

D1.3 – Two-level authoring widget software and documentation

Citation for published version (APA):

Boytshev, P., Stefanov, K., Mascarenhas, S., & van der Vegt, W. (2019). *D1.3 – Two-level authoring widget software and documentation: RAGE – WP1 – D1.3*. RAGE project.

Document status and date:

Published: 01/01/2019

Document Version:

Publisher's PDF, also known as Version of record

Document license:

CC BY-NC-SA

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

<https://www.ou.nl/taverne-agreement>

Take down policy

If you believe that this document breaches copyright please contact us at:

pure-support@ou.nl

providing details and we will investigate your claim.

Downloaded from <https://research.ou.nl/> on date: 09 Sep. 2021

Open Universiteit
www.ou.nl





RAGE

Realising an Applied Gaming Ecosystem

Research and Innovation Action

Grant agreement no.: 644187

D1.3 – Two-level authoring widget software and documentation

RAGE – WP1 – D1.3

Project Number	H2020-ICT-2014-1
Due Date	January 31, 2019
Actual Date	January 24, 2019
Document Author/s	PB, KS, SM, WV
Version	1.0
Dissemination level	PU/ RE
Status	Draft/ Final
Document approved by	WW



Document Version Control			
Version	Date	Change Made (and if appropriate reason for change)	Initials of Commentator(s) or Author(s)
0.1	2018.12.28	Writing of Section 3.	PB
0.2	2018.12.30	Writing of Section 4.1.	PB
0.3	2019.01.02	Writing of Section 4.2.	PB
0.4	2019.01.05	Writing of Section 5.1 and 5.2.1	PB
0.5	2019.01.06	Writing of Section 5.2.2, 5.3 and 5.4	PB
0.96	2019.01.22	Fixing figure indices in 6.1. Rewriting of Section 6.2 with a solution. Writing of Section 6.3.	PB

Document Change Commentator or Author		
Author Initials	Name of Author	Institution
PB	Pavel Boytchev	Sofia University
KS	Krassen Stefanov	Sofia University
SM	Samuel Mascarenhas	INESC ID
WV	Wim van de Vegt	OUNL

Document Quality Control			
Version QA	Date	Comments (and if appropriate reason for change)	Initials of QA Person
	Jan 23	Minor	WW

TABLE OF CONTENTS

	Page
EXECUTIVE SUMMARY	6
1 INTRODUCTION	7
2 OVERVIEW OF WP1 TOOLS	8
3 SOFTWARE COMPONENT AUTHORIZING TOOLS	10
3.1 The Data-centric approach and the RAGE Metadata Editor	10
3.1.1 Design principals	11
3.1.2 User interface	12
3.1.3 Metadata verification	13
3.2 The User-centric approach and the Asset creation Wizard.....	15
3.2.1 User interface	15
3.2.2 Metadata verification	17
3.2.3 New metadata elements.....	21
3.3 Comparison of both approaches	22
4 COMPLEMENTARY SOFTWARE COMPONENT TOOLS.....	24
4.1 Taxonomy tools	24
4.1.1 RAGE Taxonomy Viewer.....	24
4.1.2 RAGE Taxonomy Selector	27
4.1.3 RAGE Taxonomy Editor	30
4.2 Web Portal Tools	33
4.2.1 RAGE Repository Manager	33
4.2.2 RAGE Component Viewer.....	35
4.2.3 RAGE Artefact Manager.....	36
5 CONFIGURATION AUTHORIZING TOOLS.....	38
5.1 Goals and design.....	38
5.2 Implementation details	40
5.2.1 Preliminary prototype.....	40
5.2.2 Current implementation	41
5.3 Parameters of the authoring tool	43
5.4 Distribution.....	47
6 CASE STUDIES AND EVALUATION	48
6.1 The use of RAGE Configuration editor wizard for client-side assets	48
6.2 Case study on the application of the Configuration Editor to FAtiMA Toolkit	52
6.3 Evaluation summary	58
7 CONCLUSIONS	59
REFERENCES	60

LIST OF FIGURES

Figure 1: Cascading processing of metadata schema files	11
Figure 2: Metadata meta-editor constructing a metadata editor in real time	12
Figure 3: Opening page and the metadata hierarchy in the Metadata Editor	13
Figure 4: Meta-metadata shown (in red fine-print)	14
Figure 5: Display of internal metadata nodes	15
Figure 6: Snapshots from the RAGE Software Asset Wizard	17
Figure 7: The last step of the wizard with metadata completeness scores	17
Figure 8: Embedded Taxonomy Viewer	25
Figure 9: Collapsed and expanded concept nodes	25
Figure 10: Horizontal and vertical layout of taxonomy concepts	26
Figure 11: Multilingual support in taxonomies	26
Figure 12: The RAGE Taxonomy Selector	27
Figure 13: End-nodes and top-levels	27
Figure 14: Complex taxonomy in the Taxonomy Selector	28
Figure 15: Single or multiple concept selection	28
Figure 16: Architecture of embedding a Taxonomy Selector	29
Figure 17: Taxonomy Selector embedded in a Metadata Editor	29
Figure 18: Moving a node in the Taxonomy editor	30
Figure 19: Structural changes in a taxonomy	30
Figure 20: Creating new taxonomy concepts	31
Figure 21: Untranslated taxonomy concepts	31
Figure 22: The Taxonomy Editor in advanced mode	32
Figure 23: Repository architecture	34
Figure 24: Repository Manager workflow	35
Figure 25: The RAGE Metadata Viewer	35
Figure 26: The viewer as an intermediate layer between the manager and the wizard	36
Figure 27: Artefact Manager Workflow	37
Figure 28: The in-game use of RAGE software components	38
Figure 29: Configured software components	38
Figure 30: RAGE Configuration Editor Wizard	39
Figure 31: Generated Configuration Editor and configuration file	40
Figure 32: Metadata meta-editor constructing a configuration editor in real time	41
Figure 33: Using the wizard and the editor	41
Figure 34: Unification of Configuration Wizard and Configuration Editors	42
Figure 35: Configuration wizard and editor duality	43
Figure 36: Visual layout of the Configuration Wizard	45
Figure 37: Tracker Asset Settings Editor Definition	49
Figure 38: Generated Tracker Asset Settings Editor	50
Figure 39: Version and dependencies Editor Definition	51
Figure 40: Generated Version and dependencies Editor	52
Figure 41: Description of FATiMA configuration	53
Figure 42: Description of FATiMA configuration (cont)	54
Figure 43: Generated configuration editor for FATiMA	55
Figure 44: FATiMA configuration editor with data	55
Figure 45: FATiMA configuration editor with more data	56

LIST OF TABLES

Table 1: Partial verification within the meta-editor	13
Table 2: Semantic grouping of metadata	16
Table 3: Verification rules in the wizard	17
Table 4: Weight of metadata groups	19
Table 5: Englishability of some texts	20
Table 6: Comparison of metadata editors (developer’s perspective).....	22
Table 7: Comparison of metadata editors (user’s perspective)	22
Table 8: Examples of generated concept identifiers	32
Table 9: Interpretation of the wizard’s interface	45
Table 10: Supported customization properties.....	46

EXECUTIVE SUMMARY

In this deliverable, we give detailed description of all end user tools developed in the frame of WP1, with the special emphasis on the Configuration Editor Wizard, described in Chapter 5. So far, significant parts of the asset creation infrastructure have been reported in previous deliverables D1.1 and D1.4. In order to counteract the fragmentation of the descriptions of different parts in different documents and annexes, we have opted for providing a comprehensive overview of the full range of authoring tools created in Task 1.3. We felt a coherent, full overview is needed, because of the interdependencies of the various tools that we created to accommodate the asset creation methodology. The deliverable summarises the various component authoring tools, taxonomy tools and component management tools, reported before, but also presents the new Configuration Editor Wizard: this tool allows component developers to define and generate a tailored configuration editor alongside their component, which then can be used by the user of the component (viz. game developers) to configure the component for technical integration in different software and gaming platforms. The very idea of having “an editor for creating editors” explains the reference in the deliverable’s title to “two-level authoring”. Readers already familiar with the contents of the previous deliverables may want to go immediately to chapter 5, which presents the Configuration Editor Wizard and a detailed description of two case studies used to test and demonstrate the effectiveness of the Configuration Editor Wizard.

1 INTRODUCTION

This deliverable is a culmination of the efforts done in WP1. The main goal of WP1 was to develop a new innovative methodology, relevant specifications and all needed tools in order to support game asset development.

The RAGE project is unique in that it is not focussed exclusively on the development of applied games, but in parallel on the development of an ecosystem for the creation, exchange and storage of assets¹ that could, in turn, be incorporated within different games targeting multiple audiences and domains. In this context, often existing approaches and methodologies used by game development companies, cannot be directly adopted, reused or repurposed, as they typically focus on design and development of singularly domain-specific games.

A RAGE Asset is a self-contained solution that demonstrates economic value potential, based on advanced technologies related to computer games, and intended to be reused and repurposed in a variety of game platforms and scenarios (cf. D1.4).

The RAGE assets contain advanced game technology components (software), enriched and transformed to support applied games development. They are combined with value added services and attributes, such as instructions, tutorials, exemplars, best practice guides, instructional design guidelines and methodologies, connectors to the major game development platforms, data capacity, and content authoring tools/widgets for game content creation. Such artefacts should facilitate their usage. The assets are technologically underpinned by an asset meta model and framework for interoperability, and they are shared, exchanged and reused in the form of RAGE Asset packages.

The RAGE meta model defines the structure of the RAGE Asset package and the metadata format, used to describe different components of the RAGE Asset package.

The RAGE Asset Development Methodology provides details of the activities and actions required to develop a validated RAGE Asset package. The proposed methodology is based on established software development practices and processes and enables software specialists to develop specific software components that will form RAGE Asset core, applying their own preferences and methodologies. Initially the primary users of the RAGE Asset Development Methodology were RAGE Asset package developers and a First draft version of this methodology was tested when developing some of the RAGE Assets thus providing early identification of potential problems or issues.

On the base of the RAGE Asset Development Methodology, we defined all relevant standards and protocols, to be used by applied gaming assets for exchanging information with other game components, LMS, evaluation systems, HRD systems, etc. The main result was the game asset metadata model, needed to describe all the main features of the gaming assets (Georgiev et al., 2016) and the RAGE client-asset architecture (Van der Vegt et al., 2016).

In order to use this model, we have designed and developed a web-based back-end infrastructure supporting full asset (code, metadata, documentations, etc.) development, packaging, sharing and distributing. Our infrastructure includes an asset repository and a metadata editor for creating/editing assets descriptions and defining elements.

Although some tools have already been described in previous deliverables (D1.1, D1.4), we have opted for presenting a comprehensive overview of the full range of authoring tools created in Task 1.3 in order to counteract fragmentation. We felt a coherent, full overview is needed, because of the interdependencies of the various tools that we created to accommodate the asset creation methodology. Chapter 2 provides a quick overview of the repository system and its tools. Chapter 3 elaborates the software component authoring process and tools. Chapter 4 presents a series of complementary tools re taxonomies and asset management. Chapter 5 presents the configuration authoring tool (Configuration Editor Wizard) that we developed to prepare and support the configuration of the gaming assets into different software and gaming platforms. Readers already familiar with the contents of the previous deliverables may want to go immediately to chapter 5, which presents the Configuration Editor Wizard and a detailed description of two case studies used to test and demonstrate the effectiveness of the Configuration Editor Wizard.

¹ In this document the term “asset” refers to a software component”, possibly enriched with manuals, tutorials and other artefacts (cf. D1.4). The terms are often used as synonyms.

2 OVERVIEW OF WP1 TOOLS

The RAGE Software Asset Repository is at the core of the asset development infrastructure (see Task 1.3) and the main function of the repository is to facilitate the process of RAGE Asset development, sharing, exchange, reuse and repurposing. The repository (and its associated tools) is a subsystem of the full RAGE infrastructure, which is loosely coupled with the RAGE publication portal (cf. D1.1, chapter 6).

The architecture Software Asset Repository makes no assumptions about the specific metadata being vocabulary proposed other than that it will 1) access certain common fields for reporting purposes and 2) expect version and dependency information to be present.

The Asset software repository is the core of the asset development infrastructure. It is used to store and manage access to:

- Reusable game assets
- Artefacts – resources within game assets
- Metadata for game assets and for artefacts
- Relationships between assets – dependencies, related assets, etc.

The Asset software repository leverages the discovery, development and reuse of game assets and artefacts. It will help both game asset developers and consumers in all the activities concerning game assets life-cycle.

The Asset repository is complemented by various front-end tools.

RAGE Taxonomy Tools (viz. Viewer, Selector, Editor)

The taxonomies that are used in RAGE's asset metadata schema (Georgiev et al., 2016) represent hierarchies of concepts and controlled vocabularies. They are used to provide prefixed sets of values for some metadata elements. The front-end taxonomy tools in RAGE are actually a set of three tools – Taxonomy Viewer, Taxonomy Selector and Taxonomy Editor. They are generic tools that are all written in JavaScript, share a common core and can be embedded in a web page.

The design principles of the taxonomy tools are: (1) minimalistic user interface and set of features; (2) consistent visual layout; (3) multiplatform support through HTML5 and JavaScript; and (4) multilingual interface and taxonomy content.

The RAGE Taxonomy Viewer is used to browse taxonomies. It provides several layouts and interface languages. The RAGE Taxonomy Selector is used to pick concepts from taxonomies. This tool supports the functioning of other front-end tools, like metadata editor, asset configuration tools, various authoring tools, etc. The RAGE Taxonomy Editor is used to edit a taxonomy. It supports structural changes to a taxonomy (such as adding, deleting and reallocating concept nodes around the hierarchy) and content changes (such as changing concept names and concept translation in several languages).

RAGE Metadata Editor

This front-end tool provides the functionality to edit the metadata associated with an asset. The editor hides the internal metadata complexity and constructs a flexible dynamic interface.

When a metadata file is loaded by the editor, it extracts the schema definition of the metadata and builds the user interface. The hierarchy of metadata is represented as nested blocks which nesting goes as deep as it is defined by the schema. After the metadata are edited by the user, the editor generates an XML representation of the metadata and sends it back to the server.

Asset Creation Wizard

Finally, the asset creation methodology, the quality assurance considerations and the asset metadata requirements were merged together and implemented into a single asset creation wizard, which supports and guides asset owners through the process of asset submission to the Ecosystem portal (Deliverable D1.1). It complements the metadata editor as a demand coming from RAGE asset developers. The wizard design, which was based on an analysis of the asset submission workflow, decomposes the submission process into 8 subsequent steps, while a limited subset of metadata fields are qualified as mandatory. The wizard was used and evaluated by all RAGE's asset developers.

The purpose of the RAGE Wizard is to assist in preparing the metadata description of a software asset. It splits the efforts into several annotated steps. This style of presenting the metadata is

not suitable for the case when users just want to view the asset. To access this problem we developed the RAGE Metadata Viewer.

The RAGE Metadata Viewer

This tool extracts the metadata of a software asset and presents it in a structured page.

The main purpose of the viewer is to provide an easy-to-read description of a software asset with appropriately grouped metadata. Additionally, the viewer interface is printer-friendly – i.e. if the user prints the page, the viewer hides navigation and control buttons and prints only the metadata. This is suitable for creating off-repository records of assets.

In the rest of this Deliverable, we describe all software tools in full details, presenting their dependencies and main usage scenarios.

All tools are available at <https://bitbucket.org/account/user/rage-su/projects/RT>

3 SOFTWARE COMPONENT AUTHORIZING TOOLS

The goal of the RAGE project is to support applied gaming industry by making available reusable game software components. The RAGE software components contain both a software core, implementing the functionality of a component, and a rich set of complementary content (tutorials, examples, methodologies, etc.)

All components are described by additional data, called metadata (Georgiev et al, 2016). Currently the model envisions a significant amount of metadata to be collected for each component, used for its description, classification, searching and filtering. In order to programmatically analyse and process these metadata, they are presented in a well-specified format. Due to the amount of metadata for a component, it is not feasible for a human to work directly with this format – users would need a metadata editor, which is a software tool for presenting and manipulating metadata to assist people.

There are several approaches for making a metadata editor. Two of these approaches are implemented in the RAGE project and described in this deliverable:

1. Data-centric approach – i.e. the most important element is the metadata and all other elements, including the user, are adjusted to accomplish more effective metadata processing. This is the first approach used to make a metadata editor. The resulting software is called *RAGE Metadata Editor*.
2. User-centric approach – it considers the user as the most important element and all other elements are processed in a way to make user's experience more comfortable and confusion-free. This is the other approach used to make a metadata editor for RAGE. To distinguish it from the previous editor, the second one is named *RAGE Metadata Wizard*.

3.1 The Data-centric approach and the RAGE Metadata Editor

The RAGE Metadata Editor is software for editing metadata. It is actually a metadata *meta-editor* for components, packages and other entities. It is designed to simplify the user interface and to adhere to some flexibility in metadata structure. The metadata entry form is split in several tabs, which use standard web controls for user input – text boxes, list boxes, buttons and links. The visual appearance of the form is customizable through widgets and CSS styling.

The meta-editor collects descriptions of the metadata from schema files by processing XML Schema structures. Then it dynamically generates a specific metadata editor for a given metadata set. The generated user interface depends on the structure of the metadata and the collected descriptions. The interface uses customizable widgets to present appropriate visual layouts for different metadata. When editing is completed, the editor performs basic verification of the metadata and converts it back into the same format as the input metadata.

The concept of generating metadata editor from metadata schema is not new. *MDEdit* is an example of such editor (Suleman, 2003) using the same general concept, but with different implementation. *MDEdit*, however, does not process nested structures and the schema files are extended with new formatting tags. Another editor is *EUOSME* (European Open Source Metadata Editor). Its interface is close to the RAGE meta-editor, but, it is tuned to geospatial metadata, which are not part of the RAGE metadata model (Grasso, Craglia, 2010). The *OLR3-Editor* is described as able to accommodate any metadata description. However, it “assumes only local data, which are stored in the OLR3 database, and is not yet adapted for working with distributed data.” (Kunze, Brase, Nejd, 2002). Finally, the *ARCADE Authoring Tool* (Architecture for Reusable Courseware Authoring and Delivery) uses DTD and XSD content templates together with XSL presentation templates. As a result it generates XML/WML course content and HTML course presentation (Bontchev, Iliev, 2003). This authoring tool supports hierarchies and could be extended by additional export modules. The *ARCADE Authoring Tool* is not immediately applicable to the context of RAGE, as both projects have different goals and domains.

3.1.1 Design principals

The design of the metadata editor addresses several core principles. The editor hides the internal hierarchy and complexity of the metadata representation. An XML schema defines the metadata properties, relations and constraints. Some items are implemented in different ways and the editor hides these differences. Thus it presents the metadata to the user in a unified way.

For example, the cardinality of attributes in the schema is defined as `use="required"`, while the cardinality of tags is defined as `minOccurs="1"`. Another example is the use of controlled vocabularies. They are implemented in three distinct ways: as schema enumeration types `<enumeration>`; as external taxonomies; and as `xml:lang` attributes.

Another core design principle is the provision of flexibility – the metadata editor is not bound to a single fixed and predetermined metadata structure. Instead, it reads its definition schema and reconstructs the structure of the metadata.

Currently the RAGE software component schema is composed of private RAGE-specific definitions enriched by industry-standard definitions from the a general schema for web-based assets ADMS (Europeana's Asset Description Metadata Schema), the vocabulary for interoperability between web-based data catalogues DCAT (Data Catalog Vocabulary), the vocabulary describing resources in a searchable way DCTERMS (Dublin Core Terms), the semantic description of people and their social relations FOAF (Friend-of-a-friend Schema) and the specification for conceptual description of metadata models for web resources RDF (Resource Description Framework) – Figure 1.

