



Project Deliverable Report

Deliverable 2.2 – Existing Services – integrated

Work Package	2
Task	2.2, 2.4
Date of delivery	Contractual: 01-10-2009 Actual: 01-10-2009
Code name	D2.2 Version: 1.13 Draft <input type="checkbox"/> Final <input checked="" type="checkbox"/>
Type of deliverable	Report
Security (distribution level)	Public
Contributors	Reinhard Dietl, Fridolin Wild, Bernhard Hoisl, Robert Koblichke (WUW), Berit Richter, Markus Essl, Gerhard Doppler (BIT MEDIA), Traian Rebedea, Stefan Trausan-Matu (PUB-NCIT), Philippe Dessus (UPMF)
Authors (Partner)	Reinhard Dietl, Fridolin Wild, Bernhard Hoisl, Robert Koblichke (WUW), Berit Richter (BIT-MEDIA), Paola Monachesi (UU), Kiril Simov (IPP-BAS), Traian Rebedea (PUB-NCIT), Sonia Mandin, Virginie Zampa (UPMF)
Contact Person	Fridolin Wild (WUW)
WP/Task responsible	WUW
EC Project Officer	Ms. M. Csap
Abstract (for dissemination)	This document focuses on existing software artefacts from project partners and their availability to be turned into web-based and service-oriented applications. Therefore, a technical description of every task is given. A strategy is also described focusing on the software development process in relation to the project's timeline.
Keywords List	LTfLL, Widgets, NLP, Services, Integration, Development

Table of Contents

Executive Summary	3
1. Introduction.....	4
2. Positioning Showcases (WP4).....	6
2.1 Positioning from Portfolios (T4.1).....	6
2.2 Conceptual Development (T4.2).....	8
3. Support and Feedback Showcases (WP5)	12
3.1 Interaction Analysis (T5.1).....	12
3.2 Assessing Textual Products (T5.2)	15
4. Social and Informal Learning Showcases (WP6)	20
4.1 Task 6.1 – Knowledge Sharing Network.....	20
4.2 Task 6.2 – Social Component	24
5. Overview on Showcase Characteristics	32
6. Development Strategy	33
6.1 Position of Software Development within the Project	33
6.2 Development Workflow	35
7. Versioning Policy.....	41
References.....	43

Executive Summary

Within this deliverable, a current snapshot on the iteratively developing prototypes is given. The integration of these showcases focuses mainly on turning the existing technology into web-based, service-oriented applications, and structuring their software architecture along the concept elaborated in the previous infrastructure deliverable D2.1. This involves mainly the re-use of existing technology (hence the name ‘showcases’), although several of these task showcases already involve new developments.

The task showcases documented here encompass two prototypes per work package, one for each task outlined in the description of work. This includes:

- T4.1: Positioning from Portfolios
- T4.2: Conceptual Development
- T5.1: Interaction Analysis
- T5.2: Assessing Textual Products
- T6.1: Knowledge Sharing Network
- T6.2: Social Component

The presentation of the integration of existing services is wrapped up by an overview on the interoperability characteristics of these showcases.

Additionally, the development strategy pursued is elaborated in section 6, thereby providing more details about the software development process and the development workflow. Finally, section 7 updates the versioning policy in brief.

This deliverable d2.2 interfaces closely with the deliverables d3.1 and d3.2: whereas this document focuses on the technical (architectural) documentation, the scenario deliverables d3.1 and d3.2 focus on the user perspective.

Details on the pedagogical workflow along with details on infrastructure options including scalability of the proposed solution and the trade-offs evolving from technical decisions are delivered in the integration report.

1. Introduction

Within the first deliverable D2.1 of this work package on integration, a concept had been elaborated for a service-oriented development framework that is strong enough to integrate the technologies considered essential for the tasks of the work packages 4, 5, and 6 while at the same time respecting their heterogeneity. This concept incorporates the segmentation of the involved systems and components into three layers: a layer each for widgets, services, and data storage. The service layer itself can further be divided into service logic and application logic thus encapsulating the service invocation, execution, and delivery routines and the application specific processing routines respectively.

Following this encapsulated approach with well-defined interfaces has many advantages in the sense of interoperability, transferability, and reusability of the deployed software components. A widget-based architecture seems to be the right choice to handle the very heterogeneous software developments of the different partners. Defining clear interfaces makes it possible to have loosely-coupled and spatially distributed applications working together.

Therefore, it is essential to ‘widgetise’ the deployed software which means having a GUI-output being able to be viewed in any web-browser at the client-layer (web-widget) which can be addressed using web-services. A first step in this direction is the widgetisation and servicification of existing showcases: partners need to plan their further developments for the version 1 of integrated services in form of widgets and services. This means not necessarily adapting all existing showcases – as these are only prototypes; it means, however, testing the adaptation where the showcase forms already a basis for future developments.

Within this deliverable, a current snapshot on the iteratively developing prototypes is given, thereby following the former described conceptual approach. The integration of these showcases focuses mainly on turning the existing technology into web-based applications and structuring their software architecture along the concept touched upon above and elaborated in more detail in deliverable D2.1. This involves mainly the use of existing technology, although several of the task prototypes already involve new developments.

The task prototypes documented here encompass two prototypes per work package, one for each task outlined in the description of work. This includes:

- T4.1: Positioning from Portfolios
- T4.2: Conceptual Development
- T5.1: Interaction Analysis
- T5.2: Assessing Textual Products
- T6.1: Knowledge Sharing Network
- T6.2: Social Component

2. Positioning Showcases (WP4)

The work within work package 4 on positioning is organized into two complementary tasks. Task T4.1 focuses on the analysis of portfolios in order to determine a learner's current standing (position) in a given domain. Task T4.2 on the contrary concentrates on diagnosing a learner's conceptual development. The following subsections outline the prototypes developed for showcasing both tasks.

2.1 Positioning from Portfolios (T4.1)

As outlined in D2.1, the web service architecture is composed of four layers. The following sections will outline the R-based LSA framework for web services applied to this 4-tier-architecture.

On the **client layer**, a web browser can use any HTTP based communication mechanism to access web services via RESTful requests. The server hosting the service framework handles the request by invoking the applications on the service layer and dispatches any parameters that have been passed to the server.

Within the **service layer**, the service framework relies on code written in R which is used to transform, validate, and then communicate the parameters to the application logic layer. Invocation of the underlying R-scripts is handled by an Apache server, which is equipped with a module called 'Rapache' (Horner, 2009). On arrival of a request for an R web service at the Apache server, Rapache invokes an instance of the R shared library, executes the required R web-service script and enables the R framework to access all information that has been passed to the Apache server by the client (via the HTTP protocol).

After computation of the application logic – successful or not – the service layer returns a custom XML structure representing the result object. The actual structure to be used can be freely chosen by the web service implementer. This individual choice is given to developers as it enables quick development of XML interactions for simple tasks without limiting the implementation of a full-fledged communication architecture based on a standard XML protocol (e.g. SOAP), which would have been the case if e.g. XML-RPC would have been chosen as the standard protocol.

The **application logic layer** consists of a service-specific R function which is able to perform LSA processes with only the parameter list passed by the service layer.

The backend layer ensures the presence of appropriate access functions to the current space warehouse implementation in the environment ('scope') of the dynamic LSA processes.

Decoupling the LSA process logic from the service layer framework has the following advantages:

1. A dedicated LSA process logic developer can implement logical functions with neither knowing, nor caring about the actual implementation of the space warehouse on the one hand and the service layer on the other hand.
2. The architecture the usage of the storage access methods provided by the storage layer, which makes the handling of large spaces on the backend layer independent of the application logic.
3. Having the whole application logic present as a single R language object enables the manipulation of the logic itself using R, enabling the optimization e.g. of the data handling by reducing copy-procedures of values (Oehlschlägel, et al., 2008).

The LSA process logic then returns a result object, which is – depending on the task –any R object, including lists, arrays or even binary image data which may be generated by a graphics implementation. This data is then passed to the service layer for transformation and communication to the client layer.

The **backend layer** provides help for implementers of LSA process logic who may face several problems concerning the retrieval of the space objects:

- *Storage requirements for spaces*: Depending on the application scenario of the LSA process logic, spaces can be very large and might not fit into the main memory of the client, nor server.
- *Transfer bottlenecks*: Depending on the storage device, retrieval of a space from that device may be slow.
- *Retrieval overheads*: storage mechanisms like compression or serialization create a computational overhead when retrieving a space.
- *Simultaneous access*: Some applications scenarios (like web services) may require instant access to space objects for multiple clients at the same time.

The considerations above lead to the development of three types of space storage mechanisms which can be accessed by the applications layer using a common interface:

- The *monolithic warehouse approach* keeps a central R instance permanently open, holding all space objects previously calculated in main memory. LSA processing logic is passed into this R instance and executed there, locally. This approach eliminates all overhead created by copying large space objects between storage media as they are accessed directly from memory. It has to be kept in mind that the central R instance holding the spaces is unavailable to other requests until the LSA processing logic has finished.
- The *inter-instance copy approach* keeps – like the monolithic approach – all spaces in main memory. Application logic is – in contrast – not executed in the storage R instance, but rather, a copy of the original object is passed to the Rapache instance, freeing the central R instance's access interface again as soon as the copy has been generated.
- The *serialisation approach* keeps all space objects in a binary file on the server's hard disk. This has the advantage that on most servers, hard disk space will by far exceed main memory, and for that reason, storage should be less a problem. On the downside, (de-)serialization of space objects may be – depending on the hardware used – a time consuming task which may slow down the LSA process.

2.2 Conceptual Development (T4.2)

Following the proposal of D2.1, the architecture of the showcase prototype for monitoring the conceptual development of a learner can be segmented into four layers. A client layer communicates through a service layer with the application logic that again is supported by particular functionalities (most notably data storage and retrieval functionalities) coming from a back-end layer (see Fig. 1).

