

GRAPPLE

D1.3a Version: 1.0

"Stand-alone" Adaptive Learning Environment Specification

Document Type	Deliverable
Editor(s):	Kees van der Sluijs
Author(s):	Paul De Bra, David Smits, Evgeny Knutov
Internal Reviews:	Kai Hoever (IMC), Frederic Kleinermann (VUB)
Work Package:	1
Due Date:	01-02-2009
Version:	1.0
Version Date:	24-02-2009
Total number of pages:	33

Abstract: This deliverable describes the architecture of the GRAPPLE stand-alone adaptive learning environment, or GALE for short. GALE has an architecture consisting of adaptation, user modelling and domain modelling components that are built around an *event bus* with which all communication can take place using SOAP messages. The adaptation process consists of a pipeline of "processors" and "modules" that can be easily extended to add new adaptation functionality. Extensive caching is used to ensure adequate performance (short response times). This deliverable also describes how a large part of GALE can be connected to the GRAPPLE infrastructure to enable the non-stand-alone use of GALE's adaptation, user modelling and domain modelling components.

Keyword list: adaptive learning environment

Summary

This deliverable describes the Stand-Alone Adaptive Learning Environment of GRAPPLE, or GALE for short. It also describes a large number of interfaces needed by authors and designers in order to configure the adaptation and presentation to their own desires, and in order to know how to extend GALE with new functionality. Since a large part of the stand-alone environment consists of the Adaptation Engine which is common between the stand-alone adaptation environment and the environment that can be used in combination with an LMS this deliverable also describes the core adaptation part of the GRAPPLE infrastructure as a whole.

In order to accommodate all current and future adaptation functionality requirements the adaptation engine is highly extensible and configurable. This deliverable shows the structure of "processors" and "modules" used to perform the adaptation, and it explains the "event bus" that is used for communication between functional parts of GALE, namely the adaptation engine, the user model service and the domain model service.

This deliverable also explains how to install GALE and set up and configure a first example course.

The deliverable starts with a description of different architectures and functionality of adaptive hypermedia and learning systems, so as to provide a context for understanding the history and background of the design choices made for GALE.

Authors

Person	Email	Partner code
Paul De Bra	debra@win.tue.nl	TUE
David Smits	d.smits@tue.nl	TUE
Evgeny Knutov	e.knutov@tue.nl	TUE

Table of Contents

SUMMARY 1

AUTHORS 2

TABLE OF CONTENTS 2

TABLES AND FIGURES..... 4

LIST OF ACRONYMS AND ABBREVIATIONS 4

1 INTRODUCTION 5

2 A COMPARISON OF ALE ARCHITECTURES 6

2.1 Domain Model Comparison 6

2.2 User Model Comparison 8

2.3 Adaptation Model (Adaptive Engine) Comparison 11

3 THE CORE ALE ARCHITECTURE 13

3.1 The GALE components 14

3.2 The "process" of handling a request for a concept 15

3.3 Configuration file hierarchies 17

3.4 Defining the Processor pipeline 19

3.4.1	Default variables in the resource	20
3.4.2	Default variables in the servlet context.....	21
4	PROCESSOR DETAILS	21
4.1	Helper Processors	21
4.2	The XMLProcessor (10-40).....	22
4.3	The LayoutProcessor (0-1)	22
4.4	The FrameProcessor (10-50)	23
4.5	The Plug-in Processor (0-0).....	23
5	COMMUNICATION WITH THE DOMAIN MODEL AND USER MODEL.....	23
5.1	The EventHash class.....	24
5.2	The GALE DM service	24
5.3	The GALE UM service	25
6	AUTHORING GUIDE	26
6.1	Installing GALE as a stand-alone ALE.....	26
6.2	Authoring ConceptConfig.xml files	27
6.3	Configuring the Plug-in Processor	28
6.4	Configuring the LayoutProcessor and FrameProcessor	28
6.5	Configuring the XMLProcessor.....	29
6.6	GALE expressions.....	31
	REFERENCES	32
	APPENDIX A: DIFFERENCES BETWEEN GALE DESCRIPTION AND FIRST PROTOTYPE	33

Tables and Figures

List of Figures

Figure 1: Core GALE architecture 13

List of Tables

Table 1: Summary of Domain Model properties 7
 Table 2: Summary of Domain independent and Domain dependent User Model properties 10
 Table 3: Adaptation Model (Adaptive Engine) properties comparison table 12
 Table 4: Events supported by domain model services 25
 Table 5: Events supported by user model services 26

List of Acronyms and Abbreviations

AHA! (or AHA)	Adaptive Hypermedia Architecture (also used as prefix for other terms)
ALE	Adaptive Learning Environment
AM	Adaptation Model
CAM	Conceptual Adaptation Model
DM	Domain Model (this includes the Adaptation Model)
GALE	GRAPPLE Adaptive Learning Environment
GRAPPLE	Generic Responsive Adaptive Personalized Learning Environment
LMS	Learning Management System
SOAP	Simple Object Access Protocol
UM	User Model

1 Introduction

The core of the adaptive functionality in GRAPPLE is delivered through what we call the **GRAPPLE Adaptive Learning Environment**¹ (or GALE). This deliverable describes not only GALE itself but also the components that enable GALE to be used either within the larger GRAPPLE framework or **stand-alone**, without connection to the framework or to a learning management system (or LMS). GALE draws from previous research into adaptive learning from different GRAPPLE partners. The core architecture is based on an (almost) complete rewrite of AHA!, the Adaptive Hypermedia Architecture that was developed at the TU/e. The most recent public description of AHA! that can be found in [2] still refers to the monolithic AHA! version 3, whereas GALE has a modular architecture and the adaptation flexibility and extensibility needed for GRAPPLE. GALE is sometimes also referred to as AHA! version 4, and a number of modules, variables or methods have names starting with AHA or aha for historical reasons.

GALE is centred around an **event bus**. GRAPPLE components (including components of the ALE itself) can send events to the event bus and can listen for events that are sent to the bus by other components. An event can be a small/simple message like that a user has accessed a certain concept of an application (or course), and can be as large as posting the complete part of the user model of the current user related to an application. The communication with GALE's event bus can be done through SOAP messages. (This is a configuration option). In a stand-alone setting (without LMS or other GRAPPLE components) the communication overhead of SOAP can be avoided.

Many adaptive learning applications have been developed in the past. Although the research has focused on the adaptive methods and techniques that were used (and their benefit for the learning process) these research prototypes were also characterized (and remembered) by their specific look and feel. In GRAPPLE it is important to give application/course developers maximum freedom in deciding not only how the adaptation works but also in the design of the look and feel of their application. Although (to keep authoring simple) GALE comes with a predefined presentation and adaptation template, it is also reasonably easy for authors to create their own look and feel for their applications without making their learning material depend on that look and feel. This document describes how to create your own template(s).

The presentation/adaptation flexibility is achieved through two sets of configuration files. At the conceptual level the configuration files decide how to adapt and present concepts of a specific application (at any position in the conceptual structure). At the content level each directory containing resources (files or pages) can contain a configuration file to decide how to adapt and present the resources in that directory. It is thus possible for authors and course developers to share a GRAPPLE server infrastructure while deciding independently on the desired adaptation and presentation of their course material.

GALE needs to be able to serve adaptive learning material to users no matter how they approach GALE, be it directly or through an LMS. The current plan for GRAPPLE is to allow users to log in and identify themselves through a single-sign-on infrastructure. To anticipate different ways of user identification/authentication the "login manager" is a separate module of GALE. An administrator can configure GALE to indicate which login manager (implementation) to use. When an external single sign-on infrastructure (such as Shibboleth or OpenID) is used access to GALE is completely transparent. When no external service is used a login manager is available that presents a simple authentication form. When the user accesses adaptive information content (like a page or concept) the browser first presents a login form, and after filling that out the requested content is presented.

Last but not least, the issue of performance is crucial in GALE. Adaptation is performed between the learner submitting a request (asking for a course page or for the evaluation of a test for instance) and receiving an (adapted) reply. In order to achieve sub-second response times GALE needs to have all the information needed to perform the adaptation readily available. In GRAPPLE this information roughly exists in two places: in GALE and in other "parts" of the GRAPPLE environment. In GRAPPLE there may be several databases containing information about a single learner. An LMS may keep track of completed courses (or course sections and tests). The LMS may also be involved in part of the evaluation of the learner's knowledge. Some external applications may also be used to train specific skills, etc. The GRAPPLE infrastructure offers connectivity between such components, and offers storage of and reasoning over user information that is stored in a distributed way. But such information is not *instantly* available and can thus not be requested and obtained in the short period of time between the learner's request and GALE's reply.

¹ In the future this might be renamed to Generic Adaptive Learning Environment.

GALE solves this through a caching mechanism. Information coming from different components of GALE itself is cached, and after a request GALE waits for responses from its internal components (like a user model or UM component). Communication with GRAPPLE components outside GALE is done asynchronously: requests are sent and responses result in cache updates, but information that is needed for the adaptation is always taken from the cache without waiting for a reply from an external component. Application developers should take this "lag" in the availability of information from non-GALE components into account as much as possible.

2 A comparison of ALE architectures

Before explaining the architecture of the ALE developed for GRAPPLE (GALE for short) we present the outcome of a comparative study of adaptive learning environments (or adaptation engines) so as to identify the commonalities between these ALEs that should be present in the GALE. We perform this comparison for the three main parts of an adaptive application, according to the AHAM reference model [1]: Domain Model, User Model and Adaptation Model.

2.1 Domain Model Comparison

Each adaptive application must be based on a Domain Model (DM), describing how the conceptual representation of the application domain is structured. This model indicates relationships between concepts and how are they connected to content presentation in terms of fragments, pages, chapters, information units, pagelets or any other structure encapsulating information about a concept.

The domain model of an adaptive hypermedia application usually consists of the following components: concepts and concept relationships. A concept represents an abstract information item from the application domain. In all systems the concepts form a hierarchy. As a result each concept can be either an atomic (primitive) concept or a composite concept that has children concepts (sub-concepts) and a description of how do they fit together.

For example in Interbook [3] a textbook is structured as a hierarchy of chapters and sections with atomic presentations, tests or examples. The pages (and sections) are connected to a structure of concepts, indicating for each page what required (prerequisite) knowledge there is for the page, and which outcome concepts the page teaches something about. In KBS Hyperbook [5] the system uses a knowledge base which consists of so-called 'Knowledge Items' which are essentially concepts. In this respect each document from the document space is indexed by some concepts from the knowledge base which describe the content representation and hierarchical structure. In APeLS [4] the concepts are encapsulated into a 'Narrative' structure where each narrative it can be hierarchically split into sub-narratives.

Each system proposes its own way to encapsulate content information: in a form of a Pagelet (in APeLS), which contains content and a content model, or it may be an Information Unit just encapsulating content information as in KBS-Hyperbook. And these Information Units are indexed to map the Knowledge Items structure. In the AHAM model [1] and in the AHA! system [2] content representation is based on pages, which are the units of information presented to the user for each interaction. Pages consist of fragments, what can be conditionally included (but which cannot be changed) by the AHS and which represent the lowest level in the concept hierarchy.