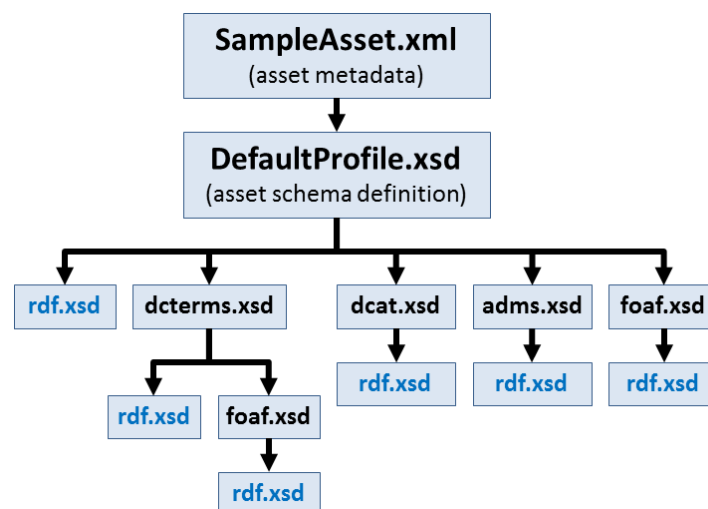


Figure 1: Cascading processing of metadata schema files

The most important underlying principle in the RAGE Metadata editor is that it is actually a *meta-editor*, not an editor. As a metadata meta-editor, it builds a metadata editor in real time as shown in Figure 2.

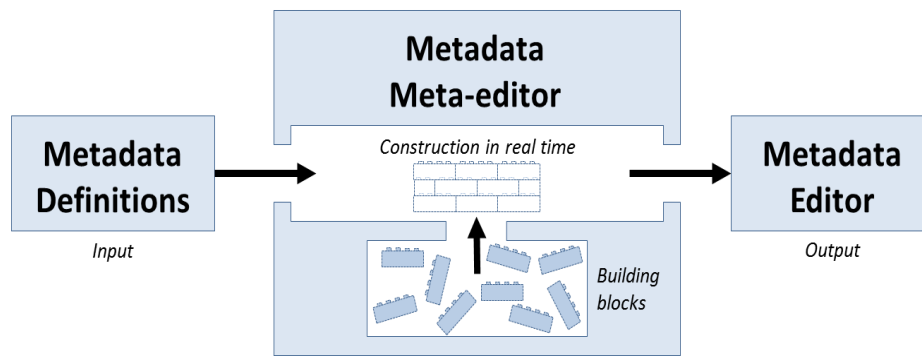


Figure 2: Metadata meta-editor constructing a metadata editor in real time

The meta-editor is equipped with a collection of building blocks and construction algorithms. The input data contain heterogeneous definitions of metadata, like the metadata model itself, additional schemas, taxonomies and styling preferences. Then the meta-editor constructs the graphical user interface of an editor, which is presented to the user.

The benefits of using a meta-editor, instead of editor are:

- Changes in the structure of metadata do not require changes in the meta-editor. This provides the flexibility to adapt the metadata model with minimal impact on other project software.
- If another metadata schema is given to the meta-editor, it will generate another editor. In this way the initial intent of building an editor of asset metadata is implemented as a general purpose metadata editor that could process other RAGE entities like components, packages and artefacts.

The most notable disadvantage of building a meta-editor is that it is more complex piece of software and requires significantly more efforts. To ease future improvements and modifications of the meta-editor, it relies on widgets that encapsulate the underlying variety of metadata definitions.

3.1.2 User interface

A typical initial view of a RAGE Metadata Editor is shown in the left snapshot in Figure 3. The metadata elements are grouped in several tabs, called *Main*, *Classification*, *Status*, *License*, *Solution* and *Usage*. There is one additional tab, named *All*, which lists all metadata in a single page.

Tabs contains metadata elements in a specific category: *Main* – metadata about the general properties of the software component: title, description, type, date, language, and access URL; *Classification* – metadata describing and classifying the component: keywords and taxonomy classifications; *Status* – metadata about the current version and development status, as well as relation to other components: version, version notes, status, maturity level, custom metadata, related components and dependencies with other components; *License* – metadata about the people and organizations related to the component and its license: creator, publisher, owner, and license; *Solution* – metadata about the software and its components: description, requirements, implementation, design, engine, platform, programming language, and tests; and *Usage* – metadata about component installation, customization, configuration and usage.

The last two categories contain metadata for artefacts – component resources, like data or documentation files, which have their own set of metadata: name, reference, type, creator, version, format, license, etc.

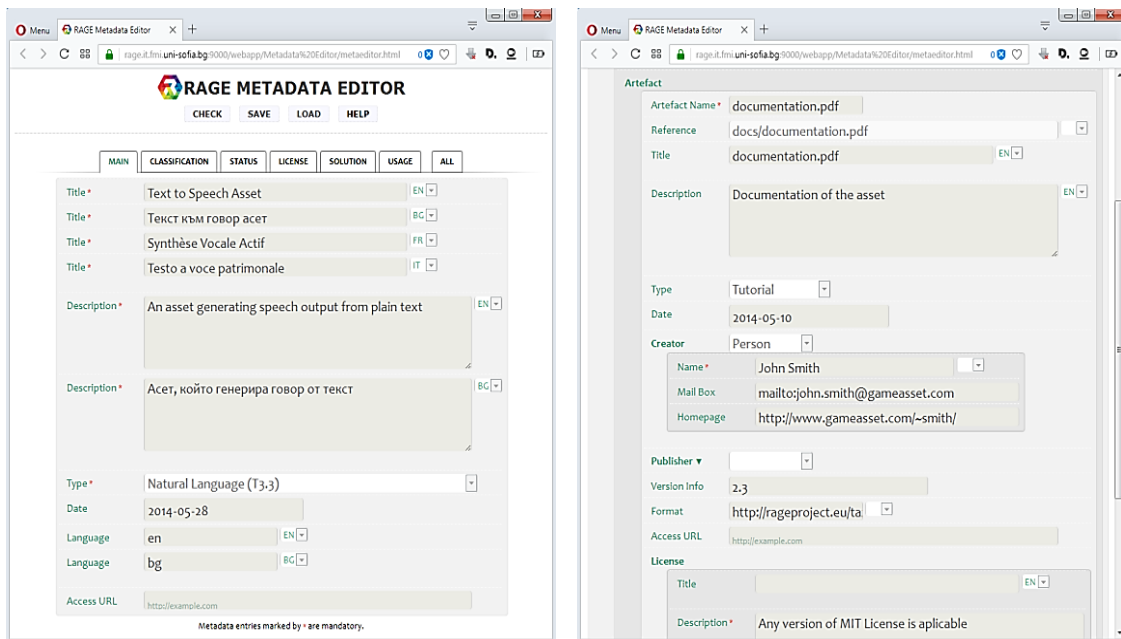


Figure 3: Opening page and the metadata hierarchy in the Metadata Editor

The metadata elements are positioned vertically from top to bottom and follow the order and the nesting defined in the metadata schema. The actual appearance of the elements depends on the browser. The hierarchy of the metadata model is preserved in the editor. Nested metadata are represented as nested blocks, which can be expanded, collapsed, duplicated or deleted.

Most of the metadata use standard user interface control elements like text boxes, list boxes and buttons. The only exception is the section for component classification, where users select concepts from taxonomies, relevant to the component. The visualization of the classification section is managed by another RAGE tool – the *Taxonomy Selector* (cf. chapter 4.1.2).

3.1.3 Metadata verification

The definition of metadata in RAGE may be done via the metadata meta-editor, or via direct upload of components. Thus the verification module of the metadata is not a part of the meta-editor. As a result, the meta-editor does not provide a full verification. It only supports some rudimentary measures to partially verify the metadata entered by the user. Details are provided in Table 1.

Table 1: Partial verification within the meta-editor

Item	Description	Means of verification
Metadata structure	Metadata is a complex hierarchical structure conforming to the metadata model.	The metadata model is encoded in the metadata schema. The meta-editor creates the user interface in accordance to the schema. The location of metadata in the model's hierarchy is enforced on the user interface.
Missing metadata	Some metadata are optional and may be omitted, other are compulsory.	There is a visual indication for compulsory metadata – red asterisks beside the labels. Additionally, when the metadata is prepared for the server, the user is notified about all missing compulsory metadata.
Conditionally missing metadata	Some compulsory metadata are inside an optional block. They are treated compulsory only if the block is not empty.	When the metadata is prepared for the server, the editor explicitly checks all compulsory metadata inside optional metadata. This check is performed on the full depth of the hierarchy. The user is notified if there is missing compulsory data in optional blocks.

Date metadata	Some metadata hold dates as strings.	The editor uses the HTML5 date input type. If the browser supports it, the user will fill a predefined template for the date or select a date interactively. For older browsers the date metadata is accepted as a string without verification of the contents.
Schema-based controlled vocabulary	Some metadata may have a value from a fixed list of values coded in the schema.	The editor generates a list box with the available values, so the user cannot select a value, which is not allowed. The only exception is the “empty” value, which is identical with lack of data.
Taxonomy-based controlled vocabulary	Some metadata may have a value from RAGE taxonomy.	The editor generates a list box with the available taxonomy concepts, so the user cannot select unknown concepts. If the taxonomy is changed and some metadata become invalid, they will be skipped by the editor as if they are “empty” values.
Classification taxonomy	The user may add classification based on selected concepts from selected taxonomies.	The editor allows the user to select only taxonomies that are compatible with the taxonomy tools and exist in the RAGE repository. When the user selects concepts from these taxonomies, the same verification as in the taxonomy-based controlled vocabulary is used.

Apart from this partial verification, the meta-editor has a special developer’s mode. This mode is used during the development of the meta-editor while the actual asset metadata records are still incomplete or incorrect. When the tool works in this mode, it provides additional internal details, such as links between visual elements and the metadata schema, log files of metadata properties, toggling metadata visibility, etc. (see Figure 4 and Figure 5).

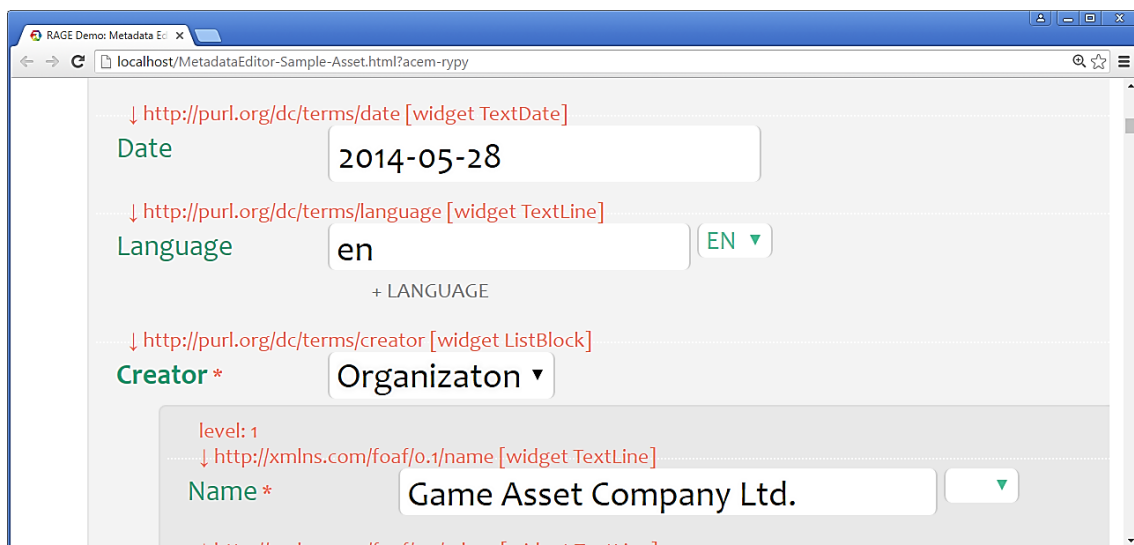


Figure 4: Meta-metadata shown (in red fine-print)

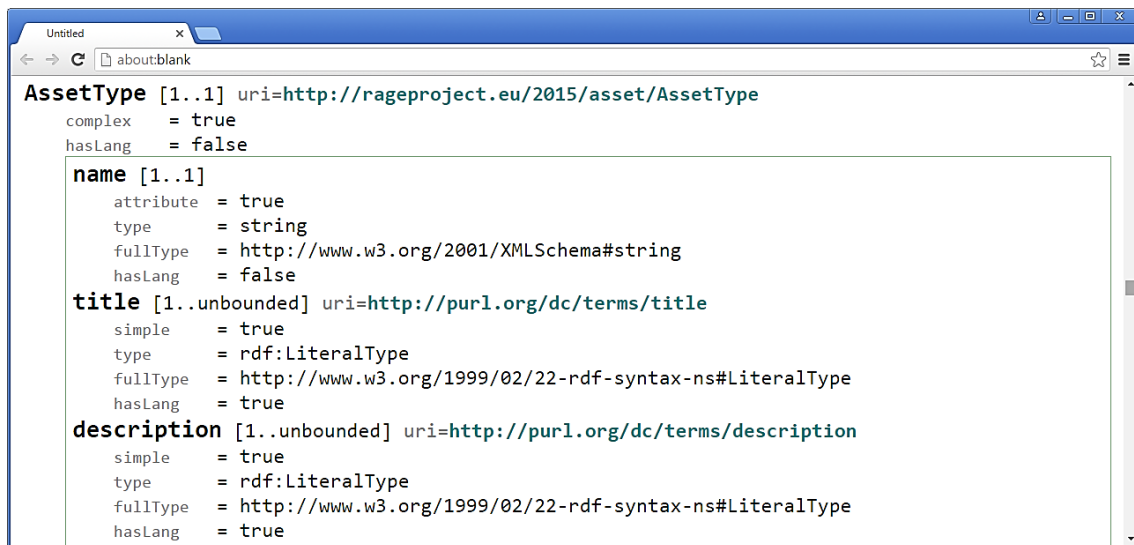


Figure 5: Display of internal metadata nodes

In addition to the developer's mode, the meta-editor may be configured at run-time. It may change the source of metadata (retrieving metadata from the server or from a local file), the visual appearance (by using alternative CSS files), or the metadata schema.

3.2 The User-centric approach and the Asset creation Wizard

The second approach of designing and implementing a metadata editor is focused on the user and the user experience. This shifts both the design and the internal architecture of the editor. While the data-centric meta-editor is targeted towards experienced users, who know the details of the metadata model of RAGE assets, the user-centric metadata editor is targeted to external component developers, who are not expected to be familiar with how the description of components is presented in the metadata records. As a result, the user-centric metadata editor was closer to a wizard software, rather than to a general editor. Thus the name of this metadata editor is set to *RAGE Asset Metadata Wizard*.

Although both the meta-editor and the wizard are used to modify the same record of metadata, they follow completely different approaches. This significantly affects the implementation of both tools. The metadata wizard is built from scratch. It does not use any internal components from the meta-editor. The data processing is straightforward, because the interface and the analysis of metadata are hardcoded in the source code. The wizard provides several advantages over the data-centric meta-editor: it guides the developer through several steps of describing the component; the interface is simple, easy to use and self-explanatory; no any preliminary knowledge of the metadata model is required; indicators shows the level of completeness of the description and the verification is done as early and complete as it is possible.

3.2.1 User interface

Considering the actual needs of the contents of the component description, the wizard shows only a minimal set of 20 compulsory and 14 optional preselected metadata elements. The other elements are not used by the wizard, but they are still accessible by the data-centric meta-editor. The preselected metadata elements are semantically clustered into eight groups. Each group forms a step of the component description process – see Table 2.

Table 2: Semantic grouping of metadata

Group / page	Description	Metadata elements
About	General information about the component, e.g. title, description, logo, access URL, etc.	5 compulsory 3 optional
Classification	Information about target platforms, programming language and applied computing keywords.	1 compulsory 6 optional
Status	Contains software version, version notes, commit reference, development status.	3 compulsory 1 optional
License	Details about licenses, conditions and potential restrictions.	1/2 compulsory 1 optional
Contacts	Information about owners and creators of the component.	2 compulsory
Resources	Files or references of the software, documentation, tests, etc.	2 compulsory 5 optional
Quality	Information about the component's quality.	3 optional
Submission	Review of component description completeness, self-declaration form and component submission.	No metadata elements

The appearance of the wizard is presented in Figure 6. The left snapshot is the first step containing the general asset description (group About), the right snapshot is the 7th step, which contains the self-declaration form (group Quality).

Component Wizard

STEP 1
STEP 2
STEP 3
STEP 4
STEP 5
STEP 6
STEP 7
STEP 8

STEP 1
STEP 2
STEP 3
STEP 4
STEP 5
STEP 6
STEP 7
STEP 8

About the component 100% completed

Welcome to the RAGE Component Wizard. It will guide you through the process of describing your component. First we'll create a general description of the component.

Name
This is the first information that a user will see about your component. Try to think of a name that is informative, short and engaging.

Real-Time Arousal Detection Using Galvanic Skin Response

One sentence description
Please provide a one-line description of what your component can do. This text is crucial as it will show up in the search results and helps users to decide whether or not they want to continue with the component.

The asset detects player's arousal based on measured Galvanic Skin Response (GSR), or Electro-Dermal Activity (EDA), and can be used for adaptation of digital games and other

Short non-technical description
Please explain in 3-4 lines without technical details what your component can do. This text will be used for short advertisements on the web and in search results. Supported [Markdown syntax](#).

Galvanic Skin Response (GSR), also referred to Electro-Dermal Activity (EDA), is directly dependent on the activity of the sweat glands and is often used to index the autonomic arousal. GSR offers a popular and affordable way for detection of player's arousal in adaptive digital games and other affective computing applications. The asset produces some real-time features of GSR signal measured from a particular player and, as well, indicates the level of

Technical description
Please briefly explain some of the technical details of the component. Explain: (1) what the component does; (2) what it could be used for; (3) what inputs are needed; (4) what outputs it produces. Supported [Markdown syntax](#).

The asset produces real-time features of GSR signal measured from particular player, together with the level of player's arousal using a user-defined scale. It receives a filtered raw signal from a simple, low-cost biofeedback device allowing sampling rate up to 0.8KHz.

STEP 1
STEP 2
STEP 3
STEP 4
STEP 5
STEP 6
STEP 7
STEP 8

ABOUT
CLASSIFICATION
STATUS
LICENSE
CONTACTS
RESOURCES
QUALITY
SUBMISSION

Submission 100% overall completed

This is the last step. Please, take a moment to review the completeness of your component description. If necessary, you can go back to any of the previous steps and change or update your description.

1. About	100%
2. Classification	100%
3. Status	100%
4. License	100%
5. Contacts	100%
6. Resources	100%
7. Quality	100%
OVERALL:	100%

Congratulations
The description of your component is complete (as far as we could tell). All mandatory and optional elements are present. You may now save the description to the RAGE server.
Thank you for your excellent work.

Publishing
Please, select whether you want your component to be published to the RAGE portal, when it is submitted to the RAGE repository. Published components can be accessed by all RAGE users.

No, do not publish the component to the RAGE portal.

Yes, do publish the component to the RAGE portal.

I declare that:

The information that I have entered is correct to the best of my knowledge

I take sole responsibility for all content published and activity that occurs under my account. I have previously received consent from persons whose contact

Figure 6: Snapshots from the RAGE Software Asset Wizard

All pages share the same visual style. The upper side contains the sequence of steps (1 to 8) and explanations about the steps. Each metadata element in these pages has its own title and description, which are used to reduce the user confusion about the purpose of the metadata.