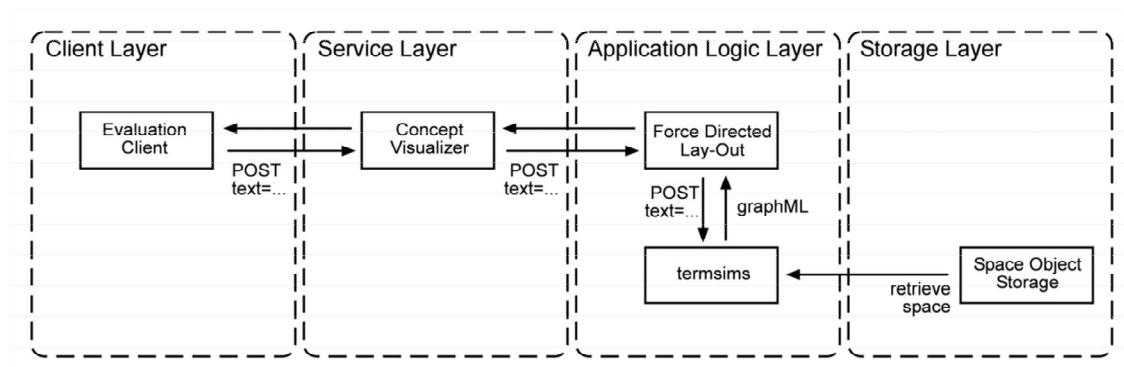


Fig. 1: Architectural layers of the system.

The **client layer** (see Fig. 2) can be seen as running the visible user interface of the prototype, typically via a web browser. The evaluation client consists of two components, the first being a form with a text area into which learners copy & paste their textual input. The second widget is a visualization of the concept graph which is called upon submitting the textual input. This graph then renders the relationships between the key concepts triggered by the textual input through a force-direction lay-out: terms semantically close to each other will be placed in short distance of each other, while terms distant will be positioned further apart from each other. The colors of the concept map indicate to which cluster the terms belong. More details on the calculation will be outlined below in the description of the other layers.

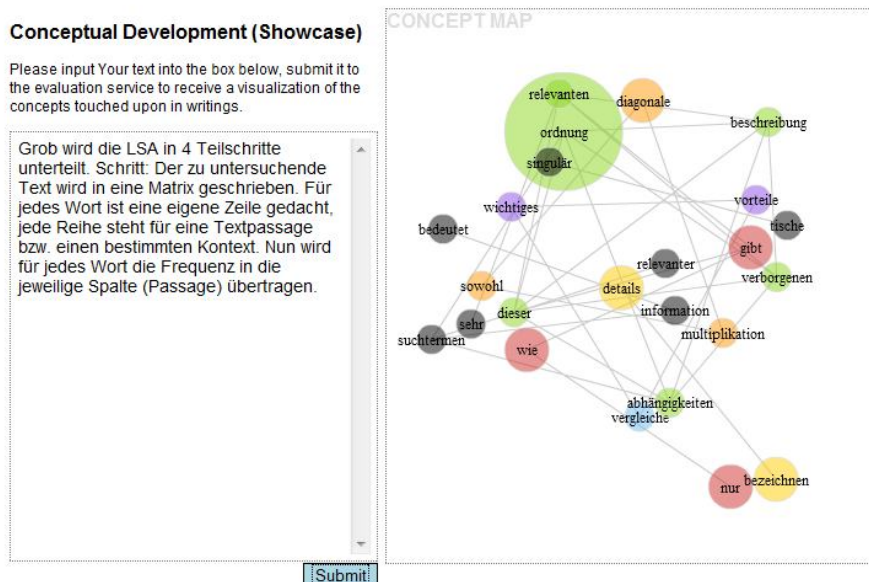


Fig. 2: Screenshot of the user interface.

The **service layer** holds the counterpart to the client layer: it provides the html-pages, receives the textual input via HTTP using POST, dispatches it to the force-directed lay-out and returns an HTML-wrapped flash application in order to deliver the requested visualization.

The **application logic layer** consists of two components; both of them can also be accessed directly as services. The force-directed lay-out is a flash application written in flex that uses the Prefuse Flare visualization toolkit (see <http://flare.prefuse.org/>) in order to render the underlying graph of conceptual relations among the resulting concepts with the help of a physics simulation of interacting forces. By dispatching the received input text to the second component – termsims –, the resulting graph is reconstructed from the

graphML transport format (see <http://graphml.graphdrawing.org/>) and is dynamically visualized on the flash stage.

This second service serves to calculate term-to-term similarities by first folding the textual input into a pre-existing latent-semantic space and by extracting those most prominent terms from the resulting lower-order text vector in this space by filtering for the 30 most frequent terms that load with a frequency in the latent-semantic space higher than a given threshold of .05. These terms are considered to be concepts describing the textual input in this lower-order latent-semantic space. The size of the node is calculated to be ten times the value of their frequency in the latent-semantic space (plus 1).

By subsequently calculating the term-to-term cosine distances of these terms, a graph wrapped into graphML can be returned that contains the concepts as nodes (labeled with the corresponding term) and all term-to-term cosine distances above a given threshold as edges.

For example, the resulting concept list might contain the terms ‘dog’, ‘cat’, and ‘table’; dog and cat might have a cosine distance of .7 in this space (being higher than the threshold of .6), whereas both dog and cat have a cosine distance of .3 to table; thus, the termsims service would return only an edge between the nodes ‘dog’ and ‘cat’, but none involving the node ‘table’.

Additionally, the termsims service returns colors for each node depending on which cluster it has been assigned to using kmeans with (*number-of-nodes / 4*) clusters.

The **storage layer** contains one latent-semantic space constructed from a flexible subset of the 18 million documents large and freely available PubMed/Medline corpus, nine million of which included an abstract.

To generate a proper LSA space for the medical field from these documents, we first extracted these documents from XML to a format more suitable for the subsequent computations. As we eventually needed the document (bag-of-words) data in R, we chose to use the R text mining (tm) package (see Feinerer et al., 2008) for data extraction as it provides facilities to convert a corpus to a term-document-matrix.

However, it soon became obvious that we would run into memory problems when trying to hold the complete text corpus in memory with R, so we went from storing full text representations of the documents to representing documents by indices relative to the corpus vocabulary only. This not only heavily cut down memory usage, but also sharply

sped up the following bag-of-words matrix calculation, which would have been another serious bottleneck in terms of computation speed. To further reduce the corpus' memory usage we also applied some basic preprocessing (stemming, stopword filtering, number removal, lower case, etc.) while reading in the text content.

As we wanted to create custom spaces based on topics or concepts, we first analysed the PubMed topical information (the MeSH headings) and extracted this data into a MySQL database to subsequently retrieve the corresponding document ids for to a given MeSH heading (and its children). To extract the MeSH headings to the database, we modified the PubMed data extraction demo from the Lingpipe Java libraries for our needs.

We also chose the R text mining package, because its term-document-matrix (TDM) facility used a sparse matrix as its data format, which was a must for us if we wanted to incorporate a larger corpus such as PubMed. As the R LSA package (Wild, 2009) could not yet handle sparse matrices for its singular value decomposition (SVD), we interfaced the `svdlib.c` from Berry (1992) in the modified version from Rhode (2009) that can handle sparse matrices as input data. A new release of the package is currently being prepared.

Using aforementioned techniques we could successfully calculate a test space that includes all 9 million abstract-holding PubMed documents and a vocabulary of more than 330.000 (frequency-filtered) terms.

However, if there is more detailed information available about the topical requirements, a smaller and more efficient space can be calculated around one or more topics. For this purpose we adapted the R text mining package again: Now the user only needs to provide the topics (given as MeSH descriptors) to the text mining package, which in turn generates a new corpus based on the documents of interest extracted from the document base. From this corpus the user may then easily generate a (preprocessed) textmatrix that is finally used as input for the singular value decomposition.

This way we created latent-semantic spaces for the MeSH descriptor 'skin' (containing information from 49.414 documents) as well as a space containing the MeSH descriptors 'pharmacology', 'drug interactions', and 'accident prevention' (built from 24.346 documents).

3. Support and Feedback Showcases (WP5)

Work in this work package 5 is organized alongside two tasks. Task T5.1 concentrates on recommendations derived from interaction analysis of learners. Task T5.2 focuses on recommendations based on assessing student writings.

3.1 Interaction Analysis (T5.1)

Two applications have been developed for the analysis of collaborative chat conversations in order to provide automatic feedback and grading to the tutors: Polyphony Analyzer and ChAMP (Chat Assessment and Modeling Program). As the implementation of these systems had started before D2.1 was available and because they have been intended to be used by a limited number of tutors for testing and validation purposes mainly, they were not designed as services, but as stand-alone desktop systems, therefore the client-server architecture was not used. Nevertheless, they were partially implemented a four-tier architecture composed of a three-tier modified MVC (Model-View-Controller) and a data processing layer. The MVC has the restriction that the Model (Data) layer and the View (Presentation) layer do not communicate directly, but only through the Controller as Fig. 3 of the architecture displays below.

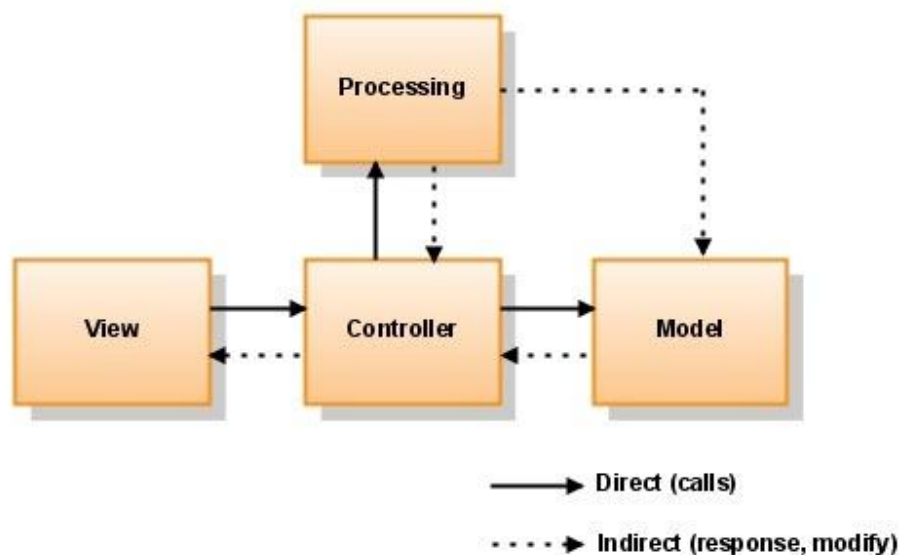


Fig. 3: Architectural layers of the system.

