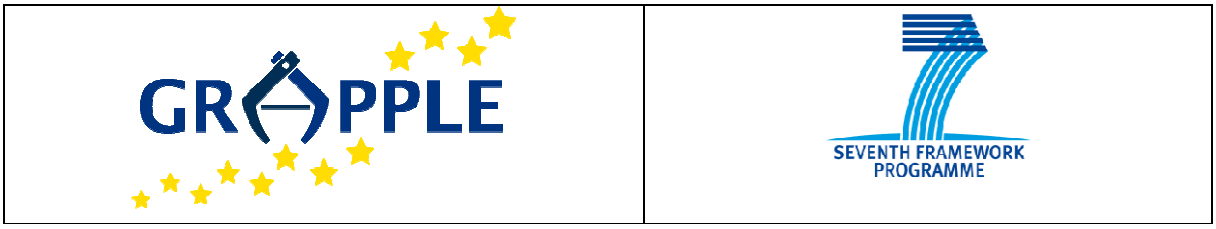



# GRAPPLE

D1.3b Version: 1.0

## “Stand-alone” Adaptive Learning Environment, first implementation

| | |
|---|---|
| **Document Type** | Deliverable |
| **Editor(s):** | Kees van der Sluijs |
| **Author(s):** | Paul De Bra (TUE), David Smits(TUE), Evgeny Knutov(TUE) |
| **Internal Reviews:** | Alexander Nussbaumer(UNIGraz), Andrea Lorenzon(GILABS) |
| **Work Package:** | 1 |
| **Due Date:** | 01-11-2009 |
| **Version:** | 1.0 |
| **Version Date:** | 10-02-2010 |
| **Total number of pages:** | 34 |

**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 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.

This deliverable is presented as an update to D1.3a in order to make it self-contained. Section 2 is essentially unchanged as it provides background information only. All other sections have been changed significantly.

# 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

# Tables and Figures

**List of Figures**



**List of Tables**

# List of Acronyms and Abbreviations

| | |
|---|---|
| AE | Adaptation Engine |
| AHA! (or AHA) | Adaptive Hypermedia Architecture (also used as prefix for other terms) |
| AHS | Adaptive Hypermedia System |
| ALE | Adaptive Learning Environment |
| AM | Adaptation Model |
| CAM | Conceptual Adaptation Model |
| CRT | Concept Relationship Type |
| DM | Domain Model (this includes the Adaptation Model) |
| GALE | GRAPPLE Adaptive Learning Environment |
| GRAPPLE | Generic Responsive Adaptive Personalized Learning Environment |
| GEB | GRAPPLE Event Bus |
| GUMF | GRAPPLE User Model Framework |
| KI | Knowledge Item |
| LMS | Learning Management System |
| LO | Learning Objective |
| 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 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[2] 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. In a stand-alone setting SOAP is only needed for the authoring tools to send a new Conceptual Adaptation Model (CAM) to GALE. SOAP is not needed once applications are running and need no updates from the author(s).

Many adaptive learning applications have been developed in the past. Although the research has focused on the adaptation 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 predefined presentation and adaptation templates, 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 in various ways. GALE uses an overall configuration file that can be used to provide defaults for adaptation and presentation. Deviating from defaults can be done on a per concept basis. Concept inheritance can be used to group concepts that behave similarly. GALE can easily be extended and configured to use additional means of configuring presentation and adaptation. GALE is delivered with a default "admin" application that defines a "layout concept" of which the attributes can be inherited by other concepts to use the same look and feel.

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. GRAPPLE allows users to log in and identify themselves through a single-sign-on infrastructure based on Shibboleth. 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 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 concept/content is presented. Details on how GALE communicates with other GRAPPLE components can be found in deliverable D7.1b.

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 used 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, with which GALE communicates through the GRAPPLE Event Bus (GEB). 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 *asynchronous* connectivity between such

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

[2] See Section 2.1 for an explanation of the term *concept* in adaptive learning applications.

components (through GEB), and offers storage of and reasoning over user information that is stored in a distributed way, using the GRAPPLE User Model Framework (GUMF). Information from GUMF 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. 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. With GALE it is possible to use AJAX to use incoming user model updates to update a page that is already shown to the user, but this may be uncomfortable as the text (or other elements in the presentation) may suddenly change without any action by the user.

# 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. The comparison was published in [6], independent of GRAPPLE. The models and systems that are compared are chosen because they have reasonably different and complementary functionality. More systems exist, even within the partners of the GRAPPLE consortium.

## 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 they are 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 GRAPPLE we distinguish between concept relationships that have a meaning in the *subject domain* and relationships that have a *pedagogical* meaning. The Domain Model (DM) contains subject domain relationships (like kind-of, same-as, special-type-of) and the Conceptual Adaptation Model (CAM) contains pedagogical relationships (like prerequisites). The types of pedagogical relationships that can be used are defined through the CRT tool (CRT stands for Concept Relationship Types). In the introductory comparison we present in this section we do not make that distinction between relationships in DM and CAM.

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 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.

