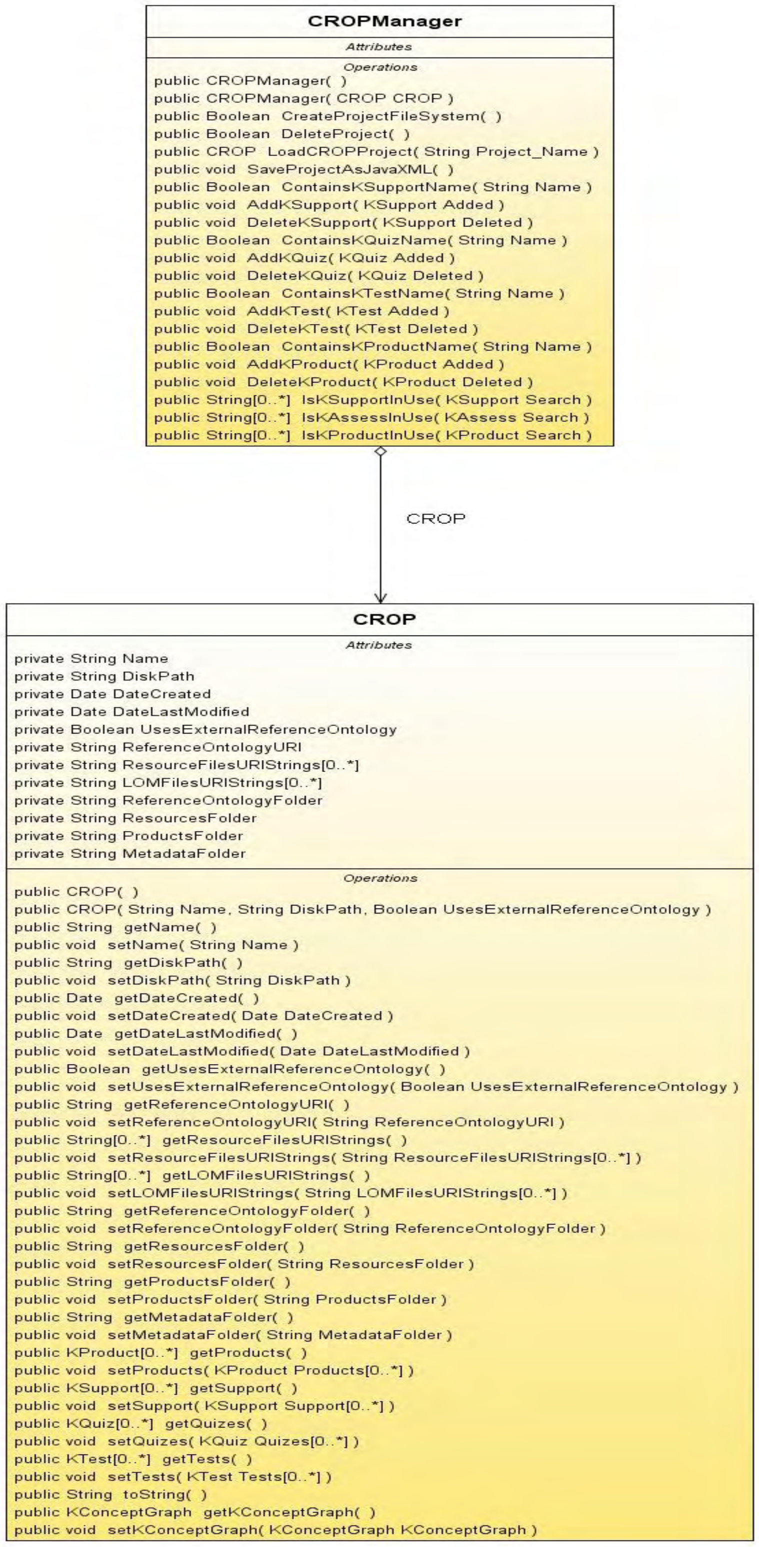

CROPManager
Attributes
Operations
public CROPManager( )
public CROPManager(CROP CROP)
public Boolean CreateProjectFileSystem( )
public Boolean DeleteProject( )
public CROP LoadCROPProject(String Project_Name)
public void SaveProjectAsJavaXML( )
public Boolean ContainsKSupportName(String Name)
public void AddKSupport(KSupport Added)
public void DeleteKSupport(KSupport Deleted)
public Boolean ContainsKQuizName(String Name)
public void AddKQuiz(KQuiz Added)
public void DeleteKQuiz(KQuiz Deleted)
public Boolean ContainsKTestName(String Name)
public void AddKTest(KTest Added)
public void DeleteKTest(KTest Deleted)
public Boolean ContainsKProductName(String Name)
public void AddKProduct(KProduct Added)
public void DeleteKProduct(KProduct Deleted)
public String[0..*] IsKSupportInUse(KSupport Search)
public String[0..*] IsKAssessInUse(KAssess Search)
public String[0..*] IsKProductInUse(KProduct Search)
CROP
CROP
Attributes
private String Name
private String DiskPath
private Date DateCreated
private Date DateLastModified
private Boolean UsesExternalReferenceOntology
private String ReferenceOntologyURI
private String ResourceFilesURIStrings[0..*]
private String LOMFilesURIStrings[0..*]
private String ReferenceOntologyFolder
private String ResourcesFolder
private String ProductsFolder
private String MetadataFolder
Operations
public CROP( )
public CROP(String Name, String DiskPath, Boolean UsesExternalReferenceOntology)
public String getName( )
public void setName(String Name)
public String getDiskPath( )
public void setDiskPath(String DiskPath)
public Date getDateCreated( )
public void setDateCreated(Date DateCreated)
public Date getDateLastModified( )
public void setDateLastModified(Date DateLastModified)
public Boolean getUsesExternalReferenceOntology( )
public void setUsesExternalReferenceOntology(Boolean UsesExternalReferenceOntology)
public String getReferenceOntologyURI( )
public void setReferenceOntologyURI(String ReferenceOntologyURI)
public String[0..*] getResourceFilesURIStrings( )
public void setResourceFilesURIStrings(String ResourceFilesURIStrings[0..*])
public String[0..*] getLOMFilesURIStrings( )
public void setLOMFilesURIStrings(String LOMFilesURIStrings[0..*])
public String getReferenceOntologyFolder( )
public void setReferenceOntologyFolder(String ReferenceOntologyFolder)
public String getResourcesFolder( )
public void setResourcesFolder(String ResourcesFolder)
public String getProductsFolder( )
public void setProductsFolder(String ProductsFolder)
public String getMetadataFolder( )
public void setMetadataFolder(String MetadataFolder)
public KProduct[0..*] getProducts( )
public void setProducts(KProduct Products[0..*])
public KSupport[0..*] getSupport( )
public void setSupport(KSupport Support[0..*])
public KQuiz[0..*] getQuizes( )
public void setQuizes(KQuiz Quizes[0..*])
public KTest[0..*] getTests( )
public void setTests(KTest Tests[0..*])
public String toString( )
public KConceptGraph getKConceptGraph( )
public void setKConceptGraph(KConceptGraph KConceptGraph)

Figure 6.8 the CROP package Class Diagram

6.3.5 *Customization* package Class Diagram

Figure 6.9 shows the class diagram for the *Customization* package. A single class is defined here· the ***Customization*** class· which is responsible for two (2) types of customization via its methods:

Look and Feel: sets the Look and Feel (graphics style) for the application.

Work Directory: sets the workspace where CROP projects are stored.

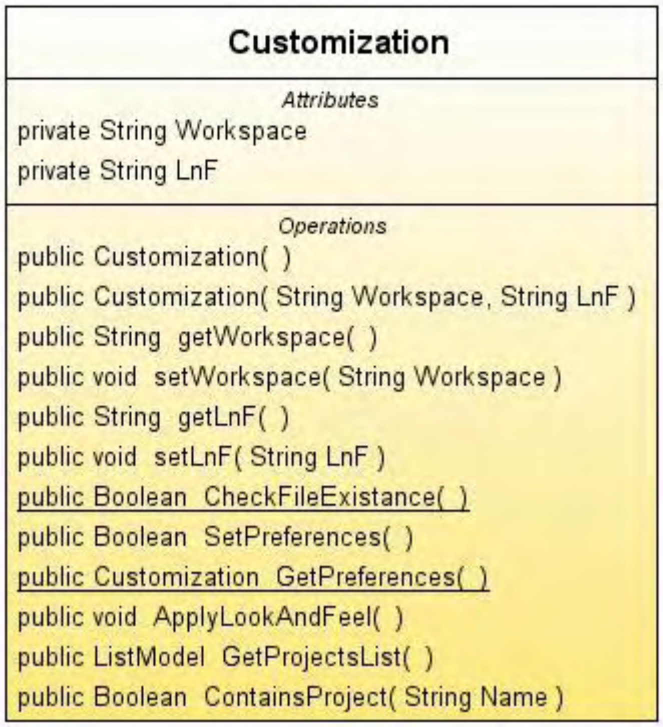

```
Customization
Attributes
private String Workspace
private String LnF
Operations
public Customization( )
public Customization( String Workspace, String LnF )
public String  getWorkspace( )
public void  setWorkspace( String Workspace )
public String  getLnF( )
public void  setLnF( String LnF )
public Boolean  CheckFileExistance( )
public Boolean  SetPreferences( )
public Customization  GetPreferences( )
public void  ApplyLookAndFeel( )
public ListModel  GetProjectsList( )
public Boolean  ContainsProject( String Name )
```

Figure 6.9 Customization package Class Diagram

6.3.6 *Help* package Class Diagram

This package also contains a single class· the ***HelpSetTreeModel*** class· which is responsible to load the help set for the CROP Editor application. It implements the TreeModel Java interface and it is a File System extractor for a given folder. Figure 6.10 shows the class diagram for Help package.