Different technologies were used for the implementation of these applications: Polyphony Analyzer was developed in C#.NET (using the additional library WordNet.NET), while ChAMP was executed in Java. The latter is using the following additional libraries: Jazzy for spellchecking, Prefuse for social networks modeling, JFreeChart for generating charts, JWI (Java Wordnet Interface) for interacting with Wordnet and MTJ (Matrix Toolkit for Java) for EVD - eigenvalue decomposition - and SVD - singular value decomposition. The configuration of the applications is done by using external configuration files and special internal classes.

View (Presentation) Layer

The view contains the user interface and controls and is responsible for the interaction with the user and displays the results of the processing. Polyphony Analyzer is a multi-windowed application, while ChAMP uses multiple tabs. The communication with the controller is done inside the event (action) handling mechanism used for each control and all the parameters needed to handle the event processing are transmitted to the controller. After the controller finishes processing the event, he passes back the results that should be interpreted by the view in order to be properly displayed to the user by modifying the controls. A simple way of communicating between the view and the controller is by using messages that have a type that is related to the actions that should be taken by each layer and a container that encapsulates all the parameters needed to process the message.

Controller Layer

The controller is used for communication between the view and the model, thus being responsible of modifying the data layer accordingly to the user's actions in the view and by transmitting back the information that should be modified in the view to reflect the processing of the data. It receives messages from the view and performs the operations corresponding to the message type and parameters. These messages modify the data model or the state of the controller. The modification of the data stored in the model is done either by passing the message further to the model view, if the operations are simple, or through the data processing layer, if complicated operations are necessary. After the processing of the data is done, the controller receives the results of the processing and constructs a new message that is communicated to the view that contains the information that should be modified by the view. As the applications are not very complex, the communication between the three other layers performed by the controller is synchronous, using a single processing thread.

Model (Data) Layer

The model stores the data processed by the application. Initially, the data contains only the chat conversation that is parsed from an external XML (or HTML) transcript file.

When proceeding with processing the chat, the model is enriched with new information about the results of the processing. Due to the low memory requirements required for storing and processing the data, it is stored in the main memory as long as the application uses it. The model modifies the underlying data either directly when requested by the controller or indirectly when the processing layer modifies the data in the model to store its results. When the model finishes processing the data, it notifies the controller and transmits to it the information that is needed to change the view. Part of the processed data can be exported in different formats depending of the application: Polyphony Analyzer exports XSL spreadsheets and text files, while ChAMP exports XML files.

Data Processing Layer

The data processing is responsible for performing the natural language and social network processing tasks that are more difficult. It is instructed by the controller to start the analysis and, after finishing its task, updates the model, thus enriching it, and notifies the controller. Because the data processing layer is separated from all the other tiers, the functions that perform the processing tasks can be modified without altering any of the other layers or by performing only minor modifications.

Service-Orientated Approach

For the integration in the LTfLL architecture described in D2.1, the two systems will be transformed in web services. The view requires the most significant adaptation, as the desktop based view must be transformed in a web browser usable one, by using HTML controls and forms, plus modifying the interaction pattern with the server. This way, the View layer shall become the Client layer, while the rest of the application shall reside on the server. The rest of the tiers are easily mapped on the D2.1 architecture: the Controller is the Service layer and must be adapted to respond to HTTP requests, while the Data Processing layer is equivalent to the Application Logic layer and the Model layer to the Backend layer, with minor modifications.

The services that shall be developed for task 5.1 are intended to provide support and feedback for collaborative chat conversations and discussion forums for students, tutors and teachers. The service implementation is very loosely based on the current systems, as the services shall employ a different approach, integrating some of the features used for the two systems and improving them by implementing new features. There shall be two distinct services that shall use similar approaches, especially for the data processing techniques: a chat service and a forum service. These services shall be decomposed into a number of distinct sub-services – for example, the chat analysis service shall have at least three sub-services: one for feedback generation, one for providing grading for each participant and one for an enhanced visualization of the conversations. The Client layer

shall use the widget platform based on PHP, JavaScript, AJAX and Flash/Flex or Java applets/JavaFX. The Backend layer shall be used to store the data in memory and retrieve it from a persistent storage, such as a relational or XML database. We opt for a relational database used together with an ORM (Object-Relational Mapping) like the Hibernate framework for Java. The Service layer shall provide the end-points of the services and the communication between the outer-world (requests/responses) and the Data and Application Logic layers.

The most important layer of the service shall be the Application Logic layer because this is where all the processing is going to be performed. Considering its complexity, the processing layer can be decomposed into 7 different sub-layers:

1. Basic processing and NLP (Natural Language Processing) pipe: spelling correction, stemmer, tokeniser, POS tagger;
2. Linguistic ontology (e.g. Wordnet) interfacing sub-layer;
3. Domain ontology and semantic sub-layer – either built by experts or automatically extracted from various web sources (e.g. using Wikipedia and Wiktionary);
4. Social network analysis sub-layer;
5. Advanced NLP and discourse analysis sub-layer: identification of cue phrases, speech acts, rhetorical schemas, lexical chains, co-references;
6. Advanced discourse analysis: adjacency pairs, implicit links, discussion threads, argumentation, transactivity;
7. Polyphony sub-layer: includes modules for examining inter-animation, convergence and divergence.

The first four sub-layers perform distinct operations that are not inter-related one with another, but they are all used by the sub-layers 5-7 that provide more advanced functions. Moreover, each sub-layer in the interval 5-7 uses the outputs provided by all sub-layers that have a lower identifier (e.g. sub-layer 6 uses the outputs from all sub-layers 1-5), thus composing a processing layer stack that has at the base sub-layers 1-4 and at the top sub-layer 7. All the results provided by the Application Logic layer are saved to the Backend layer through the Service layer.

3.2 Assessing Textual Products (T5.2)

Apex (Lemaire, et al., 2001) is a system which delivers automatic feedback to students who read course texts and then summarise them. It invokes LSA for both proposing texts to read, as a search engine, and for measuring how well the summary matches the given

course text. The student is involved in two main interaction loops. First, as many course texts as the student wants are proposed upon request (see Fig. 4). The student assesses each of the texts read as understood (summarisable) or not (reading loop, see Fig. 5). Then the student can freely choose among the following tasks: write a summary from each of the read texts he understood (writing loop, see Fig. 7), read a new text or create a new request, depending on his response and the availability of further texts (see Fig. 5 and 6). During the writing loop, the student can ask *Apex* to assess his summary (i.e., whether he captured the gist of the course text, see Fig. 8 and 9), and then can revise it accordingly or go to the next summary (see fig. 9). The following use cases describe the human-system interactions.

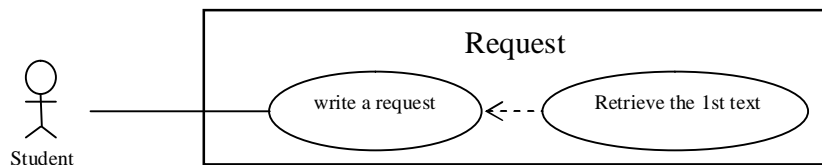


Fig. 4: The student indicates key words about the topic to work on and types a request.

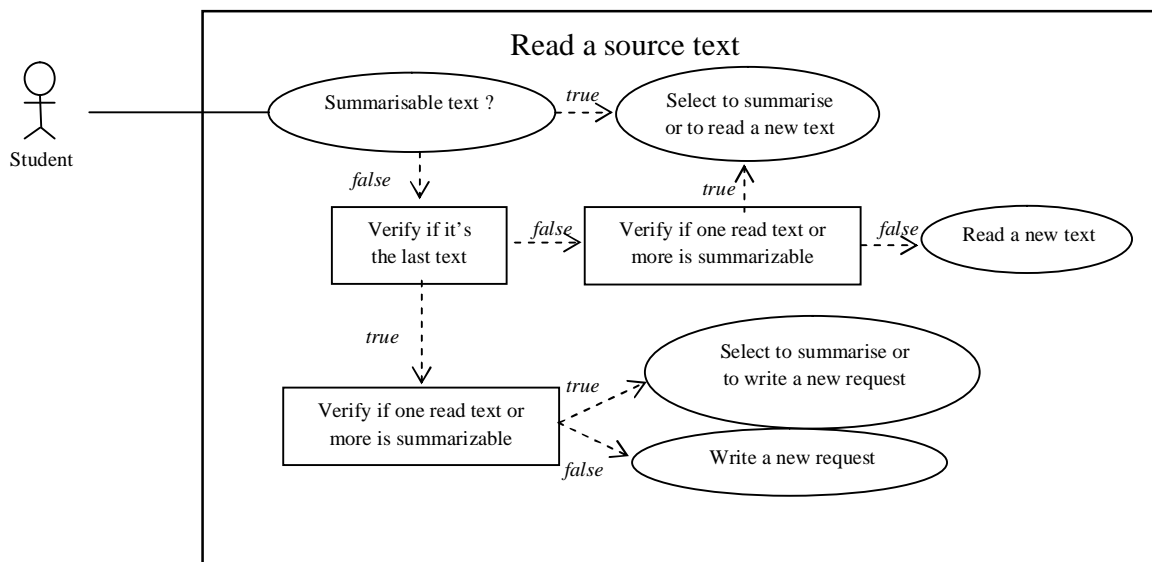


Fig. 5: The student reads a text and determines if he can summarise it.