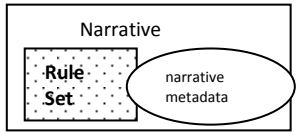
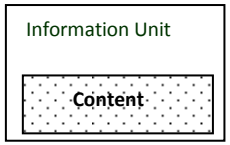
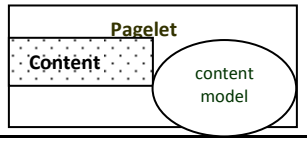
| | | | *AHAM* | AHA! | KBS Hyperbook | APeLS | Interbook |
|---|---|---|---|---|---|---|---|
| **concept** | Concept | an abstract representation of an information item from the application domain | ***Concept information:*** - <attribute, value> pairs - sequence of anchors - presentation specification **atomic concepts** - represent single fragment of information, **composite concepts** use child attribute to specify sequence of composite concepts | **Concepts** like in AHAM with restrictions also have type; and associated with a template, (can have only fixed number of attributes) | **Knowledge Item (KI)** - abstract representation of domain knowledge (e.g. if, class, run_method) - may also be a compound structure | Encapsulated **in Narrative Model Metadata** each narrative may add a new Concept and corresponding Narrative Rules   | **Glossary Entries** = **Domain Concepts** |
| | Concept relationship | represents semantic relationship between concepts | **Authored semantic linking** between concepts in a form of: <C1,C2,T,A> (T=type) - link; prerequisite; inhibit; part (compositional) (A = attribute value) | types of relationships: Fragment / Link / Contain | **Dependency graph of the KI-s =>** semantic links between Information Units (IU) Each IU is connected to one or more KI presenting which concept represents corresponding content in IU | In a form of a **relationships map in the Narrative Model** | **Concept relationships** (*navigational paths between glossary items*) types: 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 | **Knowledge Items (KI)** - index Project Units and Info. Units | n/a | **Glossary Entries** index Domain Concepts are indexed on textbooks (bookshelves) |
| **content** | Content data presentation | content unit structures | Pages and Fragments (page may consist of several fragments) | Pages and Fragments | Information Units (IU)   | Content info. is presented in a form of a Pagelet which may belong to a certain content group (see below)   | Content info is presented in a **Textbook** (shelf of textbooks) **Glossary** (glossary entries provide link to a certain textbook and connection to a certain domain concept) |
| | Content grouping | content grouping according to similarity of presentation, objectives, etc. | n/a | n/a | **Project Units** are mapped on Information Units Project Unit defines a number of KI has to be learnt to fulfill project goal | **Content Group** (content pagelets are organized in a group fulfilling the same Learning Objective LO) | 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

## 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.

| | | | AHAM | AHA! | KBS Hyperbook | APeLS | Interbook |
|---|---|---|---|---|---|---|---|
| **User Goal / objectives** | | overall learning goal stated by interaction with user | User follows a link to a (different) page | User follows a link to a (different) page | **- for Direct Guidance** and<br>**- for Goal based Learning:**<br>**Knowledge Items (KI) to be learnt are selected by user Goal** (with triggering event for AE) **consists of KI array**<br>**- 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**<br>**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 |
| | | preferences (font types, pictures, examples, size, etc.) | n/a | Link coloring (default or defined) | n/a | n/a | n/a |
| | | cognitive/learning style | n/a | Can be authored (not offered as a default option) | 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 | (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 Modeling 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.

Note that unlike in Table 2 we no longer list Interbook in Table 3 as Interbook can be emulated completely by AHA! so its adaptation model is a subset of that of AHA!.

| | | 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<br>**INT** - internal (system internal)<br>**EXT** - external (requires external feedback) | **INT:** AE pre- and post stores updated knowledge coefficients of the user knowledge in UM<br>**EXT**: tests | **INT:** AE updated knowledge coefficients of the user knowledge in UM (*readpages*)<br>**EXT**: authored tests | **EXT: direct feedback**<br>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<br>**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<br>- links adjustment<br>- page construction<br>- next page is shown to the user | - updated UM attributes<br>- links adjustment<br>- page construction<br>- 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**<br>**=>** rendering of the course model int***o* personalized Course Content**<br>**-** 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**<br>**- event**<br>**- condition**<br>**- action**<br>**+ propagation** (execution of other rules) **+ phase** (phase rules grouping) | **ECA** (Event-Condition-Action) rules, consisting of<br>**- event**<br>**- condition**<br>**- action**<br>**+ propagation** (execution of other rules) | **object oriented conceptual modeling 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**<br>based on **JESS** (Java Expert System Shell) is a rules language is based on CLIPS, which is a LISP-like language.<br>**Rule-Based language (Facts->Rules)**<br>(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 meting 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**<br>- Fragment inclusion/selection<br>- Links hiding/annotating/destination changing<br>- 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.