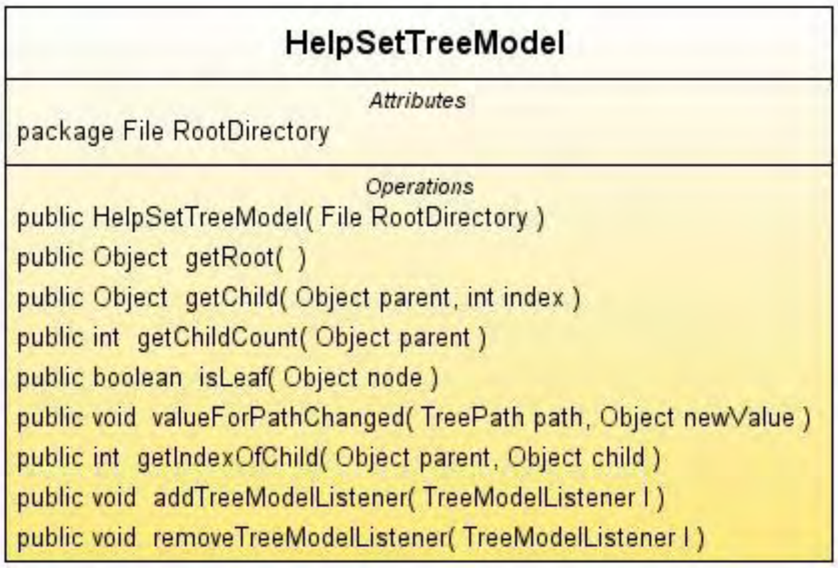

HelpSetTreeModel
Attributes
package File RootDirectory
Operations
public HelpSetTreeModel(File RootDirectory)
public Object getRoot( )
public Object getChild(Object parent, int index)
public int getChildCount(Object parent)
public boolean isLeaf(Object node)
public void valueForPathChanged(TreePath path, Object newValue)
public int getIndexOfChild(Object parent, Object child)
public void addTreeModelListener(TreeModelListener l)
public void removeTreeModelListener(TreeModelListener l)

Figure 6.10 Help package Class Diagram

6.3.7 *ReferenceOntology* package Class diagram

ReferenceOntology package class diagram is shown in Figure 6.11. It contains a single class· the ***ReferenceOntologyHandler*** class that contains methods to create, load and save OWL ontology files for a given CROP project. A ReferenceOntology performs as an input for the content ontology (KConceptGraph) of a CROP Learning Object.

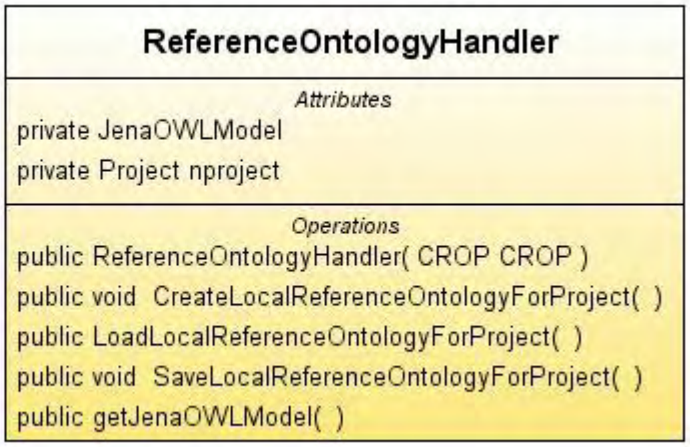

ReferenceOntologyHandler
Attributes
private JenaOWLModel
private Project nproject
Operations
public ReferenceOntologyHandler(CROP CROP)
public void CreateLocalReferenceOntologyForProject( )
public LoadLocalReferenceOntologyForProject( )
public void SaveLocalReferenceOntologyForProject( )
public getJenaOWLModel( )

Figure 6.11 ReferenceOntology package Class Diagram

6.3.8 *Resources* package Class Diagram

The Resources package class diagram (Figure 6.12) contains also a single class named ***ResourceManager***. This class supports digital file management in association with a CROP project instance. This methods include add/remove URI from resources and importing a file (copy to local folder).

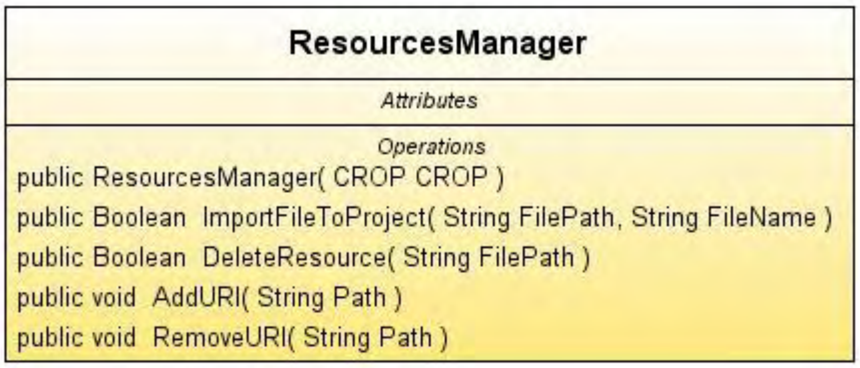

ResourcesManager
Attributes
Operations
public ResourcesManager(CROP CROP)
public Boolean ImportFileToProject(String FilePath, String FileName)
public Boolean DeleteResource(String FilePath)
public void AddURI(String Path)
public void RemoveURI(String Path)

Figure 6.12 Resources package Class Diagram

6.3.9 *UI* package Class Diagram

This package (Figure 6.13 (a, b, c, d)) contains all Graphical User Interface Components for the CROP Editor. Chapter 7 demonstrates the application at runtime showing screenshots that illustrate all the components that are implemented to serve CROP Editor features. This is an extended package implementation and the main characteristics are listed below giving an overview of the implementation and the actual design methods used:

UICDa.jpg(6.13a)

UICDb.jpg(6.13b)

UICDc.jpg(6.13c)

UICDd.jpg(6.13d)

- ***Application main window***: for the main CROP Editor window I used the Java *JFrame* class. In this window a CROP project is hosted and provides the appropriate views for each structural component of the project. Figure 6.14 shows the UML Component Diagram for the CROP Editor Views.

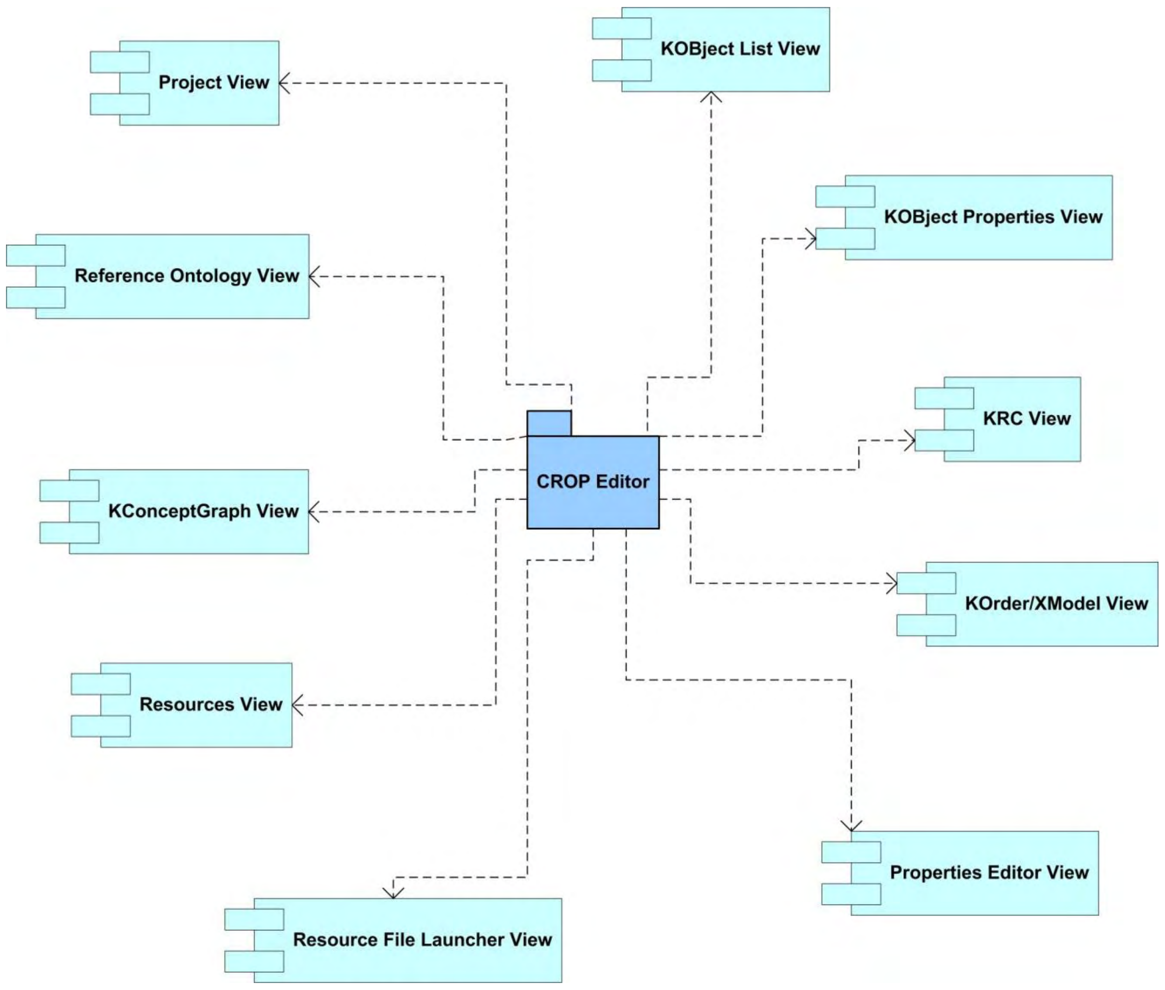


Figure 6.14 CROP Editor Views Component Diagram

- ***Panels***: all the component view of the editor are hosted in Java *JPanel* forms. Each panel may have a toolbar to edit the object that is currently in process.

- ***Lists***: there is a variety of vectors and grouped objects as it is cited at this section. These groups are modeled using the *ListModel* Java interface and are shown inside *JList* components. The ListModel interface is highly modifiable and can be extended to host any type of sequential objects (vectors, lists, arrays etc.).

- ***Scenes***: Graph scenes (visual components) are used to demonstrate the three (3) graph types that appear in CROP framework: KConceptGraph, KRC Graph and XGraph. This component is fully editable and implements many handlers such as the *ObjectSceneLister* interface to manage object selection. It provides menus over the various object data types and is customizable and extensible according to the developer's needs. It also supports Scene view methods to zoom, layout and change the graph representation inside the panel host.

- ***Drag & Drop***: this feature is implemented to increase user – application interaction efficiency (usability). An example usage is to drag class name from the ontology panel and drop it to the KConceptGraph Scene or populate a KRCNode object list with KObject instances by dragging an object (KResource or KProduct) and drop it on the KRCNode image in the KRC Grpah Scene.

- ***Docking***: all the application is developed using a Multiple Document Interface (MDI) style. That means that the main window can perform as a host (parent component) for various other windows (panels – child components). It used as virtual desktop screen where additional panels are placed and supports docking capability (move, dock, undock, replacement, resize etc.). The internal components are called *dockables*.

- ***Update***: Most of the UI components are automatically updateable according to the user actions.