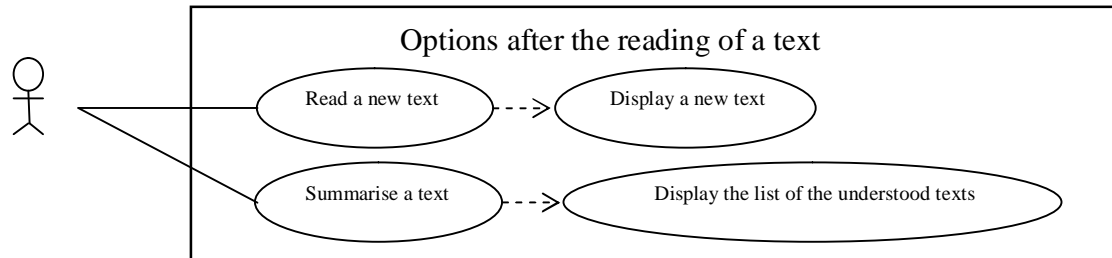


Fig. 6: The student chooses if he wants to write a summary or read a new text.

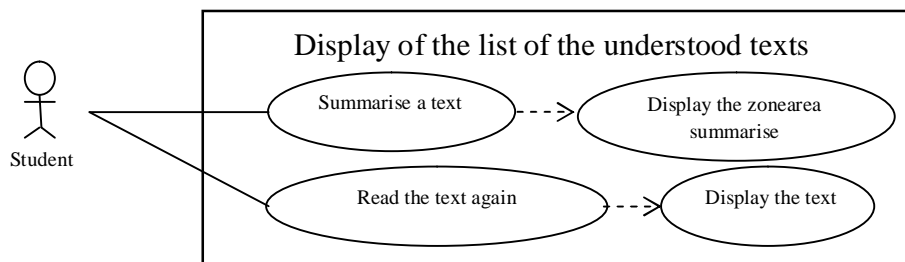


Fig. 7: The student determines if he wants to summarise or re-read a text.

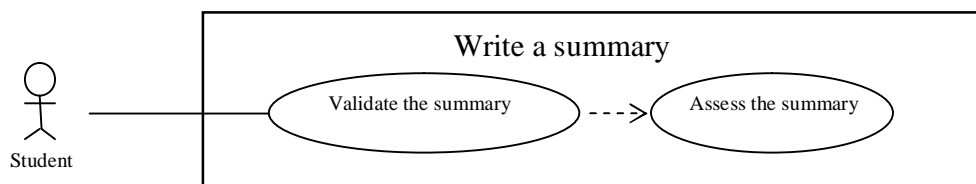


Fig. 8: The student writes a summary and asks an assessment.

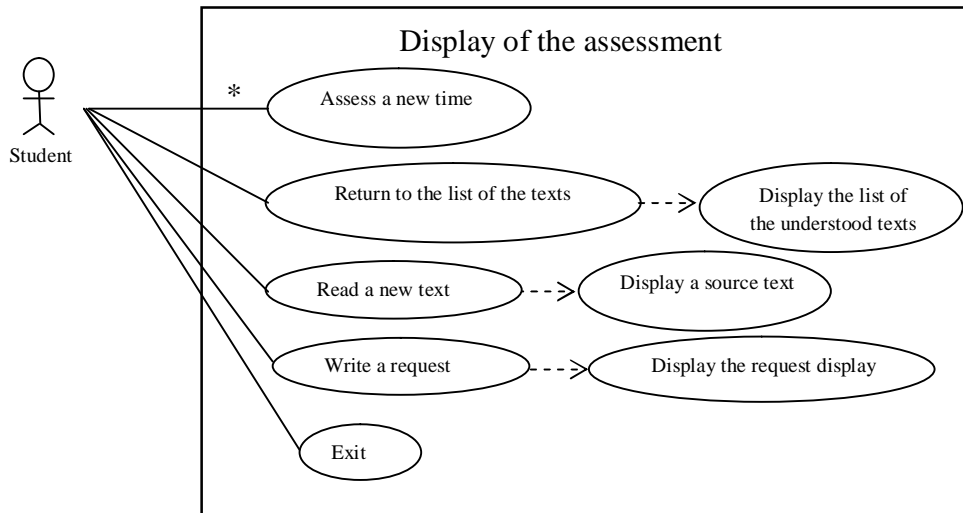


Fig. 9: The student peruses his evaluation and determines that he wants to do after that.

The student actions lead to the execution of different functions which are required to send requests to a server, on one hand to store any data and on the other hand to use LSA (via the bellcore application, currently being replaced by the R infrastructure).

Client layer

Clients use a web interface. It lets users read texts, summarise them and peruse assessments about the work completed. The interface code is in HTML / PHP.

Service layer

This layer links the interface and LSA. In accordance with the users' actions, every parameter is sent to the server with POST/GET method. The parameters are used in C scripts which give the possibility to invoke the LSA program or to recover user data from text files (one by session).

Application logic layer

C scripts are invoked. They are able to perform LSA from parameters passed or to recover data directly from files. The LSA application (currently still the Bellcore application) returns a result file. In this file, semantic proximities are required. The service layer transforms this file to communicate data to the client layer.

Storage layer

LSA needs a semantic space to function. We compute it from a text corpus, which depends on the user's knowledge level and the domain taught. Since computing a semantic space is processing time demanding, we use semantic spaces computed in advance. If we want LSA to make comparisons in processing a new document (in addition to the corpus), we don't compute a new semantic space, and we use specific LSA functions (*tplus* and *syn*, i.e., "fold in" technique).

Descriptions of functions

Below, the PHP functions which are used are described. Some of them invoke C functions:

- *Affiche_requete*: the user has to write his request
- *Fait_choix*: this function proposes to go on to read or to write a summary
- *Fait_selection*: this function sends the first text and recovers data about the understanding of this text. The first text is selected depending on the LSA results. It invokes from a C function (*selectFirstText.c*) which invokes the *tplus* and *syn* LSA functions in turn.
- *Recup_compris1*: this function invokes a C function (*understoodUser.c*) to update the session file.
- *Affiche_newText*: this function selects the next text and refreshes the session file. The text is selected depending on LSA results. LSA is invoked from a C function (*newText.c*) which invokes the *syn* LSA function in turn.
- *Recup_choix*: this function recovers user choices from each step.
- *Afficher_text_compris*: this function provides to the user the list of the read and summarizable texts (displayed as a whole or only the first sentence) as well as the *Apex* assessment. Then the user can either summarise a given text, or re-read it.
- *Ecrire_text*: this function displays the form used to summarise a text and store the user summary.
- *Fait_choix2*: if there is no other text to display, *Apex* proposes to write the summaries of the understood texts or to write a new request
- *Eval_text*: this function allows the assessment of a summary by LSA, its storage and its display. LSA is invoked from a C function (*understoodLSA.c*) which invokes *tplus* and *syn* LSA functions.

4. Social and Informal Learning Showcases (WP6)

The work within work package 6 on supporting social and informal learning is divided into two tasks. Task T6.1 deals with the creation of a knowledge sharing network, whereas task T6.2 focuses on adding a social component to the public knowledge.

4.1 Task 6.1 – Knowledge Sharing Network

This document describes the architecture and the services which will be part of the Common Semantic Framework (CSF).

CSF Resources

The data stored in CSF will be in XML format. The notion to be used here is *resource*. Typical resources are ontologies, lexicons, learning materials, communication notes, comments, web links, etc. Each resource is connected with the following additional information:

- DTD or XML Schema – definition of the structure of the XML documents that are instances of the resource
- Internal elements and external elements. Internal elements contain information which will be processed by the CSF – editing, storing, searching, visualizing, etc. External elements are pointed from the resource and they generally are processed by external tools – for example, PDF viewer is used to open PDF documents.
- Tools for processing of the elements of the resource.
- Visualization rules for the resource – how the elements of the resource are presented to the user, how the user can manipulate them, etc.
- Search Schema(s) for the resource.
- Resource creation. The following modes are envisaged – automatic creation by an external services, manual by the user, or mixture of the previous ones (some elements of the resource are generated automatically, others are entered manually).

This information will be called *Resource Information*. The XML documents for a given resource will be called also *Resource Documents*.

Basic Level Services of CSF

The basic level of CSF provides services for all the basic operation of XML documents that represent resources. These services are: DTD/XML Schema Management; Resource

Information Editing; Tools Declaration; Search Schema Editing; Store Resource Documents; Retrieve Resource Documents; Visualize Resource Documents.

DTD/XML Schema Management

This service will support: the declaration of a DTD/XML Schema; editing of existing DTD/XML Schema; validation checks of the resource documents; export of DTD/XML Schema.

We will reuse the corresponding modules from CLaRK System, implemented by us, (<http://www.bultreebank.org/clark/index.html>). CLaRK is a perfect editor for XML documents and DTDs. It is integrated with XPath processor, DTD Validator and tools that can support users for XML documents transformations. The full version needs to be implemented.

Resource Information Editing

This service will support: editing of resource information; import and export of resource information.

For each kind of resource information an editor will be implemented. The actual information will be represented as XML documents which will facilitate the exchange of such information.

To be implemented.

Tools Declaration

This service will support the declaration of external tools (services). The declaration will contain the following information: where the tool is located; the type of the tool arguments; the type of the result. Usually the external tools will be used for navigation over the web, browsing different type of documents (PDF, RTF, etc.), for creation of resources (complete or partial). The tools will be called under some events within CSF.

To be implemented.

Search Schema Editing

This service will support the creation and modification of search schema. One of the basic components of CSF is an XML oriented search engine. In order the resource documents to be searchable at least one search schema needs to be connected to the resource. The search schema contains definitions of the context of search, terms of search and retrievable elements. When a search schema is defined an index for it is created and resource documents can be indexing in it.

It is implemented as an extension of Lucene engine (<http://lucene.apache.org/>).

Store Resource Documents

This service will support storing of resource documents (local or remote). During the storing corresponding index will be modified appropriately.

It is implemented using the file system at the moment, but it could be extended to web repository.

Retrieve Resource Documents

This service will support retrieval of resource documents (or their elements) from the document repository. The retrieval could be done from a list of documents or via XML Search Engine. The first option is similar to navigation over file system. The second option is via Lucene engine based XML search engine. This search engine provides a query language tuned to search schemas. The query is evaluated over an appropriate index and the service returns a list of documents and/or their elements that match the query.

It is implemented as an extension of Lucene engine (<http://lucene.apache.org/>).