A concept relationship is a meaningful connection between concepts. In AHAM it is represented as an object with a unique identifier, attribute-value pairs, presentation specification and a sequence of (two or more) specifiers which represent tuples of concept and anchors IDs, direction of relationship and presentation specification. Each concept relationship has a type (e.g. direct link, inhibitor, 'part of' or prerequisite). Such a Domain Model structure representation captures the types of relationships that can be encountered in most AHS systems. In KBS-Hyperbook one may see the dependency graph of all the KI's (knowledge items), in AHA! there are binary relationships of arbitrary types, and in APeLS there is a form of relationships map in a Narrative Model, by which adaptive logic is represented.

In Table 1 we present a summary of Domain Model properties of each system (model): its generic definition (is the second column) followed by how it appears in: AHAM, AHA!, KBS-Hyperbook, APeLS and Interbook.

Table Legend:

We should acknowledge that each row in the table envisions a comparative description of a particular system property or aspect which we consider represent more or less the same system functionality, on the other hand table shows all the differences both in approach, implementation or a set of properties provided

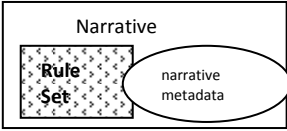
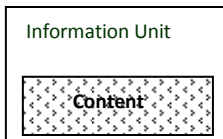
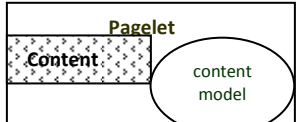
		AHAM	AHA!	KBS Hyperbook	APeLS	Interbook		
concept	Concept	an abstract representation of an information item from the application domain	<p>Concept information:</p> <ul style="list-style-type: none"> - <attribute, value> pairs - sequence of anchors - presentation specification <p>atomic concepts - represent single fragment of information, composite concepts use child attribute to specify sequence of composite concepts</p>	<p>Concepts like in AHAM with restrictions also have type; and associated with a template, (can have only fixed number of attributes)</p>	<p>Knowledge Item (KI) - abstract representation of domain knowledge (e.g. if, class, run_method)</p> <ul style="list-style-type: none"> - may also be a compound structure 	<p>Encapsulated in Narrative Model Metadata</p> <p>each narrative may add a new Concept and corresponding Narrative Rules</p> 	<p>Glossary Entries = Domain Concepts</p>	
	Concept relationship	represents semantic relationship between concepts	<p>Authored semantic linking between concepts in a form of: <C1,C2,T,A> (T=type) - link; prerequisite; inhibit; part (compositional) (A = attribute value)</p>	types of relationships: Fragment / Link / Contain	<p>Dependency graph of the KI-s</p> <p>=> semantic links between Information Units (IU)</p> <p>Each IU is connected to one or more KI presenting which concept represents corresponding content in IU</p>	In a form of a relationships map in the Narrative Model	<p>Concept relationships (<i>navigational paths between glossary items</i>) types:</p> <ol style="list-style-type: none"> 1) First-Page, 2) Sub-Section, 3) Domain_Concept, 4) Bookset 5) Loginpage, 6) Requirement, 7) Outcome, 8) Fragment 	
	Indexing	explicit indexing options: mapping concepts, projects, etc.	n/a	n/a	<p>Knowledge Items (KI)</p> <ul style="list-style-type: none"> - index Project Units and Info. Units 	n/a	<p>Glossary Entries index Domain Concepts are indexed on textbooks (bookshelves)</p>	
content	Content data presentation	content unit structures	Pages and Fragments (page may consist of several fragments)	Pages and Fragments	Information Units (IU)		<p>Content info. is presented in a form of a Pagelet which may belong to a certain content group (see below)</p> 	<p>Content info is presented in a Textbook (shelf of textbooks)</p> <p>Glossary (glossary entries provide link to a certain textbook and connection to a certain domain concept)</p>
	Content grouping	content grouping according to similarity of presentation, objectives, etc.	n/a	n/a	<p>Project Units are mapped on Information Units</p> <p>Project Unit defines a number of KI has to be learnt to fulfil project goal</p>	<p>Content Group (content pagelets are organized in a group fulfilling the same Learning Objective LO)</p>	n/a	
	Storage	content storage part of the AHS	Within-Component Layer	Within-Component Layer	Domain Model	Content Domain	Textbooks / Textbook shelves	

Table 1: Summary of Domain Model properties

by each system, and a difference or similarity in terms used to describe system functionality. E.g. "Content Grouping" presented in APeLS or KBS Hyperbook is implemented either in a way of grouping similar content chunks in APeLS content groups or grouping a sequence of concept and associated content in Project Units fulfilling similar user objectives. At the same time the "content storage part" property may show that different systems store content information in different systems sub-components, either encapsulating concept and content structures in one model or trying to separate them in order to provide content reusability facilities.

2.2 User Model Comparison

The User Model (UM) has to be created and kept up-to-date to represent user knowledge, interest, preferences, goals and objectives, action history, type, style and other relevant properties that might be useful for adaptation. Some systems also look into the environment in which an application is used, device properties, work context, etc.

UM properties are usually separated into domain dependent and domain independent properties. The user typically has as domain-independent properties an identity, name, password, all with simple (atomic) values, but UM may also have more complex properties such as a collection of groups the user belongs to, preferences, a number of learning styles, work environment, and so on. The domain-dependent properties of UM (for a given user) typically consist of some entities, objects or concepts, for which we store a number of attribute-value pairs. For each entity there may in principle be different attributes, but in practice most entities will have the same attributes. Therefore, to implement UM we may use a table structure, in which for each entity the attribute values for that entity will be stored.

As domain dependent properties we see that most entities in UM represent concepts from DM, forming overlay over DM, mapping the user's domain-specific characteristics like knowledge of concepts over the domain knowledge space. There may be more Domain dependent properties like: test results, learning objectives which can be problem solving tasks or short term objectives. Typically these need to be represented in DM as well in order to make use of them for adaptation. Thus, even for properties like learning goals the UM will be an overlay of DM, however not all Domain dependent properties should necessarily belong to an overlay, there can be aggregation properties like an "average knowledge" or auxiliary like "has seen introduction page", which are difficult to express in UM as an overlay.

In the table below goals and objectives are separated from the domain dependent user properties in the sense that they are treated as separate instance (sometimes even a separate Goals layer) which strictly deal with the goal representation, statement of the goal and its mapping on a concept structure.



User Goal / objectives		AHAM	AHA!	KBS Hyperbook	APeLS	Interbook		
		User follows a link to a (different) page	User follows a link to a (different) page	- for Direct Guidance and - for Goal based Learning: Knowledge Items (KI) to be learnt are selected by user Goal (with triggering event for AE) consists of KI array - for Project based learning: Goal and Project repository	Learning Objective - state the goal of learning procedure	User stated/assigned learning goal		
User Goal statement	Goal statement by the user	n/a	n/a	1) User defined 2) Proposed	1) User defined	1) User defined		
System internal objective	Goal interpreted in terms of Adaptive Engine and DM	Concept to learn (one step at a time) (stated with triggering event for AE)	Concept to learn (one step at a time)	Project (consists of project units mapped on KI) or KI to learn for Guidance tour to reach a certain goal	LO is mapped to a certain content group that has to be learned (decision on LO can be done runtime (based on learner and environment info))	represented as a set of concepts to be learned		
properties	domain independent	user common static parameters	yes	yes + authored attributes	yes	yes	yes	
		experience / background	n/a (not stated explicitly, but can be considered and expressed in UM)	n/a (not stated explicitly)	n/a	n/a	n/a	n/a
		preferences (font types, pictures, examples, size, etc.)	n/a	Link colouring (default or defined)	n/a	n/a	n/a	n/a
		cognitive/learning style	n/a	Can be authored (not offered as a default option)	n/a	n/a	Supported via Narratives (each Narrative supports different pedagogical approach dealing with the same course meeting the same LO)	n/a
		explicit user environment settings (time, place, etc.)	n/a	n/a	n/a	n/a	(e.g. device dependent narratives mentioned)	n/a

domain dependent	Knowledge	represented by an array of concept and a number of attributes for each content entity (<attribute, value> pairs) representing user knowledge of each concept (knowledge, interest, ... and etc.)	represented by an array of concept and a number of attributes for each content entity	KV - Knowledge vector = array of Knowledge Items [K1, K2, Kn], each is weighted according to user confidence in this Knowledge	Competencies.learned - describes users prior-knowledge described with the same vocabulary (concepts) as Narrative (DM)	knowledge attribute - value estimating Users knowledge on each Concept
	learning objectives	n/a (tracked by AE)	n/a (tracked by AE)	n/a	Competencies.required - describes user learning goal (minimum knowledge learner should acquire to complete a course)	
	problem solving task (short term user goal)	yes (next page guidance = local guidance)	yes (next page guidance = local guidance)	direct guidance	yes (course authoring dependent)	

Table 2: Summary of Domain independent and Domain dependent User Model properties

2.3 Adaptation Model (Adaptive Engine) Comparison

The System has to adapt the presentation, the information content and the navigation structure to the user's level of knowledge, interest, navigational style, goals, objectives, etc.. Thus an Adaptation Model (AM) has to be provided, indicating how concept relations in DM affect user navigation and properties update (for instance whether the system should guide the user towards or away from information about certain concepts). In GRAPPLE the adaptation model is described at an abstract, conceptual level in the CAM. For the adaptive learning environment, or *adaptation engine*, these conceptual adaptation rules need to be translated into a concrete rule language supported by the GALE engine.

The table below compares the different ways in which adaptive systems implement "adaptation rules", user modelling, etc.. GALE is not mentioned in the table as its functionality is not yet completely determined. Here is a description of the main GALE properties:

Some systems follow an approach of "forward reasoning" in which an event leads to a conditional action that in the case of an AHS means a UM update. These updates lead to more conditional actions, etc. To some extent this is comparable to "forward chaining" in rule-based reasoning systems. Through forward reasoning one can calculate high-level UM properties, and have their values ready immediately when needed. Other systems use "backward reasoning", trying to deduce UM values from events that have happened, somewhat like how rule-based reasoning systems may use "backward chaining" to find evidence for a proposition. Through backward reasoning high-level properties can be calculated from lower-level (stored) information without the need to calculate the high-level properties when the events occur. An example of information that is typically calculated (forwards) when a user requests a concept is the user's *knowledge* of that concept. An example of information that is typically calculated (backwards) when a link to a concept needs to be presented is the *suitability* of that concept for the user. GALE tries to offer a truly generic adaptation engine that can perform both types of reasoning. In addition to that GALE is also able to execute arbitrary (Java) code during both forward and backward reasoning, in order to meet extensibility requirements.

GALE not only has an internal UM but can also communicate with GRAPPLE's User Modelling Framework (UMF), both to notify the UMF of actions performed by the user but also to be notified of learning outcomes from applications outside GALE (for instance when the learner takes a test using LMS functionality).

GALE does not prescribe what the meaning is of values in the user model. Typically a numerical value for the user's "knowledge" of a concept will represent *how much* the user knows about the concept, but it is equally possible to have the interpretation of *the probability that* the user knows the concept.

Adaptation rules in GALE are *event-condition-action* (ECA) rules, but because the condition and action may contain arbitrary Java code (including method calls) this is a much more powerful rule system than for instance in AHA! 3.