6.3.10 *KPlayer* package Class Diagram

This package is a case study for the Learning Object Player of CROP LOs. KPlayer consists of the algorithms and the application of them in order to run an execution model (XModel) for a given KProduct (i.e. a Learning Object). The Class Diagram of the KPlayer as it is until now is shown in Figure 6.15. The aim of the KPlayer is to perform simultaneously with other necessary components that will arise from the requirement analysis in a universal Instructional Environment, integrating an XModel, a StudentModel and a report mechanism to support and assess learning.

kplayercD.jpg (6.15)

6.3.11 *Tools* package Class Diagram

This package (Figure 6.16) contains classes that are responsible for many operations in CROP Editor. Five (5) Java classes are defined here:

- ***FileHandling*** class: contains methods for copying a file, convert a byte stream to file, write a file into a Java String object etc..
- ***HTMLFormatConverter*** class: supports transformation from HTML format to RTF format via a middleware application involving XML file handling.
- ***Images*** class: contains methods to transform image files into reusable Java objects (Image or ImageIcon).
- ***Serialization*** class: this class is the key for load and save mechanism in CROP Editor. Two methods are defined here:
  - ✓ ***SerializeToJavaXML***: takes input an object of any kind and maps its internal structure into a reusable XML file format.
  - ✓ ***DeSerializeFromJavaXML***: takes input an XML File and maps the XML schema and values into the desired object implementation.
- ***Strings*** class: this class contains to validate Java Strings, say check if a String object is of null value, empty or contains spaces.

toolsCD.jpg (6.16)

6.4 CROP Editor Application Features

In this Subsection I cite the CROP Editor features and the actual Learning Object designer's (educator) available functionality. To avoid complication more details for the Editor functionality are given in the next Chapter using the runtime screenshots.

A CROP Learning Object designer is able to:

- ✓ Create, Load and Save a CROP Project.
- ✓ Create, Edit and Save reference ontology in OWL format or View an existed external ontology via the URI where the ontology file is stored.
- ✓ Select a class name from the reference ontology and add it into the KConceptGraph (content ontology) Scene Panel.
- ✓ Define dependence relations (KHasPrerequisite instances) between KConcept instances visually in the graph.
- ✓ Edit the KConceptGraph by adding or removing edges or nodes.
- ✓ Create digital resource files in RTF format using an internal RTF Editor.
- ✓ View and Edit the Resources List.
- ✓ Navigate on the Web and import URI addresses to use them as resources.
- ✓ Create and Edit KResources (KSupport, KQuiz and KTest objects): this involves a selection mechanism for TargetKConcept and LOM file association.
- ✓ Preview the implemented KResource instances and edit their properties.
- ✓ Create, Load and Save KProducts. Edit the KRC Graph structure and populate KRCNode objects with KObject instances. Update KRCEdges according to the KConceptGraph implemented dependencies.
- ✓ Implement execution models for a given KProduct selecting default or custom creation mode for the XGraph. In custom

creation case, the designer can implement new XNode objects using the Drag n Drop functionality from a component palette and edit their properties using a very simple menu and define appropriate XEdges.

- ✓ Customize via a wizard the application Look and Feel or the workspace to store the implemented CROP projects.
- ✓ Take help tips using the Help support mechanism that exists with update features.

6.5 Additional Tools and Technologies Integration

For the CROP Editor development I used some additional tools and technologies to collaborate with the existing Java functionally. These tools include:

A docking framework: For that purpose I used MyDoggy Java Docking Framework available at [17].

A Reference Otology Handler: to load, view, edit and save reference ontology for the CROP project I used the Protégé OWL API v.3.3.1 which is available at [18].

A Look and Feel applicator: to set application Look and feel I used the Substance Java Look and Feel package available at [19].

A Collapsible Panel implementer: in order to save screen space I used collapsible panels apart from the docking framework. The package I used is the Japura v.1.2.1 package available at [20].

An HTML/RTF file Editor: for that purpose I used the FCKEditor available at [21] that is integrated in a single Java component implemented in the DJNative Swing Component Library available at [22].

An HTML to RTF converter: using the FCKEditor the result is in HTML format. In order to transform it to RTF I used two technologies: the JTidy library that performs HTML structure checking and correction available at [23] and the Apache Formatting Objects Processor (FOP) available at [24] which supports HTML to RTF conversion via an XML transformation (with XSL application as middleware).

A visual graph scene editor: graph scenes in CROP Editor are used to provide editing functionality for the KConceptGraph, KRC Graph and XModel Graph. For this purpose I used the Java Visual Library v.2.0 available at [25] which is an integrated to NetBeans IDE tool and can be retrieved as plug-in module and used separately in Java applications.

6.6 Application Prerequisites and Setup

In order the application to be executable the host system must meet the following prerequisites:

1. Installed Java Runtime Environment (JRE) version 5 or later that can be found at [14].
2. Operating System Microsoft Windows XP or later. The application has been tested also in Ubuntu Linux.
3. Web Browser: Microsoft Internet Explorer or Mozilla Firefox.

To setup the application in the system you have to:

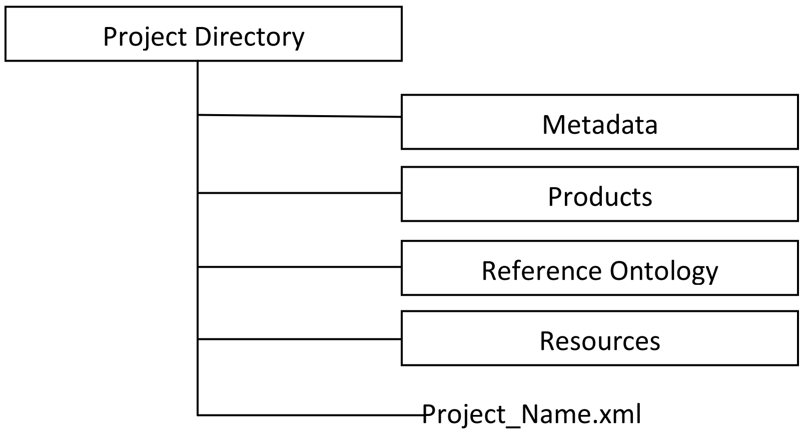
1. Copy the "*Runt*" folder from the attached CD into a system folder in the hard disk drive.
2. In the *dist* folder create a folder and name it *CROP Editor Help*. Any file (Web page, PDF file etc.) you put there will be shown in the Help set list of the Editor.

3. Run the *CROPEdit.jar* file from the dist directory and customize the application selecting a workspace directory and the desired Look n’ Feel.

CHAPTER 7 – CROP EDITOR DEVELOPMENT EXAMPLE AT RUNTIME

In this Chapter I present the CROP Editor at runtime showing screenshots and explaining more the application functionality. In this demonstration I use an example to proceed that involves a Learning Object that teaches the user (learner) how the CROP Editor works.

Figure 7.1(a) shows the CROP Editor main window when the application starts plus the *Project Menu* where the user can choose either to open an existing project or create a new one. For this example I will create a new project (Figure 7.1 (b)) called *CROP Editor Usage* and in the Panel that appears I give the name and select to use a custom Reference Ontology (leave unchecked the Checkbox named as *Uses external reference ontology*).If I choose to use an external reference ontology the panel below the name input text field becomes enabled and I can navigate through web ontology resources providing the OWL file URI and testing its existence or usability by pressing the *Test named button.* After creating the project all Menus are available. In this design phase a CROP folder is created inside the workspace defined in the customization and all the necessary subfolders. In case that I select to use a custom reference ontology an owl file also is constructed and saved in the reference Ontology sub – Folder of my project. The file system hierarchy of a crop project is shown below:



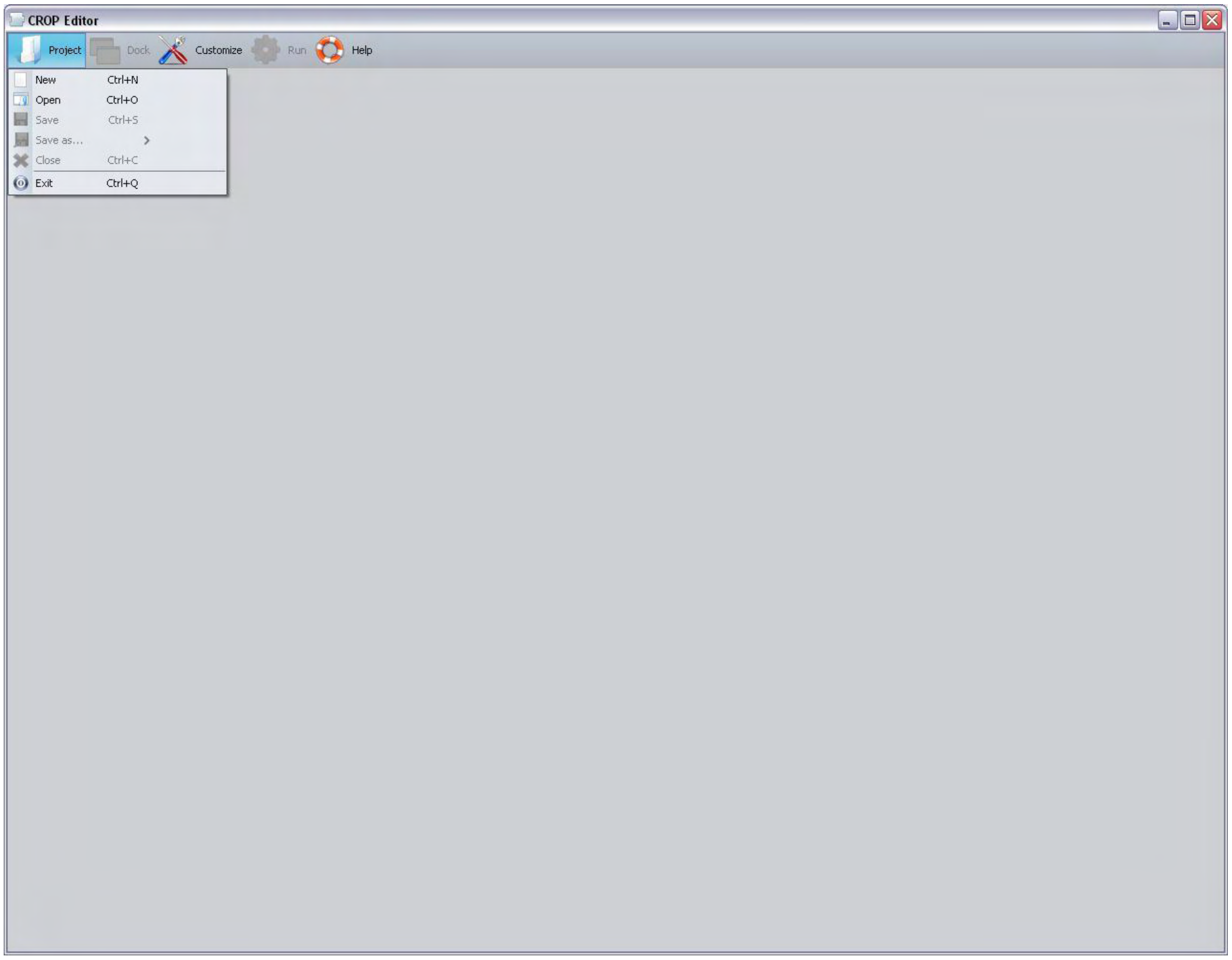


Figure 7.1 (a) CROP Editor Main Window

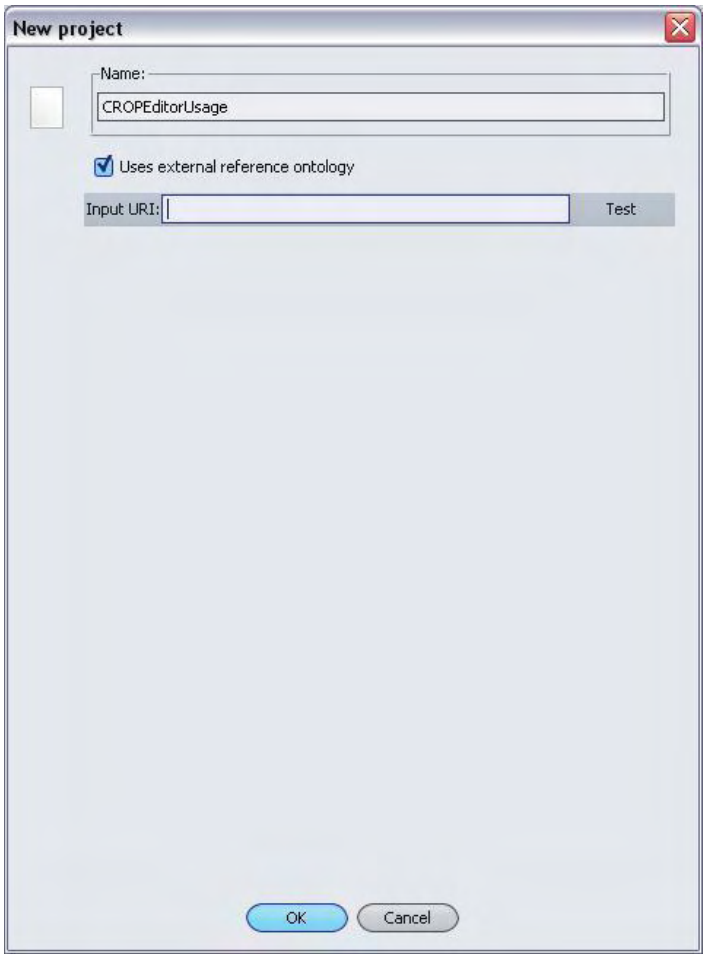
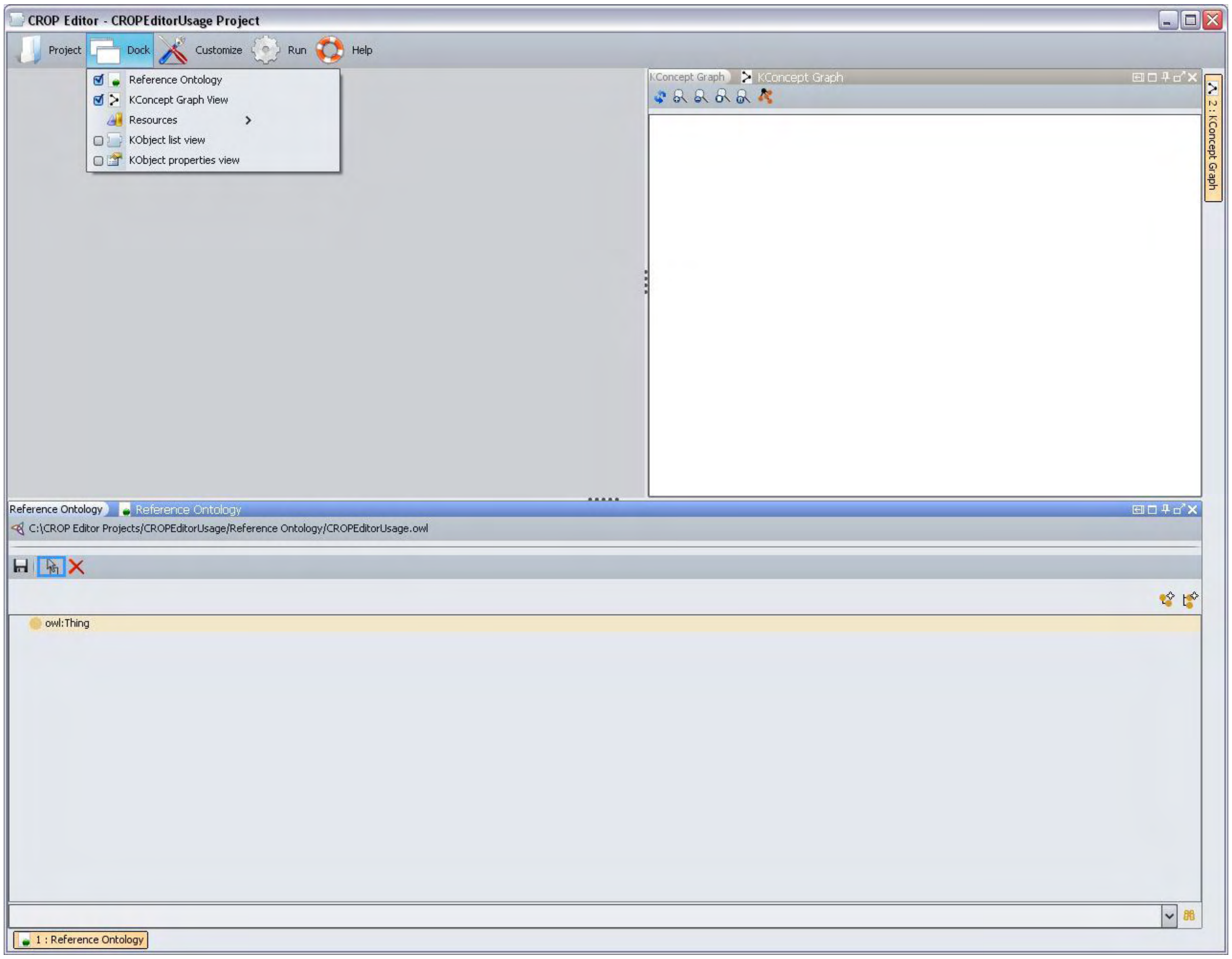


Figure 7.1(b) the panel to create a New CROP Project

Next, in Figure 7.2a I expand the *Dock Menu* and I choose to dock the ReferenceOntology View Panel and the KConceptGraph View Panel. These two panels have a very tight interaction relation because of the implementation mechanism that guides to select classes from Reference Ontology Panel and add them as KConcept instances in KConceptGraph Panel. Using this feature I construct the KConceptGraph. This action presupposes that the designer has implemented tha appropriate classes in the Reference Ontology panel which is an extension of the *SelectClassPanel* implemented in the Protégé OWL API [18]. Moreover, I design the dependence relations between KConcept nodes by pressing Ctrl + left click on the source node and dragging the edge over the target node. The designer can any time remove a KConcept node or a KHasPrerequisite edge, unless this node is not used as a TargetKConcept of KObject. Removing a node causes the associated edges to be removed too. In the KConceptGraph Scene panel I can zoom in various ways or perform a Layout algorithm. Figure 7.2(b) shows the implanted constructions.



C:\CROP Editor Projects\CROPEditorUsage/Reference Ontology/CROPEditorUsage.owl

Figure 7.2(a) Reference Ontology and KConceptGraph panels.