Visualize Resource Documents

This service will read the rules for visualization and will generate a concept map which is displayed to the user. The user will be able to navigate over it, edit it, stored it and run on request external tools. There will be two versions of the service – standalone and web-based one. The first will require installation of VUE system locally. The web-based one will be simplified version of the standalone one.

This service is under implementation within the Visual Understanding Environment (VUE) – <http://vue.tufts.edu/>.

Extended Layer of CSF

At this level of CSF the actual resources and services for the tasks within LTfLL project will be provided. They will include the actual definition of resources with their resource information. The following resources are envisaged at that phase of the project: Ontology Management Service; Lexicon Management Service; Document Annotation Service; Social Media Services (Task WP6.2).

Ontology Management Service

This service will comprise three services: Ontology Storing and Reasoning; Ontology Translation; Ontology Filtering.

Ontology storing and reasoning service provides the basic functionalities for accessing information (explicit or implicit) from ontology. This includes:

- registration and deregistration of ontology models;
- listing of direct and indirect sub-concepts;

- listing of direct and indirect super-concepts;
- listing of individuals for a concept;
- listing of concepts to which an individual belongs;
- listing of properties defined for a concept;
- structural and logical consistency check of the ontology model registered;
- extraction of a registered ontology model;
- generation of ontology fragments (with/out sub-concepts; super-concepts; sibling concepts; property relations and range concepts; with property restriction super-concepts for a concept supplied as a parameter);
- generation of a sub-hierarchy, containing super-concepts and/or sub-concepts with a pointed step for a supplied as a parameter concept.

For implementation of these functionalities Pellet OWL reasoner is used (<http://www.mindswap.org/2003/pellet>). The necessary Web services are already implemented and they will be reused within LTfLL project.

Ontology translation service converts ontology in simplified graph representation and also translates the concept and relations names in a natural language using the lexicon service. The result of this service will be the main way in which ontology will be presented within CSF as a resource. For this resource we will define a search schema, a set of visualization rules. The actual service is already implemented as our own software module and an appropriate web service is defined for it. The incorporation of the resource in CSF will be implemented within LTfLL.

Ontology filtering service will apply rules for simplification of the ontology in order it to be better understood by the user. This service will be implemented as our own module within the project.

Lexicon Management Service

This service supports the alignment of lexicons with the ontology. It provides functions for access of terms for a concept; concepts expressed by a term; addition of new terms; access to definitions. It is already implemented as our own software module. An appropriate web service is defined for it.

Document Annotation Service

This service comprises two services: text annotation and image annotation. The text annotation is implemented as a language pipe performing tokenization, POS tagging, lemmatization, and semantic annotation. We already have implementations of language pipes for several languages (from LT4eL project). Where it is possible we will reuse them. Additionally, we will augment these language pipes in order to achieve better semantic annotation. The implementation is done by using third parties software for

languages different from Bulgarian. The integration and the augmentation will be done within CLaRK system (<http://www.bultreebank.org/clark/index.html>). In order to support web services we extended CLaRK system to be used in pipes and as a server. This service will be used for annotation of each textual element of the resources within CSF such as learning objects, comments, definitions, etc. The image annotation works with images in multimedia documents. It provides an image editor for selection of regions in the image and mechanisms for annotation of these regions with concepts from ontology. The resulting annotation is stored in the document and can be used for searching and other processing. The image annotator is already implemented.

Social Media Services

These are services which will be implemented with LTfLL project and they will elicit new knowledge and recommendations from social media.

4.2 Task 6.2 – Social Component

The Common Semantic Framework (CSF) will be a knowledge sharing system that connects learners to resources and learners to other learners by means of user profiles, ontologies, social tagging and social networks. In order to connect learners to resources, we relate the tags that emerge from the social media applications to the (lexicalizations of the) concepts of an existing domain ontology. To this end, we extract the domain knowledge based on social tagging. This domain knowledge, covering relations between topics is compared with that attested in ontologies and is eventually used to populate them. In this way, we aim to complement the formal knowledge represented by ontologies with the informal knowledge emerging from social tagging improving thus the possibility of retrieving appropriate material for a more personalized learning experience.

We also aim at connecting learners to other learners. To this end, the content the learner is searching and selecting can be used as a trigger to get him in touch with other users who have tagged or used this content before him. To this end, the learner will focus on the learning objects that are produced by people who are relevant for the domain he studies and/or people the learner trusts. The system will monitor the changes that appear in his network with respect to content and users and will recommend how the learner should update the network (by adding new peers or removing old ones) and answering the learner's questions about relevant materials and peers on a given topic. In this way, we add a trust dimension to the search since a learner will trust the objects produced, tagged or recommended by his own network.

This wide array of tasks that aim to integrate the heterogeneous sources asks for clear separation of the relevant components in order to keep the system maintainable, scalable and easy to use. The different levels of interaction which exist at different timescales (real time responses to queries, asynchronous notification of new information and data aggregation components) ask for a clear division of layers and associated components that are going to perform the tasks needed. The CSF contains components for data storage, various types of task dependent visualization and other components for transforming stored data in a given repository and inferring and extracting new information from it. Strong decoupling of components in the separate layers is of vital importance for future development and extension of the CSF.

Our system architecture (a specification of how various components/services should interact) is based on an augmented Model-View-Controller architectural pattern. This architectural pattern strongly supports clear separation between various types of services/components and is widely used and understood in Software Engineering practice. We have added (similar to Task 5.2, see section 3.2) a separate data-processing layer which is loosely coupled to the controller for computationally intensive tasks. We think that the separate processing layer can ease development on actual long-running computational tasks which are not directly linked to the view layer in any way. This could for example include a crawler which extracts data from social media applications continuously. This is in sharp contrast to the controller layer that is actually very much involved in passing and transforming information between the model and the view layer. The view layer contains the components needed for presentation and visualization of data provided by components in the controller layer.

We will now introduce a description of the view layer, a description of the data model used, the controller layer, data processing layer and we will illustrate the usefulness of the design using an example.

Model Layer

Data which is aggregated from various sources and needs to be available at a later time is stored in the model layer. Examples include the data aggregated by the crawler which includes connections between tags, resources and users, ontologies and lexica. The data needs to be stored in the model, because real-time aggregation is not possible within the time-constraints defined by the visualization (results need to be available within a

second), so some form of caching and or pre-computation of partial results is needed in order to meet these demands.

The model layer thus caches results retrieved from various sources which can include ontologies, details about the structure of various social networks and so on. In order to accommodate the various information sources the data model needs to store triples of type (User, Resource, Tag), as these triples are the way to represent folksonomies (Mika, 2007). The Data model represented in Figure 10 below consists of these primary entities plus the necessary relations between them and also some entities like keyword and concept needed for disambiguation purposes.

The User represents the user of the application and his peers, the resource can represent any type of content that a User can create or use (videos, bookmarks, blog entries, etc.), the tag is a word that is used to annotate a resource and the community is a virtual group of users where a number of tags are used. Additionally the crawler needs a queue to store the URLs to be visited and the next time of the visit for that resource. The relations are the ones between users (friendship, followers, etc.), community membership, relation between users and resources and the relation between user, resources and tags.

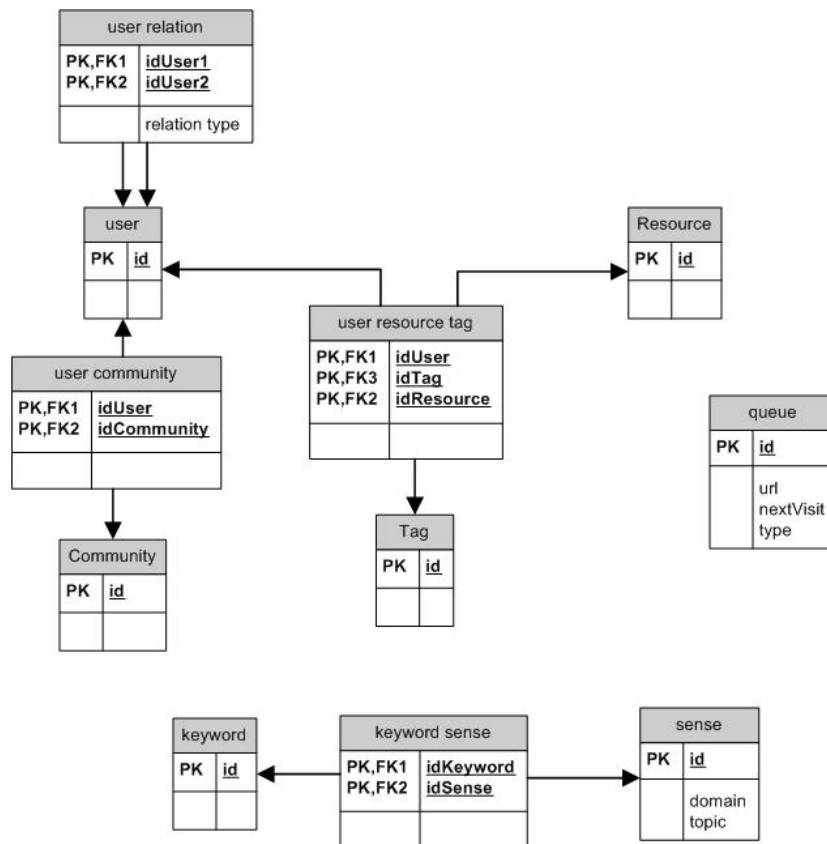


Fig. 10: ER data model

The data model depicted above is primarily aimed at storing the data retrieved by the crawler, but other information sources exist in the model layer as well. This includes ontologies, lexica and pre-computed term-document-vectors. The imported ontologies can both be domain-specific such as the LT4eL ontology on computing that is reused in this project, but larger background ontologies like dbpedia are also included when a domain ontology is either insufficient or non-existent. Each ontology can be queried and stored separately if required.

The data model for the crawler is currently implemented using a MySQL database. There are plans for the next version to move to an RDF repository (Sesame, Mulgara) that will allow easier integration with ontologies like the LT4eL ontology and with other lexica. Components in the controller directly interact with the database. For the moment ontologies are cached from the web and stored as files in a repository which can then be loaded into the Ontology Management Service in the controller layer. The integration between the folksonomies and the resources in the ontology is currently ongoing.