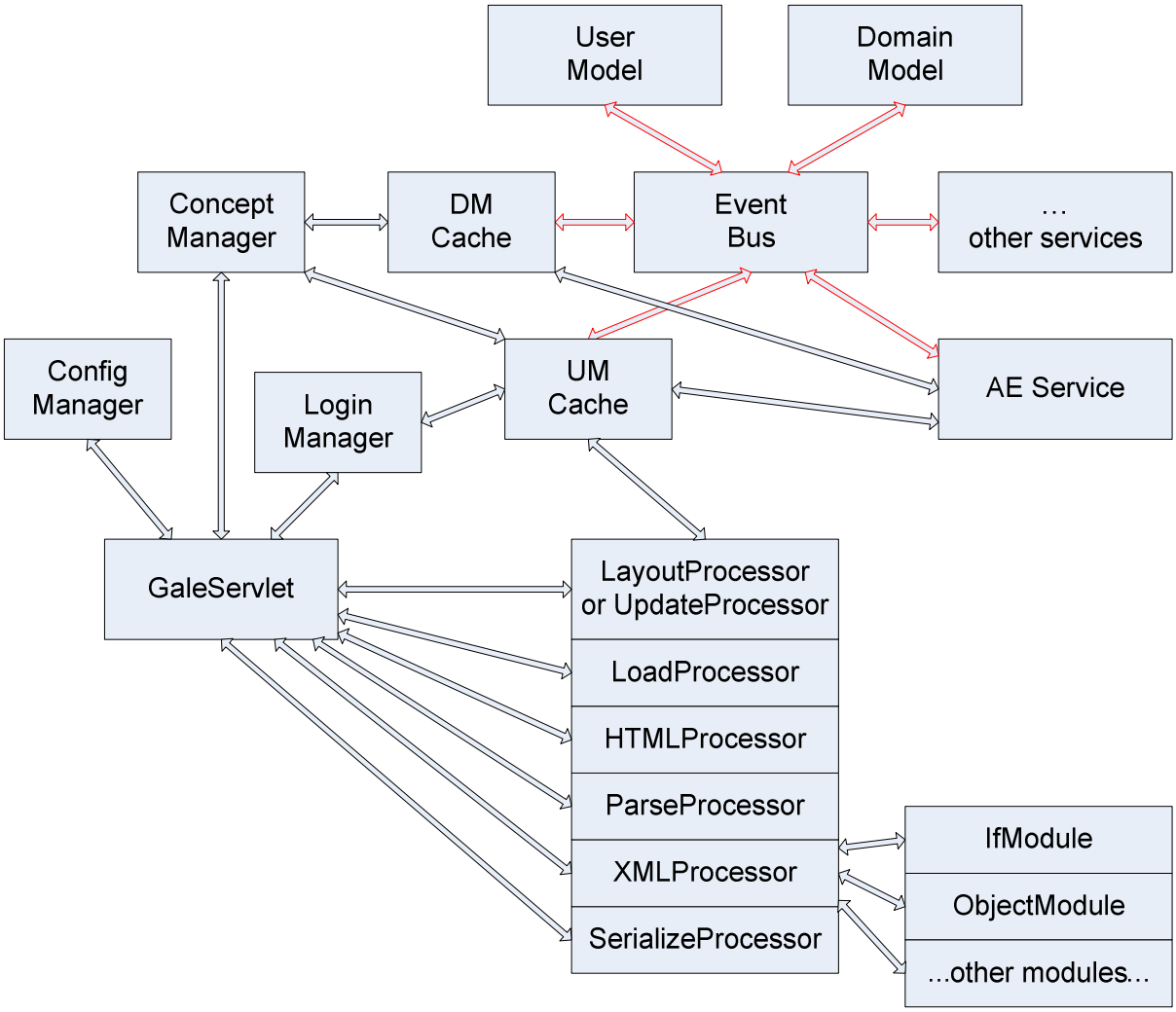



Figure 1: Core GALE architecture

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**[3]. 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.

GALE uses Spring (www.springsource.org), a well known Inversion of Control container, to configure and instantiate all components. In this deliverable we will often refer to “standard” functionality of GALE whereas in reality the functionality can be modified by changing the galeconfig.xml configuration file.

[3] This Event Bus is **not** the GRAPPLE Event Bus (GEB) but GALE connects to GEB through its internal event bus. The internal event bus can work synchronously whereas GEB can only work asynchronously.

The servlet, called GaleServlet, coordinates most of the work done by GALE as a direct result of requests by users. The servlet is a wrapper around GaleServletBean, that is set up by Spring. The real work is done by processors, modules and plug-ins. The UM cache and DM cache provide transparent access to the entire domain (and adaptation) model and user model. Whenever necessary they communicate with the event bus to retrieve, store or update domain model and user model data. The processors, modules and plug-ins load resources (like XML files), perform adaptation to some parts (in the case of XML, 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).[4] These structures are obtained by compiling *Conceptual Adaptation Models* (CAMs) for which WP3 develops authoring tools. The CAM editor (described in D3.3b) notifies GALE of (committed) changes to a CAM via the GRAPPLE event bus (described in D7.1b). 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 Modeling Framework (GUMF) developed in WP2 and 6. GALE needs a fast internal UM service for fine-grained UM data, whereas GUMF is used to communicate higher-level UM information between different GRAPPLE components.
- The communication within GALE is centred around the internal *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. Different implementations of the event bus exist. There is a highly "local" implementation that only uses method calls and there is a SOAP implementation (the local implementation is used by default). If external domain model or user model services are required, the SOAP implementation should be used. GALE communicates with other GRAPPLE components via the GRAPPLE Event Bus and it is thus not necessary to use the SOAP implementation within GALE.
- The *DM Cache* and *UM Cache* components ensure the immediate availability of data from the domain-, adaptation and user models. The DM Cache hides the fact that actual domain/adaptation model information may come from different services and may be stored differently. They exist purely for performance and ease-of-use reasons and are not elaborated upon in the remainder of this document.
- The *AE Service* listens to updates on the EventBus from DM and UM. It is responsible for keeping the DM Cache and UM Cache that reside in the Adaptation Engine part of GALE in sync.
- The *GaleServlet* 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. As mentioned above an overall Spring configuration file (called galeconfig.xml) is used to set up and configure every component in GALE. As we shall see below the default concept manager makes GALE even more adaptable, through processors and modules.
- The *Login Manager* is responsible for ensuring that the user is properly identified and authenticated. GALE contains a DefaultLoginManager for stand-alone use. Additional login managers can be used to make GALE collaborate with external authentication services. A LinkLoginManager is available for simple (but insecure) single sign-on between an LMS and GALE. (This was used in the initial prototype of the GRAPPLE infrastructure.) Within GRAPPLE the IdPLoginManager is the preferred