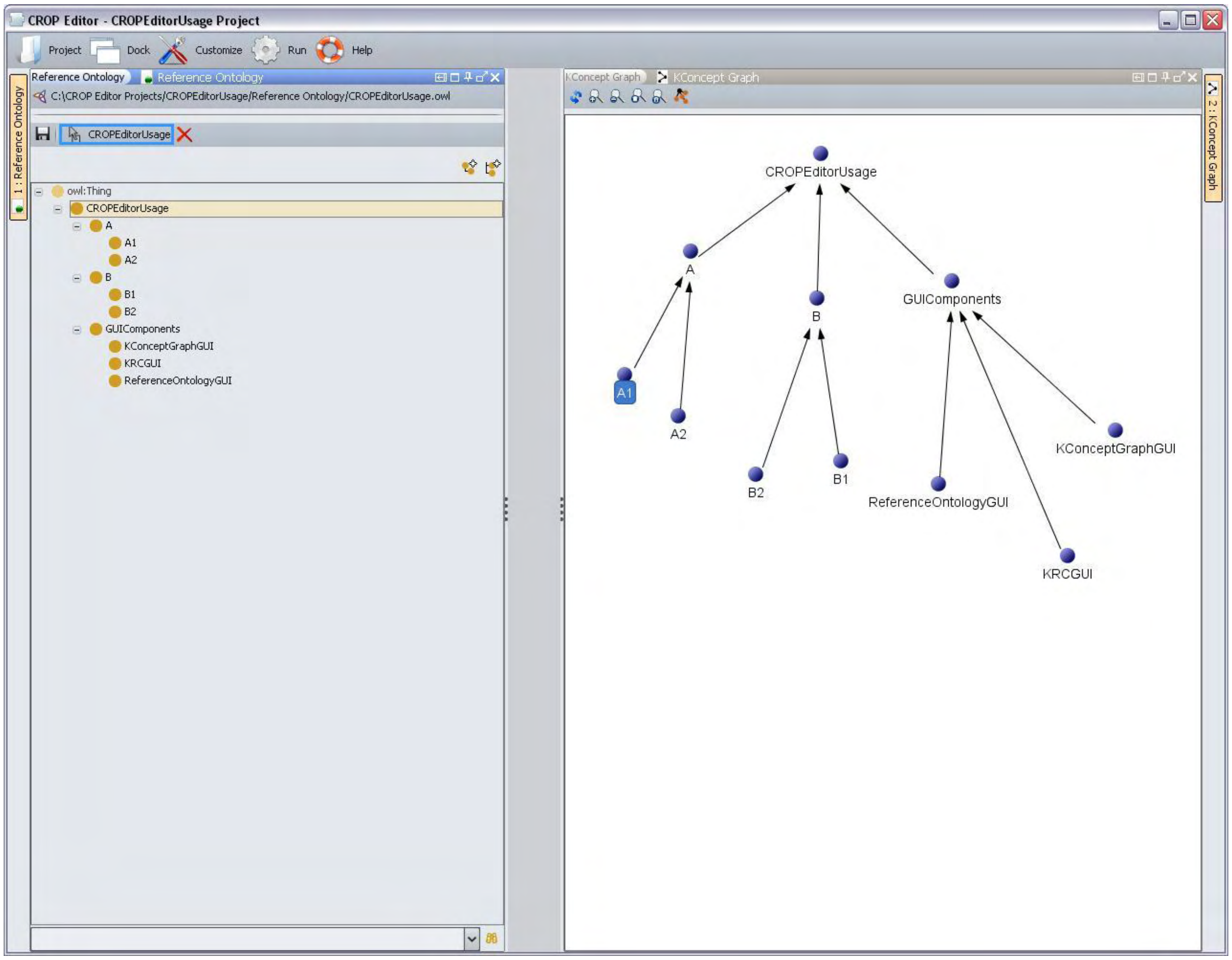
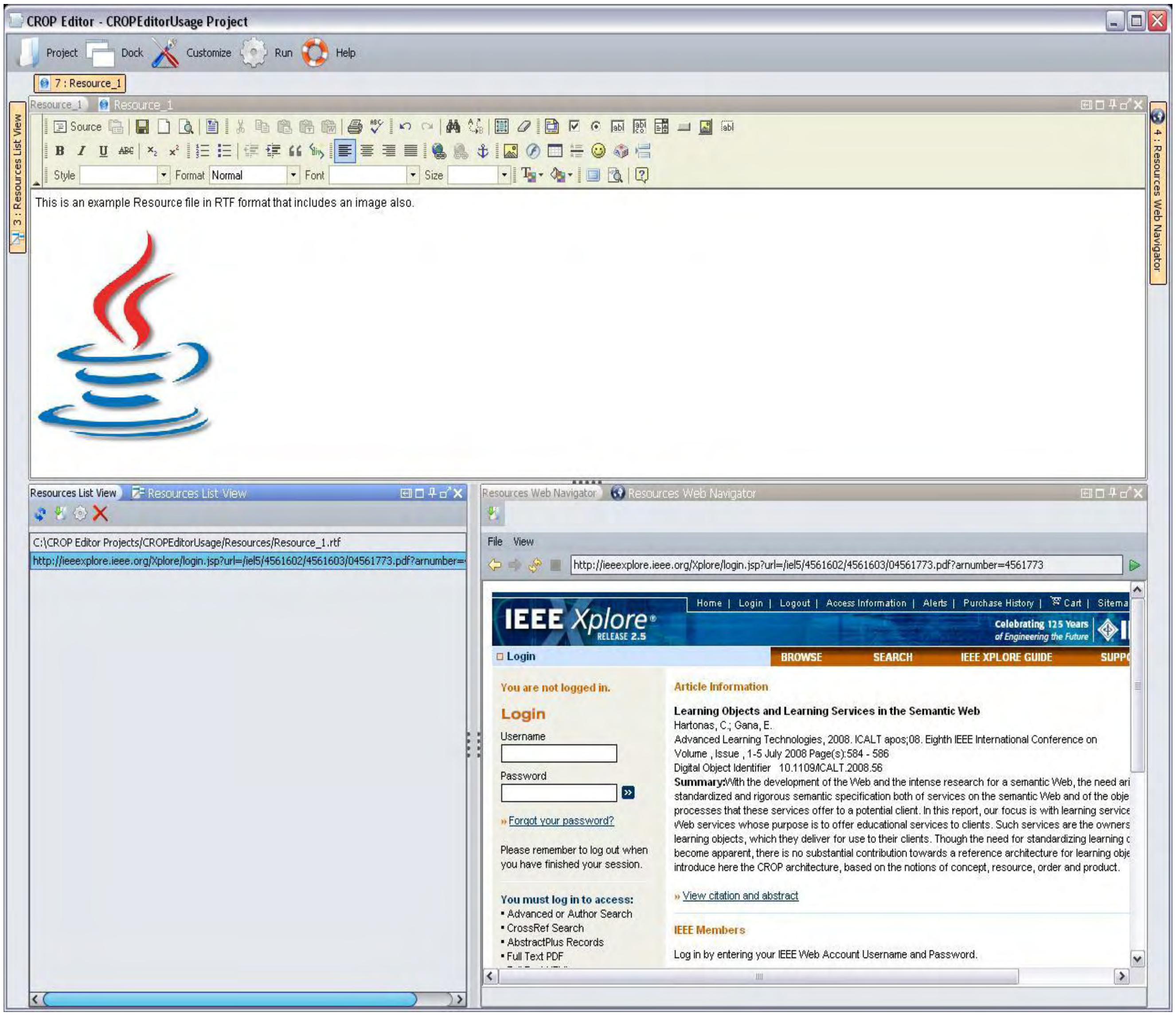


Figure 7.2 (b) an ontology – KConceptGraph construction example

Following, Figure 7.3 shows the *Resources Menu* panels: The *HRML/RTF New Resource Panel*, the *Resource List View Panel* and the *Web Resource Navigator* Panel. The FCKEditor [21] integrated in the DJNative Swing Project [22] offers full HTML rendering and edit capabilities. Pressing the *Save* button this HTML page is saved in RTF format as explained in the previous Chapter. The Web Navigator panel launches a *built in* Java Web Browser that is also integrated in [22]. The Resources List View Panel lists the available resource files and URI that are stores in the Resource project sub-folder.



This is an example Resource file in RTF format that includes an image also.

C:\CROP Editor Projects/CROPEditorUsage/Resources/Resource_1.rtf
http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/4561602/4561603/04561773.pdf?arnumber=

http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/4561602/4561603/04561773.pdf?arnumber=4561773

You are not logged in.

Login

Username

Password

» Forgot your password?

Please remember to log out when you have finished your session.

You must log in to access:
- Advanced or Author Search
- CrossRef Search
- AbstractPlus Records
- Full Text PDF

Article Information

Learning Objects and Learning Services in the Semantic Web
Hartonas, C.; Gana, E.
Advanced Learning Technologies, 2008. ICALT apos;08. Eighth IEEE International Conference on
Volume , Issue , 1-5 July 2008 Page(s):584 - 586
Digital Object Identifier 10.1109/ICALT.2008.56
Summary: With the development of the Web and the intense research for a semantic Web, the need ari
standardized and rigorous semantic specification both of services on the semantic Web and of the obje
processes that these services offer to a potential client. In this report, our focus is with learning service
Web services whose purpose is to offer educational services to clients. Such services are the owners
learning objects, which they deliver for use to their clients. Though the need for standardizing learning c
become apparent, there is no substantial contribution towards a reference architecture for learning obje
introduce here the CROP architecture, based on the notions of concept, resource, order and product.

» View citation and abstract

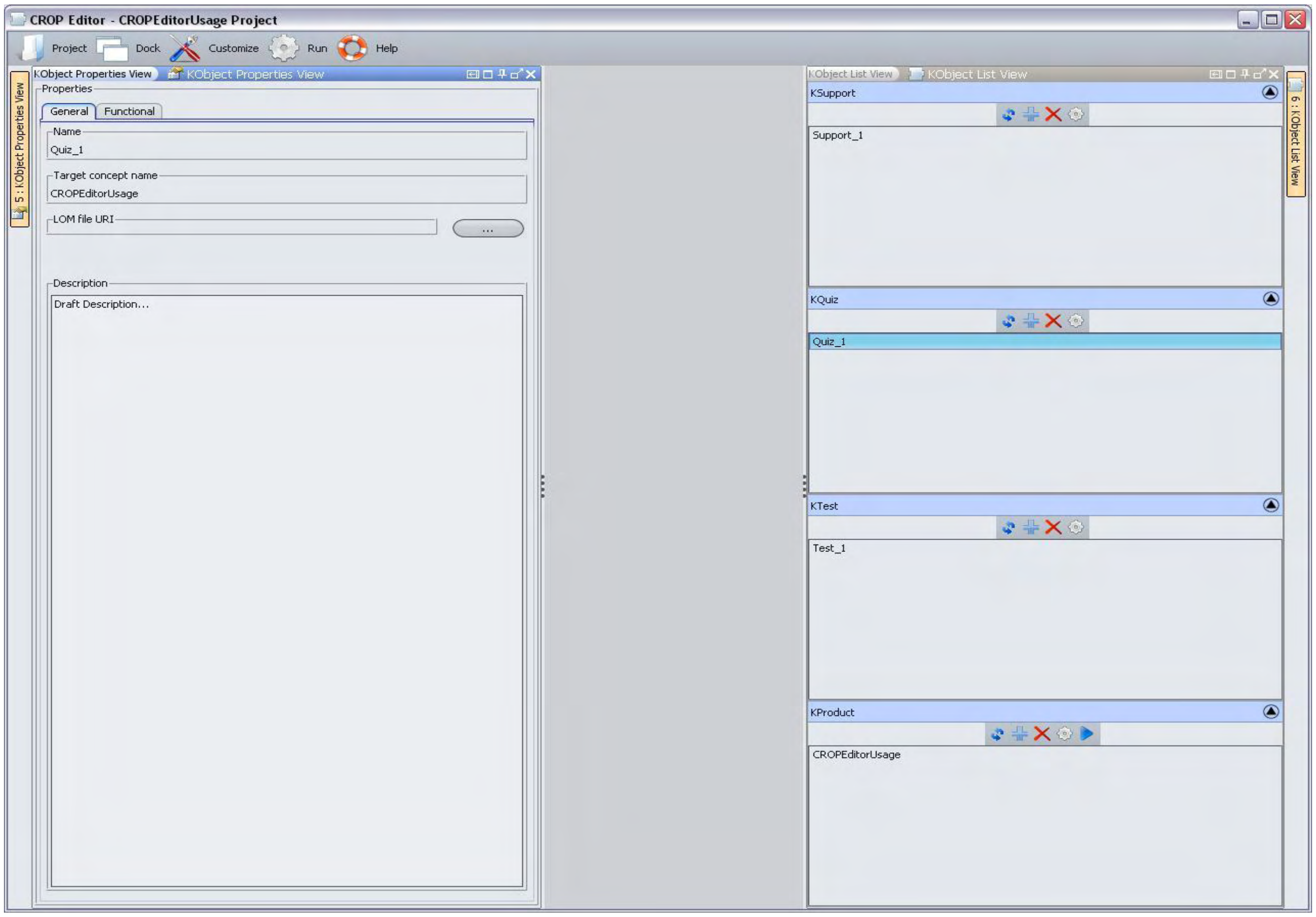
IEEE Members

Log in by entering your IEEE Web Account Username and Password.

Figure 7.3 *Resources Menu* dock able panel views

Figure 7.4 (a) shows the CROP Editor with KObjectList View and KObject Properties View Panels docked. The KObjectList View Panel offers navigation between the KSupport, KAssess and KProduct instances, while it provides add/remove and launching functionality using the toolbars. Selecting a KObject Instance the KObjectProperties View Panel is automatically populated with the KObject properties respectively. For this example I created one instance per object type and I have selected the KQuiz listed object to show in the Screenshot its properties. When the user clicks the *add* button for each KObject panel, a dialogue appears that prompts the user to provide a *name* for KObject instance and select a KConcept from the list. This will be TargetKConcept. This panel is presented in Figure 7.4 (b).