View Layer

The view layer contains components which are responsible for the interaction with the user. The view layer needs to be independent of the data model used and interacts with the controller in order to acquire data needed for some type of visualization or sends feedback back to components in the controller layer. In order to obtain flexible, reusable and web-enabled components in the view layer widgets will be used. Widgets are small web applications that provide some usually simple function and can be easily embedded into any other web application. The use of the augmented MVC-pattern means that we will constrain this very wide notion of 'widgets' somewhat. Widgets are only allowed to communicate with components from the controller layer which export web services. A good decomposition of the user interface into separated widgets decouples functionality and makes it easy to adopt the interface to different tasks or user's needs or preferences.

Widgets will realize different visualization demands such as:

- navigating an ontology fragment with associated resources
- a social network graph showing connections between users
- search the social network for resources or friends and display the results as a list
- show the definition of a concept

In order to support the visualization of graph-based visualizations an extra dependency is required in order to display and manipulate this type of visualization. We have opted to use the java-based visualization suite called VUE for this goal. VUE is to be embedded into one or more widgets when graph-based visualization is needed. VUE allows us to search, modify, add, auto-layout and navigate graph-based visualizations which can include graphical images. VUE can import existing VUE graphs (a custom XML-based format) or existing RDF-files, though the automatic visualization capabilities for that type of data is sub-optimal. The VUE-component which resides in the view layer will communicate components from the model layer through REST-style web service communication for graph acquisition and feedback.

Controller Layer

The application logic in the controller layer takes care of the interaction with the view and the data model. The main job of the controller layer is to integrate various information sources in order to pass this information on to the view layer. In order to

integrate the information sources a great variety of tasks resides in the controller layer which have no direct visual feedback associated with them, these components have been moved to the processing layer. The controller layer components that are exported as web-services so that they are available to the view are:

- *Visualize Resource Documents* – generating graphs from various information sources (ontologies, resources, related tags) for the view layer
- *Lexicon Management Service* – managing lexica (already described by Task 6.1)
- *Crawler Management* – managing the actions of the crawler (start/stop)
- *Crawler scheduling* – managing the tasks of the background search processing
- *Folksonomy Query Service* – managing the queries coming from the users through the widgets and send them to the data processing layer for running the actual search queries
- *Ontology Management Service* – storing, loading, filtering and querying ontologies for various properties
- *Document Annotation Service* – adding metadata to resources (already described by Task 6.1)
- *Store Resource Documents* – storing submitted resources in the repository (already described by Task 6.1)

Example of the preparation of an ontology-backed visualization in the controller layer (cf. also the WP 6.2a scenario):

- specify ontology-selection (using Ontology Management Service described by Task 6.1)
- associate resources with selected concepts through the Folksonomy Query Service
- convert resources, concepts, lexicalisations to a special purpose VUE-graph with the Visualize Resource Documents service
- the VUE-graph is passed to a component in the view layer (VUE)

The Ontology Query Service extracts a portion of an ontology which is then augmented directly with lexical items extracted from the lexica. This is then fed to the VUE-generator which generates a special purpose VUE-graph which is backed by an ontology fragment (concepts and the relations to another), resources associated with the tag and possible lexicalisations (synonyms). The VUE-file is then passed onto the view layer. At this point all of the tasks mentioned in this section are realized by java-objects calling native methods. We envisage that we can split all the various components into web services which allow for greater flexibility. The controller will export its functionality through either REST-style webservice with XML or JSON used for data exchange or SOAP for more complicated interactions.

Processing Layer

Components which are present in the data processing layer perform functions that run somewhat autonomously from the controller and or view. This often involves computation or data aggregation that take too much time to generate results, run continuously or don't have direct visual feedback. Processes that are computationally intense also reside in this layer (examples include term-document-vector calculation that gets stored as sparse matrices) and the search algorithms that create indices on various data.

Examples of components that reside in this layer include:

- a crawler which runs continuously in the background.
- a keyword extractor (Lemnitzer et al. 2007, Killing et al. 2008) which extracts important terms from resource documents (extracting keywords is too computationally involved for direct user feedback through the view)
- ontology enrichment component
- loading ontologies through the Ontology Management Service (or alternatively: generating a shallow ontology from social networking data)
- load lexicon and associate lexical entries with concepts in ontology (using Lexicon Management Service described in task 6.1)
- calculating related tags using cosine similarity and cooccurrence-based methods and storing these in the model

For providing accurate search results some work has to be done independently from the search queries in the background. This, for example, involves extending the ontology with new concepts and relations coming from the social media aggregated by the crawler. This is done by analyzing, clustering and sorting the existing information in the data model in order to enrich an existing ontology. All the data that is calculated in this layer will be stored in the same model layer which is shared with the controller. This allows for immediate use by the controller and prevents further coupling between the controller and processing layer.

Architecture Example

We will illustrate the complete architecture by considering the crawler and associated services (cf. also scenario WP 6.2b).

The crawler works by extracting data from social networking applications using web service APIs provided by these applications. The simplified architecture for the crawler is presented in Figure 11 below. The data is stored in the data model described in a previous section.

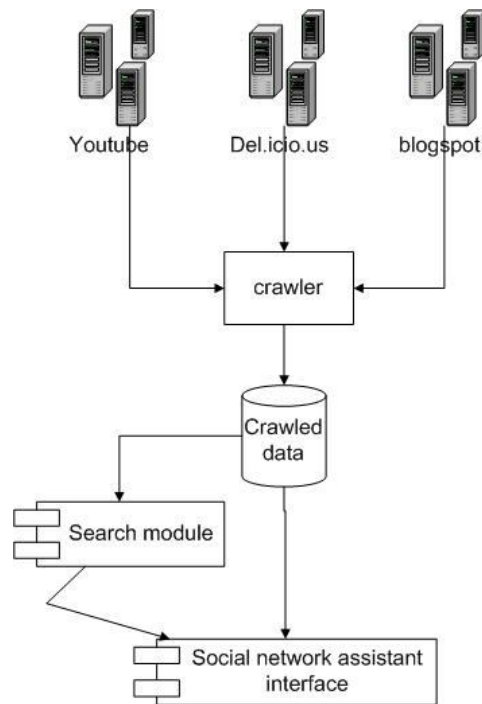


Fig. 11: Simplified architecture of the crawler

When applying the augmented MVC-pattern on this crawler-specific architectural picture it becomes clear at which layer each of the components reside. The 'crawler' in the processing layer, 'Crawled data' in the model, 'Search module' in the controller and the 'Social network assistant interface' in the view layer. The diagram has been slightly simplified so that the controller which exists between the 'Crawled data' and the 'Social network assistant' is left out, but in reality it is present.

5. Overview on Showcase Characteristics

	T4.1	T4.2	T5.1	T5.2	T6.1	T6.2
Encapsulation	well-defined interfaces	well-defined interfaces	well-defined interfaces	partly	partly	well-defined interfaces
WP2 Integration	full	full	easy possible	possible	possible	easy possible
Documentation	partly	partly	partly	partly	nearly full	
License	fully open source	fully open source	partly open source	partly open source		
Client layer						
Widgetisation	easy possible	easy possible	possible	possible	adaptations needed	easy possible
Visualisation	HTML, CSS, JavaScript	HTML, CSS, Flash	HTML, CSS, Java Applet	HTML, CSS, JavaScript		
Service layer						
Servicification	full	easy possible	possible	adaptations needed	adaptations needed	easy possible
Technology	XML, REST, AJAX	XML, REST over HTTP	XML, REST	REST over HTTP	VUE, XML, REST	SOAP, WSDL, XML
Application logic layer						
Programming language	R, PHP	R, Flare	C#.NET, Java, Python, PHP	C, PHP		Ruby, Java
Storage Layer						
Technology	MySQL	file system, MySQL	XSL, text files, XML	text files	XML	MySQL

6. Development Strategy

6.1 Position of Software Development within the Project

Development Lifecycle

As this project’s software outcomes will be very heterogeneous in their implementation characteristics it is important to define a suitable development strategy for this task. Within the first deliverable D2.1 the basic software development and release process for the whole project period was defined. Enhancing this first approach one step further, Fig. 12 displays a simplified project lifecycle, emphasising on the software development process important for the proposed development strategy (bold items represent development specific tasks).

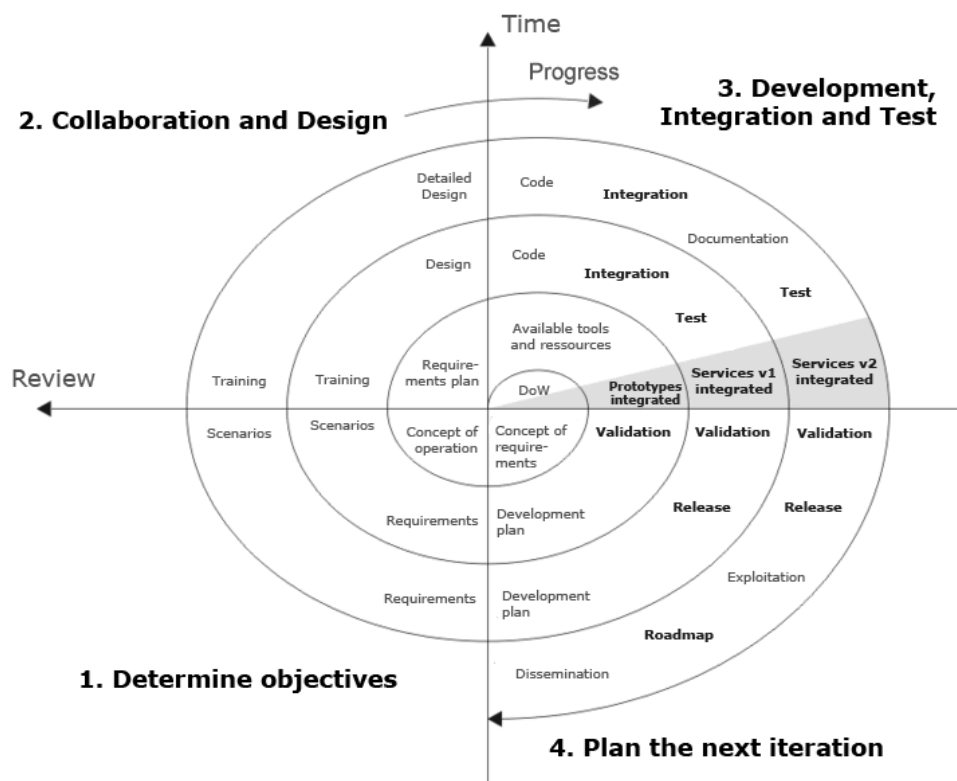


Fig. 12: Simplified project lifecycle emphasising software development process; adapted spiral model from (Boehm, 1988)

As specified in deliverable D2.1 software releases are defined to be prepared in line with the submission of the deliverables of the different work packages. With respect to WP 2 this means, that the project's lifecycle is an iterative process having altogether three main loops: (1) 'Existing services (showcases) integrated', (2) 'Services v1 – integrated' and (3) 'Services v2 – integrated'.