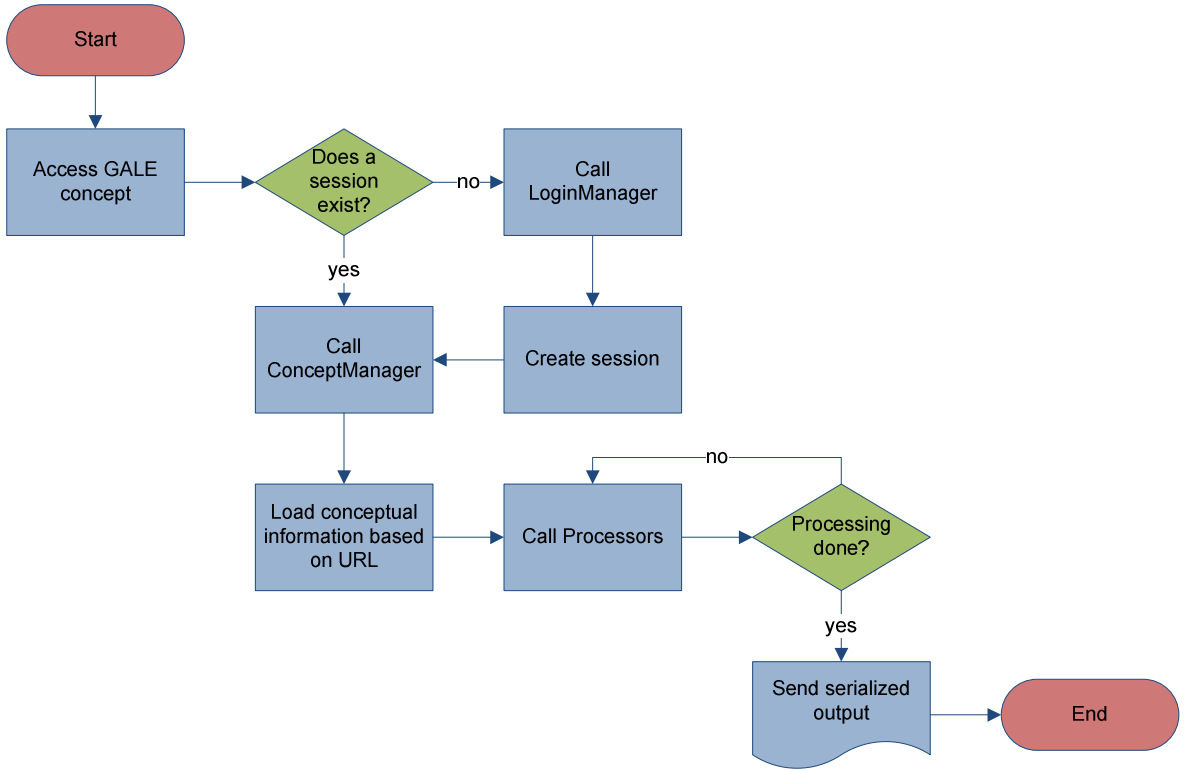
[4] In the GRAPPLE Authoring Tool (GAT) there is a separation between DM and CAM but in GALE there is not: both together form the GALE Domain Model.

gin manager is it uses authentication and authorization through Shibboleth[5]. Deliverable D7.1b has more information on this.

- The *Concept Manager* is responsible for determining the requested concept. Typically it will use the http request to somehow identify the concept being requested (this is what the default implementation does). Since it is called before any processor, it is an ideal place to interpret the request URL in some special way. This is used by the default implementation to allow ${home} and ${lib} URLs to access the GALE home directory and GALE WEB-INF/lib directory. GALE also supports "plug-ins" for handling special types of requests. For instance the "Password" 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 GaleServlet 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.
2. The user needs to be identified. GaleServlet calls the *login manager* to obtain information about the user. For stand-alone GALE use (not using the Shibboleth 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 (DefaultLoginManager) 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.*

[5] See http://shibboleth.internet2.edu/.

c. Since UM cache will not have cached the user model, it will communicate with the user model service through the event bus.

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. GaleServlet now calls the *concept manager* in order to find out how to handle the request. If the request is for a concept, the concept manager will determine the identity of the requested concept. Otherwise it tries to meet the request in some other way, depending on the specific implementation.

4. 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 Spring configuration file indicates 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 that is called is the *UpdateProcessor*. It signals an EventManager that the 'access concept' event has occurred. The EventManager has handlers defined in galeconfig.xml. The default EventAccessHandler executes the event code of the concept as defined in the domain model. The resulting changes to the user model are posted on the event bus, and subsequent changes made by any registered UM service are integrated in the UM cache.

   b. The first processor touching the resource is the *LoadProcessor*. It is responsible for retrieving the actual resource associated with the concept (which can be a local file or can be a resource that has to be retrieved from some other server through http). The name of the resource is found in the `resource` attribute of the concept. Note that this attribute value may be "computed" and may have been updated by the *UpdateProcessor* that runs first. 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.

   c. Optionally the *LogProcessor* then adds an entry to a global log file (access.log by default). The id of the user, date, request, referrer (that may be present in the HTTP request), the name of the requested concept and the resulting resource are logged, for possible later analysis.