The possibilities for adaptive navigation support and adaptive presentation in GALE are an extension of AHA! 3. Sections 3.4 and 6 explain the many adaptation features GALE offers.

		AHAM	AHA! (version 3)	KBS Hyperbook	APeLS
Generating sequence	Generating adaptive content sequence (whether adaptive content is generated in a sequences or by one step at a time)	n/a (one page at a time)	n/a (one page at a time)	Depth-first-traversal algorithm checking the system's estimation of the student's knowledge of those KI s that are prerequisites for the actual goal	Narrative author must ensure that the customized courses produced from the narrative contain concept and Pagelet sequences to maintain coherency and logical flow
User observation	Getting feedback about user knowledge after completing project / goal / reading course INT - internal (system internal) EXT - external (requires external feedback)	INT : AE pre- and post stores updated knowledge coefficients of the user knowledge in UM EXT : tests	INT : AE updated knowledge coefficients of the user knowledge in UM (<i>readpages</i>) EXT : authored tests	EXT: direct feedback learner is asked about his own performance or domain expert may be asked to judge student's performance this grades knowledge user has on KI	INT : system updates user competency EXT : test possible if supported in Narrator (and corresponding rules)
Output of an AE	Output of an Adaptive Engine affecting different models aspects	- updated UM attributes - links adjustment - page construction - next page is shown to the user	- updated UM attributes - links adjustment - page construction - next page is shown to the user	Updating UM (KI coefficients in user model updated to current state) after completing the project (based on direct user feedback and probability calculation)	personalized course model => rendering of the course model into personalized Course Content - updated UM learned competencies on completing the course
certainty	Does system use any probabilistic aspect?	n/a	n/a	uses Bayesian network	n/a
Adaptation Rules	Rule types used in AE to drive adaptation process	ECA (Event-Condition-Action) or CA (Condition-Action) rules - event - condition - action + propagation (execution of other rules) + phase (phase rules grouping)	ECA (Event-Condition-Action) rules - event - condition - action + propagation (execution of other rules)	object oriented conceptual modelling language Telos - uses deduction rules, where: - everything in the knowledge base is a proposition ; each proposition has four components named respectively: from, label, to and when	Narrative Rules based on JESS (Java Expert System Shell) is a rules language is based on CLIPS , which is a LISP-like language. Rule-Based language (Facts->Rules) (Different facts make a rule applicable and it is asserted)
Adaptive navigation	guiding the learner by customizing the link structure or format		links with 3 possible states (desired, undesired, uninteresting)	Navigation is done according to generated sequential trail for a guidance tour or a PU sequence for a selected Project meeting students learning objectives	Navigation is done according to a course model and rendered content (narrative authors should ensure that produced content Pagelet sequence follow logic and structure of the course)
Adaptive presentation	customization of course content to match learning characteristics specified by the UM		done by Page Constructor - Fragment inclusion/selection - Links hiding/annotating/destination changing - Learning style (optional)	Information Unit content presentation is provided to the user	AE output is personalized course model which is rendered to s personalized Course Content for a particular user (Personalized Course Model Renderer - XSLT processor)

Table 3: Adaptation Model (Adaptive Engine) properties comparison table

3 The core ALE architecture

Figure 1 gives a rough sketch of the core architecture. Below we describe how the different components are actually used when the learner issues a request for a "concept" (which can be a request for a high-level goal or a concrete *page* from a course). For basic understanding of GALE it is sufficient to read up to Sections 3.1 and 3.2. More technical details follow in later sections. As GALE is based on a redesign of AHA! the name AHA! still appears in some components. We have a first partial prototype of GALE available with this deliverable, which does not yet conform completely to the description in the deliverable. Appendix A lists details of the prototype that deviate from the description. (Deliverable D1.3b, due in month 21, will implement all the functionality of D1.3a, and more, that will be described in a document to accompany the D1.3b software deliverable.)

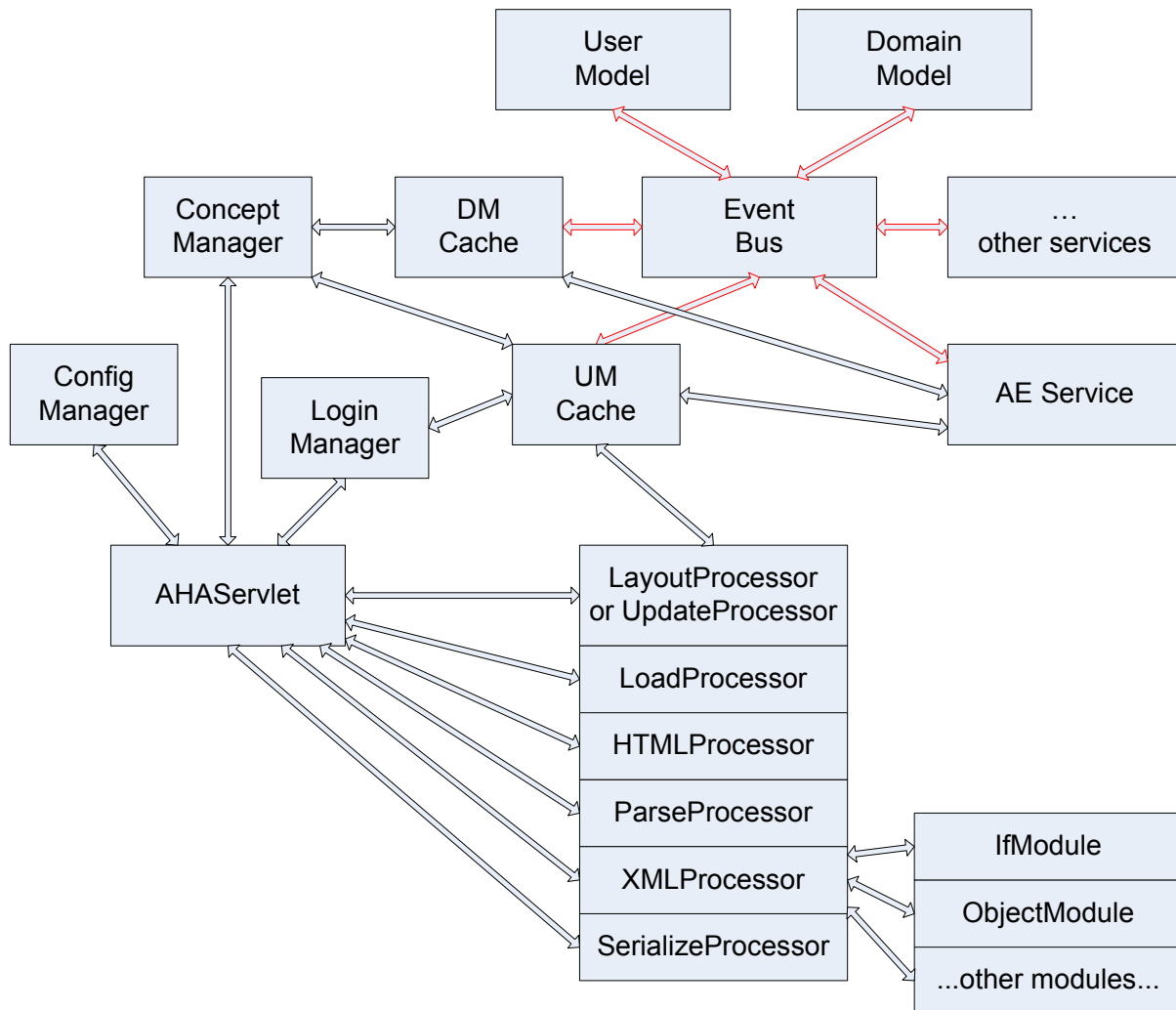


Figure 1: Core GALE architecture

GALE is implemented using Java and Servlet technology. It should therefore work with any (up to date) Servlet container. Within GRAPPLE we will mainly test it with Apache Tomcat, on different platforms. GALE presents itself to the "outside world" in two ways: through a **servlet** interface and through the **event bus**. The event bus is typically used to connect GALE to other GRAPPLE components (except perhaps for the login process), and the servlet is used for accessing GALE as (or actually through) a web server. Communication with the event bus (the red arrows) can be done through SOAP messages, whereas the communication between the servlet and other components is through method calls. The different "processors" and "modules" also have an "invisible" way of communicating: through shared in-memory data structures.

The servlet, called AHAServlet, coordinates most of the work done by GALE as a direct result of requests by users. The real work is done by processors, modules and plug-ins. Through the UM cache they notify the event bus of the action(s) that are going on so that the user model gets updated. They load resources (like XML files), perform adaptation to some parts (depending on the occurrence of certain tags) and then produce output to be sent in the reply to the user (or user's browser). We will look at a scenario of what goes on below. We first briefly explain the components of Figure 1. Later we will explain more in detail how each of the components actually do their work, guided by configuration files (that make GALE highly customizable).

3.1 The GALE components

This section contains a *brief* description of the different ALE components. A detailed description follows in Section 4.

- Figure 1 shows a *Domain Model* or DM component, which handles a description of an application domain (or course), including adaptation rules (or an *Adaptation Model* or AM). These structures are obtained by compiling *Conceptual Adaptation Models* (CAMs) for which WP3 develops authoring tools. The CAM editor (which does not exist yet at the time of this writing, but is described in deliverable D3.3a) notifies GALE of (committed) changes to a CAM via the event bus. (See Appendix A, point 1 for a description of authoring for the first prototype, in the absence of a CAM editor.)
- GALE needs its own *User Model* (or UM) service in order to operate in stand-alone mode, and in order to store detailed information about the user that is used for the adaptation and not needed by any other GRAPPLE component. The user model contains application-independent information about the user, called the *entity*, as well as application-dependent information, of which the structure is derived automatically from DM. (Note that GALE's UM service is different from the GRAPPLE User Modelling Framework developed in WP2 and 6.)
- The communication within GALE is centred around the *Event Bus*. Components send requests to the event bus, and other components may listen to these events and handle them. This may result in a reply sent to/through the event bus as well. Section 5 shows some of the communication that goes on. GALE may "optimize" requests in order to improve overall performance. For instance, when GALE needs to know the user's knowledge of a concept it will not just request that value, but all the information about all the concepts of the same course (for this user). The UM cache will capture that entire user model part and as a result there is no need to ask for the next concept's knowledge, perhaps a few milliseconds later. Different implementations of the event bus exist. There is a highly "local" implementation that only uses method calls and there is a SOAP implementation (which is used by default). A truly stand-alone GALE installation can improve overall performance by using the "local" implementation.
- The *DM Cache* and *UM Cache* components ensure the immediate availability of data from the domain-, adaptation and user models. They exist purely for performance reasons and are not elaborated upon in the remainder of this document.
- The *AHAServlet* is the main "coordinator" of most of the work performed by GALE. It maintains the session, loads and activates *processors* that work on the resource (or file) associated with the requested concept and sends responses back to the user (browser).
- GALE is very adaptable. A general configuration file (currently still called *ahaconfig.xml*) tells the main servlet which login manager to use, which concept manager implementation to use and which protocol to use for the event bus. As shown below the default concept manager makes GALE even more adaptable.
- The *Login Manager* is responsible for ensuring that the user is properly identified and authenticated. GALE contains a login manager for stand-alone use. Additional login managers can be used to make GALE collaborate with external authentication services such as Shibboleth².
- The *Concept Manager* is responsible for determining which resource to load when a concept is requested, and which processors to use to process/adapt that resource. The concept manager needs information from both the *domain model* (DM) and the *user model* (UM) in order to decide which resource should be processed for the requested concept. Once this decision is made the

² See <http://shibboleth.internet2.edu/>.

concept manager uses *ConceptConfig* files to determine how to process the resource. First the directory tree (path) for the resource is used to load *ConceptConfig* files that specify how to process and present the resource (or file or page). An author can group pages into directories and use *ConceptConfig* files to indicate a preference for how pages from each directory should be presented. The concept manager then searches a directory tree corresponding to the concept hierarchy of the current application (or course). The *ConceptConfig* file for the requested concept can override the processing/adaptation/presentation preferences indicated for the resource. It is thus possible to adapt and present the same resource (page) differently when it is used in two different applications (or courses).