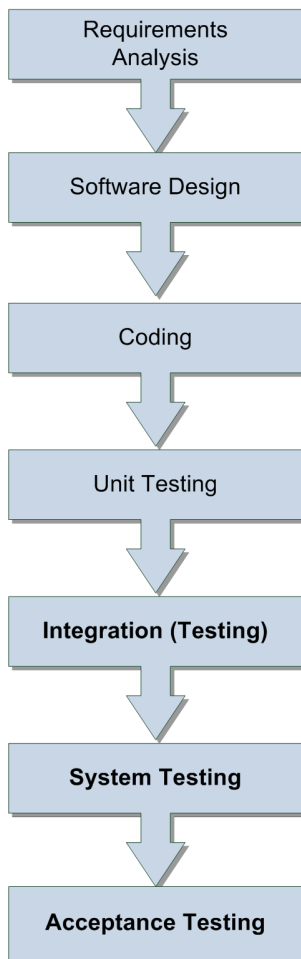
The (1) initial iteration has its starting point from the Description of Work (DoW) and defines first software requirements which are met at large in the architecture, design, and implementation of the prototypes reflecting the different partners' showcases. After the technical validation, the outcome is a revised development plan with adapted requirements which are influencing the scenario-based design process. Being now in the (2) second iteration a new software design is elaborated causing new development of software products which have to be tested and integrated in the general WP2 infrastructure. After having a stable first version of developed services, a product release is planned freezing the development at that stage. The technical validation is done like in the first loop, resulting in a new development plan and therefore emerging adapted requirements. As it can be assumed – being in the (3) last iteration – that the software development process has become as stable as the software outcomes, a last detailed design phase is the initial task for the final development, integration, and testing phase resulting in a second version of developed software products. At the end of the project a final release is done having well tested and stable software products which can be integrated in existing applications as defined in the requirements.

Software Development Process

It can be assumed that coding of applications is mostly done in-line with typical looking processes well known in software development. By having a look at Fig. 13 a software development process can be seen focusing on test and integration tasks (bold).

Fig. 13 can be seen as a sub-system of the entirely development process throughout the whole project. This means that each stage of the development process has to be more or less run through in each iteration of the project lifecycle.

By having a closer look at Fig. 13 all development starts with a requirements analysis, followed by defining the functional specification and the software architecture and the system design. Then the actual coding of software parts takes place. Unit testing can be seen as the first stage of the dynamic test process and is the smallest testable part of an application. Until now all development specific issues are done more or less by the different WPs themselves.



After having assured that the individual software components are fit for use, the application is integrated within the WP2 infrastructure. Along with the integration, testing takes place to ensure that the program can run on WP2’s infrastructure and to collect requirements for and guarantee transferability and interoperability.

Yet, it is not finally decided if the project’s software outcomes should consist of a single coherent system or of individual software components loosely coupled together. By sticking to a widget based design approach, this decision can be postponed to a later point in time when the different software parts are more evolved (e.g. after validation of stage ‘Services v1 – integrated’). This software architectural decision affects very much the kind of system tests performed.

The last stage of the software development process forms acceptance testing, which is done using at the one side a technical validation and at the other side a stakeholder driven qualitative validation approach.

Fig. 13: Adapted and simplified software development process

6.2 Development Workflow

As for the actual development, a process is defined which ensures an easy way to develop, share, integrate and test different software components. The process has to ensure that every partner can build software according to his/her preferences but has to stick to a minimal set of general rules which guarantee good communication and cooperation. This is important because every partner will have to collaborate with others at a certain stage in the development lifecycle.

Therefore, two separate processes are defined meeting the needs of software developers. It is distinguished between (1) partners who uses WP2 infrastructure for developments and (2) partners who uses external infrastructure.

Developments on WP2 Infrastructure

Every developing project partner is approved to have access to the WP2 development and test infrastructure. WP2 ensures that partners are granted appropriate access to suitable programming environments meeting their requirements. Therefore, project partners can develop their software components on a ‘common and typical’ server infrastructure to guarantee at least a minimum level of transferability and interoperability. For developments following this approach a workflow has been defined which is displayed in Fig. 14¹. There a typical development process is shown starting with an initial integration of already existing software at project partner’s side and ending with a new stable package release consisting of an up and running service on the WP2 live system. Light blue activities intended to be in the scope of project partner WPs, whereas light red activities are duties of WP2.

The initial integration of existing software has to follow the workflow of external infrastructure developments to ensure to have an already integrated system for further developments (partner WPs and WP2 involved). The software can then be extended by the project partner resulting in a new revision (e.g. at the end of a day) which units have to be tested to verify internal correctness. After successful unit tests a commit to the global SourceForge repository is done. This development cycle is iterated until a major revision is deployed. As software developments are done already on the WP2 infrastructure integration tests are done automatically by partner WPs (otherwise components would not operate correctly). Major revision of software outcomes are also tested for their validity by WP2. After confirming a correct major revision it is stated as being an integrated revision. If any error occurs which cannot be solved by WP2 partner WPs are informed which have to adapt the software.

As stated in former sections software releases are defined to be done in line with the submission of the deliverables of the different WPs. This minimum requirement must be met, but there is nothing to say against a software release between two deliverables, if a new stable version is deployed and satisfactorily tested. This means if a major revision is stated stable by the involved WPs a new package release can be instantiated. For this reason WP2 will deploy the already successfully integrated software on the test system also on the live server. If the installation and configuration fails for any reason, partner WPs are informed which can adapt the software with the help of WP2. As this adaptation will result in a new revision, it has to pass through all the stages of testing once again. At

¹ For all activity diagrams displayed in this paper it is assumed that the project partner has already been granted access to the WP2 infrastructure and that the required development environment is set up. Therefore, this process is not illustrated in the corresponding figures.

the end of the development process a correct working service should be up and running on the live server and being accessible by everybody from the outside.

At every step in the development process documentation of the software has to be done. But to ensure a traceable development process protocols of this work have also to be maintained. More information can be found in section 7.

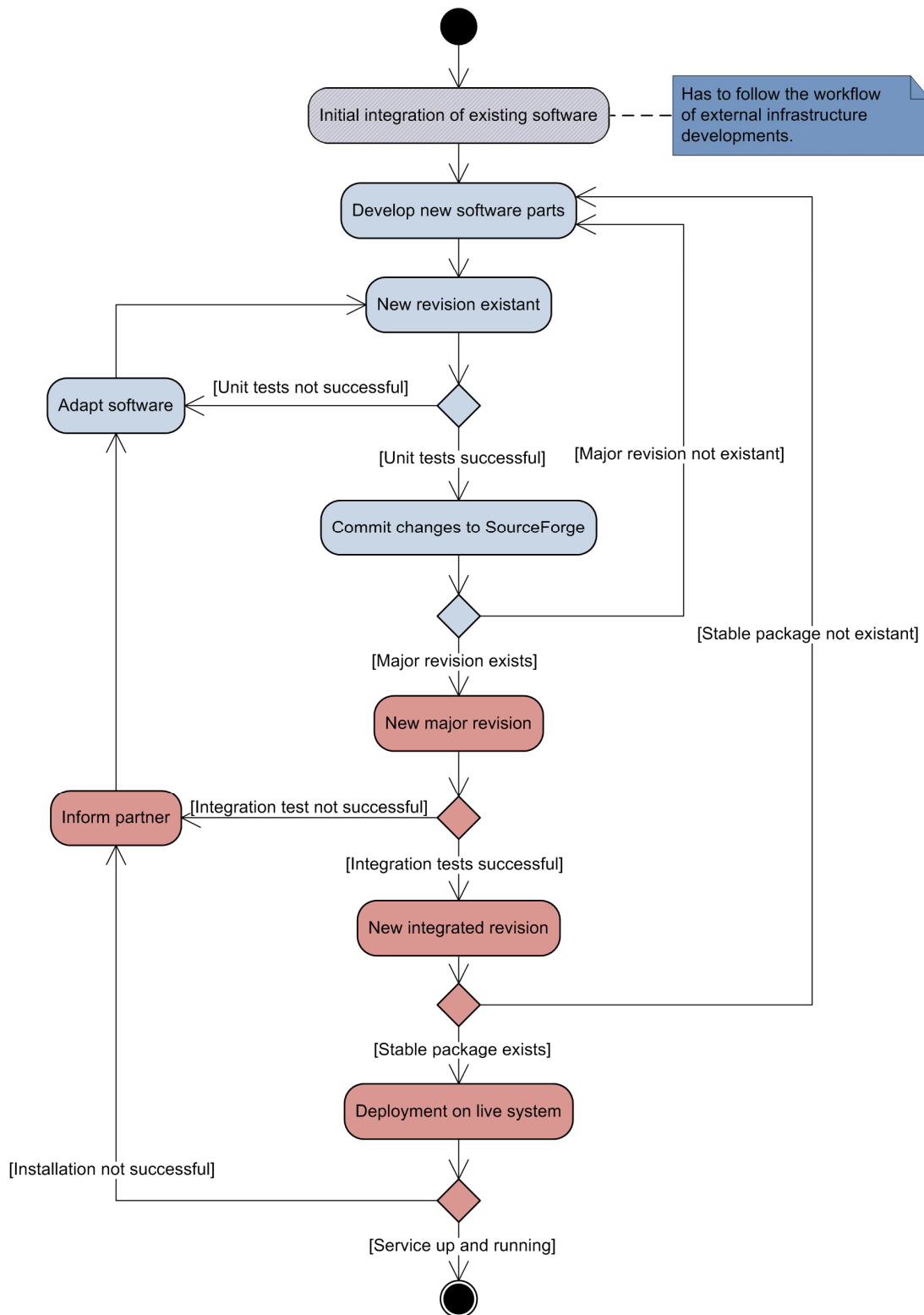


Fig. 14: Activity diagram of workflow for developments on WP2 infrastructure

Developments on External Infrastructure

Some partner WPs feel the need to develop software on their own infrastructure. Therefore, for them the WP2 servers are not the best choice as a programming environment. It should be no problem that partner WPs deploy programs on their own as long as it can be guaranteed that developed software systems can run on WP2's infrastructure to test their transferability.