   d. The next processor checking out the data is the *HTMLProcessor*. If the mime type is some form of HTML (but 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.

   e. 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, using the open source dom4j parser.

   f. 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 Spring configuration file indicates 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 adding (a configuration for) them to galeconfig.xml. (The device adaptation described in D4.4 uses this GALE extensibility.)

   g. Optionally, the *LayoutProcessor* generates a frame-like structure using tables, by creating an (in-memory) XML document that contains the views (any class that implements the LayoutView interface) embedded in a table that defines the layout. This XML document has a placeholder element where the actual content should be. The LayoutProcessor then decreases the level and sets the "redo" flag on the resource. This will cause the container (GaleServlet) 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 LayoutProcessor a second time, this is recognized and the actual adapted content is added to the table structure.

h. When the DOM tree is adapted the *SerializeProcessor* generates the textual XML representation and presents that to GaleServlet as an *InputStream*. For resource types that do not have specific processors associated with them GaleServlet 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 GaleServlet calls a special PlugIn that may generate its own output. These plug-ins set the level to 100, which for GaleServlet means that the output was already generated by the plugin itself. Examples of such plug-ins are the Password and the Logout PlugIn.)

## 3.3 Configuration

In GRAPPLE adaptive applications are derived from a *Conceptual Adaptation Model* or CAM (see deliverable D3.3b). The CAM defines the concepts of the application and pedagogical relationships between concepts. In order to be as generic as possible 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 and which are defined while authoring the *concept relationship types* (or CRTs).
- 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.) For each concept GALE uses a UM attribute *layout* to determine the layout to be used for that concept.

GALE uses an extendable configuration mechanism. Configuration information can determine the (default) layout, the way in which user model variables should update link presentation, the processors to use, the stylesheet to use and much more. The 'current configuration state' in GALE may depend on the user model, the concept requested, the actual http request, elements in galeconfig.xml, and anything else available to GALE.

The Spring configuration file (galeconfig.xml) sets up a *config manager* and a set of *config resolvers*. Requests for configuration information are directed to the *config manager*, and are identified by a URI. The fragment part of the URI is used to encapsulate the name of a specific attribute requested; the remainder of the URI is used to identify a specific *config resolver*. For instance, the PresentationConfig class is a *config resolver* identified by the URI 'gale://gale.tue.nl/config/presentation'. It defines several attributes and one of them is called 'css'. This would allow a request to the config manager with URI 'gale://gale.tue.nl/config/presentation#css'. The actual value returned depends on the implementation of PresentationConfig, which in turn may use anything available to GALE to produce a result (like the current user model, the current concept, the http request, etc.).

Below is a list of the configuration URIs available (and used) by default. The URIs use a "gale:" protocol part in the URI which indicates to gale that this is a special gale URI. No request using "gale:" is ever sent over the network, so it is not a "real" protocol.

- `gale://gale.tue.nl/config/presentation#css`
  - returns: `java.lang.String`
  - The default css is '${home}/gale.css'. This can be overwritten in galeconfig.xml, by specifying the 'defaultCSS' property in the 'PresentationConfig' bean.
- `gale://gale.tue.nl/config/presentation#layout`
  - returns: `org.dom4j.Element`
  - Might return null to indicate no layout should be used (the default). This is overwritten by a '#layout' attribute in the current concept. This attribute should contain serialized XML that makes up the layout to use.
- `gale://gale.tue.nl/config/processor#list`
  - returns: `java.util.List<nl.tue.gale.ae.ResourceProcessor>`

- Relies on galeconfig.xml to supply a list of processors. It expects the named bean ‘processorList’ to contain such a list.

- `gale://gale.tue.nl/config/link#classexpr`
  - returns: `java.lang.String`
  - Returns the expression that should be evaluated to determine the css class of links to a particular concept, based on user model variables about that concept. The default is ‘(${#suitability}?(${#visited}>0?\"neutral\":\"good\"):\"bad\")’. This can be overwritten on a per concept basis by the domain attribute ‘#link.classexpr’.

The default *config resolvers* can be extended (sub classed), to allow more complex means of storing and distributing configuration information.

## 3.4 Defining the Processor pipeline

The configuration data in ‘gale://gale.tue.nl/config/processor#list’ is used as a list of processors. After the concept manager was successful in determining the identity of the concept to show, the processor list is used to call each individual processor.

A resource in GALE is basically a collection of named objects. Each resource has a 'level', which 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.

The minimum and maximum levels for each processor can be changed in the Spring configuration file, to change the order in which processors are called. However, one should do this with care because the output level of a processor depends on the actions a processor has taken and cannot be configured in the Spring configuration file. So the possible changes in the execution order are somewhat limited.

If the level of a resource is still 0 after all processing is done, the GaleServlet 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 GaleServlet 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

Though we use Spring to set up and configure all components of GALE, and Spring normally recommends to make relations between components explicit in configuration files and avoid using hard-coded references to beans, we explicitly chose to ignore this based on some technical reasons[6]. This means that almost all components are accessible through the resource. There is a convenience wrapper called GaleContext, that contains static and instantiated methods to access almost all components. A reference to GaleContext is available to GALE expressions as the variable ‘gale’.

# 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.gale.ae.ResourceProcessor`. It is called by the GaleServlet with the current resource as a parameter. If the processor decides to do anything with the

[6] GALE allows arbitrary Java code in its adaptation model. To allow the greatest flexibility we want this code to have access to all GALE components. This code is unavailable to Spring, so we cannot rely on Spring to inject references to the components.