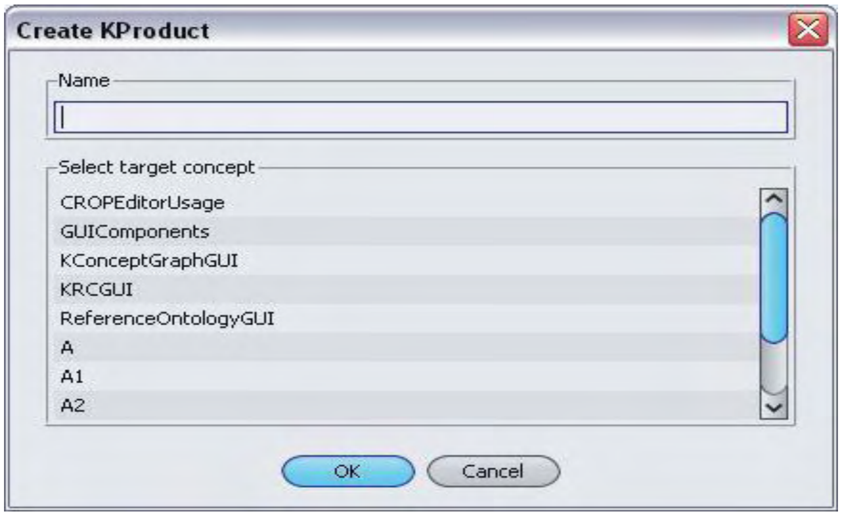
The Properties View Panel provides for every KObject type a *General* Tab pane where various information are shown: the name, the target concept, the reference ontology URI and also supports LOM document association and description text input. In case of a KSupport instance there is a selection mechanism to associate this object with a resource file URI. In case of KQuiz, KTest and KProduct instances one additional tab pane is added to the properties panel· the *Functional* named pane· In KQuiz mode this pane supports *Question* text input, *Answers* definition and success/failure text input. In KTest mode the designer can select a sequence of implemented KQuiz instances. In KProduct mode this Tab pane is named *KOrder* and the designer can create XModel instances and launch their XGraph Editor that is discussed and presented further. Figure 7.4 (c) shows screenshots of the above.



Name: Quiz_1
Target concept name: CROPEditorUsage
LOM file URI:
Description: Draft Description...

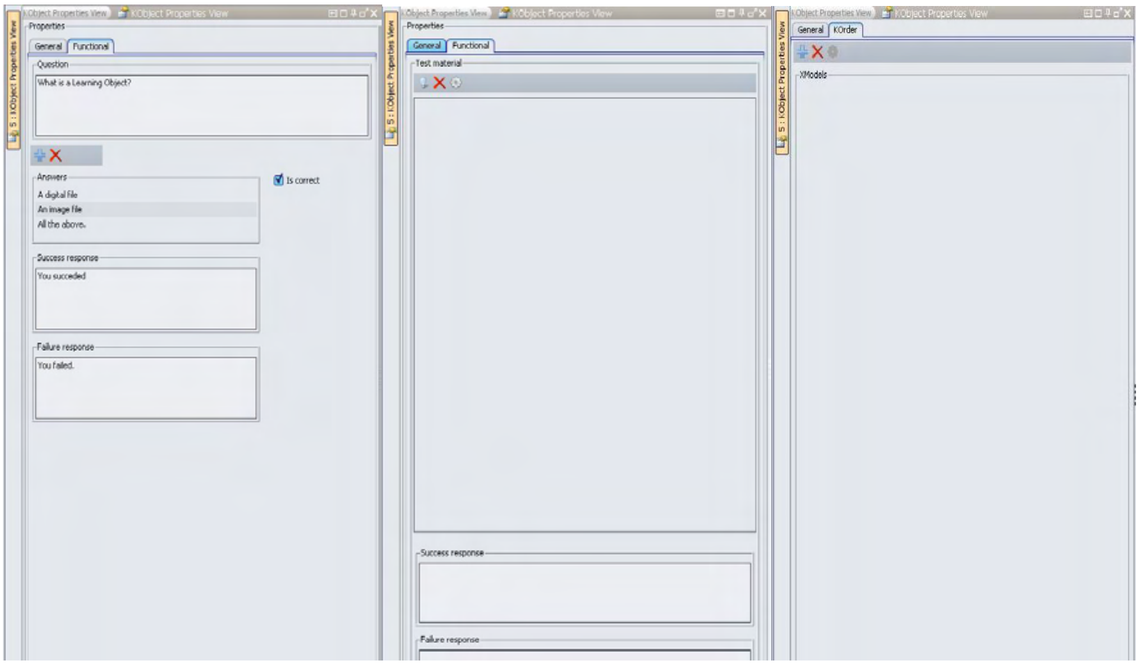
KSupport: Support_1
KQuiz: Quiz_1
KTest: Test_1
KProduct: CROPEditorUsage

Figure 7.4 (a) KObject List and Properties View Panels



Select target concept:
CROPEditorUsage
GUIComponents
KConceptGraphGUI
KRCGUI
ReferenceOntologyGUI
A
A1
A2

Figure 7.4 (b) Create a new KObject instance



What is a Learning Object?

A digital file
An image file
All the above.

You succeded

You failed.

Figure 7.4 (c) KQuiz, KTest and KProduct *Functional* Tab panes.

Continuing this presentation in Figure 7.5 I present the KRC Graph editor. This Editor is also implemented using a Visual Graph Scene as in KConceptGraph. Creating a KProduct instance by default a single KRCNode is created· the node that corresponds to the selected TargetKConcept. Selecting a KRCNode the designer can update its dependencies using the KRCNode *properties panel* from the Dependencies collapsible panel area. This panel shows the node type for the selected KRCNode. Furthermore, the designer can implement KResource instances using the menu of each KRCNode by right – clicking on it. In addition, a Drag n Drop functionality is supported here, so to pick KObject instances from the KObjectList View Panel and drop them on a KRCNode updating in that way its node type. For this demonstration I have implemented two (2) KProducts. One has as TargetKConcept the *CROPEditorUsage* concept and the other the concept *A.* To show the functionality of the KRC Graph Editor I have updated the node type of the KRCNode instance name as *A_node* and have added to its node type the second KProduct instance that has as TargetKConcept the *A* concept. It is important here to note that a the assignment of a KObject instance to the KObjectList (i.e. the node type) of a given KRCNode, presupposes that this node and the selected KObject have the same TargetKConcept.

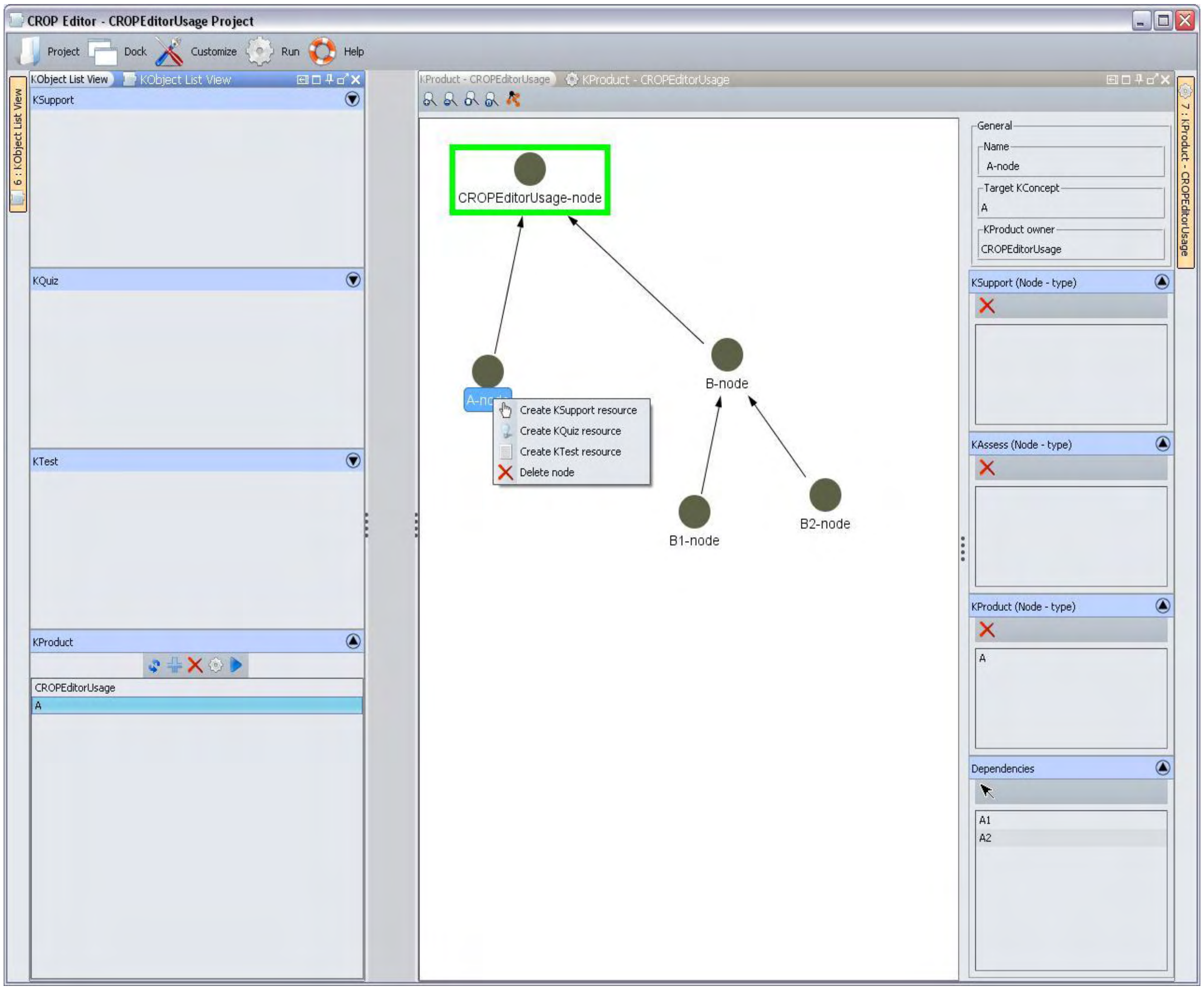


Figure 7.5 the KRC Graph Editor

As already told previously the Properties View panel for a KProduct supports the implementation of XModel instances. Figure 7.6 shows the XGraph Editor. When the designer decides to add an execution model a panel appears and prompts for *Name* input and *XModelType* (Custom, DefaultHorizontal or DefaultVertical) selection. When the XModel is of custom type the XGraph editor is available. Except from the main XModel XGraph Editor, when an XConceptNode is selected the XGraph for this node appears for editing. The only difference between the XModel XGraph and the XConceptNode XGraph is that the XConceptNode graph accepts XResourceNode and XsubProductNode instances that can be added from the list that appears in the panel and contains the KObjectList of the selected XConceptNode.