3.2.2 Metadata verification

Each page of the wizard shows the percentage of completeness of the metadata on the page. Showing an indicator of completeness is a major requirement for the wizard, along with the requirement of guiding the user through a series of steps. While developers describe their software components in the wizard, it continuously calculates and updates two scores. Apart from the completeness, there is also a score for the component quality. Both scores are recorded in the metadata, but users cannot access and modify them directly.

The scoring algorithm of the wizard is designed to provide as much details and granulated verification as what is reasonably possible. Counting just the presence of metadata and considering whether it is compulsory or optional does not provide enough details – this would be just a rough indicator for completeness.

Although the metadata completeness is shown in each wizard's page (see Figure 6), they are also summarized in the last page of the wizard. Figure 7 is a snapshot of a fragment of this page.

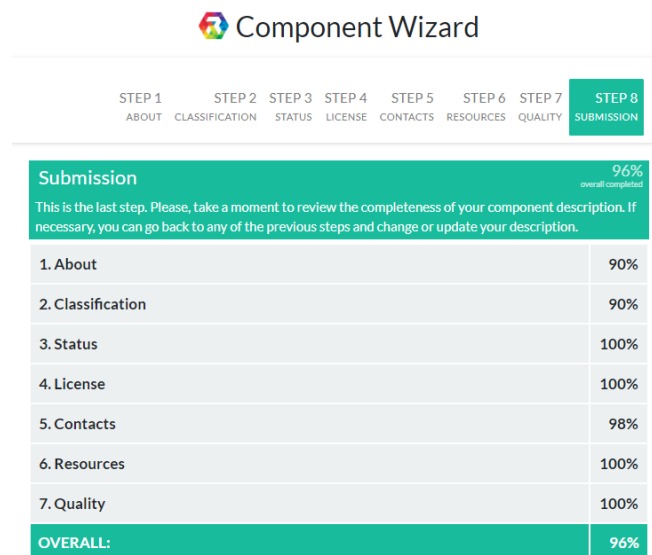


Figure 7: The last step of the wizard with metadata completeness scores

The user describing a component can see the completeness for each group of metadata elements, as well as the component's overall completeness. If compulsory elements are missing, then the wizard shows a warning.

The wizard fine-tunes the scoring by attaching verification rules for each metadata element. Most of the rules inspect not only the presence or absence of metadata, but also the contents of the metadata. Thus the wizard uses an approach, which could capture minor differences based on the actual values. The list of defined rules is presented in Table 3.

Table 3: Verification rules in the wizard

No	Rule	Description
----	------	-------------

1	Default value	The metadata value is the same as the default value, provided automatically by the wizard.
2	“Others” value	Selected is “others” option from a list of predefined values, so the actual value is to be provided as custom metadata.
3	Common value	The value is too common (e.g. the name of the asset is “RAGE asset”), a more specific value is expected.
4	Empty	The value is missing or is an empty string.
5	Optional empty	The value of an optional metadata element is missing or is an empty string.
6	Internal link	The value is a link, URL or URI to a local resource or service within the RAGE server.
7	Length (N)	The number of characters in the value is less than N.
8	Words (N)	The number of words in the value is less than N.
9	Phrases (N)	The number of comma-separated phrases is less than N.
10	Sentences (N)	The number of sentences in the value is less than N.
11	No connection	No connection to the RAGE server, the value is check against cached taxonomies.
12	Version	The version number not N.N, N.N.N or N.N.N.N.
13	Invalid date	The date is either invalid, or its formatting is invalid.
14	Strange date	The date is too old or it is in the future.
15	Custom metadata (N)	The formatting of the custom metadata is not split into at least N pairs <i>name=value</i> .
16	URL	The URL is not correct.
17	Path	The absolute or relative path is not correct.
18	Email	The email address is not correct.
19	English	The value does not appear to be a text in English.
20	Always	This rule is always applicable – it is the termination rule.

Each metadata element has a maximal weight from 0 to 10, defining its importance. The most important elements, like the component title, has weight 10. Less important ones have smaller weights, e.g. the development status is 8, the programming language is 6, the component date is 5.

Depending on the actual value, some metadata may have their weights reduced. This reduction contributes to the fine granulation of completeness indicators.

The overall completeness score is the ratio of the total actual weight of all metadata and the total of their maximal weight (from 0 to 236). Each of the steps in the wizard has its own completeness score, which is calculated in the same way as the overall score, but it considers only the metadata

from the step. Table 4 shows the maximal score of each group and its contribution to the overall score. The distribution of weights reflects the relative importance of the metadata groups, thus the two most important fragments of the metadata are the general descriptions of the component and its resources.

Table 4: Weight of metadata groups

Group	Maximal Score	Contribution to the total score	Number of rules
About	62	26.3%	32
Classification	28	11.9%	8
Status	31	13.1%	11
License	25	10.6%	6
Contacts	25	10.6%	12
Resources	40	16.9%	10
Quality	25	10.6%	4
Submission	-		7
Total	236	100%	90

There is a list of rules attached to each metadata element (the total number of rules in each metadata group is shown in the last column of Table 4). Each rule verifies a specific aspect of the metadata value, and will reduce the weight of the element if the rule is not followed.

The score of a group of metadata elements M_i and also the score of the whole component metadata is calculated as:

$$S = \sum (1 - P_{i,\min(j):A_{i,j}}) S_i$$

where S_i is the maximal score of M_i , $P_{i,j}$ is the reduction factor (penalty) from the j -th rule for M_i ; and $A_{i,j}$ is *true* if the j -th rule is applicable or effective to metadata element M_i .

When the metadata value is checked, only the first failing reduction rule is applied – this is the reason to use *min(j)* in the formula. For example, the metadata element containing component's keywords has a maximal initial weight 3. The value is evaluated by checking its rules from top to bottom, stopping at the first applicable rule. The rules for the asset's keywords are:

- **Rule 4:** If the value is empty, then the element's score is 0 (i.e. 100% reduction)
- **Rule 15:** If the value contains less than 3 keywords, then the score is 2.7 (i.e. 10% reduction)
- **Rule 19:** If the value looks like non-English text, then the score is 1.5 (i.e. 50% reduction)
- **Rule 20:** If none of the above rules are activated, then the score is 3 (i.e. no reduction at all)

Some metadata elements have only one or two rules, while other may have up to six rules. Usually, the more complex or important element is, the more sensitive it is to its content, and more rules are checked.

An example for such sensitive element is the asset description. Its rules are: rule 4 (the description is empty), rule 1 (the text is same as the default value), rule 7 (is it less than 10 characters), rule 8 (is it less than 3 words), rule 10 (is it just one sentence); rule 19 (does it sound like a text in English) and rule 20 (termination rule).

The reduction factors for the rules are not fixed, because the same rule for different metadata elements may impose different impact.

Many of the rules are general checks, like number of words, out of range dates, etc. Other rules are based on regular expressions, like rule 18 that verifies syntactically an e-mail address `/\S+@\S+\.\S+\/` or rule 16 for checking a resource URL `(http|ftp|https)://[\w-]+(\.[\w-]+)+([\w-.,@?^=%&:/~+#-]*[\w@?^=%&:/~+#-])?`.

The most complex implementation is that of Rule 19, which checks whether some text sounds like English. It is neither feasible to have a huge dictionary with English words, nor accessing external linguistic services. Instead, the wizard adopts its own statistical algorithm, which evaluates how close to English is some text (“englishability”) by evaluating three factors: variety of character pairs V_p , frequency of character pairs F_p and variety of characters V_c .

$$E = 10V_p + F_p - V_c = 10 \frac{|pairs|}{|text|} + \frac{\sum(F_{i,j} - 15)}{|text|} - \frac{\min(|text|, 26)}{|chars|}$$

where $|text|$ is the length of the text in characters, $|chars|$ is the number of distinct characters, $|pairs|$ is the number of distinct pairs of characters and $F_{i,j}$ is the frequency of specific pair of characters based on seven novels (Leon, 2008). The calculated value E is a metric for the “englishability” of the text. If $E > limit$ for some limit, then the text is considered to be in English.

The formula is shaped experimentally by testing different metadata strings, harvested from the existing component descriptions. The most influential and English-language-specific factor is the frequency component F_p . The other two factors are present to fine-tune the result, because the frequency statistics for short text does not provide adequate results.

For most elements the limit for E is set to 20, except for the keywords, which have limit 14, because they may contain non-English texts like acronyms, abbreviations, product version numbers, etc. Table 5 shows the calculated “englishability” of two English texts and three texts which are not considered as English.

Table 5: englishability of some texts

Text	englishability				English?
	$10V_p$	F_p	V_c	E	
Empty RAGE Asset	6.88	19.44	1.78	24.53	Yes
This asset is linked through a game and it produces a visual representation of player and group performance statistics.	5.04	40.76	1.30	44.51	Yes
Asdf asdf asdfasdfasdf	1.82	16.36	5.50	12.68	No
Blah-blah	3.33	3.78	2.25	4.86	No

9c8w37nroc iuerh ncs8qe	4.35	5.91	2.09	8.17	No
-------------------------	------	------	------	-------------	----

3.2.3 New metadata elements

The RAGE Metadata wizard is developed after the metadata editor. This changed some of the requirements for the wizard in respect to the metadata. Namely, new metadata were added, like short, long, general and technical descriptions, component logo, component artefacts, quality assurance self-declaration form, etc. Some of these elements were not present in the metadata model. However, the wizard has its own metadata model which is transparently translated to and from the RAGE metadata model. The new metadata elements are:

- *Component quality* – the value is a numeric score, calculated upon a self-declaration form
- *Detailed description* – a new longer description of the component
- *Promotional description* – another variant of the description, which is used in promotional materials
- *Technical description* – description with technical details about the platform, the software, the protocols, etc.
- *Commit URL* – a link to GitHub commit resource for the component, it is useful for the component developers and component users.
- *Component completeness* – the automatically calculated completeness score is stored in the metadata, so that other tools may access it directly.
- *Coding style* – a part of the self-declaration form; describes the coding style of the component software.
- *Architectural conformance* – describes the component conformance to a set of requirements and specification.
- *Software testing* – list the types of internal tests passed successfully by the component.
- *Self-declaration* – indicates whether the self-declaration form is completed.

The solution adopted in the wizard is to implement new metadata elements as *custom metadata* elements. The RAGE metadata model contains custom, user-definable elements with the sole purpose of extending the model with new metadata.

To make a seamless transition between the two metadata models, the wizard automatically converts custom metadata from the RAGE metadata model to normal metadata elements in the wizard's model and vice versa.

The only way for a user to understand that some metadata are stored as custom metadata is to view the component through the data-centric meta-editor, because it shows the metadata as it is stored. The other option is to monitor and inspect the actual XML data stream between the wizard and the server.

Except for these new metadata elements, there are other elements that the wizard supports – these are resources (artefacts), such as data files, source code, tutorials, documentation, configuration files etc. Although these resources are included in the RAGE metadata model, they are not implemented in the meta-editor in the same way as in the wizard.

The incoming metadata has two indicators of each resource – its type and its location section within the metadata model, however, sometimes these two indicators are conflicting with each other. On the other hand, the wizard supports only sections, but not the same set of sections as the metadata model. To resolve this discrepancy, the wizard translates incoming types and sections into wizard's sections. Before the component metadata are sent to the server, the wizard's sections are converted back into metadata sections.

Finally, there is yet another new metadata element, which is also a resource – this is the component logo – an image, shown along with the component title and description, both in the RAGE portal and in promotional materials. The wizard handles the component logo as a

standalone metadata element. Internally, it is represented as an image resource and is stored as one of the component's artefacts.

3.3 Comparison of both approaches

The first working version of the meta-editor is developed in early 2016. Its purpose is to edit the component metadata by exposing the full complexity of the metadata model. An internal review in 2017 revealed that component developers outside the RAGE consortium might find it difficult to use the data-centric meta-editor, as they are not familiar with the metadata mode structure, and they are not expected to be familiar with it.

The RAGE Metadata Wizard was developed in March and April 2017. Although it has the same purpose – to edit component metadata – its approach and implementation are conceptually different. Instead of being data-centric like the meta-editor, the wizard is designed as a user-centric tool. It is not possible to compare both tools in terms of which one is better and which one is worse, because they have their own specific advantages and disadvantages.

For low-level work (internal component developers and RAGE developers) the data-centric meta-editor might be more appropriate, because it shows the metadata as it is – both its contents and its hierarchy. For high-level work (external component developers and RAGE users) the user-centric wizard is more suitable, because it hides the complexity of the metadata and presents it in a comprehensive way. A side-by-side comparison of the meta-editor and the wizard is presented in the following Table 6 and Table 7.

Table 6: Comparison of metadata editors (developer's perspective)

Criteria	Data-centric Meta-editor	User-centric wizard
Implementation efforts	High	Moderate
Development time	An year	A month
Supporting and upgrading efforts	No efforts for changes in the metadata model Significant efforts for changes beyond that	Moderate efforts
Performance	Slower, data passes several transitions	Faster, processing is computationally simple
User interface	Generated in real-time	Prebuilt and fixed
Metadata verification	Rudimentary	Complete, excepts for artefacts

Table 7: Comparison of metadata editors (user's perspective)

Criteria	Data-centric Meta-editor	User-centric wizard
Target users	Asset developers and data administrators	External asset developers
Complexity of metadata presentation	Hierarchical	Flat and simplified

Filling metadata elements	Homogeneous, unguided	Heterogeneous, guided
Indicators	Only missing compulsory metadata	Detailed indicators of metadata completeness
Graphical interface	Focus on completeness and accuracy	Focus on navigation and comprehension
Viewing others' assets (read-only mode)	Same interface. All UI elements are forced into read-only mode.	Separate tool – the metadata viewer

In terms of implementation the data-centric meta-editor is much more complex with its cascading dataflow through several metadata-processing phases. The flexibility in the design allows upgrading and improvement – the meta-editor generates an editor in real time following a metadata schema. So if the metadata is modified, the meta-editor will create another editor. However, if the changes go beyond the schema, then the efforts to implement them would be significant.

The metadata wizard has a moderate complexity; its implementation is simpler and straightforward. As all metadata elements are hardcoded in the wizard, any modification, even the slightest one, should require modification of the source code.

Because of its internal complexity, the meta-editor has slightly lower performance; the user interface needs a second to be generated, because a number schema files and taxonomies are downloaded over the internet. The tool performs only rudimentary metadata verification, relying on the server to run a complete verification.

The wizard, on the other hand, is faster, its interface is prebuilt, and it performs complete verification of the metadata in real-time. The only exceptions are the artefacts – the wizard expects the RAGE server to signal back problems with the uploaded resources, such as duplicate names, unsupported file types, etc.

In terms of users' experience, the meta-editor and the wizard are also quite different. The target users of the meta-editor are internal asset developers and data administrators, who are familiar with the metadata model. These users can work with the full complexity of the metadata hierarchy, including several levels of metadata blocks nesting. The wizard is more appropriate to external asset developers, who are not expected to know the details of the metadata model. This is because the wizard shows a flat oversimplified list of elements.

In terms of the user interface, the RAGE meta-editor indicates only missing compulsory data, while the wizard shows detailed indicators of metadata completeness, accompanied by hints and suggestions. Additionally, the wizard's interface is focused on easier navigation and guided data entry.

4 COMPLEMENTARY SOFTWARE COMPONENT TOOLS

4.1 *Taxonomy tools*

Taxonomy is a science of classification of objects based on their properties and relations. Taxonomies are represented as hierarchies of concepts. Apart from classifications, taxonomies may represent controlled vocabularies.

The classification of software components is implemented as describing their various aspects via concepts from taxonomies. Within RAGE, taxonomies are used as a source of classification values from preselected hierarchies of concepts as well as values from predefined controlled vocabularies. For example, some of the metadata elements are bound to taxonomies and their values must be concepts from these taxonomies.

In RAGE, several tools have been built to manage the taxonomies:

- *Taxonomy viewer* – used to show a taxonomy and navigate in it.
- *Taxonomy selector* – used to select one or more concepts from a taxonomy.
- *Taxonomy editor* – used to modify the concepts and the hierarchal structure in a taxonomy.

These three tools are not implemented as separate applications. They are software modules, which are to be embedded in a web page for on-line or off-line taxonomy management. The tools share common code and appearance. The communication between the user software and the taxonomy tools is via a simple API in JavaScript.

The overall design principal of the taxonomy tools are a minimalistic user interface and set of features, a consistent visual layout, multiplatform support through JS/HTML5 and multilingual interface and content, if present.

4.1.1 *RAGE Taxonomy Viewer*

The Taxonomy Viewer is a tool for browsing and viewing a taxonomy. It is the simplest tool of the set of taxonomy tools. The Taxonomy Selector and Taxonomy Editor could also be used to browse a taxonomy, but the Taxonomy Viewer is safer to be used, because all program code and visual interface for taxonomy modification is removed.

The tool presents a taxonomy in several layouts, several scales and several languages if the taxonomy concepts are multilingual.

Figure 8 shows the default view of the Taxonomy Selector, when embedded in a web page. The name of the taxonomy is used as a root for the taxonomy tree. The individual concepts are arranged and connected depending on the taxonomy hierarchy. Concept nodes, which have narrower concepts sub-nodes, could be collapsed or expanded.

The bottom right part of the tool contains buttons for changing the visual layout of the taxonomy and the language of the concepts.

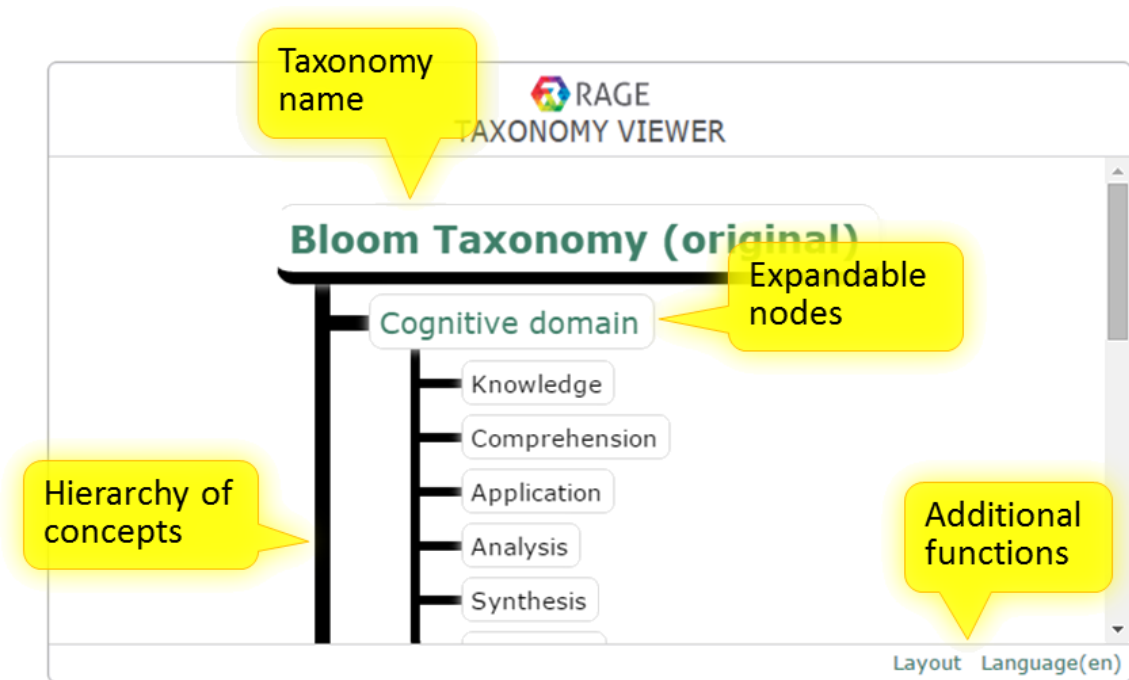


Figure 8: Embedded Taxonomy Viewer

The navigation in larger taxonomies is helped by collapsing and expanding concept nodes. Figure 9 represents the original Bloom taxonomy default layout of the Taxonomy Viewer with collapsed concept nodes (left), with one expanded node (middle) and all expanded nodes (right).