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 used are based on the list provided by the config key 'gale://gale.tue.nl/config/processor#list'. 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.

By default the configuration key 'gale://gale.tue.nl/config/processor#list' uses a Spring bean from galeconfig.xml called 'processorList' to retrieve the processor list. New processors, modules, plugin, etc. can be defined in galeconfig.xml.

In the descriptions that follows the variable 'gale' refers to an instance of GaleContext. With each processor we list the minimum and maximum level at which the processor can operate on a resource. The minimum and maximum level for a processor is listed between parentheses in section 4.1. A processor can only be executed when the current concept's level is in the right range and by setting the concept's level, the processor can indicate which other processors can be executed next. Even when a processor leaves the level unchanged it is not executed again.

## 4.1 Helper Processors

**LoadProcessor (0-2)**

The LoadProcessor makes the resource data available as an inputstream. It tries to retrieve a URL for the resource by using the 'resource' attribute of the current concept. If that is not found, it will use 'gale:/empty.xhtml' as the URL by default. The URL's location is opened and stored as an input stream in the resource under the name 'stream'. The mime type is based on the URLs extension and stored under 'mime'.

The level of the resource is set to 5.

**LogProcessor (2-10)**

The LogProcessor logs a request for a resource in an access.log file. The location of this file depends on the LogManager defined in galeconfig.xml. By default this location is the 'log' directory inside the 'tomcat/webapps/gale' directory. To change this to e.g. /home/gale/logdir you would add:

```
<property name="logDir" value="file:/home/gale/logdir" />
```

The user id, access time, request URL, referrer URL, concept URI and actual resource are logged.

Other GALE components may also use the logDir property to choose a location for special-purpose logs. The MCModule for multiple-choice tests for instance (see Section 6.3) creates a mctest.log file with all test results.

The level of the resource is unaltered. (So whether the LogProcessor is used or not does not influence which other processors can be executed.)

**HTMLProcessor (5-5)**

The HTMLProcessor creates proper xhtml data from any html data. If the mime type indicates html data, then the data in the inputstream (gale.stream()) is used to create a new inputstream that contains proper xhtml data. This new inputstream 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 inputstream of the resource (gale.stream()) is used as the data to parse. The resulting xml DOM document is stored as an `org.dom4j.Element` 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 (gale.xml()) back to an inputstream, if the mime type suggests xml data. The resulting inputstream is stored in the resource as the new 'stream' object.

The level of the resource is set to 95. This tells the GaleServlet that all processing has been done but that the result has not yet been returned to the browser. The GaleServlet will do this. Should the level be 100 the GaleServlet 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.gale.ae.processor.xmlmodule`). The modules that the XMLProcessor should use are specified in galeconfig.xml.

When the XMLProcessor traverses the DOM tree, it passes control to the various modules as it encounters their tagnames. The modules can change the structure of the DOM tree. (The XMLProcessor itself does not change the document at all.) In Section 6.2 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 (10-50)

The LayoutProcessor allows the use of separate views (generated by *view processors*) with a presentation structure, implemented using html tables. The various views are defined as classes that implement `LayoutView` (found in the package `nl.tue.gale.ae.processor.view`). It creates an (in-memory) XML document that contains the views embedded in a table that defines the layout. This document has a placeholder element where the actual content should be. The LayoutProcessor then decreases the level and sets the “redo” flag on the resource. This will cause the container (GaleServlet) 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 LayoutProcessor a second time, this is recognized and the actual adapted content is added to the table structure.

## 4.4 The AjaxProcessor (10-80)

The AjaxProcessor inserts a reference to a piece of JavaScript code (`${home}/ajax.js` by default) that can be used to asynchronously refresh pages to reflect changes in the user model. The AjaxProcessor generates a unique id (guid) that identifies this resource so that the AjaxPlugin can decide whether this resource needs to be updated (in the browser) or not.

The ajax.js code causes the browser to periodically send a request to the server to check whether the browser view needs to be updated. The AjaxPlugin always generates an AjaxEvent that can be used to update a record of the time the user is reading the current page.

By default the AjaxProcessor is not used. To use Ajax you have to uncomment its definition in the Spring configuration file.

The level of the resource is set to 50.

## 4.5 The PluginProcessor (0-0)

The PluginProcessor 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/gale/concept/gale://gale.tue.nl/admin/index?plugin=Logout').

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 GaleServlet that it need not do anything any more.) This is done for instance by the AjaxPlugin when it decides that the current page need not be updated.

# 5 Communication with the GALE Domain Model and User Model

The domain model for 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 URI), 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* request[7], 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 6.5). This code is executed by an `nl.tue.gale.ae.event.EventHandler` specified in galeconfig.xml (by default this is `nl.tue.gale.ae.event.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 of 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 a URI that identifies the concept, a specific attribute and the user. Concept names themselves can be any URI that includes a host name, but does not use the user-info and fragment parts. The user-info part is used to identify a user and the fragment part is reserved for the attribute.

An example of a URI for a concept is:

```
gale://grapple-project.org/Milkyway/Planet
```

Note that the URI is purely used for identification. It does not refer to a real protocol and/or a real hostname.

Concepts have attributes and properties. The attributes of a concept define information that should be calculated on the fly or stored in the user model. Properties of a concept define DM information that can be used for adaptation or for presenting information (like a "title", or the URL of an image). Attributes also have properties, mainly used for specifying whether the attribute is read-only, persistent, public, should be kept stable, etc., but perhaps also used for "printable" information such as a label or title. Attributes and properties define the variables the author can refer to in code (for the notation see section 6.3).