- GALE also supports "plug-ins" for handling special types of requests. For instance the "PasswordConfig" plug-in lets users change their password (for stand-alone use of GALE) and the "Logout" plug-in lets users log out explicitly (not waiting for automatic logout after a timeout).

3.2 The "process" of handling a request for a concept

Either through the "location" field of the browser, or by clicking on a link anchor in any web page (on any web site, including an LMS where the user may be logged in) a user may send a request (an HTTP GET request) for a concept to GALE, or more concretely to the web server that hosts the *AHAServlet* servlet. This request starts the process described below:

1. If this is the first request the user sends since starting the browser no session will be associated with that request, so a session is initiated. The URL of a request to GALE always contains the name of a *concept* which also identifies an *application* (or *course*) to which this concept belongs. The (name of this) application is used below. (Plug-ins, a welcome page, a search page, etc. are all represented through concepts.)
2. The user needs to be identified. *AHAServlet* calls the *login manager* to obtain information about the user. For stand-alone GALE use (not using a single sign-on facility) this is the following multi-step sub-process:
 - a. For a first request (without session information as there is no session yet) the user is still unknown. The stand-alone GALE login manager redirects to a servlet/page that prompts the user for a user id and password.
 - b. The user id is passed on to the UM cache, to request the application-independent part of the *user model* (UM) for this user. Internally GALE refers to this as the *user entity*.
 - c. Since UM cache will not have cached the user model, it will communicate with the user model service through the event bus. (The UM cache "forgets" user models when a session ends.)
 - d. UM is needed by the login manager to verify that the user has provided the correct password. If so the login manager (servlet) returns a redirect to the original URL. As a result the user's browser will request the same concept again, this time with session information.
3. *AHAServlet* now calls the Concept Manager in order to find out how to handle the request for the given concept. This again is a multi-step sub-process:
 - a. The concept manager will request the *domain model* (DM) information for the given concept from the DM cache. The DM cache may have the domain model already, when another user is already logged in and using the same application/course. But if the DM cache does not yet have the DM it will request the entire DM from the domain model service through the event bus. (Requesting the entire DM saves many future requests.)
 - b. The concept manager now requests the UM cache for the user's information related to the concept. When a user just logs in UM cache will not yet have this information and will request the whole part of the user model related to this application (course) from the UM service through the event bus. (Requesting a large part of the user model saves many future requests.)
 - c. We will only describe the case where the requested concept is associated with one or more resources (URLs). (For abstract concepts without any associated resource a presentation is generated automatically that gives access to lower level concepts from the concept hierarchy.) Each URL either identifies a concrete resource like a web page or xml file to be

loaded or specifies a query that is sent to some service that should return a resource that matches the query. The URL can also represent a request for a concept (to the same or a different server). Adaptation rules (defined in DM) determine (based on UM) which resource (URL) will be used. This process (called "page selection" in the AHAM reference model [1]) is the topic of deliverable D1.2a.

- d. After selecting the resource (URL) the concept manager now requests the UM cache for the resource (URL) associated with the concept. Using our UM cache implementation that UM information should already be in the cache.
 - e. The concept manager now goes through the hierarchy of ConceptConfig (xml) files associated with the resource (URL) and the hierarchy of ConceptConfig files associated with the concept, to build a list of ConceptConfig files that need to be used in order to decide how to handle the concept. This list is returned to AHAServlet.
4. AHAServlet calls the Config Manager which *merges* the ConceptConfig files in order to obtain a single file that has all the "processing instructions". AHAServlet then loads all the "processors" that are (potentially) needed to handle (load, adapt and serve) the resource. This process is described briefly in Section 3.3 and more in detail in Section 6.
 5. Handling the resource is a multi-step sub-process that uses *processors* (more or less in the order they are shown in Figure 1). The processors are controlled using a *level*. Processors only become active when the concept has been processed up to a certain level (a range of acceptable levels) and when a processor is finished it updates the level. This is explained more in detail later. GALE can be easily extended with new processors, that are used anywhere in the processing pipeline. The ConceptConfig files indicate which processors should be used and the levels guide a processor in the decision when to run. We now describe how a resource is handled using the default chain of processors. A detailed description of the processors follows in Section 4.
 - a. The first processor touching the resource is the *LoadProcessor*. It is responsible for retrieving the actual resource (which can be a local file or can be a resource that has to be retrieved from some other server through http). An *InputStream* is opened so that a subsequent processor can load and process the data. A possible file name extension (like .html, .xml, .jpg, etc.) is used to determine the mime type of the resource.
 - b. The next processor checking out the data is the *HTMLProcessor*. If the mime type is some form of HTML (not XHTML) the (open source) JTidy converter is used to convert the file to XHTML. The HTMLProcessor replaces the *InputStream* so that it now contains valid XHTML.
 - c. The next processor looking at the data is the *ParseProcessor*. If the file is some form of XML it converts the file into an in-memory DOM tree.
 - d. At this point the *UpdateProcessor* is invoked. It signals an EventManager that the 'access concept' event has occurred. The EventManager has handlers defined in ahaconfig.xml. The default EventAccessHandler executes the event code of the concept as defined in the domain model. The resulting changes to the use model are posted on the event bus, and subsequent changes made by any registered UM service are integrated in the UM cache.
 - e. If the file is XML the *XMLProcessor* walks through the DOM tree in order to perform adaptation where needed. The modules that may be used to perform adaptation to certain tags are loaded by the XMLProcessor. The ConceptConfig files indicate which XML tag is handled by which module. By default there are modules for handling "if" tags, "object" tags, links, variables, and some other tags in XHTML files. It is possible to add new modules and use them simply by indicating in a ConceptConfig file which modules need to be loaded to handle which tags.
 - f. Optionally, the *FrameProcessor* generates a frame-like structure using tables, by creating a (in-memory) XML document that only contains the views (any class that implements the LayoutView interface) embedded in a table that defines the layout. This document has a placeholder element where the actual content should be. The FrameProcessor then decreases the level and sets the "redo" flag on the resource. This will cause the container (AHAServlet) to start processing again from a specific level (in this case the XMLProcessor's level). Now only the views are processed and when the resource arrives at the FrameProcessor a second time, this is recognized and the actual adapted content is added to the table structure. The first prototype contains a *LayoutProcessor* that generates an HTML frameset structure. It is described Section 4.3 and also in Appendix A, point 2.

- g. When the DOM tree is adapted the *SerializeProcessor* generates the textual XML representation and presents that to *AHAServlet* as an *InputStream*. For resource types that do not appear in any *ConceptConfig* file *AHAServlet* will create this *InputStream* itself in order to then send the content back to the browser. This for instance happens with images embedded in HTML pages. (For some special resource types *AHAServlet* calls a special Plug-In that may generate its own output. These plug-ins set the level to 100, which for *AHAServlet* means that the output was already generated by the plug-in. Example of such plug-ins are the *PasswordConfig* and the *Logout PlugIn*.)

3.3 Configuration file hierarchies

In GRAPPLE adaptive applications are derived from a *Conceptual Adaptation Model* or CAM (see deliverable D3.3a). The CAM defines the concepts of the application and relationships between concepts (the structure of which is currently still to be determined). In order to be as generic as possible the GALE allows the following types of operations on concepts:

- Updating the user model when a concept is accessed. The updates are defined at a high level in the CAM but are executed using *adaptation rules* which are *event-condition-action* rules.
- Processing of the concept through *Processors* and *Modules*. This is mainly used for **content adaptation**.
- Processing of the link structure through *Modules* and *View Processors* (see Section 4.3 for view processors). A link module performs **link adaptation** and a view processor produces views onto the link structure, for instance a form of **fish-eye view**.
- Generating and adapting the layout of the application. Different parts of an application may require a different layout, and the layout may also depend on the user model. (Some users may for instance require a different fish-eye view of the link structure than others.)

Resources (pages or files) are normally structured using a directory hierarchy. At each level of this hierarchy a configuration file defines how the user model is updated and how the adaptation should be performed. The configuration of the parent directory is inherited and parts of that configuration may be replaced.

Concepts are normally also structured hierarchically. This can be done in two ways. The top level is the name of the application (for instance a course). It is possible to simply create a hierarchy using special attributes (to define parent and children), and it is possible to create a hierarchy using a concept naming convention. Currently the convention is to use a dot "." to create levels of the hierarchy in concept names, and to use a directory hierarchy for configuration files corresponding to these hierarchy levels. The configuration at the conceptual level overrides the configuration at the resource level. It is thus possible for two concepts (possibly in different applications or courses) that refer to the same resource (or page) to be presented differently depending on the concept the user has requested.

The list of concept-specific configuration files produced by the concept manager is sent to the config manager, where the xml files will be loaded and merged. The ability to merge configuration files allows for a conceptually distributed configuration. Parts of the configuration may depend on the actual location of the resource to load and a part may depend on the application that this concept is a part of, or it may depend on any other concept related information. These configuration files can all be stored in different locations. Distributing the configuration allows authors (teachers) to reuse some of each other's course pages while keeping the presentation and adaptation within their own course consistent. In course A links may be presented with an icon (checkmark) to indicate how much the user already knows about the destination concept. The author of course B may not like this and can still use pages from course A but when these pages are presented within course B they then do not have the links annotated with these icons. If desired the author of course B may also use a different threshold for deciding when a link (destination) becomes suitable than the author of A.