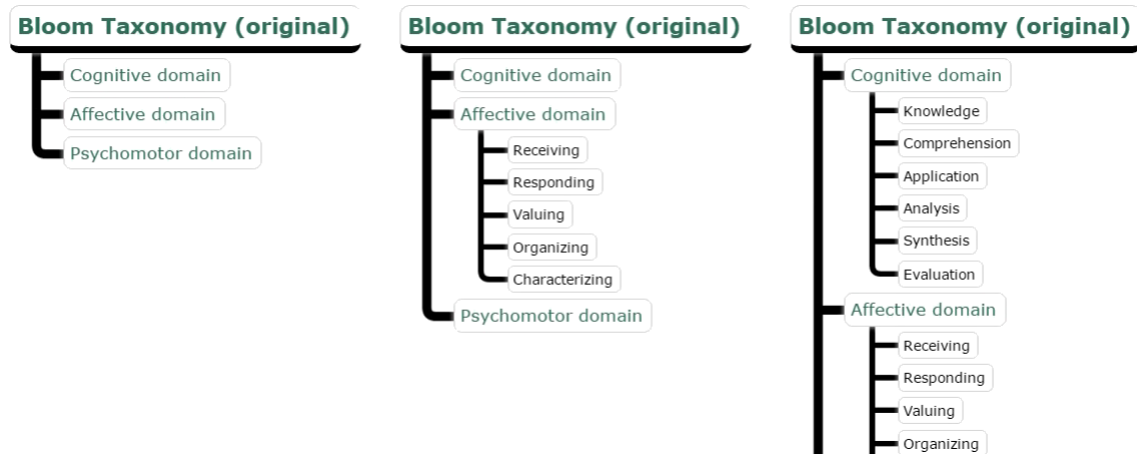


Figure 9: Collapsed and expended concept nodes

In addition to the default representation of hierarchy, the selector may also arrange concept nodes in horizontal or vertical tree. The same Bloom taxonomy from Figure 9 is shown in Figure 10 as horizontal and vertical layout.

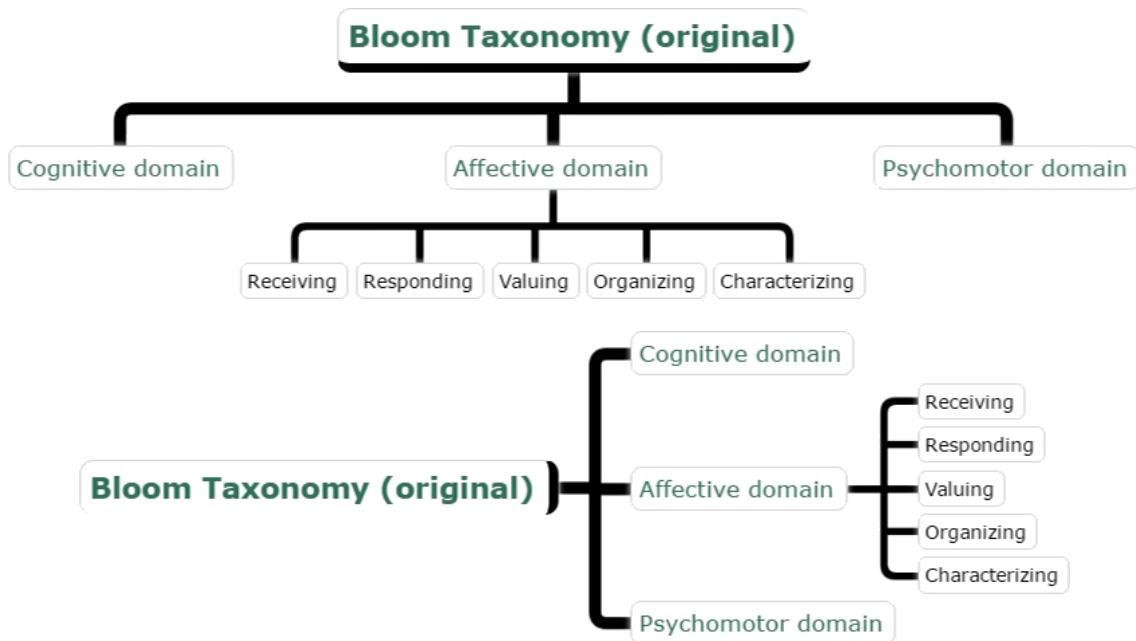


Figure 10: Horizontal and vertical layout of taxonomy concepts

If a taxonomy has multilingual support, i.e. concept nodes contain additional information about their translations in various languages, then the Taxonomy Selector will allow switching between different languages. By default the English translation is shown. Figure 11 demonstrates the Bloom taxonomy in Bulgarian. The list of supported languages is not hardcoded in the Taxonomy Viewer – it extracts the available languages from the taxonomy.

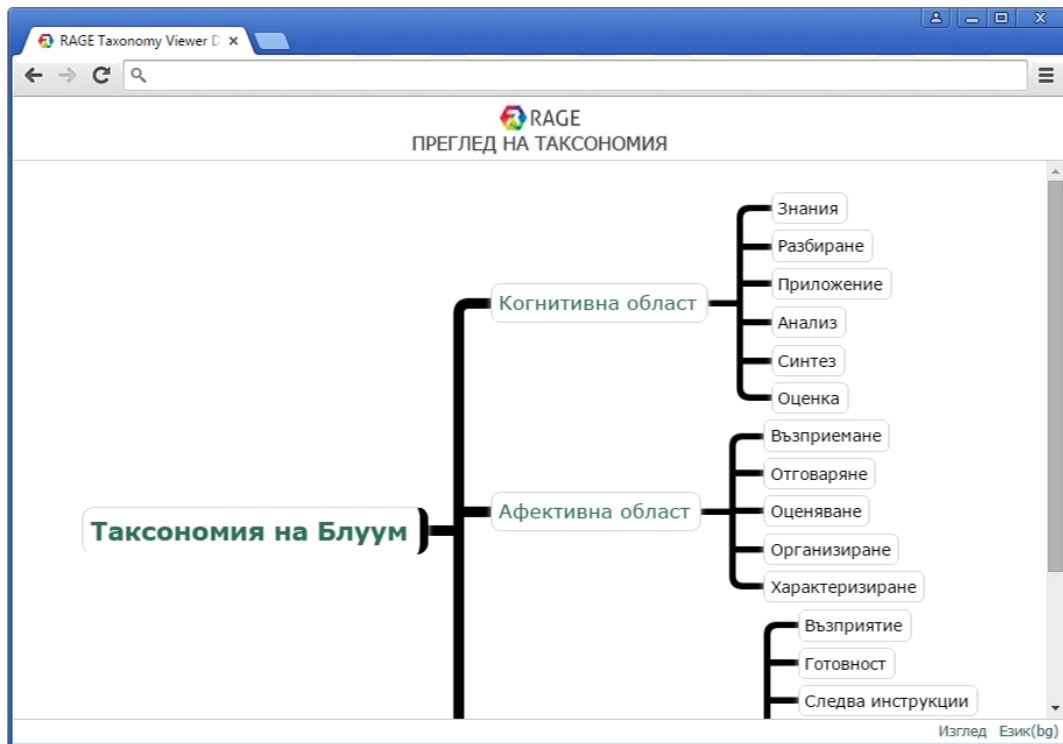


Figure 11: Multilingual support in taxonomies

4.1.2 RAGE Taxonomy Selector

The Taxonomy Selector is a tool for selecting and picking taxonomy concepts from a taxonomy. It has a similar visual appearance as the Taxonomy Viewer, but has additional interface elements. Figure 12 demonstrates the general view of the Taxonomy Selector.

There are two new buttons “Done” and “Cancel” used for confirmation or cancelation of the current concept selection. Both buttons are programmable, i.e. the user of the tool may define a custom behaviour for each of the buttons, depending on the scenario where the tool is used.

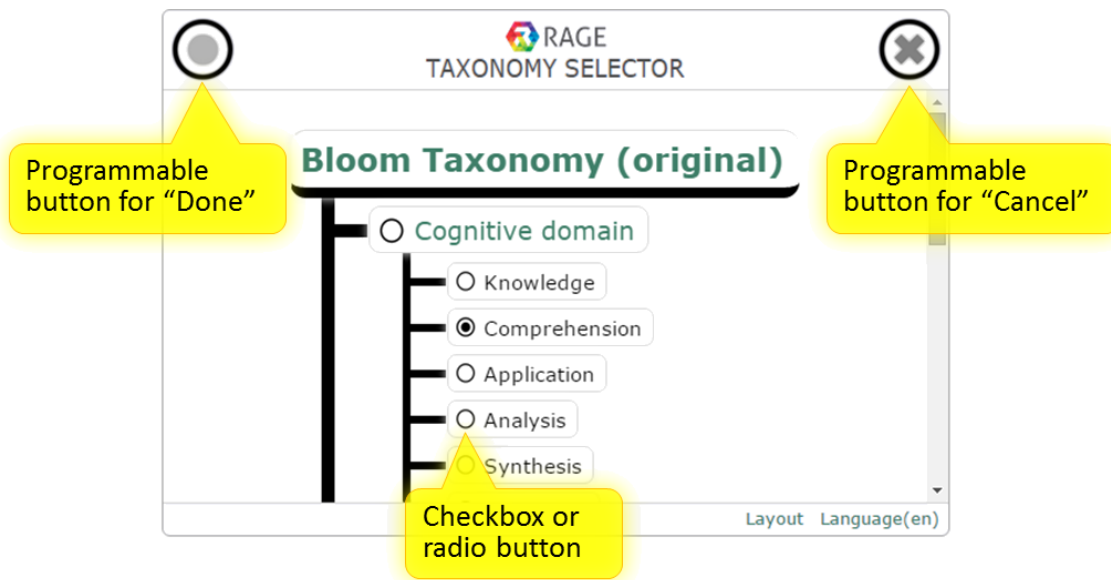


Figure 12: The RAGE Taxonomy Selector

The Taxonomy Selector is configurable in several ways. The user may set what concepts are selectable – all concepts in the taxonomy, all concept except the root concept, all end-point concepts, etc. The choice what concepts are selectable depends on the user intentions and the context.

Consider the *ACM Computing Classification System of Applied computing Taxonomy*, part of which is shown in Figures 13 and 14. In one context it might be desired to select only few end-node concept such as *Business Intelligence*, *Business Process Modelling* and *Business Process Monitoring*. In another context it might be desired to allow the selection of higher-level concepts, like *Business Process Management* or even the top-level *Enterprise Computing* – Figure 14.

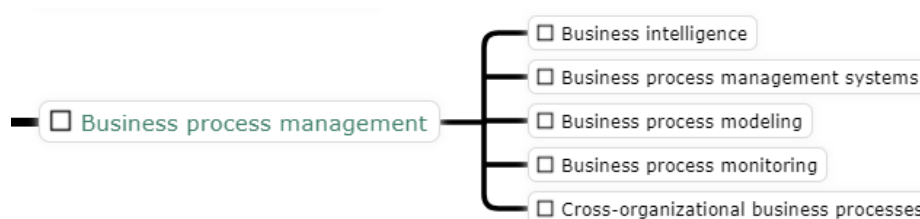


Figure 13: End-nodes and top-levels

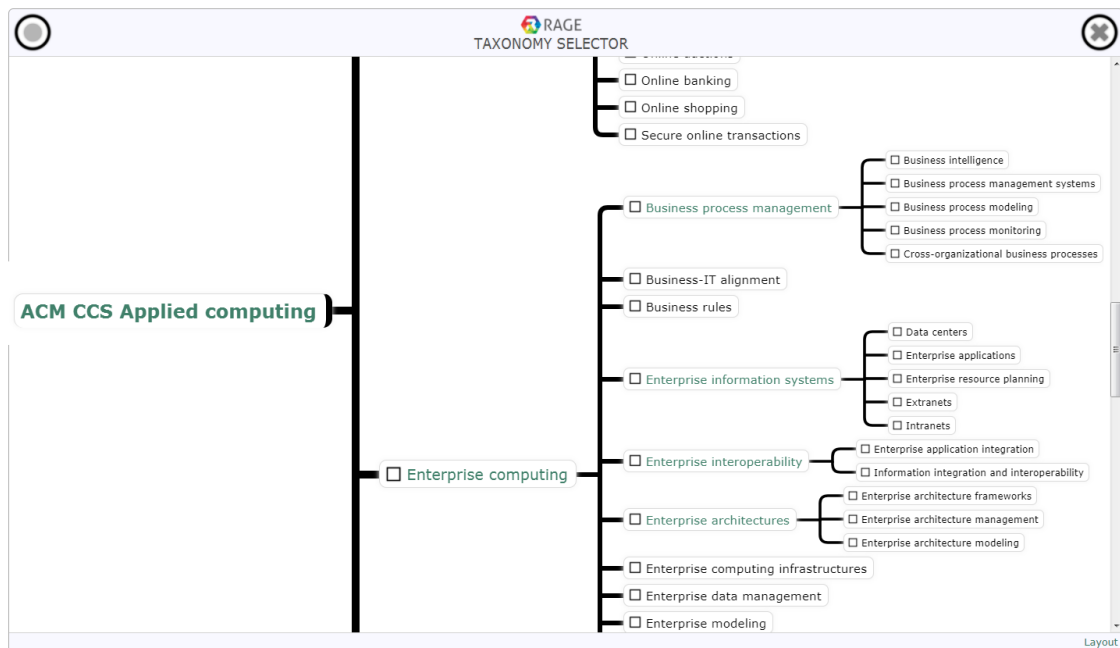


Figure 14: Complex taxonomy in the Taxonomy Selector

Another customization of the Taxonomy Selector is whether a single concept or multiple concepts could be selected. This choice affects the visual representation of the tree, as shown in Figure 15. For the single selection, each concept has a radio button, for the multiple selection it is a check box.

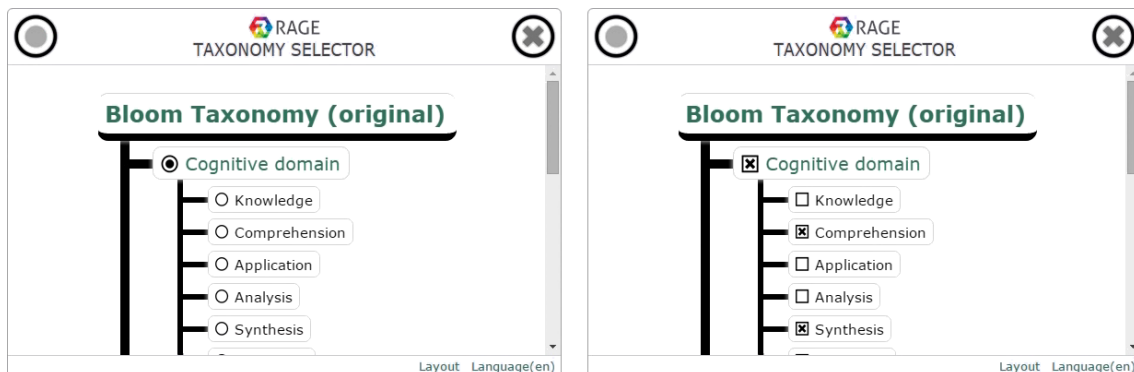


Figure 15: Single or multiple concept selection

The RAGE Taxonomy Selector is embedded in the RAGE Metadata Editor. It is used as a component for providing values for taxonomy-based and controlled-vocabulary-based metadata elements. Figure 16 represent the overall architecture of the meta-editor, which creates a metadata editor in real time and embeds in it the Taxonomy Selector. The metadata editor serves as a server to the selector:

- It provides the taxonomies (retrieved via the Software Component Services)
- It provides the current user selection of taxonomies and selected concept in them
- It accepts the new user selection and stores it in the metadata

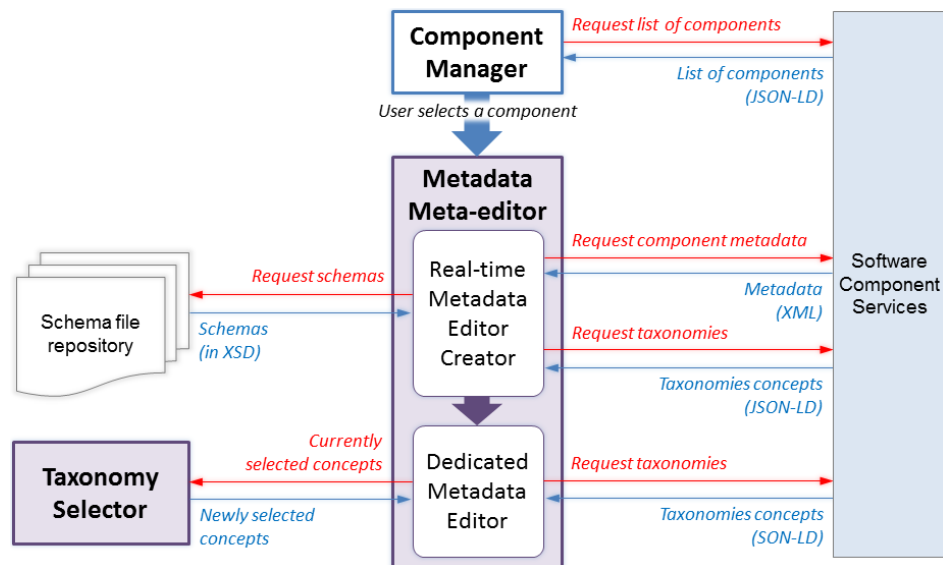


Figure 16: Architecture of embedding a Taxonomy Selector

If the taxonomy has multilingual support, the Taxonomy Selector may use any of the provided languages. However, the communication with the metadata editor transfers only the taxonomical identifiers, which are language independent.

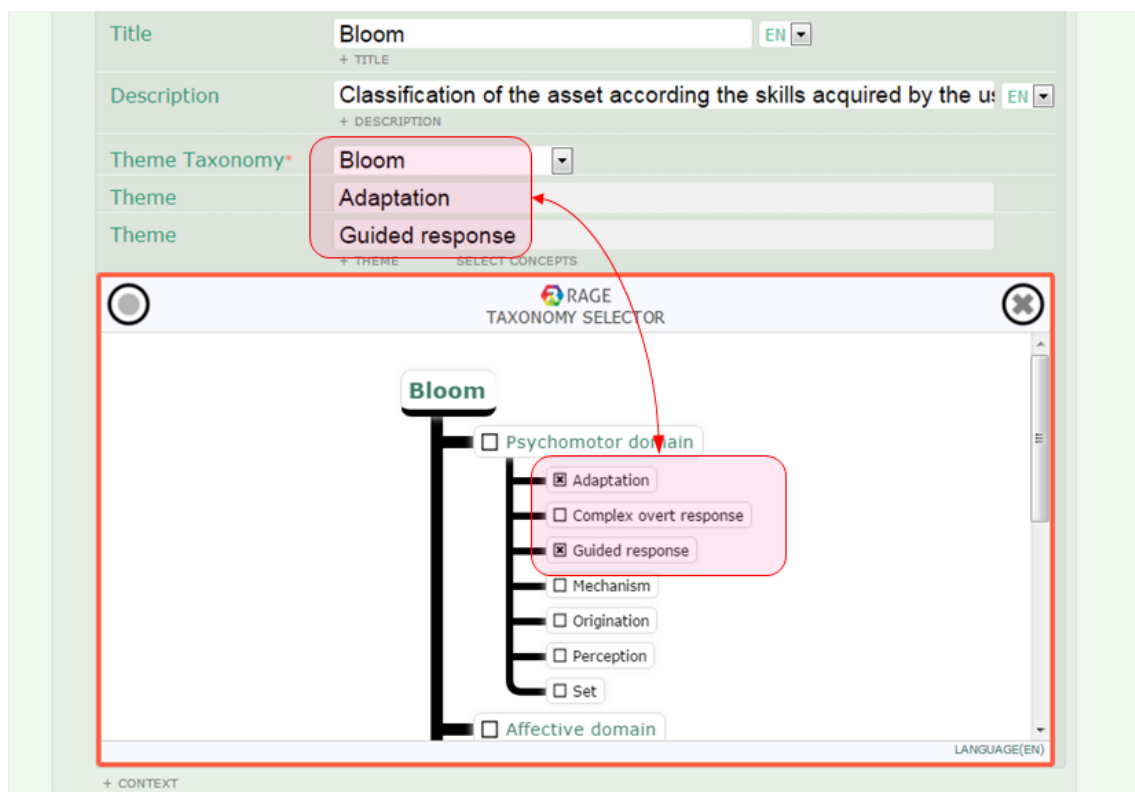


Figure 17: Taxonomy Selector embedded in a Metadata Editor

Figure 17 demonstrates the actual embedding of the Taxonomy Selector in a Metadata Editor, which is generated in real-time by the RAGE Metadata Meta-editor. The embedding is both programmatic, as the Taxonomy Selector code is included in the metadata editor, and graphical, as the Taxonomy Selector interface is inserted in the metadata editor's interface.