Attributes are referred to using the *fragment* part of the URI. To refer to the "knowledge" attribute (value) of the property we can use:

```
gale://grapple-project.org/Milkyway/Planet#knowledge
```

Properties are referred to using the *query* part of a URI:

```
gale://grapple-project.org/Milkyway/Planet#image?title
```

refers to the "title" property of the "image" attribute of the Planet concept.

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.

---

[7] 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. Stability was implemented in AHA! before but the current GALE version does not yet support it.

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.gale.dm.data` and `nl.tue.gale.um.data` and include `Concept`, `Attribute`, `ConceptRelation`, `UserEntity` and `EntityValue`.

## 5.1 The EventHash class

The class `nl.tue.gale.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 must 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.

## 5.2 The GALE DM service

The adaptation 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.

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.

Below 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):

| Method | Parameters | Result | Description |
|---|---|---|---|
| **getdm (l)** | uri:<concept-uri> | A list of the serialized entities that are part of the specified concept. | Retrieves domain model information. |
| **setdm (l)** | A list of the serialized entities that should be set. | 'result:ok' if the operation succeeded. 'result:<exception-class>' if the operation failed. | Sets domain model information. |
| **listdm (l)** | root: <concept-uri> | A list of all concept URI's starting with 'root'. | Lists all concept URI's that start with the specified URI. |
| **querydm (l)** | query: <hibernate-query> | The result of the hibernate query. | Retrieve domain model information based on a hibernate query that has direct access to the database. |
| **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

The default domain model service sends 'updatedm' events whenever changes made by a call to 'setdm' have resulted in a change to the database. 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).

## 5.3 Additional DM services

Some additional DM services are installed by default. They provide convenience access to DM information in the absence of authoring tools. The AHA3Service reads AHA! 3 style .aha files and makes the domain model and adaptation model therein available through the event bus. The format of .aha files falls outside the scope of this document. The GDOMService reads .gdom xml files. These files represent a domain and adaptation model in a very similar way as GALE uses it internally. Both .gdom and .aha files should be stored in your GALE_HOME/config directory.

The XML Schema of GDOM:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
targetNamespace="http://gale.tue.nl/gdom" xmlns:gdom="http://gale.tue.nl/gdom">
  <xs:element name="gdom">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="gdom:concept"/>
        <xs:element maxOccurs="unbounded" ref="gdom:relation"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="concept">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="gdom:attribute"/>
        <xs:element ref="gdom:event"/>
        <xs:element maxOccurs="unbounded" ref="gdom:property"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:anyURI"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="attribute">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gdom:default"/>
        <xs:element ref="gdom:event"/>
        <xs:element maxOccurs="unbounded" ref="gdom:property"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:string"/>
      <xs:attribute name="type" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="default" type="xs:string"/>
  <xs:element name="relation">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="gdom:inconcept"/>
        <xs:element ref="gdom:outconcept"/>
        <xs:element maxOccurs="unbounded" ref="gdom:property"/>
      </xs:sequence>
      <xs:attribute name="name" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="inconcept" type="xs:anyURI"/>
  <xs:element name="outconcept" type="xs:anyURI"/>
  <xs:element name="event" type="xs:string"/>
  <xs:element name="property">
    <xs:complexType>
      <xs:attribute name="name" use="required" type="xs:string"/>
      <xs:attribute name="value" use="required" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Each .gdom file consists of a series of concepts and concept relations (elements 'concept' and 'relation' respectively). Each concept element has 'attribute' and 'property' children, and one 'event' child element. The URI of a concept is specified in the 'name' attribute. Each attribute element has properties, an 'event' child and a 'default' child. The contents of 'default' and 'event' elements can be arbitrary Java code mixed with

Pseudo code as described in section 6.5. An example .gdom file can be found in the 'config' subdirectory of the gale.war file.

## 5.4 Updating the Domain Model

To support external authoring tools, GALE allows domain and application models to be sent through web services. In a simple environment, no special configuration in GALE is required, because the web service that receives and processes new models (through CAM XML files) is running by default. It's WSDL can be found at 'http://dyn070.win.tue.nl:8080/gale/services/AddContent?wsdl' (or your local version of the service). This webservice has only one parameter: the serialized CAM XML.

In a more complex environment, GALE can be configured to communicate with a GRAPPLE Event Bus (GEB). This is done by setting the 'useGEB' property in galeconfig.xml to 'true'. You should also update the 'rootGaleUrl' property, which should point to the location of your installed GALE on the internet. In the 'gebManager' part of galeconfig.xml you should update 'gebURL' and 'baseURL' appropriately.

When GALE is configured to use GEB, it will listen for the 'updateCAMModel' event. The body of this event should contain the serialized CAM XML. The event will be followed by an 'updateCAMModelResponse' event that indicates whether the update was successful ('true' or 'false').

## 5.5 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.gale.um.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)** | uri:<entity-uri> | 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)** | uri: <um-variable-uri> | The specified variable result in a EntityValue wrapper. | Retrieves user model variables. |
| **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 (I)** | 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> | A list of serialized user model variables and/or UserEntity’s that result from executing the specified Hibernate query. | The ‘nl.tue.gale.um.data’ package specifies a set of Hibernate enabled classes that are used to persist 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 variable when it does not have its value in the local cache. The UM service is expected to send ‘updateum’ events whenever a user model variable changes.