Merging of *ConceptConfig* files is done by adding information from a second xml file to a first xml file. To influence the merging process, the author of the configuration files can use a special attribute called 'merge'. This attribute can be set to any of the following values: 'add' (default), 'remove', 'replace', 'replacefirst', 'match', 'present'. Here is a list of their meaning:

- add – this is the default option and is chosen when no merge attribute is specified (an exception is the root element which has merge='replace' by default). The element from the second xml file is added to the first file.

```

<config>      <config>      <config>
  <test/>    +   <test merge="add"/>  →   <test/>
</config>    </config>                </config>
  
```

- remove – removes all elements with the same name within the current parent element from the first xml file.

```

<config>      <config>      <config>
  <test/>    +   <test merge="remove"/>  →   </config>
</config>    </config>
  
```

- replace – removes all elements with the same name within the current parent element and adds the new element.

```

<config>      <config>      <config>
  <test>      <test merge="replace">    <test>
    <a/>      <b/>                        <b/>
  </test>    </test>                    </test>
</config>  </config>                    </config>
  
```

- replacefirst – same as replace, but only the first matching element is removed.

```

<config>      <config>      <config>
  <test>      <test merge="replacefirst"> <test>
    <a/>      <b/>                        <b/>
  </test>    </test>                    </test>
</config>  </config>                    </config>
  
```

- match – matches the first element with the same name within the current parent element to the element in the second xml file and processes the sub elements recursively.

```

<config>      <config>      <config>
  <test>      <test merge="match">      <test>
    <a/>      <b/>                        <a/>
  </test>    </test>                    <b/>
</config>  </config>                    </test>
  
```

- present – adds the element in the second xml file if and only if such an element does not exist in the first xml file.

```

<config>      <config>      <config>
  <test>      <test merge="present">    <test>
    <a/>      <b/>                        <a/>
  </test>    </test>                    </test>
</config>  </config>                    </config>
  
```

```

<config>      <config>      <config>
  </config>    <test merge="present">    <test>
  </config>    <b/>                        <b/>
  </config>    </test>                    </test>
  
```

The config manager uses a smart cache so all the merging is done only when needed. These concept-specific configuration files hold information that is used by the AHAServlet to determine which processors to use. They also hold processor specific information. Every processor has its own private location in the xml files and defines its own small part of the file, using tags and attributes chosen by the designer of that processor. A dtd or schema is impossible to create, because the structure is not defined by the GALE as a whole but different parts are defined by the different processors. When GALE is extended with new processors and modules, new tags will be defined and used in the ConceptConfig files.

Each tag in a configuration file can have an attribute "expr" and a value which is an expression over the user model. If the expression returns false the element is ignored. Below is an example from an example course c2ID65:

```

<processors merge="match">
  <layoutprocessor merge="replace" expr="{c2ID65.introductorypart.done}">
    <code>nl.tue.aha.ae.processor.LayoutProcessor</code>
    <update>true</update>
  </layoutprocessor>
  <updateprocessor merge="replace" expr="!{c2ID65.introductorypart.done}">
    <code>nl.tue.aha.ae.processor.UpdateProcessor</code>
  </updateprocessor>
</processors>

```

This code means that the "layoutprocessor" is used when the user has finished (done) the introductory part of the course, otherwise the "updateprocessor" is used.

3.4 Defining the Processor pipeline

When the final configuration is available as a result of possibly merging several files, the AHAServlet loads and executes the processors defined in this configuration. An example configuration file is found below. This is not a typical *complete* configuration file (after merging individual files) but only the part that controls the *processing* of concepts and resources.

The general scheme is that for each "processor" that is defined there is a "code" tag that indicates which class file must be loaded. After that there are other tags that are specific for that processor. GALE currently follows the following conventions for determining which processor, module or plug-in to execute:

- The mime-type of a resource is used to determine which processor to execute. In reality each processor knows which mime-types it can handle.
- When a request ends with ?plugin=XXX it is handled by the plugin processor and the name (XXX) is used to decide which plug-in to use.
- The first prototype also has a ?view=XXX construct, used in conjunction with the *LayoutManager* to decide which view processor to use. (See section 4.3 and Appendix A, point 2.)

As can be seen from the file (fragment) below most processors do not require a lot of configuration information (other than the path of their class file). The XMLProcessor (see Section 4.2 for an explanation of its configuration) is a notable exception.

```

<config>
  <processors merge="match">
    <pluginprocessor merge="match">
      <code merge="replace">nl.tue.aha.general.processor.PluginProcessor</code>
      <plugins merge="match">
        <KnowledgeConfig>
          <code>nl.tue.aha.general.processor.plugin.KnowledgePlugin</code>
        </KnowledgeConfig>
        <PasswordConfig>
          <code>nl.tue.aha.general.processor.plugin.PasswordPlugin</code>
        </PasswordConfig>
        <Logout>
          <code>nl.tue.aha.general.processor.plugin.LogoutPlugin</code>
        </Logout>
      </plugins>
    </pluginprocessor>
    <loadprocessor merge="match">
      <code merge="replace">nl.tue.aha.general.processor.LoadProcessor</code>
    </loadprocessor>
    <parserprocessor merge="match">
      <code merge="replace">nl.tue.aha.general.processor.ParserProcessor</code>
    </parserprocessor>
    <htmlprocessor merge="match">
      <code merge="replace">nl.tue.aha.general.processor.HTMLProcessor</code>
    </htmlprocessor>
    <xmlprocessor merge="match">
      <code merge="replace">nl.tue.aha.general.processor.XMLProcessor</code>
      <modules merge="match">
        <ifmodule merge="replace">
          <code>nl.tue.aha.general.processor.xmlmodule.IfModule</code>
          <tags><tag>if</tag></tags>
        </ifmodule>
        <objectmodule merge="replace">
          <code>nl.tue.aha.general.processor.xmlmodule.ObjectModule</code>
          <tags><tag>object</tag></tags>
        <update>true</update>
      </modules>
    </xmlprocessor>
  </processors>

```

```

</objectmodule>
<linkmodule merge="replace">
  <code>nl.tue.aha.general.processor.xmlmodule.LinkModule</code>
  <tags><tag>a</tag></tags>
  <condition>"conditional".equals(element.getAttribute("class"))</condition>
  <classattribute>class</classattribute>
  <linkattribute>href</linkattribute>
  <annotation>
    <all>
      <badclass>
        <expr>!${suitability}</expr>
        <class>bad</class>
      </badclass>
      <goodclass>
        <expr>${suitability} & amp; & amp; ; (${visited} == 0)</expr>
        <class>good</class>
      </goodclass>
      <neutralclass>
        <expr>${suitability} & amp; & amp; ; (${visited} > 0)</expr>
        <class>neutral</class>
      </neutralclass>
    </all>
  </annotation>
</linkmodule>
</modules>
</xmlprocessor>
<serializeprocessor merge="match">
  <code merge="replace">nl.tue.aha.general.processor.SerializeProcessor</code>
</serializeprocessor>
</processors>
</config>

```

Any element that is found in the `processors` tree is considered to be a processor. It should have a subelement named `code` that contains the fully qualified name of an implementation of `nl.tue.aha.engine.ResourceProcessor`. The element of the processor itself is passed as a parameter to the processor as a means to store additional configuration.

A resource in GALE has an attribute called 'level'. The level of a resource is a number between 0 and 100 to indicate the amount of processing done on the resource. The level of a resource is used to call an appropriate processor for the current stage. Each processor has a minimum and maximum level of processing, so that a resource with a level in between those values goes through that processor. The processors are sorted first on their minimum levels and if those are equal on their maximum levels. They are executed in this order, but a specific processor only gets called when the level of the resource is within the bounds defined by that processor. By setting values a processor can thus easily indicate that some other processor (later in the sequence) should be skipped.

If the level of a resource is still 0 after all processing is done, the `AHAServlet` assumes it should try and load the requested URL directly and puts an `InputStream` object in the resource under the name 'stream'. If the level of a resource is less than 100 after all processing is done, the `AHAServlet` assumes there is an `InputStream` object in the resource under the name 'stream', and tries to send its content to the client.

3.4.1 Default variables in the resource

There are some default variables present in the resource that are put there by the `AHAServlet` at the moment the various processors are being called. They include the following:

- `context` the `javax.servlet.ServletContext` running GALE. The `servletcontext` is used to store application wide variables. There are some default variables stored here which are described in section 3.4.2.
- `request` the `javax.servlet.http.HttpServletRequest` with which the `AHAServlet` is called. The information stored here can be used by the `ConceptManager` to decide on what conceptual information to add to the resource.
- `response` the `javax.servlet.http.HttpServletResponse` with which the `AHAServlet` is called.
- `profile` the `nl.tue.aha.engine.Profile` that describes the currently logged in user.
- `config` an `org.w3c.dom.Element` describing the current merged configuration.
- `processorconfig` an `org.w3c.dom.Element` containing that part of the configuration specific to the current processor.

3.4.2 Default variables in the servlet context

The following is a list of default variables in the servlet context, put there by the AHAServlet.

- `ahahomedir` a `java.io.File` describing a location on the server that is used as the GALE home directory. This location can be defined in an environment variable called 'AHA_HOME'. If it is not set there, the path where the ALE is installed on the server will be used. If GALE runs in an unextracted war file a default path is used (`/usr/work/aha`).
- `ahaconfig` an `org.w3c.dom.Element` containing the contents of the `ahaconfig.xml` file. This file should be located in the GALE home directory described above. If it is not found there, GALE will try to load it from the `/WEB-INF` directory of the `aha` context.
- `LoginManager` an instance of `nl.tue.aha.engine.LoginManager` as described in `ahaconfig.xml`.
- `ConceptManager` an instance of `nl.tue.aha.engine.ConceptManager` as described in `ahaconfig.xml`.
- `ahadebug` an Integer value defined in `ahaconfig.xml` that indicates how much debug information GALE should write to the standard output.

4 Processor details

This chapter describes the general processors that make up the default behaviour of GALE. More details on how to configure each of the processors (and modules) are given in chapter 6. The main adaptive functionality of GALE is defined by the `XMLProcessor`. There are several helper processors that are described first, like the `LoadProcessor`, the `ParserProcessor`, the `HTMLProcessor` and the `SerializeProcessor`.

A processor in GALE is an implementation of `nl.tue.aha.engine.ResourceProcessor`. It is called by the AHAServlet with the current resource as a parameter. If the processor decides to do anything with the resource, it will probably also increase the level of the resource. In the following list of general processors the numbers after the name refer to the minimum and maximum level that the processor operates on.

Note that all processors are defined in the `ConceptConfig.xml` file(s). The adaptive behaviour of GALE can be completely changed by using different processors than the default ones described below. In particular, for the adaptation to virtual reality and simulated dialog it will be necessary to develop new modules that will perform adaptation to the appropriate xml tags. Such modules can simply be called upon by indicating their use in the `ConceptConfig.xml` file(s).

4.1 Helper Processors

LoadProcessor (0-2)

The `LoadProcessor` makes the resource data available as an input stream. It expects a `java.net.URL` variable named 'url' in the resource. It opens a connection to this URL and puts the resulting `java.io.InputStream` in the resource under the name 'stream'. The mime type of the resource is put in the resource as a string under 'mime'. It is retrieved by looking at the extension of the URL.

The level of the resource is set to 5.

HTMLProcessor (5-5)

The `HTMLProcessor` creates proper xhtml data from any html data. If the mime type ('mime') indicates html data, then the data in the input stream ('stream') is used to create a new input stream that contains proper xhtml data. This new input stream is stored in the resource as the new 'stream' object.

The level of the resource is set to 7.

ParserProcessor (5-7)

The `ParserProcessor` parses xml data of the resource. If the mime type (a string called 'mime') indicates xml data, then the input stream of the resource (an input stream called 'stream') is used as the data to parse. The resulting xml DOM document is stored as an `org.w3c.dom.Document` under the name 'xml'.

The level of the resource is set to 10.

Most of the "real" work is done using the generated DOM tree. The adaptation processors/modules manipulate the DOM representation of the document. They can increase the level of the resource bit by bit, but not exceeding the value 90.

SerializeProcessor (10-90)

The `SerializeProcessor` is activated after all the manipulations to the DOM tree. It serializes the xml data found in the resource back to an input stream, if the mime type suggests xml data. The resulting input stream is stored in the resource as the new 'stream' object.