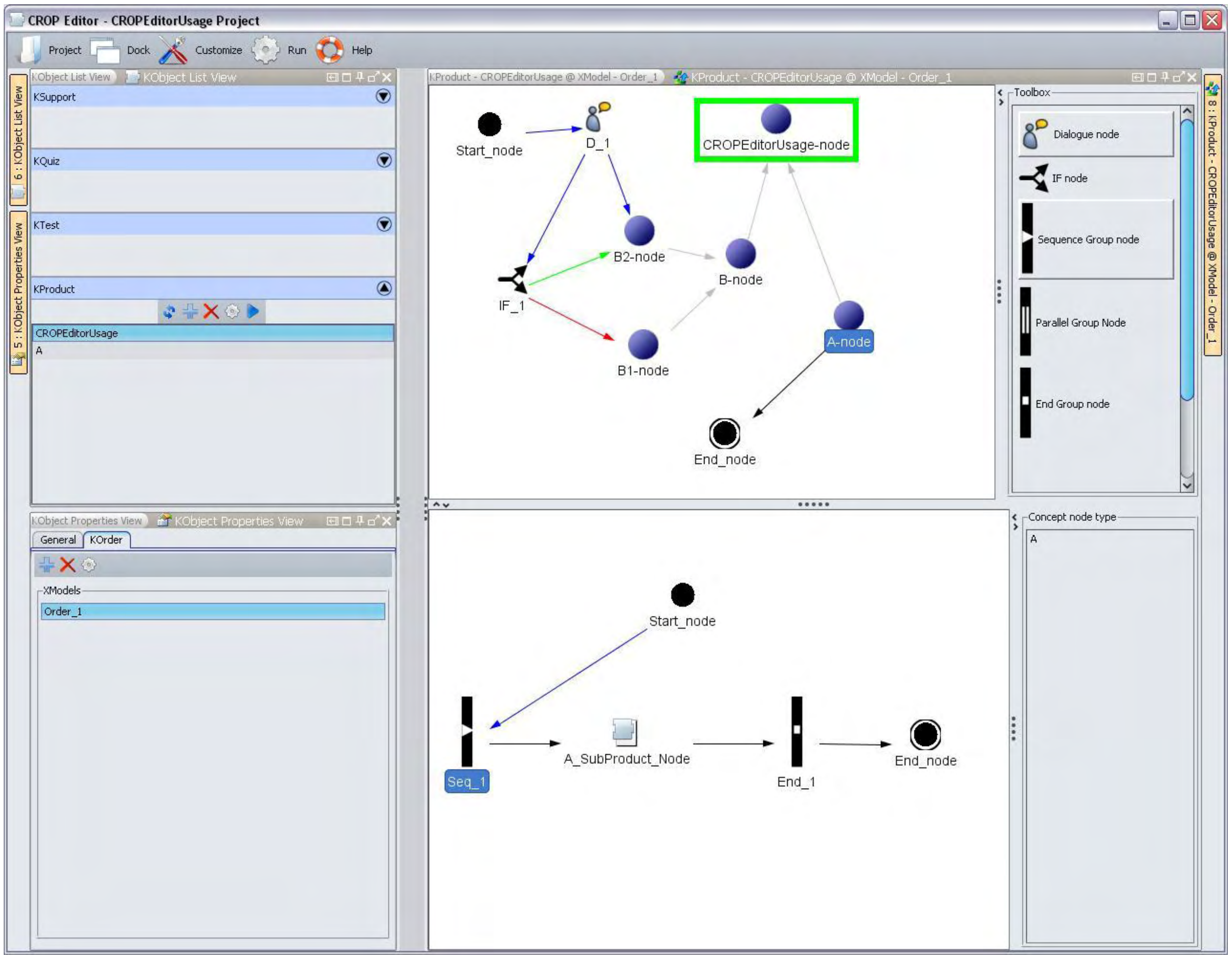


Figure 7.6 the KOrder - XModel Editor

Selecting the *Customization Wizard* from the *Customize Menu* the user can change the workspace directory and the Look and Feel of the application. Figure 7.7 shows the previous figure (7.6) using the *Motif* Look and Feel.

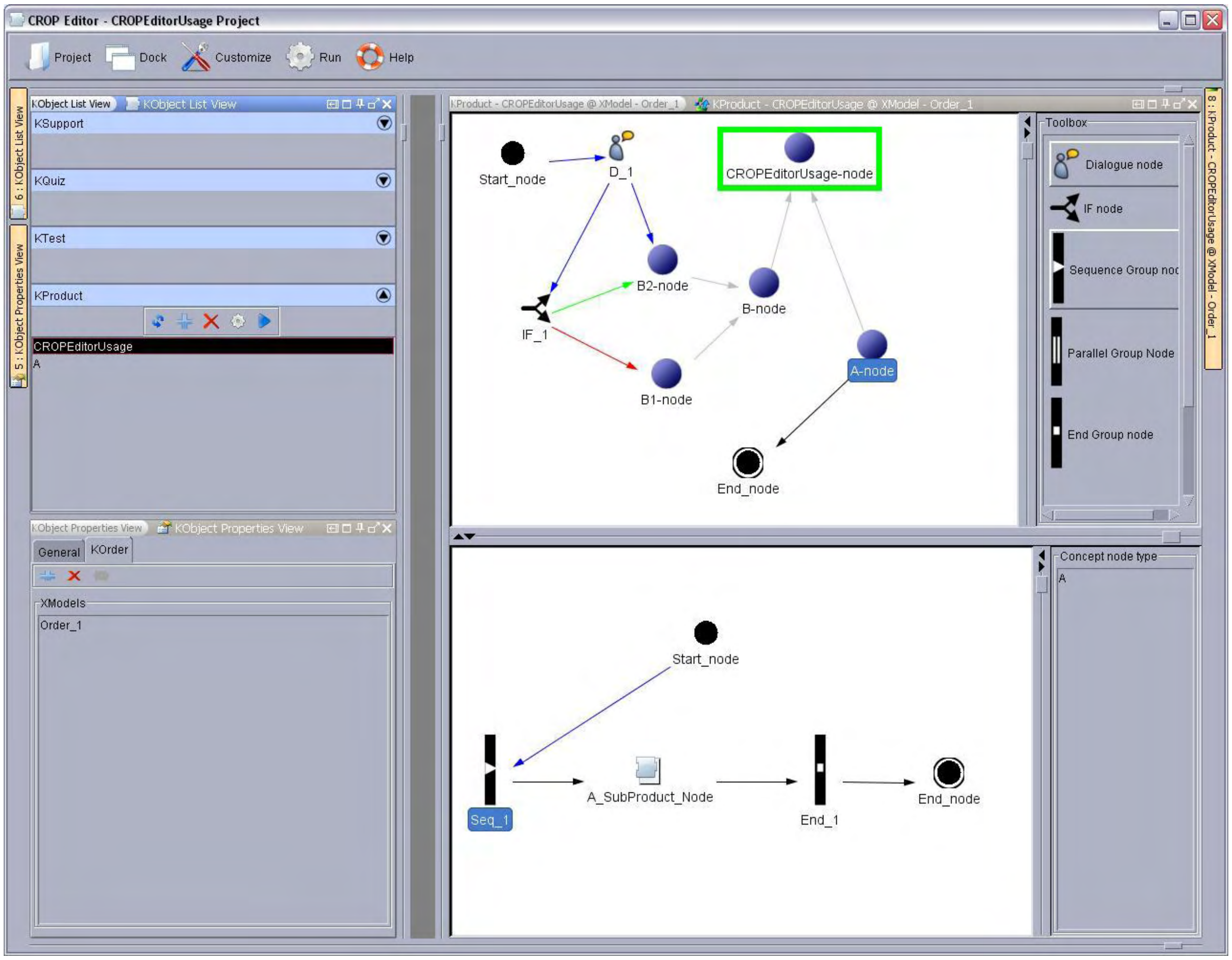
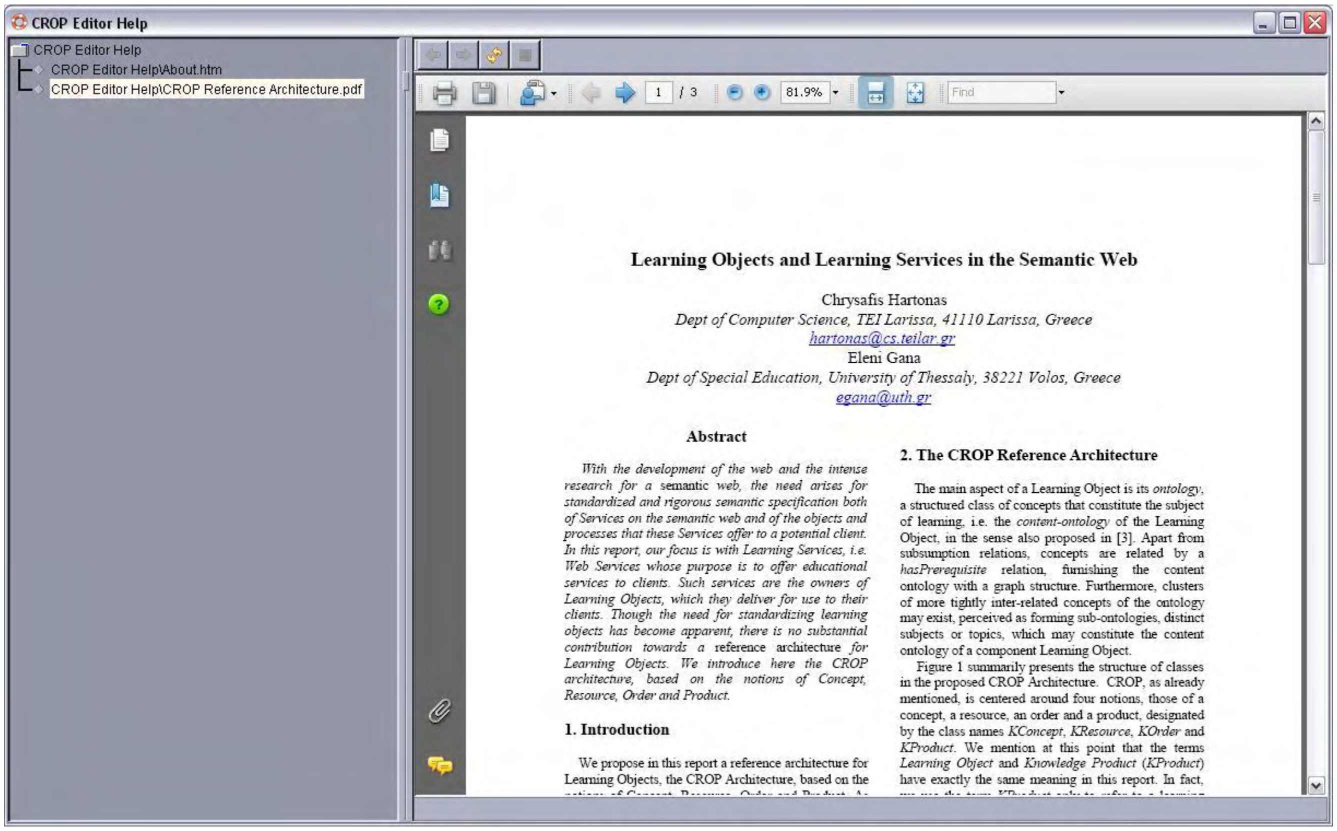


Figure 7.7 the Motif Look and Feel

Figure 7.8 shows the Help Frame of the CROP Editor. The Help set list is updateable by editing the content of the *CROP Editor Help* folder. The preview panel can render Web pages, Images, text and xml files, pdf files, while in case we want to load documents of different the appropriate applications launches automatically.