The workflow of developing software on external infrastructure can be seen in Fig. 15. Again light blue activities belong to partner WPs, while light red activities are in the scope of WP2. The development of new software parts and unit tests are all done on partner WPs infrastructure. That means that versioning and track changes are not obligatory and every partner WP has to take care of backing up their programming code on their own.

If a major revision has been developed the partner WP commit it to the main SourceForge repository. WP2 checks-out/update the version on their infrastructure and perform integration tests to ensure that the software works in their environment. As these integration tests are likely not to be performed periodically, errors will occur and WP2 will inform the partners if they cannot fix the error by themselves. The partner WP has to adapt the software and testing procedures will start again.

If the major revision is stated as a stable package – and of course after successful integration tests – a package release on the live server is done like in the workflow of developing on WP2 infrastructure in the former chapter. Again, at the end of the integration process a correct working system should be up and running on the WP2 live server.

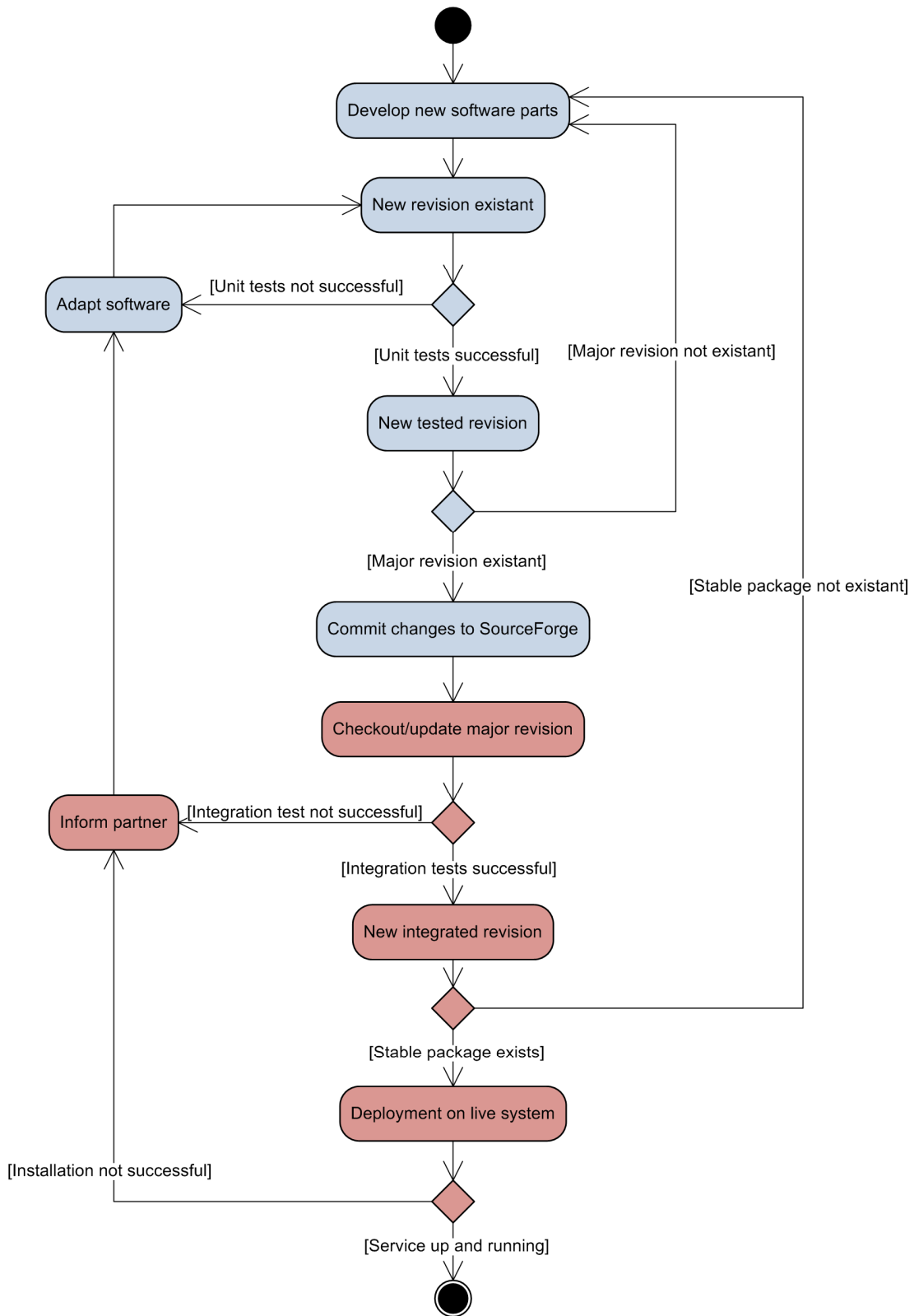


Fig. 15: Activity diagram of workflow for developments on external infrastructure

7. Versioning Policy

1) Check-in code to SourceForge by project partners:

The code checked in to the SourceForge repository should include a folder named `docs`, which contains an installation manual (`install.txt`) and a test manual (`test.txt`). For more information about the installation and test manuals follow the instructions in steps 2 and 3.

2) Installation manual (`install.txt`):

The installation manual (`install.txt`) describes how to install the service in a stepwise manner. It may contain instruction parts such as ‘Requirements’, ‘How to install’, ‘Troubleshooting’, etc. The steps described in the installation manual should be clear enough so that a technical staff that is not familiar with the corresponding service is capable of performing a working installation successfully. If the code to be installed requires libraries or packages not existing on the infrastructure, these should be listed in the ‘Requirements’ part of installation manual. The required packages should be specified in detail to simplify the installation process.

3) Test manual (`test.txt`):

The test manual (`test.txt`), describes a stepwise procedure to test the service that is already installed. The goal is to verify the service manually in order to ensure that the service meets the functionality and works properly on the production system. It should contain detailed instructions about the inputs, commands and operations to be executed during the test and the expecting output of the service. In order to ensure the successful integration to the production system the infrastructure partners (BIT MEDIA & WUW) offers detailed information about the development environment and the server hard- and software.

4) Check out the code by infrastructure partners:

The infrastructure partners (WP2) are informed automatically through the email notification of SourceForge when the project partners check in their code. If the code does not include the required installation and test manuals described in step 1 the project partner is informed by infrastructure partners about the missing files. The corresponding partner is then asked to complete the code according to instructions in step 1 and check it in again to SourceForge.

5) Service installation by infrastructure partners:

WP2 will try to install the software following the instructions in the installation manual as described in step 2. In the case of installation failures an error description is sent back to the partner, who offers the service. The partner is asked to update the installation manual or code, check it in again and offer a cooperative support to the infrastructure partners in order to achieve a working installation on the infrastructure.

6) Service test by infrastructure partners:

If the service could be installed successfully, WP2 will try to test the installed service according to the instructions in the test manual described in step 3. If the test fails or cannot be executed, an error description is sent back to the partner, who offers the service. The partner is asked to update the test manual, or fix the errors, check in the code again and offer a cooperative support to the infrastructure partners in order to achieve a working service on the infrastructure.

7) Final success report to project partners:

If the service is installed and tested successfully, WP2 informs the partners about the successful installation and testing of the service.

References

Berry, M. 1992. Large Scale Singular Value Computations, *International Journal of Supercomputer Applications* 6:1, pp. 13-49.

Boehm, B. W. 1988. A Spiral Model of Software Development and Enhancement. *Computer*. 1988, Vol. 21, 5.

Feinerer, I. and Hornik, K. and Meyer, D. 2008. *Text mining infrastructure* in R. *Journal of Statistical Software*, 25(5):1-54, March 2008.

Horner, J. 2009. rapache: *Web application development with R and Apache*. [Online] 2009. [Cited: February 26, 2009.] <http://biostat.mc.vanderbilt.edu/rapache>.

Killing, A. and Spousta, M. and Holtmann, H. 2008. *D4.1c ILIAS LMS with integrated functionalities and documentation – second cycle*. Deliverable of the LT4eL project. <http://www.lt4el.eu/extern/files/m30/D41cFINAL.pdf>.

Lemaire, B. and Dessus, P. 2001. A system to assess the semantic content of student essays. *Journal of Educational Computing Research*. 2001, Vol. 24, 305-320.

Lemnitzer, L. and Vertan, C. and Killing, A. and Simov, K. and Evans, D. and Cristea, D. and Monachesi, P. 2007. Improving the search for learning objects with keywords and ontologies. *Lecture Notes in Computer Science*, no. 4753: 202-216.

Mika, Peter. 2007. *Ontologies are us: A unified model of social networks and semantics*. *Web Semantics: Science, Services and Agents on the World Wide Web*.

Oehlschlägel, J. and Adler, D. and Nenadic, O. and Zucchini, W. 2008. *A first glimpse into 'R.ff'*. <http://www.statistik.uni-dortmund.de/useR-2008/slides/Oehlschlaegel+Adler+Nenadic+Zucchini.pdf>.

Rohde, D. 2009. svdlibc, <http://tedlab.mit.edu/~dr/svdlbc>.

Wild, F. 2009. *lsa: Latent Semantic Analysis*, R package version 0.61