The level of the resource is set to 95. This tells the `AHAServlet` that all processing has been done but that the result has not yet been returned to the browser. The `AHAServlet` will do this. Should the level be 100 the `AHAServlet` will assume that returning a result to the browser has already been done and it will do nothing.

4.2 The XMLProcessor (10-40)

The `XMLProcessor` calls modules that perform the actual adaptation to any xml document. It processes the resource only if the mime type indicates xml data. This currently includes 'text/xhtml', 'text/xml', 'application/xml', 'application/smil'. It traverses the DOM tree based on a depth first algorithm. Each xml element (tag) is handed over to a module that will adapt it, or is left alone if no module handles the tag. Any adaptation that transforms an xml element (and the DOM subtree below it) into a valid xml element (and DOM subtree) is possible provided that there is a module for this adaptation.

An `XMLProcessor` module is a class that implements `XMLProcessorModule` (found in the package `nl.tue.aha.general.processor.xmlmodule`). The modules that the `XMLProcessor` should use are specified in the configuration files (see the example configuration file in section 3.4). The `xmlprocessor` element has a subelement called 'modules' that contains a list of elements defining the modules to use. Each module defined should have a subelement 'code' that points an implementation class of `XMLProcessorModule`. It should also have a subelement 'tags' that defines which xml tags indicate an element the module will process (adapt). The module element itself is passed as a parameter to the module. It may contain additional subelements that are used by the module to further define what the module should do. In the example (in section 3.4) the `linkmodule` has a 'condition' that specifies that the link adaptation will only be performed if the element (the 'a' tag) has an attribute "class" with the value "conditional". This condition is only handled by the `linkmodule` itself. The 'condition' element has no meaning to the `XMLProcessor`.

When the `XMLProcessor` traverses the DOM tree, it passes control to the various modules as it encounters their tagnames. It's the modules which in turn can change the structure of the DOM tree. (The `XMLProcessor` itself does not change the document at all.) In Section 6.5 we explain more in detail the kinds of (content and link) adaptation performed by the current set of modules that are part of the GALE distribution. It is expected that during the course of the GRAPPLE project (and even after that) new modules will be added to handle adaptation to different xml tags. This will include adaptation as needed for VR and Simulation applications (that will be defined in WP3 and WP4).

The level of the resource is left unchanged by the `XMLProcessor`.

4.3 The LayoutProcessor (0-1)

The `LayoutProcessor` allows the use of separate views (generated by *View Processors*) using a presentation structure, implemented as an html frameset. The various views are defined as classes that implement `LayoutView` (found in the package `nl.tue.aha.general.processor.view`). Each view can generate xhtml data that it puts in the resource as an `org.w3c.dom.Document` under the name 'xml'. This will be processed by the `XMLProcessor`. The `LayoutProcessor` is present in the first GALE prototype (see also Appendix A, point 2) but is not the preferred way to specify and implement layout, because it creates a race condition: An alternative layout processor that uses css or iframes will be developed later, for use within the existing presentation templates of LMSs. Layout with an html frameset is mainly intended for the stand-alone use of GALE.

If the request is the "main" request (without any `?view=XXX` suffix), the processor generates a frameset based on information in the configuration file. This is stored in the resource as xml data under 'xml' and the level is set to 90. (Note that this will activate the `SerializeProcessor`, sending the frameset to the browser. It is the browser which will send new requests, to obtain content to be presented in the different frames.)

Typically the concept configuration file will also indicate that the `LayoutProcessor` should perform user model updates (by specifying `<update>true</update>`).

If this is the request for the 'mainview' presenting an adapted page (distinguished by the addition of `?view=mainview` to the request URL), the `LayoutProcessor` will do nothing and leave the level unchanged.

If this is the request for another named view, the `LayoutProcessor` will call the view-plugin (*View Processor*) that generates that particular view and set the level of the resource to 30. A typical use of "views" is to create

navigation aids. Nowadays most Web presentations contain a "menu" with direct links to main parts of the site. A course can use a similar menu to present an adaptively annotated list of chapters or topics. GALE comes with a predefined set of View Processors (TreeView, StaticTreeView, PathView, NextView, etc.) all of which provide a different presentation of part of the concept structure of the current application (course).

4.4 The FrameProcessor (10-50)

The FrameProcessor allows the use of separate views similar to the LayoutProcessor above. Instead of using a frameset page it uses a table structure to create layout. It creates a (in-memory) XML document that only contains the views (any class that implements the `nl.tue.aha.general.view.LayoutView` interface) embedded in a table that defines the layout. This document has a placeholder element where the actual content should be. The FrameProcessor then decreases the level and sets the "redo" flag on the resource. This will cause the container (AHAServlet) to start processing again from a specific level (in this case the XMLProcessor's level). Now only the views are processed and when the resource arrives at the FrameProcessor a second time, this is recognized and the actual adapted content is added to the table structure.

4.5 The Plug-in Processor (0-0)

The Plug-in Processor allows arbitrary plug-ins to be added, that have direct control over the main output of GALE based on the current concept and user. A plug-in has two methods called `doGet` and `doPost`, that are called when the respective http methods 'get' and 'post' are called on the current concept. A specific plug-in is called by adding its name as a parameter to the request URL (like `'http://localhost:8080/aha4/concept/tutorial.readme?plugin=KnowledgeChange'`).

Setting the level of the resource is left to the plugin to handle. The plugin may write to the http response directly and set the level to 100. (This indicates to AHAServlet that it need not do anything any more.)

5 Communication with the Domain Model and User Model

The domain model for the GALE is a set of related concepts. It includes what the AHAM model calls the *adaptation model* (or AM) and is obtained from a CAM import module (compiler). Each application can define its own subset of concepts and relations. A concept has a unique name, a set of properties, relations to other concepts, event code, and a set of attributes. Properties that are used in GALE are for example a title (to be displayed when the links to the concept are shown to the user), a description, the type of concept, a flag to indicate whether the concept should be adapted upon *each* request or only upon *the first* request³, etc. . Each concept has a set of named relations to other concepts and each relation can have properties of its own.

The event code is a GALE statement (see Section 6.2). This code is executed by an `nl.tue.aha.ae.EventHandler` specified in `ahaconfig.xml` (by default this is `nl.tue.aha.ae.EventAccessHandler`).

The user model in GALE contains an overlay of the domain model, using additional attributes, such as "knowledge", "suitability", "visited", etc.. The list of attributes (and their names) is not predefined. The attributes have no implicit meaning to GALE, but only have meaning because of the adaptation behaviour associated with them. The CAM (and its translation to adaptation rules) defines which attributes exist. The type of attributes specified in the user model can be any Java class. The first example applications only use a few types like Integer, Boolean, String and Concept (for the concept hierarchy) but allowing any data type makes GALE extensible. User model variables are accessed through their unique name. This name is the concept name, a period, and the attribute name. For instance, `'tutorial.welcome.knowledge'`, where `'tutorial.welcome'` is the name of the concept and `'knowledge'` the name of the attribute. The attributes of a concept define information that should be calculated on the fly or stored in the user model. Together with domain model attributes they define the variables the author can refer to in code (for the notation see section 6.6). Each attribute has a unique name within a concept, a type, event code, default code, and a set of

³ The possibilities for "stability" are actually more elaborate: a concept can be "always adapted", "adapted only once", "adapted once in each session" or "adapted conditionally". Users may get confused when a page looks very different when it is revisited. "Stability" allows GALE to keep the presentation stable either for a while or indefinitely.

properties. Properties of attributes are used for specifying whether the attribute is read-only, persistent, should be kept stable, etc..

The user model (UM) part of GALE uses the default code and event code to update the user model when a change occurs. The event code is executed when the value of an attribute changes in the user model (*forward reasoning*). The default code is executed to calculate the value, when there is no value in the user model database (*backward reasoning*). If an attribute is set to be not persistent, the UM-server will never store its value in the database. Instead it will recalculate it, whenever necessary.

The domain model and user model are managed by separate services registered on the event bus. All information can be requested and updated by sending appropriate 'events'. The remainder of this section discusses the specific events and their parameters as used by the GALE DM and UM servers. All data classes used throughout GALE can be found in the package `nl.tue.aha.common.data` and include `Concept`, `Attribute`, `ConceptRelation`, `UserEntity` and `EntityValue`.

5.1 The EventHash class

The class `nl.tue.aha.event.EventHash` is used as a utility class by the DM and UM server in the serialization process. It allows a list or map to be serialized as a String and it can read such a serialized String and create a list or map. Each EventHash also has a name. It is an extension of the Java class `java.util.TreeMap<String, String>`, so it supports all the default methods that TreeMaps provide. The `toString` method serializes its content. The constructor can take the serialized form as a parameter.

The generic serialized form is: `<name> : <key1> : <value1> ; <key2> : <value2> ...`

The name and keys may not contain colons (:). Semicolons (;) may be escaped using the backslash (\). The serialized form may omit the keys to present a simple list. Unique keys will then be generated automatically. The `getItems` and `addItem` method may be used to manipulate the `TreeMap` as if it were a simple list.

Both DM and UM server export caching classes (`nl.tue.aha.dm.DMCache` and `nl.tue.aha.um.UMCache`) that provide (mostly static) methods to serialize and de-serialize the specific domain model and user model classes using an `EventHash`. These strings are then transmitted through the event bus.

5.2 The GALE DM service

The adaptive engine part and the UM service part of GALE need domain model information (we consider the adaptation model to be part of the domain model). They request information by sending a 'getdm' event to the event bus. This method should be supported by at least one service on the event bus. The default DM service of GALE listens to this event and returns the requested information. It uses a database and Hibernate (www.hibernate.org) to store and retrieve information. The 'setdm' method is also supported, to allow changes and additions to the domain model. Through the 'setdm' method an authoring tool (such as the CAM editor) can "commit" its work and install a newly created application (course). Appendix A, point 3, explains the situation with the first prototype.

The adaptation engine doesn't know where the data comes from when requesting domain model information. This is all handled by the event bus. Domain model services are expected to post updates on the event bus when their content changes. They should use the 'updatedm' event to indicate such a change in content.

In Table 4 we give a more detailed description of the events generally supported by domain model services (an 'l' behind the name indicates the service listens to this type of event, an 's' behind the name indicated the service sends this type of event). The default domain model service sends 'updatedm' events whenever changes made by a call to 'setdm' have resulted in a change to the database. Authoring tools (like the CAM editor) only send 'setdm' events. The DM service decides what is new (updated) and sends 'updatedm' events. Since a domain model may also contain default values for (user model) attributes, the UM service must listen to 'updatedm' events (to update the stored default values).

Method	Parameters	Result	Description
getdm (l)	application:<app-name> or concept:<concept-name>	A list of the serialized entities that are part of the specified application or concept.	Retrieves domain model information.
setdm (l)	A list of the serialized entities	'result:ok' if the operation	Sets domain model

	that should be set. May also include the special 'remove-app:<app-name>' EventHash.	succeeded. 'result:<exception-class>' if the operation failed.	information.
updatedm (s)	A list of the serialized entities that changed in the domain model.	Ignored.	Notifies listeners on the event bus that the domain model has changed.