CROP Editor Help
CROP Editor Help
CROP Editor Help\About.htm
CROP Editor Help\CROP Reference Architecture.pdf
1 / 3
81.9%
Find

Learning Objects and Learning Services in the Semantic Web

Chrysafis Hartonas
Dept of Computer Science, TEI Larissa, 41110 Larissa, Greece
hartonas@cs.teilar.gr
Eleni Gana
Dept of Special Education, University of Thessaly, 38221 Volos, Greece
egana@uth.gr

Abstract

With the development of the web and the intense research for a semantic web, the need arises for standardized and rigorous semantic specification both of Services on the semantic web and of the objects and processes that these Services offer to a potential client. In this report, our focus is with Learning Services, i.e. Web Services whose purpose is to offer educational services to clients. Such services are the owners of Learning Objects, which they deliver for use to their clients. Though the need for standardizing learning objects has become apparent, there is no substantial contribution towards a reference architecture for Learning Objects. We introduce here the CROP architecture, based on the notions of Concept, Resource, Order and Product.

1. Introduction

We propose in this report a reference architecture for Learning Objects, the CROP Architecture, based on the

2. The CROP Reference Architecture

The main aspect of a Learning Object is its *ontology*, a structured class of concepts that constitute the subject of learning, i.e. the *content-ontology* of the Learning Object, in the sense also proposed in [3]. Apart from subsumption relations, concepts are related by a *hasPrerequisite* relation, furnishing the content ontology with a graph structure. Furthermore, clusters of more tightly inter-related concepts of the ontology may exist, perceived as forming sub-ontologies, distinct subjects or topics, which may constitute the content ontology of a component Learning Object.

Figure 1 summarily presents the structure of classes in the proposed CROP Architecture. CROP, as already mentioned, is centered around four notions, those of a concept, a resource, an order and a product, designated by the class names *KConcept*, *KResource*, *KOrder* and *KProduct*. We mention at this point that the terms *Learning Object* and *Knowledge Product* (*KProduct*) have exactly the same meaning in this report. In fact,

Figure 7.7 Help support in CROP Editor.

CHAPTER 8 – CONCLUSION

The CROP Editor application follows the CROP Reference Architecture Specification and implements the core framework for CROP Learning Objects and a wrapper to facilitate the editing process. It achieves a high – level usability and the editing process is clear according to the software – user interaction. A very important factor is the Graphical User Interface implementation which in the Editor appears to be very handy from the designer's perspective. In addition all the software components (sub-systems applied with visual UIs) are fully updateable so duplication of work inside the editor is avoided. Resources design and implementation procedure is also well defined letting the designer to select from a variety of resource types taking the advantage to select the storage URI of the actual digital resource in support learning case· or build a well structured Quiz (and Test) to assess learning.

In the context of this thesis, I provided the Editor for the CROP Learning Object Architecture. There is a lot of research and development activity left as future work. From the Editor's perspective, there are several enhancements that can facilitate even more the editing process. Some of them are listed below:

- Design and development or use of existing built – in resources editor components. This will make the framework more independent. A solution to this issue could be the OpenOffice.org SDK and API [26]. At the moment there is an implementation of a Java Bean to integrate this application (Writer, Impress etc.) into a Java application with the constraint that the OpenOffice.org Suite is installed in the host system.
- Refinement of the KResource objects developing process and enhancement off learning assessment with additional object types (exercises and interactive live tests).

- Implementation of the CROP Ontology saving mechanism. It will be very useful the designer to be able to save a CROP project (and a Learning Object consequently) in OWL format. Using this feature the consistency and completeness of a Learning Object can be evaluated with Ontology Reasoning application. This will also meet the LOM Ontology specification at [11].
- An embedded LOM Editor will also supplement the system independence and portability of the CROP Editor.

As it is presented in Chapter 6, the KPlayer sub-system of the CROP Editor has been designed in a preliminary stage. The component has to be designed and implemented to meet all the features that CROP Architecture cites and must be integrated to Editor. It is highly recommended that KPlayer has to provide a CROP Editor independent run mode also, in order to be available for learners (students) as a separate application.

As SCORM Specification [13] guides the design and implementation of learning material, a Learning Object must meet the following requirements: Accessibility, Adaptability and Reusability. Thus, inside the CROP Editor application must be integrated a sub-system that will provide the designer the ability to import existing resources and products in a project. This feature concerns a very close case study of SCORM. Furthermore, a Learning Object should be available in the meaning of searchable, accessible and retrievable. This is the actual aim of a Learning Service. SCORM also cites these needs that a learning environment must meet. Applying the SCORM guidelines in the CROP Editor along with a complete KProduct player implementation and a Student model prototype, CROP Learning Objects can be available over a Web Learning Service. These features complete the instructional environment that CROP cites.

Following the above acknowledgements, this development stage of the CROPS Modeling Framework (i.e. the *CROP Editor*) can be concluded and integrated accordingly with the other structural components that briefly presented in Chapter 1.

ACKNOWLEDGEMENTS

This work would not have been possible without the support and encouragement of my advisor, Professor C. Hartonas. He smoothly assisted me during the whole period of my elaboration over this thesis. With his inspiration, great effort and valuable time dedication for our meetings, he advised me and provided guidelines, so things to become clear. His teaching was determinative for my mind growth and inspiration to continue research in Computer Science.

I would like to thank many people that helped me overcome technical issues of this work with their experience over several difficulties I encountered and advised me without a hitch, through forum discussion boards over the internet.

Finally, I wish to thank my family for their emotional support. Their contribution was invaluable. To them I dedicate this thesis.

REFERENCES

[1] C. Hartonas, E. Gana, "Learning Objects and Learning Services in the Semantic Web", Proc ICALT 2008.

[2] C. Hartonas, E. Gana, "Adaptivity for Knowledge Content in the Semantic Web", Proc KGCM 2008.

[3] R. McGreal, "Online Education Using Learning Objects", 2005

[4] C. Larman, "Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design", 1999

[5] C. Larman, "Applying UML and Patterns", 1997

[6] R. Malan, D. Bredemeyer, "Defining Non-Functional Requirements", Architecture Resources For Enterprize Advantage, Bredmeyer Consulting, 2001

[7] T. Katsaros, Dissertation in "*Adaptive Learning Objects*", MSc in Computer Science, Staffordshire University UK, TEI of Larisssa GR, September 2008.

[8] M. Tsiakmaki, PhD dissertation in progress (advisor Prof C. Hartonas), private communication.

[9] IEEE Learning Technology Standards Committee (2002) (PDF), Draft Standard for Learning Object Metadata. IEEE Standard 1484.12.1, New York, Institute of Electrical and Electronics Engineers, http://ltsc.ieee.org/wg12/files/LOM_1484_12_1_v1_Final_Draft.pdf

[10] The Instructional Use of Learning Objects. Wiley, D. (ed) Section 1.1: Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy. Online version at http://www.reusability.org/read/

[11] IMS Global Learning Consortium http://www.imsglobal.org/

[12] Web Ontology Language W3C, Semantic Web, http://www.w3.org/2004/OWL/

[13] LOM Ontology, http://www.teilar.gr/~hartonas/lom.owl

[14] IMS Learning Design Best Practice and Implementation Guide ,Version 1.0 Final Specification at http://www.imsglobal.org/learningdesign/ldv1p0/imsld_bestv1p0.html

[15] SCORM 2004, 3rd Edition at http://www.adlnet.gov/Scorm/

[16] Sun Java, http://java.sun.com/

[17] Java Development Kit (JDK) at http://java.sun.com/javase/downloads/index.jsp

[18] NetBeans IDE at http://www.netbeans.org/

[19] MyDoggy Docking Framework at http://mydoggy.sourceforge.net/

[20] The Protégé OWL API v.3.3.1 at http://protege.stanford.edu/plugins/owl/api/

[21] The Substance Java Look and Feel Package at https://substance.dev.java.net/

[22] The Japura v.1.2.1 Java Library at http://sourceforge.net/projects/japura/

[23] The FCKEditor at http://www.fckeditor.net/

[24] DJNative Swing Component Library, The DJ Project at http://djproject.sourceforge.net/ns/

[25] The JTidy Library at http://jtidy.sourceforge.net/

[26] The Apache Formatting Objects Processor (FOP) at http://xmlgraphics.apache.org/fop/

[27] The Java Visual Library v.2.0 at http://platform.netbeans.org/tutorials/nbm-visual_library.html

[28] OpenOffice.org Productivity Suite at www.openoffice.org/

APPENDIX - JAVA SOURCE CODE FOR THE CROP EDITOR CORE

The listing below shows all the packages implemented in Java for the CROP Editor Application, as they produced in the JavaDoc pages included in the attached CD.

Packages :
cropedit
cropedit.core.KConcept
cropedit.core.KConceptGraph
cropedit.core.KHasPrerequisite
cropedit.core.KObject
cropedit.core.KOrder
cropedit.core.KOrder.Performance
cropedit.core.KOrder.StudentModel
cropedit.core.KOrder.XGraph
cropedit.core.KOrder.XGraph.XEdge
cropedit.core.KOrder.XGraph.XNode
cropedit.core.KOrder.XModel
cropedit.core.KProduct
cropedit.core.KRC
cropedit.core.KResource
cropedit.core.KResource.KAssess
cropedit.core.KResource.KSupport
cropedit.CROP
cropedit.Customization
cropedit.Help
cropedit.KPlayer
cropedit.KPlayer.Path
cropedit.KPlayer.Visual
cropedit.KPlayer.XModelCollector
cropedit.ReferenceOntology
cropedit.Resources
cropedit.Tools
cropedit.UI.CROP
cropedit.UI.Customization

cropedit.UI.Help
cropedit.UI.KConceptGraph
cropedit.UI.KObject
cropedit.UI.KOrder
cropedit.UI.KOrder.XConceptNode
cropedit.UI.KOrder.XDialogueNode
cropedit.UI.KOrder.XEdge
cropedit.UI.KOrder.XEndGroupNode
cropedit.UI.KOrder.XEndNode
cropedit.UI.KOrder.XIFNode
cropedit.UI.KOrder.XModel
cropedit.UI.KOrder.XParGroupNode
cropedit.UI.KOrder.XResourceNode
cropedit.UI.KOrder.XSeqGroupNode
cropedit.UI.KOrder.XStartNode
cropedit.UI.KOrder.XSubProductNode
cropedit.UI.KProduct
cropedit.UI.KQuiz
cropedit.UI.KSupport
cropedit.UI.KTest
cropedit.UI.MainWindow
cropedit.UI.ReferenceOntology
cropedit.UI.Resources

The following pages contain the source code for the ***cropedit.core*** Java package that implements the kernel software component· the CROP Reference Architecture implementation.