4.1.3 RAGE Taxonomy Editor

The last taxonomy tool is the Taxonomy Editor, which is used to create and modify taxonomies. It provides functionality for content changes (e.g. editing concept names and adding translations in different languages) and for structural changes (e.g. adding, reallocating and removing branches of the hierarchy tree).

Figure 18 shows a snapshot of the Taxonomy Editor during the moving of a concept node. Each node has a six-dots handler, which can be grabbed, dragged and dropped.

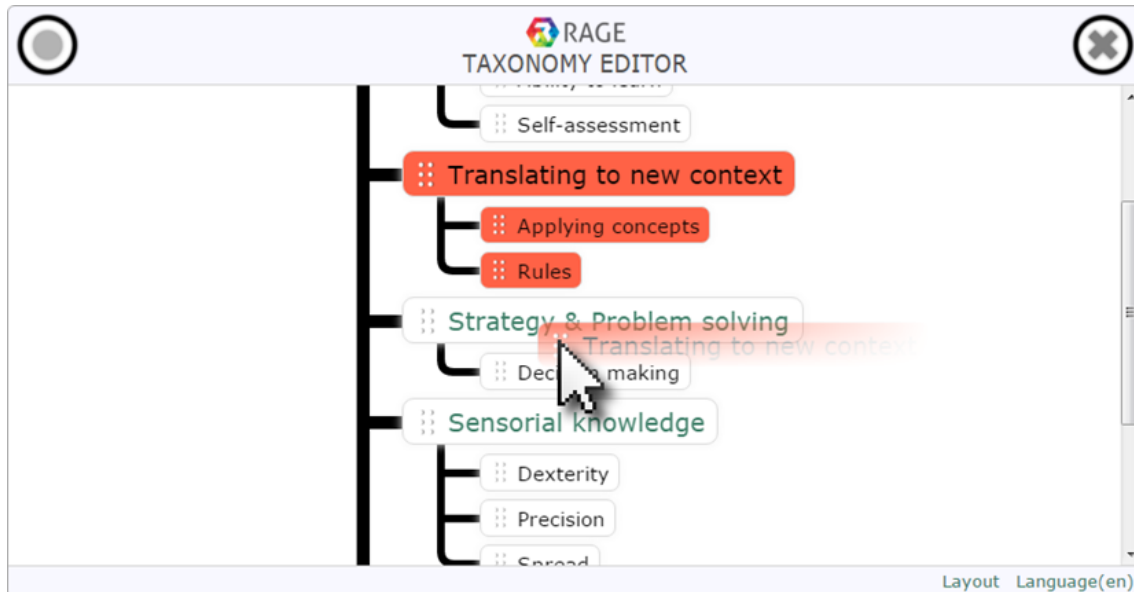


Figure 18: Moving a node in the Taxonomy editor

The structural changes allow intuitive modification of the hierarchy tree. Figure 19 demonstrates four exemplary steps. From left to right they are: moving concept A2 and all its child concepts one level deeper, so it itself becomes a child concept of A1; then moving concept A2-b one level up, so it becomes a sibling concept of A2; and finally, moving A1 at the top level.

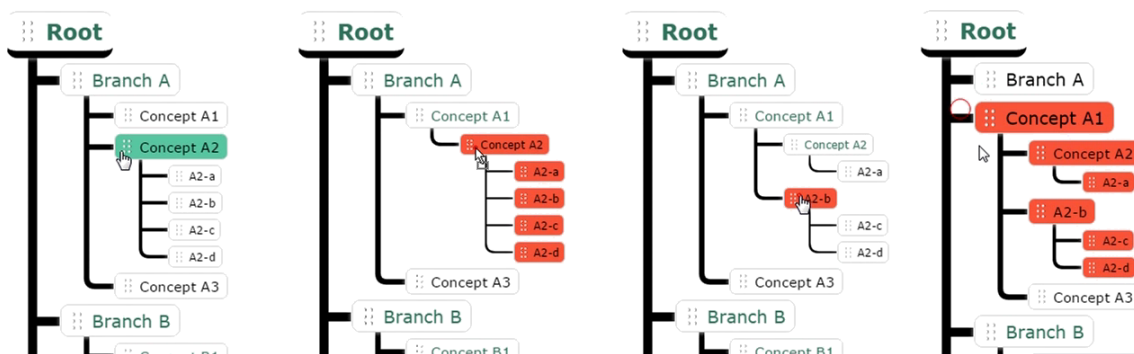


Figure 19: Structural changes in a taxonomy

The creation of new taxonomy concepts activates in-place editor of labels. Figure 20 illustrates the creation of three new nodes – A1-a, A1-b and A1-c; and moving them as child nodes of A1. The translation of the new concepts is done by switching the language. When a specific language is selected, the Taxonomy Editor shows the concept labels in this language. When a translation is not available, the Editor shows the label in English prefixed by the 2-letter ISO 639-1 language

code. For example, the concepts in Figure 21 are still not translated into Dutch, so the Taxonomy Editor shows the names in English with the prefix *nl:*.

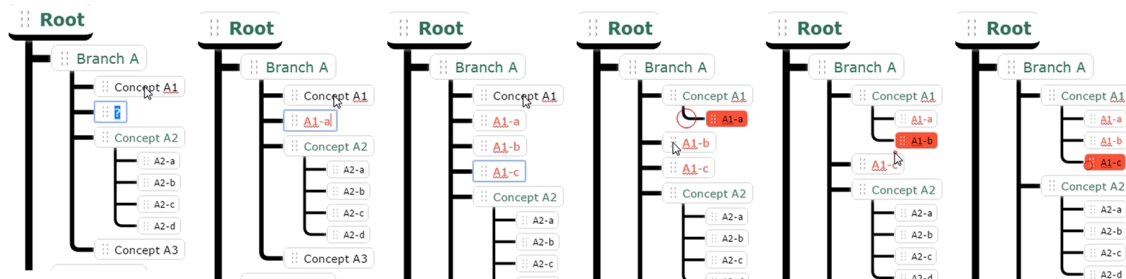


Figure 20: Creating new taxonomy concepts

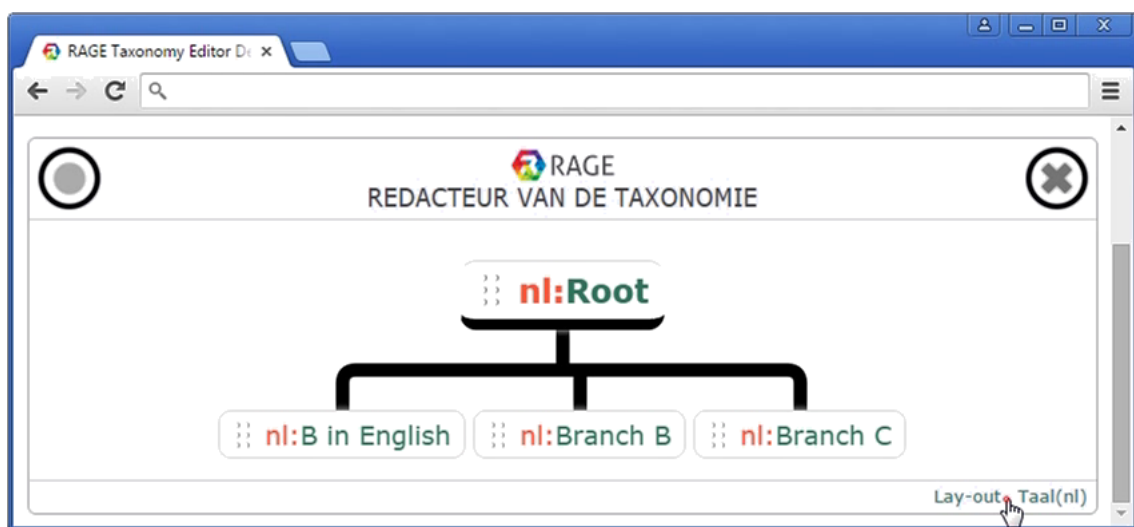


Figure 21: Untranslated taxonomy concepts

The Taxonomy Editor can be switched into advanced mode, which reveals additional functionality for the user – Figure 22. The new elements are:

- The taxonomy resource identifier is shown below the taxonomy name.
- The identifiers of individual concepts are shown below their names.
- The ID button resets/regenerates all concept identifiers.
- The JSON button exports the taxonomy in a computer-readable JSON-LD format.
- The TXT button exports the taxonomy in a human-readable text format.

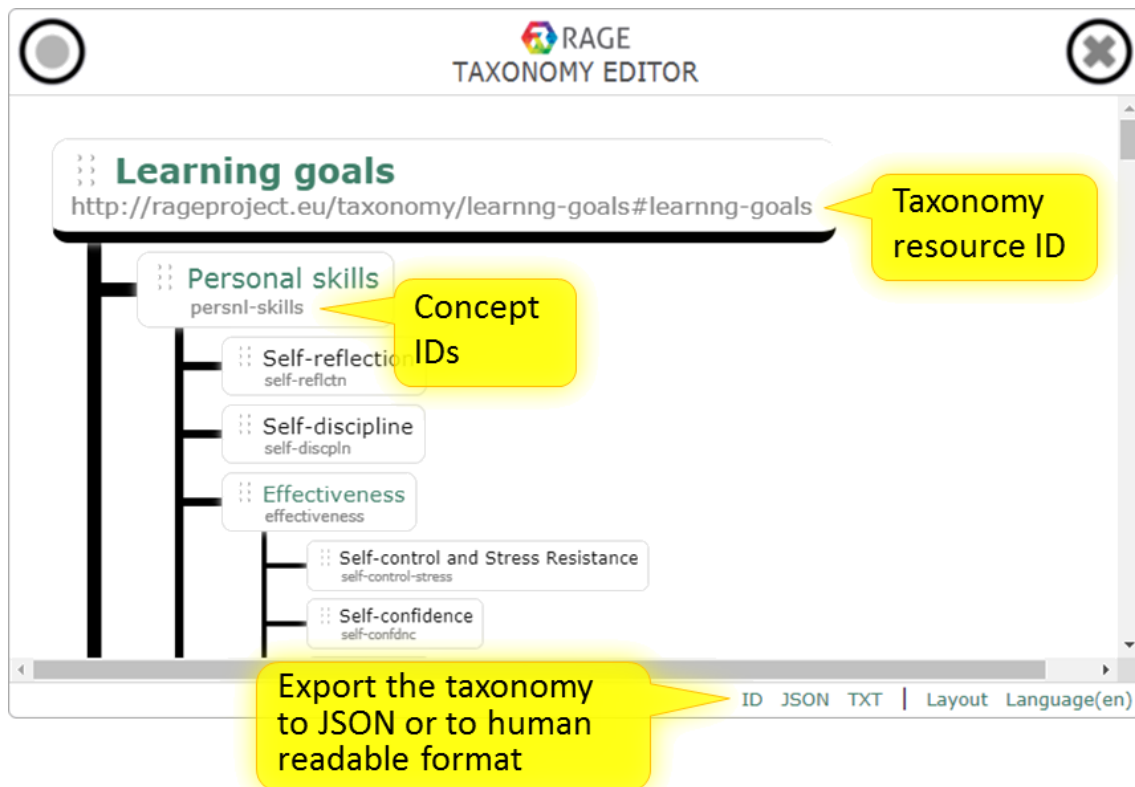


Figure 22: The Taxonomy Editor in advanced mode

The generation of concept identifiers is done automatically by an algorithm, which preserves their uniqueness within the taxonomy, generates identifiers that resemble the names, but are shorter than them. The algorithm prefers to strip out the trailing words of a concept name, if it is too long, as well as the trailing vowels of long words; and some inessential words like *and*, *of*, *the*. Some examples of generated identifiers are listed in Table 8.

Table 8: Examples of generated concept identifiers

Concept full name	Concept identifier
Principles	#principles
Learning goals	#learnng-goals
Self-reflection	#self-reflectn
Self-control and Stress Resistance	#self-control-stress
Knowledge of foreign languages	#knowldg-forgn-langgs
Recall and memorization (gaming)	#recll-memrztg-gamng
Design, preparation, anticipation, collaboration, hypothesis and assembly of simulations	#desgn-preprtn-anticptn

The identifiers are not visible for the RAGE users, they are used internally and the only formal requirement is to provide uniqueness. The extra processing work to generate these condensed identifiers is to help the system developers when they inspect manually the metadata in its native format. For example, it is much easier to understand a metadata reference like this:

<http://rageproject.eu/taxonomy/learnng-goals#plannng-organzng>

than a reference using UUID (universally unique identifier):

<http://rageproject.eu/taxonomy/learnng-goals#cab9da94-7d41-42fc-83fd-db1337de1988>

For safety reasons the Taxonomy Editor does not save modified taxonomies directly to the RAGE repository. Instead it saves them in files, which are then imported. The JSON-LD file format is used to represent the taxonomy in a way that can be imported in the repository. For internal management a taxonomy could be represented in a text format, which is suitable for manual inspections by humans. Here is a fragment of the *Learning Goals* taxonomy exported as text (the *en:* prefix indicates there is only an English translation of the concepts):

```
Taxonomy: http://rageproject.eu/taxonomy/learnng-goals#
en:Learning goals [#learnng-goals]
  en:Personal skills [#persnl-skills]
    en:Self-reflection [#self-reflctn]
    en:Self-discipline [#self-discpln]
    en:Effectiveness [#effectiveness]
      en:Self-control and Stress Resistance [#self-control-stress]
      en:Self-confidence [#self-confdnc]
      en:Flexibility [#flexibility]
      en:Knowledge of foreign languages [#knowldg-forgn-langgs]
    en:Creativity [#creativity]
    en:Research skills [#resrch-skills]
    en:Make decisions [#make-decsns]
    en:Achievement [#achievement]
      en:Problem solving [#problem-solvng]
        en:Critical thinking [#critcl-thinkng]
        en:Planning and organizing [#plannng-organzng]
        en:Reviewing and evaluating [#revwng-evaltn]
        en:Planning and organization [#plannng-organztn]
        en:Initiative and proactiveness [#inittv-proactvnss]
      en:Autonomy [#autonomy]
      en:Achievement orientation, efficiency [#achievmnt-orienttn-fficnc]
      en:Concern for order, quality and accuracy [#concrn-order-qualt]
      en:Information exploring and managing [#informtn-explorng-mangng]
  :
  :
  :
en:Other skills [#other-skills]
  en:Verbal Information [#verbal-informtn]
    en:Facts of knowledge [#facts-knowldg]
  en:Attitude [#attitude]
    en:Actions that a person chooses to complete [#actions-that-person]
  en:Motor Skills [#motor-skills]
    en:Behavioural physical skills [#behvrl-physcl-skills]
```

Exported: Sun Dec 30 2018 15:00:35 GMT+0200 (Eastern European Standard Time)

4.2 Web Portal Tools

4.2.1 RAGE Repository Manager

The initial use of the RAGE Metadata Editor needed some functionality to list and filter software components, to pick any of them and modify it. The design is focused only on the most important activities, which are necessary for the internal management of the RAGE repository at a component level:

- Listing all software components with titles, descriptions and logo.
- Searching / filtering software components by free-text keywords.
- Filtering only private components, created by the user.
- Selecting components for eventual modification, duplication or deleting.

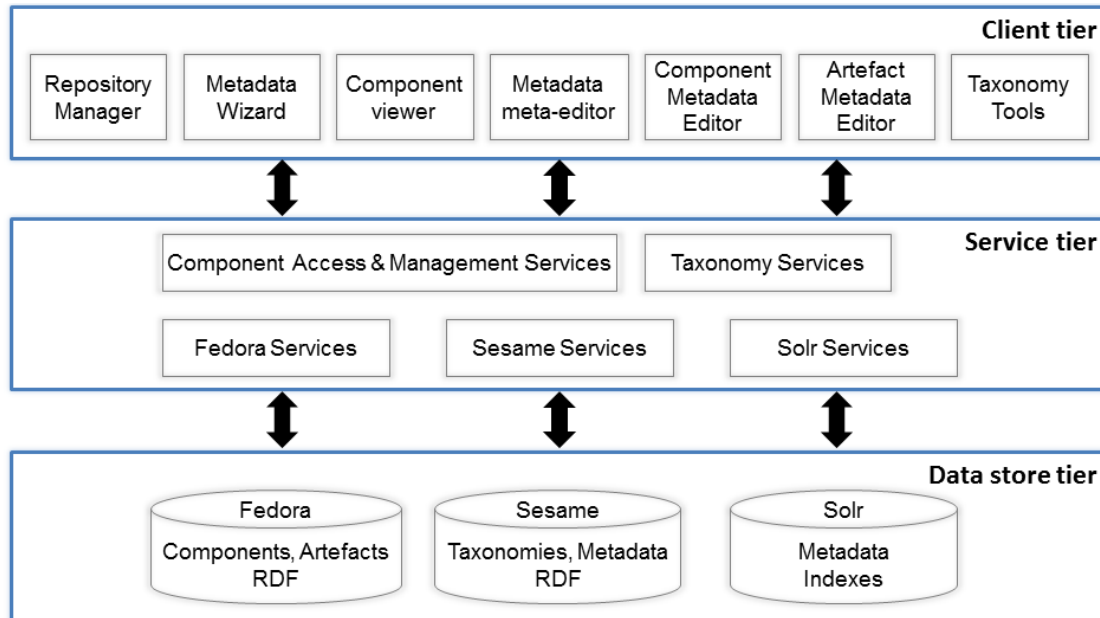


Figure 23: Repository architecture

To support the Repository Manager (as well as all other repository tools), the RAGE repository infrastructure is composed of three tiers, as shown in Figure 23: client, service and data store tiers.

- Fedora stores the software components, artefacts and metadata using RDF. Fedora is flexible, well established, scalable and durable. Fedora provides a general RESTful HTTP API for accessing repository resources through HTTP methods. It supports OAI-PMH requests on component content and metadata.
- Sesame provides efficient storage and expressive querying of large quantities of metadata in RDF and RDF Schema, as well as classification taxonomies/vocabularies. Sesame offers a RESTful HTTP interface supporting the SPARQL Protocol for RDF.
- Solr major features are full-text search, sophisticated faceted search, almost real-time indexing and dynamic clustering of data. It is used for creating full text indexes on the RAGE metadata fields and full-text and faceted searching.