Table 4: Events supported by domain model services

5.3 The GALE UM service

The adaptive engine part of GALE depends on UM services on the event bus to provide user information. The user model events are split into 'um' events and 'entity' events. The entity events communicate information about the actual user or group and its properties. Properties can be a password, whether the user is a regular user, author, administrator, or a combination of these, etc.. The UserEntity (`nl.tue.aha.common.data.UserEntity`) maintains a list of 'child' entities and one possible 'parent' entity (the group). This information can all be set and retrieved via the 'setentity' and 'getentity' events.

The actual variables that comprise the user model data for an application or course are set and retrieved via the 'setum' and 'getum' events.

Below we give a more detailed description of the events generally supported by user model services (an 'l' behind the name indicates the service listens to this type of event, an 's' behind the name indicated the service sends this type of event):

Method	Parameters	Result	Description
getentity (l)	entity:<entity-name>	The serialized UserEntity found in the database.	Retrieves UserEntity objects from the database.
setentity (l)	The serialized UserEntity that needs to be changed.	'result:ok' if the operation succeeded. 'result:<exception-class>' if the operation failed.	Sets UserEntity objects in the database.
getum (l)	application:<user-name>.<app-name>	All serialized user model variables that are part of the specified application and their values.	Retrieves user model variables from a specific application.
setum (l)	All serialized user model variables and values that should be changed.	'result:ok' if the operation succeeded. 'result:<exception-class>' if the operation failed. Followed by a list of all user model variables whose value changed because of the updates (including requested changes).	Updates user model variables.
updateum (s)	A list of user model variables and their values that have changed.	Ignored.	Notifies listeners on the event bus that the user model has changed.
updatedm (l)	A list of the serialized entities that changed in the domain model.	'result:ok' if the operation succeeded. 'result:<exception-class>' if the operation failed.	Updates the internal cache and recalculates non-persistent attributes whose default code has changed. May result in updateum events.

queryum (I)	query:<hibernate-query-string>	A list of serialized user model variables and/or UserEntity's that result from executing the specified Hibernate query.	The 'nl.tue.aha.common.data' package specifies a set of Hibernate enabled classes that are used persisting data to a database. Any query based on these classes can be used to retrieve data from the UM service.
--------------------	--------------------------------	---	---

Table 5: Events supported by user model services

The adaptation engine part of GALE will request a specific user model part (the part identified by the unique username and an application name) only once. The values retrieved are cached and updated directly in the cache and by listening to 'updateum' events on the event bus. Hence there is no need to request the information more than once (as long as the user is logged on).

6 Authoring Guide

Authoring the conceptual structure and adaptation for an application is done using DM and CAM editors. These part of the authoring process are not described here. A temporary tool is described in Appendix A, point 1. A tool for "sending" content (resources) to GALE is described in Appendix A, point 4. This chapter describes other aspects:

- How to install GALE (as a stand-alone ALE);
- How to design layout and adaptive presentation aspects of an application;
- How to extend GALE with plugins to offer additional services (besides content delivery).

Most of this chapter deals with the layout and adaptive presentation aspects. Before going into details about that we briefly describe how to install and configure GALE as a stand-alone ALE.

6.1 Installing GALE as a stand-alone ALE

GALE runs in any servlet enabled container. We have developed and tested GALE extensively on the Tomcat servlet container by Apache. To compile and run GALE under Tomcat, you will need a valid Java Development Kit (JDK 6 update 4 or later)⁴ and Tomcat version 6. If you want to compile GALE you also need Apache Ant⁵.

JDK 6: <http://java.sun.com/javase/downloads>

Tomcat 6: <http://tomcat.apache.org/download-60.cgi>

Apache Ant: <http://ant.apache.org/bindownload.cgi>

The GALE sources can be obtained through SVN. Stable versions will be posted on the GRAPPLE website, but the public anonymous SVN repository is used for the latest (development) versions. The current address of GALE is <https://svn.win.tue.nl/repos/AHA/aha4.1>.

After you have the sources and set up your JDK and Ant properly (by adding the 'bin' directories to your PATH environment variable and setting the JAVA_HOME environment variable to your java installation directory), you can compile GALE by running 'ant' in the root of the GALE distribution. This will create aha-4.0.war in the 'dist' subdirectory. You can copy this file to your Tomcat 'webapps' directory. Now you are ready to start Tomcat and go to 'http://localhost:8080/aha-4.0/' on your machine. This should bring up a standard home page that refers you to the AMt tool to start authoring applications.

⁴ The ability to have Java expressions in DM and UM and have them interpreted (compiled and run) on the fly requires JDK 6 update 4 or later. At the time of this writing this has implications for Apple users who can only run the server-side components but not the applets used for authoring in the first prototype.

⁵ Plans for a future version include a move from Ant to Maven, which should enable us to avoid bundling GALE with libraries from other open source projects.

GALE stores (nearly) every configuration file and the (DM and UM) database in the so called AHA_HOME directory. By default this is the directory where the servlet container extracts the war file (in Tomcat this would be '\$CATALINA_HOME/webapps/aha-4.0'). This directory will be overridden each time you install a new version of GALE by updating the .war file. This will most likely be undesirable. To prevent this you can set up your own AHA_HOME directory.

To set up your own AHA_HOME directory, you need to specify the AHA_HOME environment variable. If you start GALE after setting the variable it will use this directory as the base of all its configuration and resource files. GALE expects to find some configuration files in the AHA_HOME directory, so you will have to copy the following files and directories from the above mentioned directory where the servlet container extracted GALE, to your new AHA_HOME directory:

AHASTandard

author

config

database (you can omit this, if you want GALE to create a new database)

tutorial

abstract.xhtml, aha.css, ConceptConfig.xml (to the root of your AHA_HOME directory)

WEB-INF/ahaconfig.xml (to the root of your AHA_HOME directory)

After these actions you can safely install a new version of GALE without the risk of overwriting your existing database and configuration.

6.2 Authoring ConceptConfig.xml files

When GALE receives a request, the default ConceptManager builds a list of ConceptConfig.xml files to load. We explain this by an example: Assume GALE receives the request for concept 'tutorial.welcome' (via the URL <http://localhost:8080/aha-4.0/concept/tutorial.welcome>). The ConceptManager determines, based on the domain and user model that the resource with the actual content of this concept is 'aha:/tutorial/xml/welcome.xhtml'. (Deliverable D1.2a explains this resource selection process.) Then the ConceptManager will build the following list:

- \$AHA_HOME/ConceptConfig.xml
- \$AHA_HOME/tutorial/ConceptConfig.xml
- \$AHA_HOME/tutorial/xml/ConceptConfig.xml
- \$AHA_HOME/config/ConceptConfig.xml
- \$AHA_HOME/config/tutorial/ConceptConfig.xml

The first three entries are based on the resource and the last two are based on the concept name. It is no error if one or more of the files do not exist. Even if no configuration files are found, GALE will still return the actual content of the resource. The files are merged according to section 3.3. Configuration can thus be stored in any of the above mentioned files. Through the AMt tool, the author has access to the second and third file; more specifically he has access to all configuration files stored within his application resource structure.

Section 3.3 defined how the merging of multiple ConceptConfig.xml files is done by using the 'merge' attribute. There is an additional attribute available that allows adaptive merging based on the user model. This attribute is called 'expr' and it can contain any GALE expression (see Section 6.6). An element that contains an 'expr' attribute is only included in the final configuration if this (boolean) expression returns the value true.

ConceptConfig.xml files have a 'config' element as root element. (See the example in Section 3.4.) The GALE engine will look at the first 'processors' sub-element to determine the processors to use. Each child element of the processors element is expected to be a processor definition. The engine will look for a 'code' element within the processor definition and expects the fully qualified Java name of a class implementing `nl.tue.aha.engine.ResourceProcessor`. The rest of the configuration is ignored by the engine. It simply calls the processors with their definition element as a parameter. Each processor can define its own configuration. (Sections 6.4 and 6.5 provide examples.)

6.3 Configuring the Plug-in Processor

The Plug-in Processor (see Section 4.4) allows plug-ins which have direct access to the `HttpServletRequest` and `HttpServletResponse`. Plug-ins implement the `nl.tue.aha.ae.processor.plugin.Plugin` interface. The Plug-in Processor is one of the first processors that gets called by GALE. It will look at the request to see if there is a parameter called 'plugin' (like the request URL 'http://localhost:8080/aha-4.0/concept/tutorial.welcome/plugin=PasswordConfig'). If such a parameter exists it will try to find a matching definition in its own configuration. It then transfers control to the plug-in.

The Plug-in Processor uses the 'plugins' child element in its own configuration element. This element contains a list of all the plug-ins. The name of the various plug-in elements should be the same as the name used in the 'plugin' parameter. A plug-in element can contain additional configuration used by a particular plug-in. The Plug-in Processor will only look at the first 'code' child element and expects the fully qualified name of a class implementing `nl.tue.aha.ae.processor.plugin.Plugin`.

Here is a short list of some default plug-ins:

- `nl.tue.aha.ae.processor.plugin.TODOPlugin`, shows a list of references to concepts (from the current course) that have not been visited (more specifically, that have their "visited" attribute set to 0).
- `nl.tue.aha.ae.processor.plugin.DonePlugin`, shows a list of references to concepts (of the current course) that have been visited (whose "visited" attribute is greater than 0).
- `nl.tue.aha.ae.processor.plugin.PasswordConfig`, shows a page that allows the user to change his password. Will result in an error if the user is an anonymous user. (This plug-in is not intended to allow users to change a password they may have in a single sign-on facility like Shibboleth, but only in the stand-alone GALE.)
- `nl.tue.aha.ae.processor.plugin.Logout`, logs out the current user by clearing the session; (When using a single sign-on facility the only effect of "logout" will be closing the session in GALE.)
- `nl.tue.aha.ae.processor.plugin.AHA3Plugin`, allows the user to load a .aha file (AHA! 3 domain model), convert that to GALE and store it in the domain model server.

There are more plug-ins in the default package. A plug-in to execute forms, to run multiple choice tests, to allow administrators direct access to GALE expressions, to convert MOT courses and more. You can easily write your own plug-ins and add a configuration file to your course that tells GALE about the plugin.

6.4 Configuring the LayoutProcessor and FrameProcessor

The `LayoutProcessor` (see Section 4.3) and `FrameProcessor` (section 4.4) allow GALE to generate an adaptive frames structure using views. A view implements `nl.tue.aha.ae.processor.view.LayoutView`. Each view generates an in-memory xml document that will be treated as a resource by the other processors. The `LayoutProcessor` expects a child element called 'views' that defines a list of named views. This list works similar as in the Plug-in Processor (section 6.3). There is also an 'update' element that has either false or true as its value. Only if update is set to true, will the `LayoutProcessor` send an access-concept event to the `EventManager` when the frameset is generated. Otherwise you will probably want to use the `UpdateProcessor` at some other point in time to send this event.