The functioning of the tools in the client tier are also supported by additional services from the service tier: *Component Access Services* for retrieving asset packages and metadata, full-text and semantic searching and browsing for assets, *Component Management Services* for an abstract level of administering components, e.g. creating, modifying, and deleting; and *Taxonomy Services* for managing classification taxonomies and controlled vocabularies.

A first implementation of the RAGE Repository Manager was bound to the RAGE Metadata Editor and Artefact Editor created by the meta-editor (a workflow diagram is presented and explained in Figure 27, chapter 4.2.3). The second and currently active implementation of the RAGE Repository Manager is bound to the RAGE Metadata Wizard and it provides a significantly simpler workflow, shown in Figure 24:

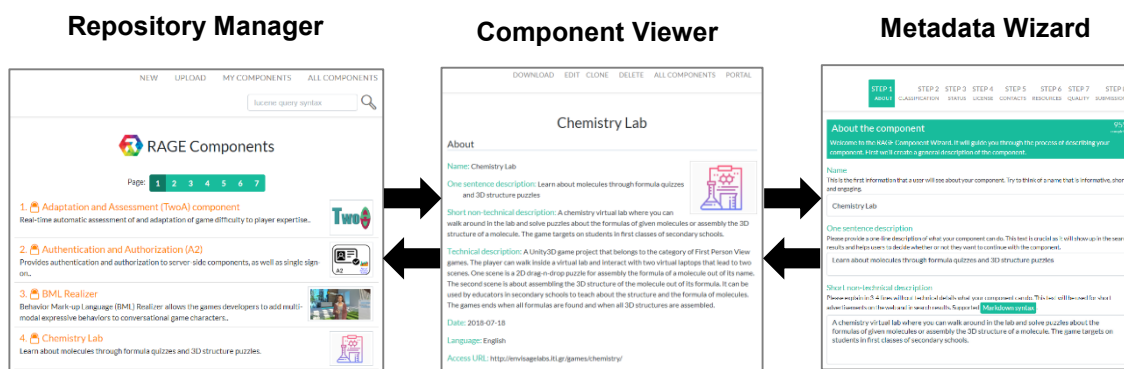


Figure 24: Repository Manager workflow

4.2.2 RAGE Component Viewer

When users access the metadata of software assets developed by others, they are not allowed to modify them. The metadata editors created by the meta-editor use a single interface – when data modifications are forbidden the Save button is removed from the web form and all data entry elements are set to read-only mode and are greyed out to visually indicate their disabled mode.

While the wizard assists at entering the asset's metadata, the interface is not suited for the case of users just wanting to view the component metadata. To address this, a separate tool was developed – the metadata viewer.

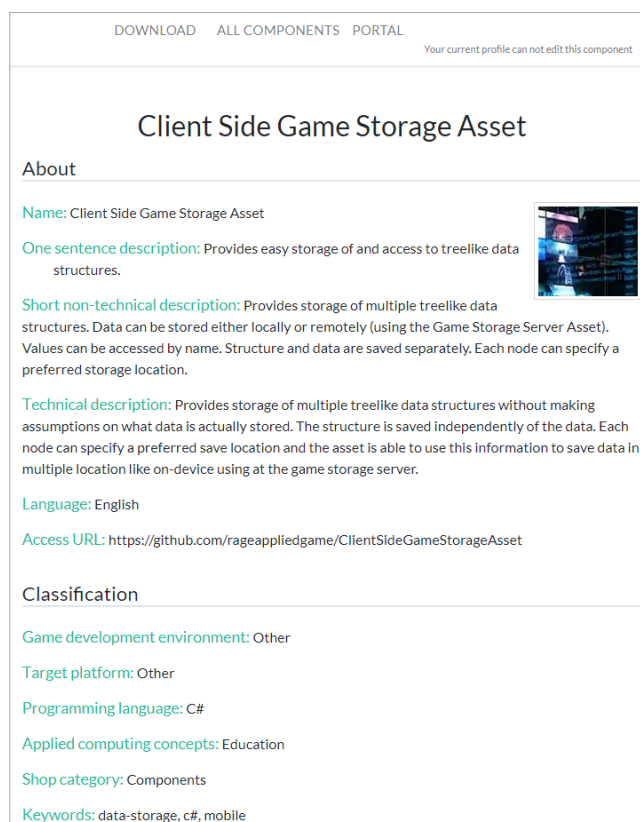


Figure 25: The RAGE Metadata Viewer

The viewer extracts the metadata of a software component and arranges them in a structured page as shown in Figure 25. This interface is suited for providing a quick overview of a component, because its metadata are presented in a more compact style; usually they fit in one or two screen pages. Additionally, the interface is printer-friendly and the component description could be printed as a hard-copy of the component dossier.

The metadata viewer shows only the metadata that are processed by the wizard. To see the full metadata, the user must use the meta-editor.

Although the purpose of the viewer is to show the metadata, it is also the intermediate layer between the front-end Repository Manager and the Metadata Wizard – see Figure 26. Component users browse and search all components in the repository within the Repository Manager. When they click on a selected component, it is opened in the Component Viewer.

Then the users can inspect the description and download it if they like to incorporate it in their game. However, if the users are the developers of this component or if they have sufficient writing permissions then they can further open it in the Metadata Wizard to edit it. When they have finished editing the asset, they automatically return to the Component Viewer to review the changes.

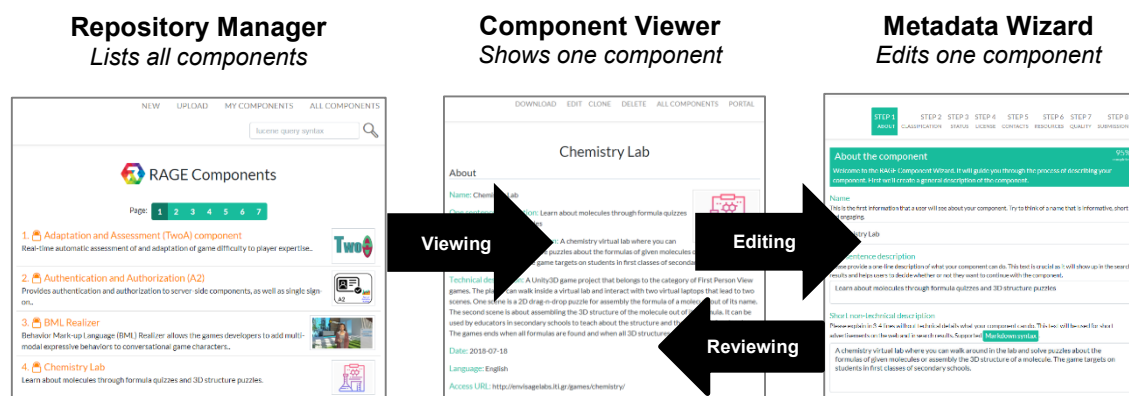


Figure 26: The viewer as an intermediate layer between the manager and the wizard

4.2.3 RAGE Artefact Manager

The RAGE Metadata Editor provides functionality to edit the metadata of RAGE software components. The component also contains additional files, called artefacts. The metadata model includes metadata describing the artefacts, but the contents of the artefacts (software code, documentation, data files, images, etc.) are not considered metadata and they are stored as separate entities from the metadata.

To facilitate full modification of a software component, the meta-editor is supported by another tool – the Artefact manager. It is used to define and modify the artefacts in a software component. Figure 27 demonstrates the workflow of the Artefact manager (the lower sequence). It is activated by the Repository Manager. It shows a list of artefacts related to a specific software component. The user may delete artefacts, add new artefacts or modify the metadata, specific to each artefact.

The metadata modification activates the RAGE Metadata Meta-editor with schemas and definitions of the artefact (rather than the whole software component). The meta-editor generates a metadata editor exclusively for the artefact metadata. The user modifies the metadata via this editor.

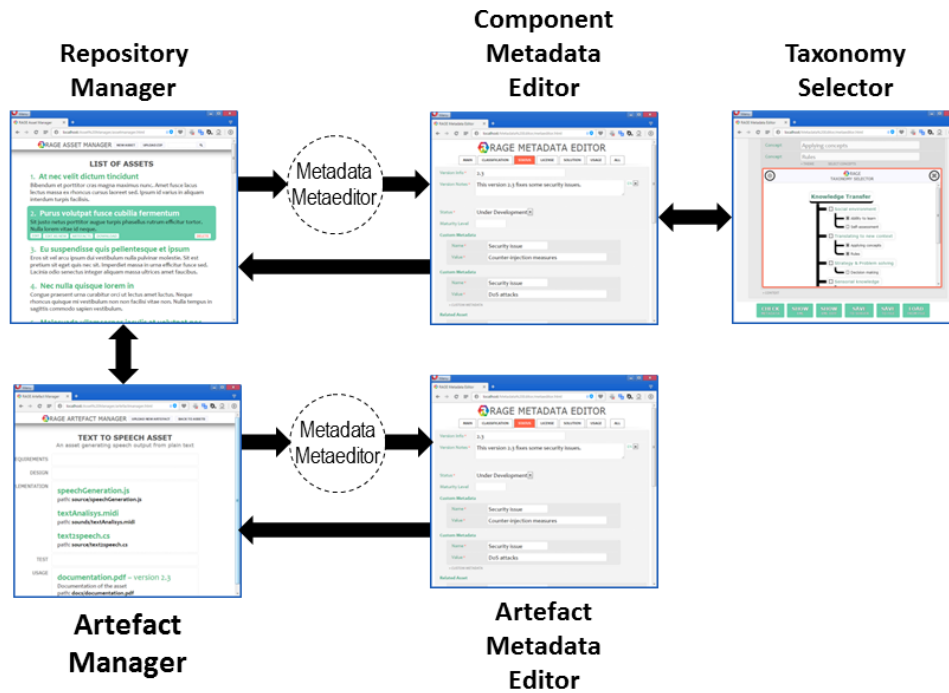


Figure 27: Artefact Manager Workflow

Figure 27 also shows the workflow when component metadata are edited (the upper sequence). The meta-editor in both sequences is the same, but the generated metadata editors are different.

The Artefact Manager is functionally coupled with the RAGE Metadata Editor and is used in connection with it. The Artefact Manager cannot be used with the RAGE Metadata Wizard, as long as the Wizard manages artefacts by itself and does not require an external tool for this.

5 CONFIGURATION AUTHORING TOOLS

5.1 Goals and design

The RAGE software components are supposed to be usable in different games. This leads to the requirement that components should be integrated in different gaming environments. Figure 28 shows a scenario where one RAGE software component (on the left) is being used in three different games (on the right), which require different configurations of the component.

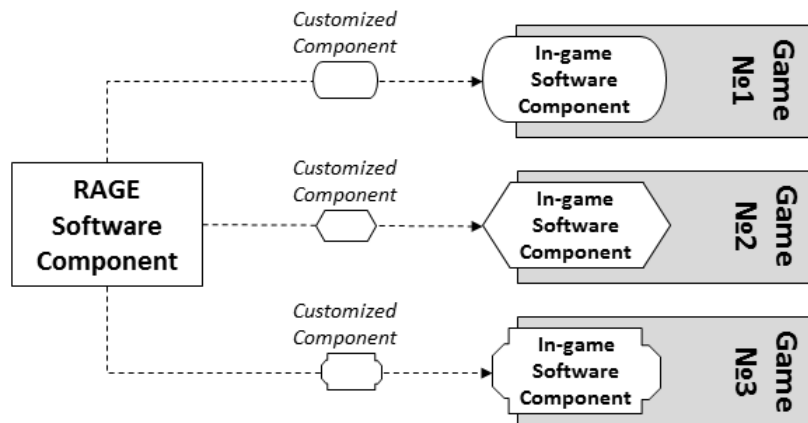


Figure 28: The in-game use of RAGE software components

Instead of providing a separate RAGE software component for each envisioned environment, component developers provide components as general software modules – i.e. one package for all types of supported games. Once a user downloads a software component, it must be configured to the specifics of the hosting game. Figure 29 shows the production of configured components. The three games use the same component, but are configured differently. This is done by providing configuration parameters to the software component, for instance with values that control its appearance, its behaviour or its data resources.

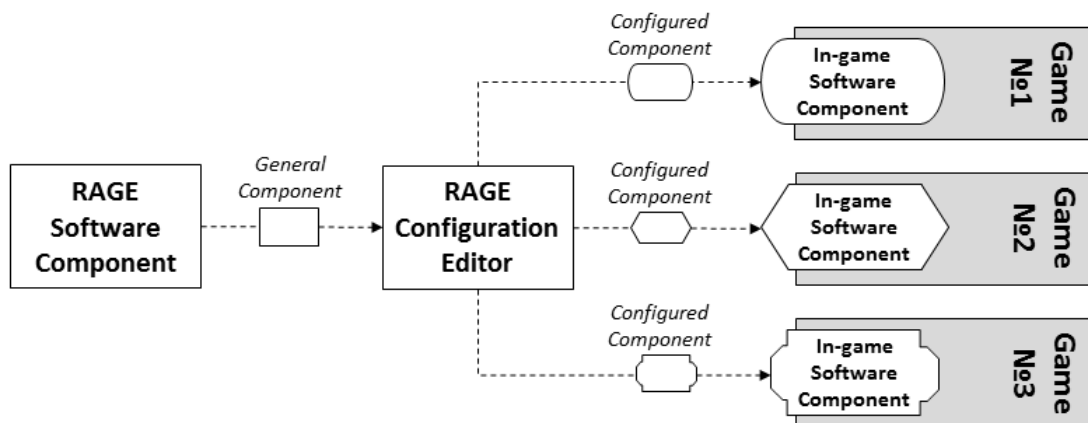


Figure 29: Configured software components

The possibility to configure a software component is functionality that is present in the component itself. The most straightforward and traditional approach is to store the configuration parameters in a file, which is then read and processed by the component. The files are created by a configuration editor, which produces XML or JSON configuration files. The role of this editor is to provide a user interface for easier definition of parameters.

Although this approach provides flexibility in term of using individual configuration wizards, it does not address the issue of simplifying component customization. The nature of each software component is unique and the configuration parameters of one component are inapplicable to another component. It is not feasible to manually build a unique configuration editor for each software component, as the number of components raises.

RAGE addresses this problem by providing a configuration authoring tool, which constructs (authors) one or more specific configuration editors tailored for each software component. This authoring tool is called the *RAGE Configuration Editor Wizard* – Figure 30.

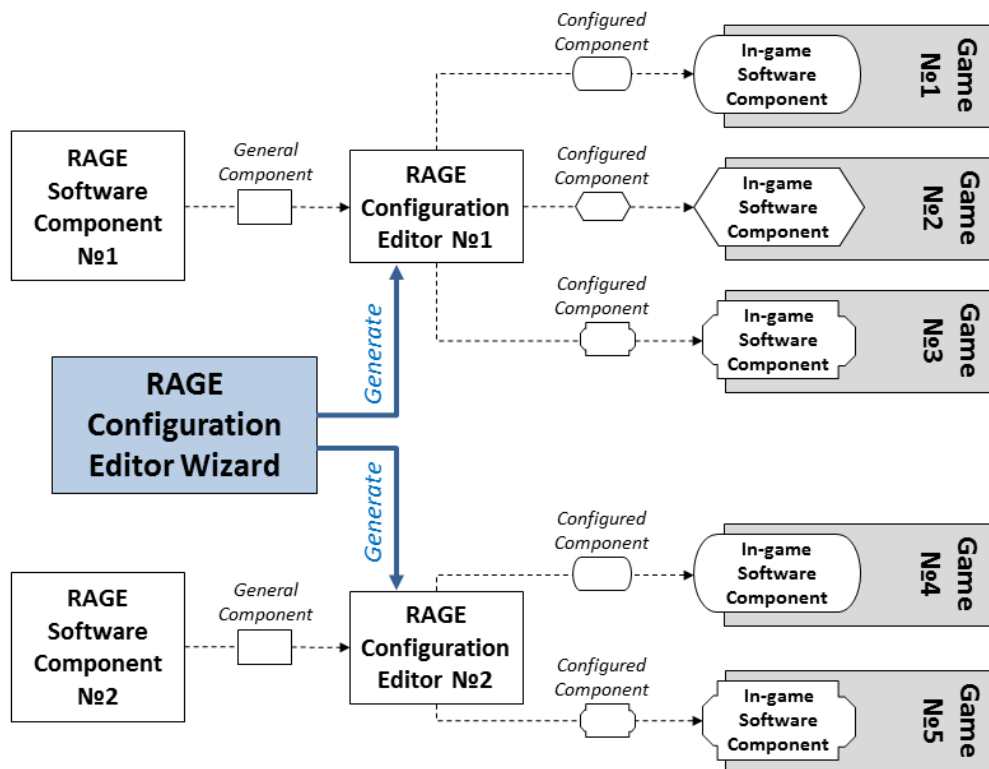


Figure 30: RAGE Configuration Editor Wizard

The major goal of the configuration wizard is to generate configuration editors. This renders the wizard as a meta-editor – a tool for authoring editors.

The design principles of the Configuration Editor Wizard are:

- This authoring tool is one tool, covering all RAGE software components, including the ones which have not been created so far.
- The authoring tool is provided as a single file, which does not require installation or configuration.
- The authoring tool can be used either off-line or on-line, on different platforms or operating systems.
- The authoring tool generates configuration editors, which are specific to a given software component. These editors create unified configuration files in a standard format, which can be programmatically processed by software components, or could be manually edited by users.
- The configuration editors created by the authoring tool are also single files, which can be used either off-line or on-line for the software component concerned.

5.2 Implementation details

5.2.1 Preliminary prototype

The first prototype of the Configuration Editor Wizard does not create separate configuration editors. Instead it uses the RAGE Metadata Editor and its meta-editing functionality. Initially, the metadata meta-editor is used to modify the metadata of software components and their artefacts. It generates a dedicated metadata editor, which is based on predefined schema definition files, describing the structure of the metadata record (i.e. the RAGE metadata model). The same core functionality can be used to transform it into a configuration editor.

The left side part of Figure 31 is a snapshot of the visual appearance of some generated Configuration Editor. On the right, is a snapshot from the XML file with configuration metadata, generated by the editor.

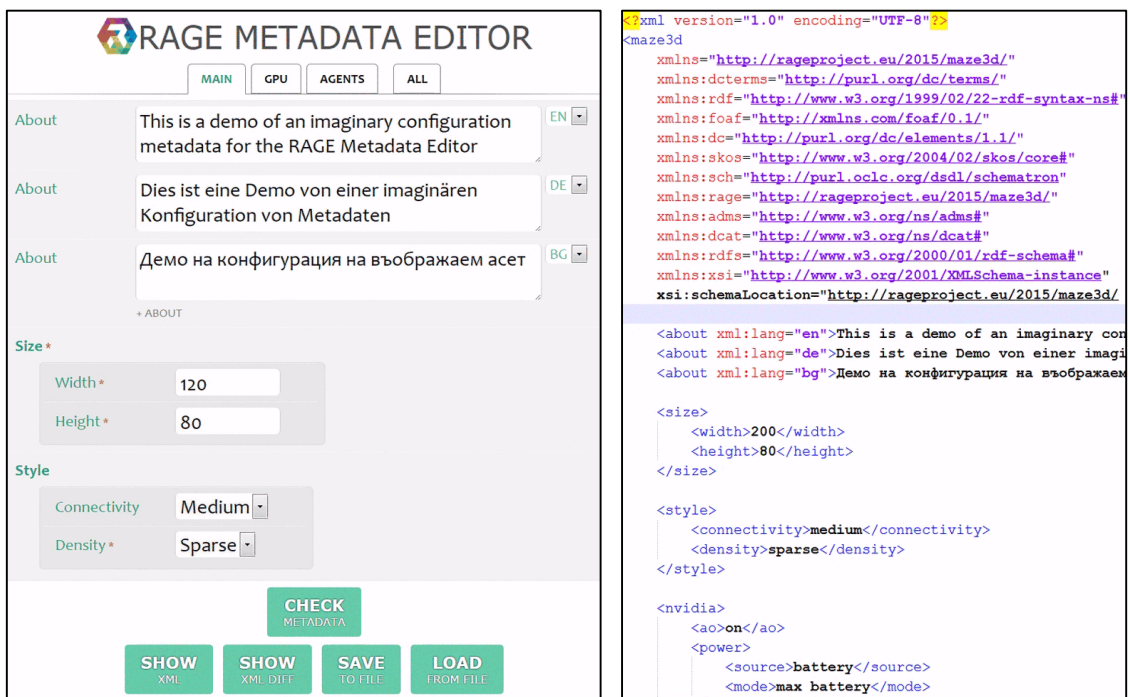


Figure 31: Generated Configuration Editor and configuration file

This flexibility allows the meta-editor to be used as an authoring tool that creates dedicated configuration editors. However, this requires the software component developers to prepare the schema files. This task introduces problems as long as developers might not be familiar with the schema definition of metadata structures in XML format.

Additionally, the meta-editor incorporates styling metadata, that control how the metadata elements are displayed on the user's screen. The styling is formatted as CSS with custom properties and parsable values.

To make schema creation easier, the preliminary prototype of the RAGE Configuration Editor Wizard was provided with a graphical interface, which developers can use to interactively generate schema and styling files, as well as control vocabularies, which can be eventually used by the RAGE meta-editor to create dedicated configuration editors. The workflow is shown in Figure 32 and it is functionally identical to the workflow of software component metadata editors in Figure 2, page 12.

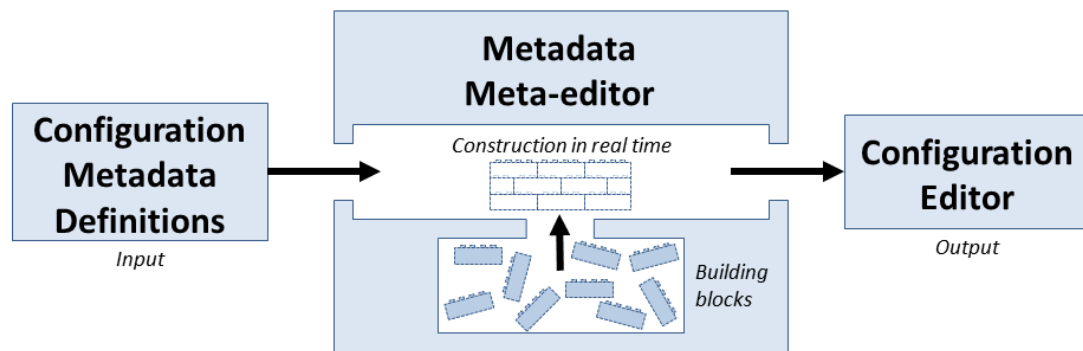


Figure 32: Metadata meta-editor constructing a configuration editor in real time

This initial prototype of the configuration editor has been used as a proof-of-concept authoring tool. Some of its features do not adhere to all the design principles, because the meta-editor is tightly coupled with the component metadata meta-editor. As a result of this coupling:

- The authoring tool is not a single file, the distribution packages contains subfolders with resource files, needed by the metadata meta-editor.
- The configuration editors created by the authoring tool required taxonomies from the RAGE server, so they could not be used as off-line tools.
- The configuration meta-editor is a piece of software that is less complex than the metadata meta-editor, however, it reuses it completely, so the configuration meta-editor is inherently more complex than actually needed.

After having created a successful proof of concept (the basic idea works!), a redesign of the editor was undertaken to achieve full compliance with the prior design principles.

5.2.2 Current implementation

The implementation of the current RAGE Configuration Editor Wizard reused a significant part of the prototype. Namely, the interface modules of the wizard and the meta-editor are reused, as well as the algorithm for generating editors. Removed is code for taxonomies and dependencies with the RAGE server, processing schema definition files, and so on. At a top level the wizard uses the same architecture as its prototype.

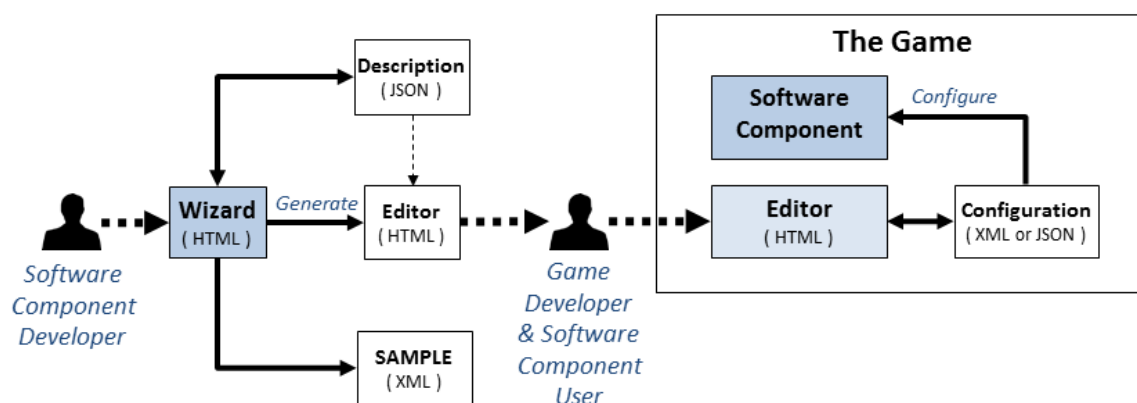


Figure 33: Using the wizard and the editor

Figure 33 pictures the intended use of the Configuration Wizard and Configuration Editor. The software component developer uses the RAGE Configuration Editor Wizard to define a description of the configuration file (in JSON format). When the description is complete, the wizard authors (creates) a RAGE Configuration Editor as a single HTML file with embedded styling and

JavaScript code. This file is made available to the game developer, the actual user of the software component. The game developer uses the editor to define and modify the configuration files (in XML or JSON format) in such a way that they enable the use of the software component within the game development environment and thereby in the game.

The wizard brings the authoring concept at a more unified level, by merging the wizard and the editor in the same file. The authoring tool is a file with two cores – see Figure 34. There is an *authoring core*, which implements the wizard-like user interface and functionality (i.e. allows definition and generation of configuration editors). The *editing core* implements the configuration editing user interface and functionality.

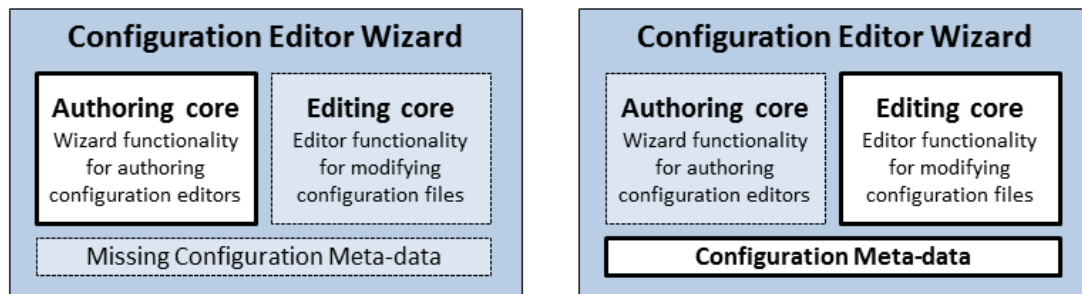


Figure 34: Unification of Configuration Wizard and Configuration Editors

Optionally, there is metadata, defining the configuration file, which is the same as the JSON description from Figure 33. The RAGE Configuration Editor Wizard (CEW) works as follows:

- If there is no metadata, CEW activates the authoring core and looks and behaves as the RAGE Configuration Wizard. When the user asks it to generate an editor, the wizard just copies itself and injects the metadata definition in the new instance.
- If there is metadata, CEW activates the editing core. It reads the metadata, generates on-the-fly the user interface for an editor and presents it to the user.

The CEW distribution package contains an example of a configuration of a software component named “Catalogue”. The file `Catalogue-Config.json` contains the JSON description of the Catalogue configuration file:

```
[{"Catalogue", {"name": "catalogue", "type": "group", "id": 10004, "group": "CatalogueGroup"}, {"CatalogueGroup", {"name": "company", "type": "group", "occurs": "0u", "id": 10009, "group": "CompanyGroup"}, {"CompanyGroup", {"name": "name", "type": "text", "occurs": "11", "id": 10012}, {"name": "person", "type": "group", "occurs": "0u", "id": 10013, "group": "PersonGroup"}, {"PersonGroup", {"name": "name", "type": "text", "occurs": "11", "id": 10014}, {"name": "phonenum ber", "type": "text", "occurs": "0u", "id": 10015}]
```

It is injected into CEW and saved as `Catalogue-Editor.html` to effectively turn CEW into a configuration editor. Inside the source code of CEW there is this fragment:

```
json = '[{"Catalogue", {"name": "catalogue", "type": "group", "id": 10004, "group": "CatalogueGroup"}, {"CatalogueGroup", {"name": "company", "type": "group", "occurs": "0u", "id": 10009, "group": "CompanyGroup"}, {"CompanyGroup", {"name": "name", "type": "text", "occurs": "11", "id": 10012}, {"name": "person", "type": "group", "occurs": "0u", "id": 10013, "group": "PersonGroup"}, {"PersonGroup", {"name": "name", "type": "text", "occurs": "11", "id": 10014}, {"name": "phonenum ber", "type": "text", "occurs": "0u", "id": 10015}]';
```

If this fragment is deleted:

```
json = '';
```

then the configuration editor will behave as configuration wizard. The left snapshot of Figure 35 demonstrates the user interface of a configuration editor, generated by `Catalogue-Editor.html`. If the metadata is removed from the file, then its activation generates a user interface for the wizard, which is a configuration authoring tool.

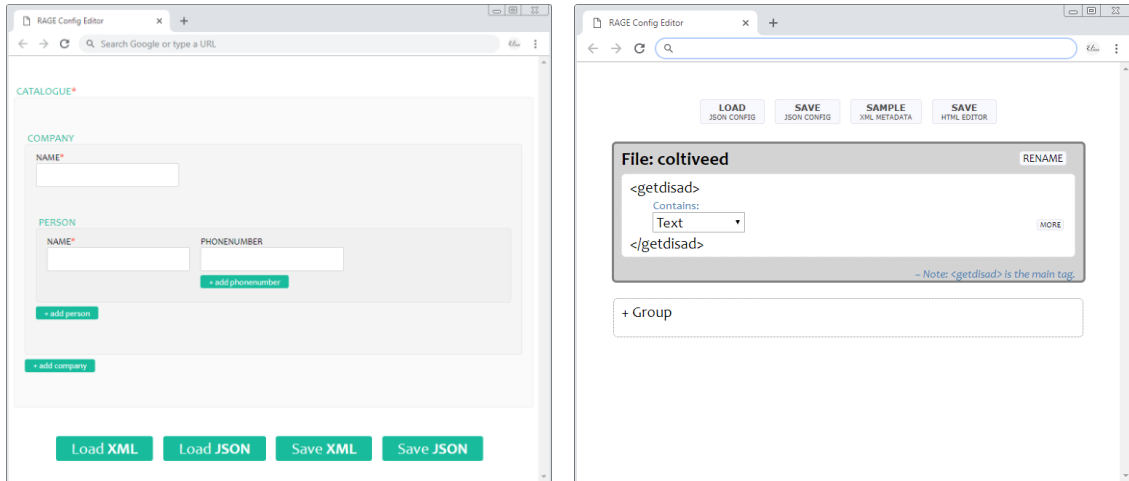


Figure 35: Configuration wizard and editor duality

This dualism of the wizard and the editor simplifies the use of the software as either a wizard, or an editor.

5.3 Parameters of the authoring tool

This section briefly describes the parameters that are used to describe the configuration file of a software component. These parameters are used to generate the editor for the configuration file. Also, the wizard uses this description to generate its own user interface, so that it mimics the XML structure of the configuration file.

An XML file is composed of nested, but not cross-overlapping tags. Here is an example of an XML configuration file.

```
<main>
  <engine>...</engine>
  <version>...</version>

  <mode>
    <x>...</x>
    <y>...</y>
    <fps>...</fps>
    <description>...</description>
  </mode>

  <mode>
    :
  </mode>

  :
</main>
```

The names of all tags, such as `<main>`, `<engine>` and `<fps>` are user-defined, i.e. the user of the wizard may think up the name of each tag. There are structures of tags, like the `<mode>` tag, which contains `<x>`, `<y>`, `<fps>` and `<description>` tags. The contents of the tags is also user-definable, as well as the cardinality of tags. In the example above, the tag `<mode>` has two instances.

A possible confusion might arise about the management of nested tags in the configuration wizard. Its interface uses the concept *group* to denote the structure of a tag without its opening and closing tag. In the following example the group is in a frame, while `<mode>` and `</mode>` are not part of the group.

```
<mode>  
  <x>...</x>  
  <y>...</y>  
  <fps>...</fps>  
  <description>...</description>  
</mode>
```

This might appear as counterintuitive, i.e. one may expect that an XML tag is to be defined as a whole – tag name and its contents. In the wizard, the group is an entity, which has its own name, different from the tag name. This approach is used to allow sharing of structures. For example, two different tags may share the same structure of their contents and the contents is defined as a single group. Such XML structure sharing is widely used in the RAGE metadata model and is “borrowed” from some schema definitions files. Dublin Core’s `<creator>` and `<publisher>`, and RAGE’s `<owner>` share the same structure, FOAF’s `<Organization>` and `<Person>` also share the same structure.

Figure 36 shows a typical interface of the wizard. The top-most block reflects the `<main>` tag; it is followed by blocks for each group of tags. The `<main>` tag contains nested tags, which are defined in *Resolution* group, while `<mode>` contains nested tags from the *ModeGroup* group.

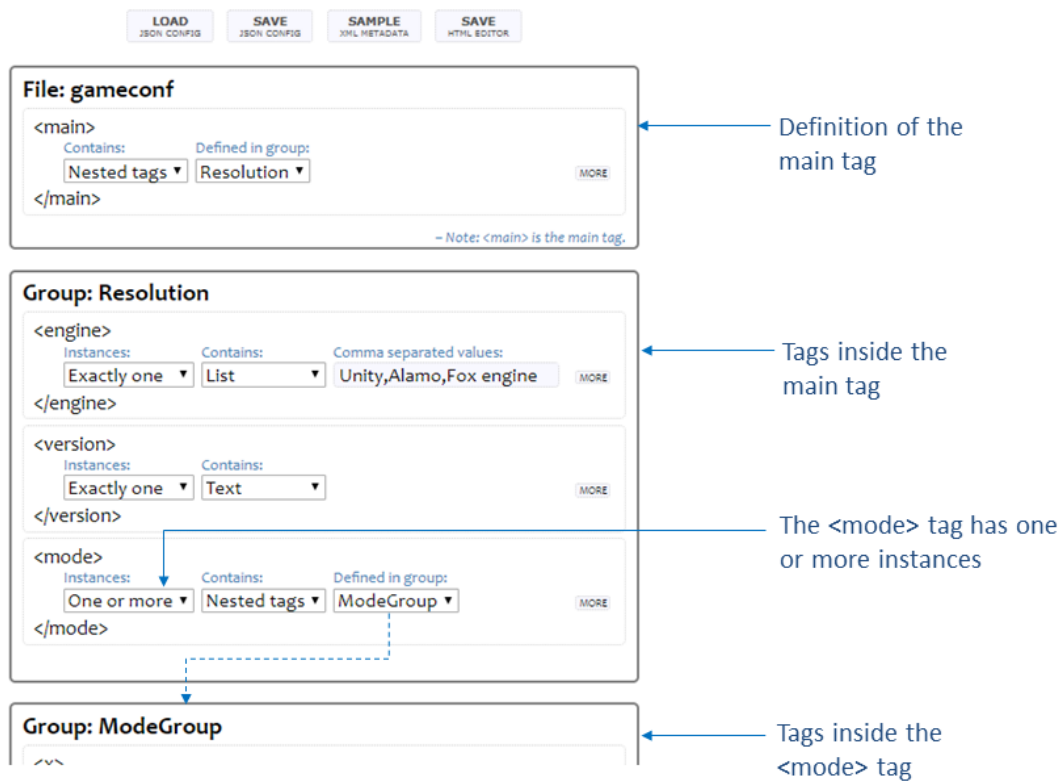


Figure 36: Visual layout of the Configuration Wizard

The interface makes it straightforward to define and understand the contents of a tag – the layout is structured almost like natural language sentences. Table 3 shows three examples from the user interface and their interpretations.

Table 9: Interpretation of the wizard’s interface

Interface	<pre> <engine> Instances: Exactly one Contains: List Comma separated values: Unity,Alamo,Fox engine </engine> </pre>
Interpretation	The <engine> tag has exactly one instance and its content is a value from the list (Unity, Alamo, Fox engine) .
Interface	<pre> <version> Instances: Exactly one Contains: Text </version> </pre>
Interpretation	The <version> tag has exactly one instance and it contains text .
Interface	<pre> <mode> Instances: One or more Contains: Nested tags Defined in group: ModeGroup </mode> </pre>
Interpretation	The <mode> tag may have one or more instances and it contains nested tags , which are defined in group ModeGroup .

Each tag has a set of configuration parameters that depend on the type of its contents. Some parameters define the value, others define how the element is visualized in the user interface of the RAGE Configuration Editor. The supported type of tags are:

- *Text*: The metadata element has a string value. Visually it is represented as a single line text entry field.
- *Paragraph*: The metadata element has a string value. Visually it is represented as a multiline text entry field. Depending on the browser, the field could be extended.
- *Integer*: The metadata element has an integer numeric value. Visually it is represented as a single line entry field.
- *Float*: The metadata element has a floating-point numeric value. Visually it is represented as a single line entry field.
- *URL*: The metadata element has a string value of URL, URI or another resource identifier. Visually it is represented as a single line text entry field.
- *Date*: The metadata element has a string value containing a date. Visually it is represented as a single line entry field. Depending on the browser, the field could be enhanced by interactive calendar and date selection control.
- *List*: The metadata element has a value from a predefined list of values. Visually it is represented as a list box. This type is used to represent controlled vocabularies or any other set of values, which are limited, e.g. Boolean values (true, false), powers of two (1, 2, 4, 8, 16, 32, 64, 128, 256), etc.
- *Nested tags*: The metadata element contains a group of other tags. Visually, they are defined in a separate block in the interface.

Table 10 lists the supported customization parameters, some of them are available for all tags, others are specific to a few of them.

Table 10: Supported customization properties

Property	Description
Contains	Defines the type of the tag – <i>text</i> , <i>paragraph</i> , <i>integer</i> , <i>float</i> , <i>URL</i> , <i>date</i> , <i>list</i> or <i>nested tags</i> .
Instances	Defines the number of instances of a tag – <i>exactly one</i> , <i>zero or one</i> , <i>any number</i> , <i>one or more</i> . If a tag may have more than one instance, in the generated configuration editor it will have a button for adding/removing instances. If a tag cannot be empty, it is marked in the editor with asterix.
Label	This is the label of the element shown in the configuration editor. If not set, the name of the tag is used as a label.
Pop-up tooltip	This is a short additional description which pop-up above each item when the mouse cursor hovers over it. If not set, no tooltip is shown.
In-line hint	This is a short text appearing inside the input box. If not set, the input box is initially empty.
CSS width	The width of the input box in CSS format (e.g. <i>20em</i> or <i>450px</i>). It is used to set the preferred width of input boxes.
CSS height	The height of the multiline input box in CSS format. It is used to set the preferred height of multiline input boxes.
Min	The minimal value for integer or float number.
Max	The maximal value for integer or float number.
Comma separated values	The element contains the allowed values, separated by commas. These values are shown in the end-user editor as a list box.