Any other child element (besides 'code', 'update' and 'views') will be regarded as a layout definition. A layout definition defines the actual frameset. It can have multiple 'type' child elements and only one 'struct' element. The combined 'type' elements define the set of concept types that this layout should be used for (if omitted, this layout will work for all concept types). The 'struct' element defines the actual structure. It can contain references to views or more 'struct' elements. It uses a similar definition as the resulting frameset to build rows and columns. An example is included below. The example actually defines a number of views that are not used in the "mainlayout" (which is allowed) and the "mainlayout" contains the predefined "mainview" which presents the actual resource (typically the xhtml page).

```

<layoutprocessor merge="replace">
  <code>nl.tue.aha.ae.processor.LayoutProcessor</code>
  <update>true</update>
  <views>
    <pathview>
      <code>nl.tue.aha.ae.processor.view.PathView</code>
      <fontsize>small</fontsize>
    </pathview>
    <childrenview>
      <code>nl.tue.aha.ae.processor.view.ChildrenView</code>
      <fontsize>small</fontsize>
    </childrenview>
    <treeview>
      <code>nl.tue.aha.ae.processor.view.StaticTreeView</code>
      <fontsize>small</fontsize>
    </treeview>
  </views>
  <mainlayout>
    <type>page</type>
    <struct cols="200,*">
      <treeview />
      <mainview />
    </struct>
  </mainlayout>
</layoutprocessor>

```

Each 'struct' element has either a 'cols' attribute defining columns, or a 'rows' attribute defining rows.

6.5 Configuring the XMLProcessor

The XMLProcessor (see Section 4.2) does the actual adaptation of xml (and thus xhtml) documents by means of modules. The configuration defines which modules the processor will use for which tags. Each module can have its own configuration again. Each module is an implementation of `nl.tue.aha.ae.processor.xmlmodule.XMLProcessorModule`. The XMLProcessor expects a child element called 'modules' in its own configuration. This element contains a list of modules in the same way as the Plug-in Processor and LayoutProcessor. Each module configuration contains a list of tags defining the elements that particular module should operate on. An example:

```

<ifmodule merge="replace">
  <code>nl.tue.aha.ae.processor.xmlmodule.IfModule</code>
  <tags>
    <tag>if</tag>
  </tags>
</ifmodule>

```

This defines the 'IfModule' as a module that is called when an 'if' element is found in the xml file that is being processed. Each module can update the xml document, using its own configuration, the user model and the domain model.

Here is a list of some default modules and how to configure them (if extra configuration is available):

- `nl.tue.aha.ae.processor.xmlmodule.ObjectModule`; this module usually works on the "object" tag. It can be configured to send access-concept events through its 'update' element (see Section 6.4). Each object is included by generating a new request URL for GALE. This URL is opened and the resulting document is added to the current in-memory xml document. The ObjectModule only works on elements that have a 'type' attribute set to 'aha/*' or 'aha' or '*/aha'. An object element can have a 'name' attribute that is interpreted as a concept name, or it can have a 'data' attribute that is interpreted as a relative URL based on the current URL of the resource content. When a concept "name" is used the same process is followed to determine the URL of the desired resource as for page concepts (see deliverable D1.2a).
- `nl.tue.aha.ae.processor.xmlmodule.HTMLModule`; This module usually works on the "html" tag. It creates a title attribute based on the concept's title property and it can be configured to generate a tag to include a stylesheet. An example child element for the stylesheet is:

```
<stylesheet>${homedir}/aha.css</stylesheet>
```

The HTMLModule also sets the base URL of the document to allow relative URL's in the document to work (like images).


```

        <expr>${suitability} &amp;& ($visited > 0)</expr>
        <icon place="front">images/WhiteBall.gif</icon>
        <view>treeview</view>
        <view>pathview</view>
        <view>childrenview</view>
    </neutralclass>
</allicons>
</annotation>
</linkmodule>

```

The 'condition' element defines the condition that determines more specifically on which tags the module should operate (only if the resulting expression is true). In the example above the 'condition' states that only links with a class attribute that has the value "conditional" are adapted. Other links are not handled by the linkmodule. You can thus write pages with non-adaptive links pointing to resources on the Web, while also including adaptive links to concepts of the course (or of some other course).

The 'classattribute' element defines the attribute that should be updated to a value defined in the annotation. In the example this is "class" just like what is used in the 'condition' but this need not be the case. The 'linkattribute' element defines the attribute that should be updated so that it becomes a valid URL. Typically this is 'href' but it can be another tag when the link adaptation is performed to documents other than (x)html.

The adaptation done to the 'linkattribute' is the adaptive selection of the link destination. Here again the process defined in deliverable D1.2a is used to select a concrete link destination (concept or resource) to be associated with the defined link destination (which must be a concept).

The link destination concept is used as a base for the expressions used in the annotation. This annotation consists of two parts: annotation of the link anchor and the addition of icons (in front of or after the link anchor).

- The link anchor is adapted by adaptively choosing a link "class". The rule:

```

<badclass>
    <expr>!${suitability}</expr>
    <class>bad</class>
</badclass>

```

means that if the "suitability" attribute of the link destination concept is false the link anchor will be presented as "bad". The actual presentation depends on the style sheet that is used. (The default GALE style sheet uses a near-black colour and no underlining for "bad" links.)

- Icons are placed in front of or after the link anchors based on the "allicons" element. The use of icons not only depends on expressions over the user model but also on the view. It is thus possible to use icons in the "treeview" but only have link anchor annotation (and no icons) in the "mainview".

Note that names like "badclass", "goodclass", etc. in the example have no meaning. They are just placeholders. We have chosen meaningful names for readability, but we could just as well have used names like "blah" and the behaviour would be identical.

6.6 GALE expressions

Any GALE expression or statement is basically a piece of Java code extended with a special notation for using user model variables. You also have access to the current AHAContext or current resource that GALE is working on from within your code. There is a reserved variable available called 'aha' that is of the type `n1.tue.aha.ae.AHAContext`. This is the actual resource (a `java.util.Hashtable`) and all variables defined in Section 3.4.1 are available.

Accessing user model variables is done by using the notation: `#{variable}`. The variable could be an attribute name like 'knowledge' or 'suitability'. In that case, GALE will add the current concept to form a proper UM variable identifier (for instance 'tutorial.welcome.knowledge'). The variable could also directly identify such a UM variable.

Setting user model variables is done by using the notation: `#{variable, expr}`. The variable is as mentioned before. The expression can be any Java expression. This will set the specified user model variable to the value of the expression.

All GALE expressions and statements are pre-parsed into real Java code and then compiled to Java bytecode. This bytecode is reused whenever possible. (This process requires the use of Java JDK 1.6.)

GALE expressions are used in ConceptConfig files as shown earlier, are used in the translation of references to concepts into references to resources, and are also used in "variable" and "if" tags. This means that the expressions may appear in all pages that are used in a course, and they cause Java code to be compiled and executed. We are aware that this poses a security risk that needs to be studied and solved before GALE can be considered final.

References

1. De Bra, P., Houben, G.J., Wu, H., AHAM: A Dexter-based Reference Model for Adaptive Hypermedia, Proceedings of the ACM Conference on Hypertext and Hypermedia, pp. 147-156, Darmstadt, Germany, 1999.
2. De Bra, P., Smits, D., Stash, N., The Design of AHA!, Proceedings of the ACM Hypertext Conference, Odense, Denmark, August 23-25, 2006 pp. 133, and <http://aha.win.tue.nl/ahadesign/>, 2006.
3. Brusilovsky, P., Eklund, J., Schwarz, E., Web-based education for all: A tool for developing adaptive courseware. Computer Networks and ISDN Systems (Proceedings of the 7th Int. World Wide Web Conference, 30 (1-7), pp. 291-300, 1998.
4. Conlan, O., Hockemeyer, C., Wade, V., & Albert, D. (2002). Metadata Driven Approaches to Facilitate Adaptivity in Personalized eLearning systems. The Journal of Information and Systems in Education, 1, 38–44.
5. Henze, N., Nejd, W. Adaptivity in the KBS Hyperbook System. 2nd Workshop on Adaptive Systems and User Modelling on the WWW, workshop held in conjunction the World Wide Web Conference (WWW8) and the International Conference on User Modelling, 1999.

Appendix A: Differences between GALE description and first prototype

1. Authoring conceptual structures for GALE

Authoring (at the conceptual level) in GRAPPLE is done through a CAM editor. (CAM stands for *Conceptual Adaptation Model*.) For the first prototype of GALE the CAM editor is not yet ready and the output language for the editor, called GAL (for *Grapple Adaptation Language* or *Generic Adaptation Language*), is being developed in parallel with the first GALE prototype. The import module in the first prototype can import AHA! Version 3 domain/adaptation models (in the ".aha" format), and MOT applications. A slightly modified AHA! 3 Graph Author tool is included that can be used to author the domain/adaptation model of an application and transfer its output directly to GALE (by sending a 'setdm' event over the event bus).

The Graph Author tool included in GALE (initially) can be used exactly as in AHA! 3, referring to resources using names like 'file:/tutorial/xml/welcome.xhtml', rather than the new syntax 'aha:/tutorial/xml/welcome.xhtml' to refer to resources located in (under) the \$AHA_HOME directory (see Section 6.1 for a description of \$AHA_HOME).

In the Graph Author expressions can be entered using the GALE syntax as described in Section 6.6. Expressions need to be placed between curly braces ({}) to have them interpreted as GALE expressions (which are Java expressions or a sequence of Java statements, using a special \${...} notation to refer to DM or UM variables).

2. The LayoutProcessor

In the first prototype the LayoutProcessor generates an html frameset. In each frame a "view" will be presented. Each frame is associated with the same concept (used in the original request), but with the name of the view added. Requests for such views are handled without calling the LayoutProcessor again. This approach has two disadvantages compared to the table structures that will be used by the next GALE prototype: 1) each request by a user generates as many additional requests as there are frames (or views), thus adding communication overhead and more load on the server, and 2) the order in which requests resulting from a frameset are handled is unpredictable. When the adaptation to a page causes user model updates (which in GALE it can when there are conditionally included objects) and these updates have influence on the presentation of a navigation menu that presentation may be different depending on this order for handling the requests.

3. The DM service in the first prototype

Beside the default domain model service, GALE includes an AHA! 3 domain model service (nl.tue.aha.dm.AHA3Service). This service only supports the 'getdm' method and uses .aha files stored in the \$AHA_HOME/config directory as its source of domain model information. The AHA! 3 domain model service allows testing of GALE before the GRAPPLE authoring tools are available.

The AHA! 3 domain model service sends an 'updatedm' event for a particular application whenever its underlying .aha file changes (it monitors the file system).

4. Sending resources to GALE

GALE currently supports the Application Management tool, adapted from AHA! version 3. The AMt-tool can be opened by following a link on the main GALE starting page (index.html in GALE's root directory) and the graph author can be loaded from within the AMt-tool. All author files are located on the server in the directory '\$AHA_HOME/author/authorfiles'. Apart from providing access to the GraphAuthor the AMt tool also offers file transfer between your local file system and the server directory tree. You can create course content on your workstation and then transfer it to the GALE server using AMt. This works roughly like popular graphical ftp or ssh file transfer tools.

An author in GALE is a user that has his 'author' flag set to true. This means that any author is automatically a user of GALE and thus can login to GALE to view content. The GALE system administrator (by default the 'admin' account, password 'admin') can add new authors or promote existing users to authors using the AMt-tool.

Each author can manage a set of applications but no application can be managed by more than one author (in the current prototype). Trying to create an application that is already managed by someone else will result in an error.