Nested tags (Defined in group)	The element contains the name of the group that defines the nested tags.
-----------------------------------	--

5.4 *Distribution*

The RAGE Configuration Editor Wizard is designed to be easily distributable. It does not require installation or any configuration.

Both the wizard and the generated editors are standalone applications (single HTML files). The software component developer uses the wizard to create a configuration editor (an HTML file) and, optionally, the structure of the file (a JSON file). Only the HTML file is given to the end user – the game developer.

If the component developer needs to provide a predefined configuration file, it can be created by the developer with the configuration file (an XML file) and given to the user together with the configuration editor or with the software component.

6 CASE STUDIES AND EVALUATION

In order to test and evaluate the Configuration Editor Wizard , we asked two asset developers to use the tool for their specific configuration tasks related to their assets. Below we describe the two case studies and the main results from the practical use of the editor.

6.1 *The use of RAGE Configuration editor wizard for client-side assets*

In this case study we describe the use of the RAGE Configuration Editor Wizard (CEW) for generating editors for the two most common XML files present in RAGE client-side assets: 1) asset settings and 2) version and dependencies. This case study was performed at the Open University of the Netherlands using CEW version 3.0: all issues with earlier versions were resolved (e.g. the problem of some missing data types). A client-side tracker asset for learning analytics, being fully compliant with the RAGE component architecture (D1.4), was used for testing.

It turned out that both editors where very simple to define using CEW. Nevertheless, two minor issues and one major issue showed up.

The file ASSETSETTING.XML (see below) is optional and is used to load and save settings of an asset at runtime. The same format is also used for packaging compile-time settings for an asset. A typical example is the settings for the tracker asset that defines a URL of the server counterpart together with some additional settings.

```
<TrackerAssetSettings>  
  <Host>145.20.132.23</Host>  
  <Port>80</Port>  
  <Secure>>false</Secure>  
  <BasePath>/api</BasePath>  
  <UserToken>a:</UserToken>  
  <TrackingCode>56d842596ff353420uazbinewmi</TrackingCode>  
  <StorageType>net</StorageType>  
  <TraceFormats>xapi</TraceFormats>  
  <BatchSize>2</BatchSize>  
  <LogFile>c:\TrackerAsset.log</LogFile>  
</TrackerAssetSettings>
```

The resulting editor definition is shown in Figure 37.

File: TrackerAssetSettings

<TrackerAssetSettings>

Contains: Defined In group: MORE

</TrackerAssetSettings>

– Note: <TrackerAssetSettings> is the main tag.

Group: TrackerAssetSettings_Group

<Host>

Instances: Contains: MORE

</Host>

<Port>

Instances: Contains: MORE

</Port>

<Secure>

Instances: Contains: MORE

</Secure>

<BasePath>

Instances: Contains: MORE

</BasePath>

<UserToken>

Instances: Contains: MORE

</UserToken>

<TrackingCode>

Instances: Contains: MORE

</TrackingCode>

<StorageType>

Instances: Contains: Comma separated values: MORE

</StorageType>

<TraceFormats>

Instances: Contains: Comma separated values: MORE

</TraceFormats>

<BatchSize>

Instances: Contains: MORE

</BatchSize>

<LogFile>

Instances: Contains: MORE

</LogFile>

Figure 37. Tracker Asset Settings Editor Definition

The editor definition can be saved as a standalone html based web page that can be opened in a web browser and can be used to generate the asset settings file for the TrackerAsset.

TRACKERASSETSETTINGS*

HOST*	PORT*	SECURE*	
<input type="text"/>	<input type="text"/>	<input type="text"/>	
BASEPATH*			
<input type="text"/>			
USERTOKEN*	TRACKINGCODE*	STORAGETYPE*	TRACEFORMATS*
<input type="text"/>	<input type="text"/>	net ▾	json ▾
BATCHSIZE	LOGFILE		
<input type="text"/>	<input type="text"/>		

Figure 38. Generated Tracker Asset Settings Editor

The only issue encountered in this editor is the lack of support for Boolean data types (the 'Secure' field is a Boolean value used to enable the https instead of http protocol). As a workaround a text field or list with values 'true' or 'false' will work.

The VersionAndDependencies.xml is a file that should be present in each RAGE client-side asset and describes the version and dependencies on other RAGE client-side assets. It is used by the implementation of the RAGE client-side asset architecture to load version and dependency information at run-time. This information can then be used by the AssetManager to generate a version and dependency report.

```
<?xml version="1.0" encoding="utf-8" ?>
<version>
  <id>ScenarioReasoner</id>
  <major>0</major>
  <minor>1</minor>
  <build>1</build>
  <maturity>alpha</maturity>
  <dependencies>
    <depends minVersion="1.2.3">Logger</depends>
  </dependencies>
</version>
```

The editor defined with this file definition results in the editor shown in Figure 39.

File: VersionAndDependencies

<version>
 Contains: Defined in group: MDK1
 </version>

- Note: <version> is the main tag.

Group: version_group

<id>
 Instances: Contains: MDK1
 </id>

<major>
 Instances: Contains: MDK1
 </major>

<minor>
 Instances: Contains: MDK1
 </minor>

<build>
 Instances: Contains: MDK1
 </build>

<revision>
 Instances: Contains: MDK1
 </revision>

<maturity>
 Instances: Contains: Comma separated values: MDK1
 </maturity>

<dependencies>
 Instances: Contains: Defined in group: MDK1
 </dependencies>

Group: dependency_group

<depends>
 Instances: Contains: MDK1
 </depends>

Figure 39. Version and dependencies Editor Definition

VERSION*

ID*	MAJOR*	MINOR*
<input type="text"/>	<input type="text"/>	<input type="text"/>
BUILD	REVISION	MATURITY*
<input type="text"/>	<input type="text"/>	alpha ▾

DEPENDENCIES

USE ASSET IDS HERE

[+ add Use asset IDs here](#)

[Load XML](#)
[Load JSON](#)
[Save XML](#)
[Save JSON](#)

Figure 40. Generated Version and dependencies Editor

The xml generated by this editor has two issues. The first, minor one is the lack of the XML declaration:

```
<?xml version="1.0" encoding="utf-8" ?>
```

at the start of the XML file. Ideally the declaration should be definable in CEW (as the encoding might change). The impact of omitting the XML declaration is not clear (as reading XML files is often supported) as it may lead to character encoding issues.

The other issue, which is a major one, is the lack of support for XML attributes needed for the <depends> tag. This issue is more fundamental as the generated editor supports both XML and JSON output and only XML natively supports the usage of attributes. A solution might be to add a checkbox in the editor for tags defining simple values (like string or integer).

For XML, depending on the checkbox value, CEW could emit the value as either attributes or tag when XML is exported.

For the JSON export one could adopt a workaround like BadgerFish () that implements lossless round trip conversions between XML and JSON.

6.2 Case study on the application of the Configuration Editor to FAtiMA Toolkit

In this case study, the configuration editor is used as a potential replacement for the custom-made authoring tools for a specific bundle of Rage components, namely, the FAtiMA Toolkit, which is a collection of components designed for the creation of characters with social and emotional intelligence. This use case was motivated in part due to the complexity of the configuration files that the FAtiMA Toolkit requires. For example, the following JSON object is an example of a possible configuration for the character's emotional state:

```

"EmotionalState":
{
  "Mood": 0,
  "EmotionalPool": [
    {"Intensity": 5, "Decay": 1, "Threshold": 1, "CauseId": 1,
      "EventName": "Event(Action-End, Player, Kick, John)",
      "EmotionType": "Reproach", "Valence": "Negative",
      "AppraisalVariables": ["PRAISEWORTHINESS"],
      "InfluenceMood": true },
    {"Intensity": 2, "Decay": 1, "Threshold": 1, "CauseId": 1,
      "EventName": "Event(Action-End, Player, Kick, John)",
      "EmotionType": "Distress",
      "Valence": "Negative",
      "AppraisalVariables": ["DESIRABILITY"],
      "InfluenceMood": true
    }
  ]
}

```

This example illustrates the need for complex data types such as an array of “emotions” that have a specific structure. Because the Configuration Editor Wizard was designed as a generic solution that is capable of handling any given component, the tool makes very little assumptions about the structure and the type of configuration that each component requires.

Figure 41 and its continuation in Figure 42 show the definition of the FATiMAQ configuration file in CEW. The emotional state complex type is defined as a group EmotionalStateGroup, which in turn contains EmotionalPoolGroup.

File: FATiMAconfig

<EmotionalState>

Contains: Defined in group: MORE

</EmotionalState>

– Note: <EmotionalState> is the main tag.

Group: EmotionalStateGroup

<Mood>

Instances: Contains: MORE

</Mood>

<EmotionalPool>

Instances: Contains: Defined in group: MORE

</EmotionalPool>

Figure 41. Description of FATiMA configuration

Group: EmotionalPoolGroup

<Intensity>
 Instances: Exactly one ▾ Contains: Integer ▾ MORE
</Intensity>

<Decay>
 Instances: Exactly one ▾ Contains: Integer ▾ MORE
</Decay>

<Threshold>
 Instances: Exactly one ▾ Contains: Integer ▾ MORE
</Threshold>

<CauseId>
 Instances: Exactly one ▾ Contains: Integer ▾ MORE
</CauseId>

<EventName>
 Instances: Exactly one ▾ Contains: Text ▾ MORE
</EventName>

<EmotionType>
 Instances: Exactly one ▾ Contains: List ▾ Comma separated values: Reproach, Distress, Other MORE
</EmotionType>

<Valence>
 Instances: Exactly one ▾ Contains: List ▾ Comma separated values: Positive, Negative, Neutra MORE
</Valence>

<AppraisalVariables>
 Instances: One or more ▾ Contains: List ▾ Comma separated values: PRAISEWORTHINESS, DES MORE
</AppraisalVariables>

<InfluenceMood>
 Instances: Exactly one ▾ Contains: List ▾ Comma separated values: true,false MORE
</InfluenceMood>

Figure 42. Description of FATiMA configuration (cont)

This description is used by CEW to generate a configuration editor specific to the FATiMA software component. When started, this editor contains no data and it looks as shown in Figure 43.

EMOTIONAL STATE*

MOOD*

EMOTIONAL POOL

INTENSITY* **DECAY*** **THRESHOLD*** **CAUSE ID***

EVENT NAME*

EMOTION TYPE* **VALENCE*** **APPRAISAL VARIABLES*** **INFLUENCEMOOD***

Reproach Positive PRAISEWORTHINESS true

+ add Appraisal variables

+ add Emotional pool

Figure 43. Generated configuration editor for FATiMA

After entering some data the FATiMA configuration editor would look like Figure 44:

EMOTIONAL STATE*

MOOD*

EMOTIONAL POOL

INTENSITY* **DECAY*** **THRESHOLD*** **CAUSE ID***

EVENT NAME*

EMOTION TYPE* **VALENCE*** **APPRAISAL VARIABLES*** **INFLUENCEMOOD***

Reproach Negative PRAISEWORTHINESS true

+ add Appraisal variables

+ add Emotional pool

Figure 44. FATiMA configuration editor with data

To test the configuration of more emotions, the user can use the “add Emotional pool” button and open a new section for emotion definition. Figure 45 shows the attempt to add two additional emotional pools and the last one of them has several appraisal variables.

EMOTIONAL POOL

INTENSITY*	DECAY*	THRESHOLD*	CAUSE ID*
<input type="text" value="2"/>	<input type="text" value="1"/>	<input type="text" value="1"/>	<input type="text" value="1"/>

EVENT NAME*

EMOTION TYPE*	VALENCE*	APPRAISAL VARIABLES*	INFLUENCEMOOD*
<input type="text" value="Distress"/>	<input type="text" value="Negative"/>	<input type="text" value="DESIRABILITY"/>	<input type="text" value="true"/>

[+ add Appraisal variables](#)

[+ add Emotional pool](#) [- remove](#)

EMOTIONAL POOL

INTENSITY*	DECAY*	THRESHOLD*	CAUSE ID*
<input type="text" value="0"/>	<input type="text" value="4"/>	<input type="text" value="3"/>	<input type="text" value="1"/>

EVENT NAME*

EMOTION TYPE*	VALENCE*	APPRAISAL VARIABLES*	
<input type="text" value="Other"/>	<input type="text" value="Neutral"/>	<input type="text" value="PRAISEWORTHINESS"/>	

[+ add Appraisal variables](#)

APPRAISAL VARIABLES*	APPRAISAL VARIABLES*	INFLUENCEMOOD*
<input type="text" value="DESIRABILITY"/>	<input type="text" value="MOODLESS"/>	<input type="text" value="false"/>

[+ add Appraisal variables](#)
[- remove](#)
[+ add Appraisal variables](#)
[- remove](#)

[+ add Emotional pool](#) [- remove](#)

Figure 45. FAtiMA configuration editor with more data

When this configuration data is saved as JSON file, it reads as follows:

```
{
  "EmotionalState": {
    "Mood": "0",
    "EmotionalPool": [
      {
        "Intensity": "5",
        "Decay": "1",
        "Threshold": "1",
        "CauseId": "1",
        "EventName": "Event (Action-End, Player, Kick, John)",
        "EmotionType": "Reproach",
        "Valence": " Negative",
        "AppraisalVariables": "PRAISEWORTHINESS",
        "InfluenceMood": "true"
      },
      {
        "Intensity": "2",
        "Decay": "1",
        "Threshold": "1",
        "CauseId": "1",
        "EventName": "Event (Action-End, Player, Kick, John)",
        "EmotionType": " Distress",
        "Valence": " Negative",

```

```

    "AppraisalVariables": " DESIRABILITY",
    "InfluenceMood": "true"
  }, {
    "Intensity": "0",
    "Decay": "4",
    "Threshold": "3",
    "CauseId": "1",
    "EventName": "Event (Action-End, Player, Show, John)",
    "EmotionType": " Other",
    "Valence": " Neutral",
    "AppraisalVariables": ["PRAISEWORTHINESS", " DESIRABILITY", "
MOODLESS"],
    "InfluenceMood": "false"
  }
]
}
}

```

The same data can be exported by the FAtiMA configuration editor as XML file:

```

<EmotionalState>
  <Mood>0</Mood>
  <EmotionalPool>
    <Intensity>5</Intensity>
    <Decay>1</Decay>
    <Threshold>1</Threshold>
    <CauseId>1</CauseId>
    <EventName>Event (Action-End, Player, Kick, John)</EventName>
    <EmotionType>Reproach</EmotionType>
    <Valence> Negative</Valence>
    <AppraisalVariables>PRAISEWORTHINESS</AppraisalVariables>
    <InfluenceMood>>true</InfluenceMood>
  </EmotionalPool>
  <EmotionalPool>
    <Intensity>2</Intensity>
    <Decay>1</Decay>
    <Threshold>1</Threshold>
    <CauseId>1</CauseId>
    <EventName>Event (Action-End, Player, Kick, John)</EventName>
    <EmotionType> Distress</EmotionType>
    <Valence> Negative</Valence>
    <AppraisalVariables> DESIRABILITY</AppraisalVariables>
    <InfluenceMood>>true</InfluenceMood>
  </EmotionalPool>
  <EmotionalPool>
    <Intensity>0</Intensity>
    <Decay>4</Decay>
    <Threshold>3</Threshold>
    <CauseId>1</CauseId>
    <EventName>Event (Action-End, Player, Show, John)</EventName>
    <EmotionType> Other</EmotionType>
    <Valence> Neutral</Valence>
    <AppraisalVariables>PRAISEWORTHINESS</AppraisalVariables>
    <AppraisalVariables> DESIRABILITY</AppraisalVariables>
    <AppraisalVariables> MOODLESS</AppraisalVariables>
    <InfluenceMood>>false</InfluenceMood>
  </EmotionalPool>
</EmotionalState>

```

The results of this case study demonstrate the practicality of the approach.

6.3 Evaluation summary

All tools described in this Deliverable were extensively tested by software components developers and end users. The methodology for testing and main results were described in D1.1.

This section lists the major positive aspects and issues with the Configuration Editor Wizard.

Positive aspects:

- It does not require set-up or installations.
- It works directly on the browser as a JavaScript application.
- It looks very good, and it worked surprisingly well to generate the editor.
- It handles the basic data types and nested complex data structures.

Observed issues:

- No direct support for Boolean values. The reason for this is the existence of multiple representations of Boolean values, like `true-false`, `on-off`, `checked-unchecked`, `yes-no`, and `1-0`. In the current version of the editor Boolean values can be implemented as user-defined lists – as an example, see the `InfluenceMood` parameter at the very bottom of Figure 42.
- No support of tag attributes and namespaces. The generated editors' import and export data as XML and JSON. XML attributes are not natively supported by JSON. There is an approach to encode attributes via special naming convention. However this would require the gaming software component that reads the configuration file to adopt the same naming convention.
- No identification for character encoding. The character encoding identification is a prefix in the XML file before the actual tags. It is not represented in JSON.
- No verification and validation. The design of the configuration editor has no knowledge about the required verification and validation of the metadata. It is expected that each software component has implemented its own module for these purposes, because configuration files could be provided by other means (including manually typing in a text editor). To minimize the code of the configuration editor, it does not verify or validate the data.

7 CONCLUSIONS

In this Deliverable, we presented the software architecture supporting the lifecycle of reusable software components for applied gaming. The main innovation is related to the combination of RAGE Asset Model and RAGE Asset Metadata Model, backed up with server-side infrastructure (repository and services) and many end user tools. The software architecture plays a pivotal role within the RAGE Ecosystem platform and is considered of strategic importance for the domain of applied gaming.

The repository as the content core system of the RAGE Ecosystem allows for flexible design and development of RAGE game assets and future search, packaging and exchange. The current architecture guarantees both scalability and durability and the approach. It also provides a high level of flexibility across different taxonomies and standards.

We further improved the architecture by providing support for Quality Assurance, asset development workflows, harvesting of assets from external systems and stores, user-friendly widgets for asset development and configuration, which provide targeted support for the gaming community.

REFERENCES

Van der Vegt, W., Westera, W., Nyamsuren, E., Georgiev, A., Ortiz, I., (2016) RAGE architecture for reusable serious gaming technology components, International Journal of Computer Games Technology, volume 2016, Article ID 5680526, 2016.

Georgiev, A., Grigorov, A., Bontchev, B., Boytchev, P., Stefanov, K., Bahreini, K., Nyamsuren, E., Van der Vegt, W., Westera, W., Prada, R., Hollins, P., Moreno, P., The RAGE Software Asset Model and Metadata Model, JCGS 2016, LNCS 9894, 2016, pp. 191-203.

Suleman, H., Metadata Editing by Schema, International Conference on Theory and Practice of Digital Libraries. Springer, Berlin, Heidelberg, 2003, pp. 82-87

Grasso, M., Craglia, M., D 2.2.3A: European Open Source Metadata Editor Developers' Guide v.1.0, EuroGEOSS, 2010.

Kunze, T., Brase, J., Nejd, W., Editing learning object metadata: Schema driven input of RDF metadata with the OLR3-Editor. In Semantic Authoring, Annotation & Knowledge Markup Workshop, 2002, pp. 22-26.

Bontchev, B., Iliev, T., ARCADE – Web-based Authoring and Delivery Platform for Distance Education. In First Balkan Conference on Informatics, Thessaloniki, Greece, 2003, pp. 293-306.

Leon, J. Frequency of Character Pairs in English Language Text, Codes and Cryptography, 2008, Available at homepages.math.uic.edu/~leon/mcs425-s08/handouts/char_freq2.pdf, September, 2017.