# 6 Authoring Guide

Authoring the conceptual structure and adaptation for an application is done using DM and CAM editors. These parts 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 5 or later) and Tomcat version 6. If you want to compile GALE you also need Apache Maven.

JDK 5: http://java.sun.com/javase/downloads

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

Apache Maven: http://maven.apache.org/download.html

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/gale/trunk.

After you have the sources and set up your JDK and Maven 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 ‘mvn package’ in the ‘master’ directory of the GALE distribution. This will create gale.war in the ‘gale/target’ subdirectory. You can copy this file to your Tomcat ‘webapps’ directory. Now you are ready to start Tomcat and go to ‘http://localhost:8080/gale/’ on your machine. This should bring up a standard home page that refers you to the AMt tool to start authoring applications.

GALE can be configured to use a home directory for storing resources and domain model information. By default this is the directory where the servlet container extracts the war file (in Tomcat this would be ‘$CATALINA_HOME/webapps/gale’). This directory will be overridden each time you install a new version of

GALE by updating the .war file, which will most likely be undesirable. To prevent this you can set up your own GALE home directory.

To set up your own GALE home directory, you need to specify the GALE_HOME environment variable. If you start GALE after setting the variable it will use this directory as the base of its configuration and resource files. Some functions of GALE (like the ‘admin’ pages) only work when GALE is able to find the resources in the GALE home directory. When setting up a different home directory than the default, you will have to copy some files to your new home directory. You can find these files and directories in the /webapps/gale directory of your servlet container:

admin
author
config
tutorial
empty.xhtml, gale.css, ajax.js

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 adaptive pages

Any file can be served by a GALE web server. The *concept manager* is the first component called to interpret the request URL to the server. It is supposed to identify the requested concept and return its URI, but it could also interpret the request URL as a location for a specific resource and return the resource without processing (which is what the default concept manager allows, see section 3.1).

By default, the first (normal) processor in the chain would be the load processor. It will load the resource specified in the 'resource' attribute of the current concept (or 'gale:/empty.xhtml' if no such attribute is found). At this point, any processor is free to modify the resource in any way or leave it unaltered.

By default GALE has extensive support for adapting xml and xhtml pages. The XMLProcessor adapts files passing through GALE when they are recognized as xml. It uses a depth-first algorithm to walk the xml tree. If any tag matches a tag specified in galeconfig.xml (in the part where the XMLProcessor is created) an XMLModule is called to process the specified tag. The XMLProcessor is namespace aware, depending on its configuration in galeconfig.xml. XMLModule's are linked to tags using the notation: '{namespace}tag'. For instance, the IFModule is coupled to the tag '{http://gale.tue.nl/adaptation}if', indicating the 'if' tag in the namespace 'http://gale.tue.nl/adaptation'. If the namespace part is omitted in galeconfig.xml, the matching process will ignore the namespace.

GALE comes with a number of XMLModule's available by default. In the description below the 'gale:' prefix refers to the namespace 'http://gale.tue.nl/adaptation'. GALE expressions are described in 6.3.

<table>
<tr><th>example</th><th>attributes</th><th>description</th></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}if -> nl.tue.gale.ae.processor.xmlmodule.IfModule</td></tr>
<tr><td rowspan="2"><pre><gale:if expr="${#suitability}">
  <gale:then>
    This concept is suitable :)
  </gale:then>
  <gale:else>
    This concept is unsuitable!
  </gale:else>
</gale:if></pre></td><td>expr</td><td>a GALE boolean expression</td></tr>
<tr><td colspan="2">The expression in 'expr' is evaluated. If it returns true, the if tag is replaced by the content of the 'then' block. If it returns false, the if tag is replaced by the content of the (optional) 'else' block.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}adapt-link, {http://gale.tue.nl/adaptation}a -> nl.tue.gale.ae.processor.xmlmodule.AdaptLinkModule</td></tr>
<tr><td rowspan="3"><pre><gale:a href="welcome"/>

<gale:a href="tour"
        exec="#{tour#start,true};"/>

<a href="/tutorial/welcome">
  <gale:adapt-link
      exec="#{#visit-tutorial,true};"/>
</a></pre></td><td>href</td><td>a relative URI string</td></tr>
<tr><td>exec</td><td>an optional list of GALE statements to be executed when the link is clicked</td></tr>
<tr><td colspan="2">If the tag to be handled is enclosed in a parent 'a' tag (any namespace), it will adapt the parent tag. The configuration key 'gale://gale.tue.nl/config/link#classexpr' is read and the</td></tr>
</table>

<table>
<tr><td></td><td colspan="2">resulting expression evaluated. The result is put in the class attribute. The 'href' attribute is updated to contain an absolute URL to the specified concept URI.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}object -> nl.tue.gale.ae.processor.xmlmodule.ObjectModule</td></tr>
<tr><td rowspan="3"><pre><gale:object data="../header.xhtml"/>

<gale:object
    name="fragments/programming"/>

<gale:object
    name="gale://gale.tue.nl/tutorial"/></pre></td><td>data</td><td>a relative URL to the current 'resource' URL</td></tr>
<tr><td>name</td><td>a relative URI to the current concept URI</td></tr>
<tr><td colspan="2">Either 'name' or 'data' should be specified. 'data' is used to refer to resources, possibly located within the same directory structure as the current resource. 'name' is used to refer to concepts, possibly relative to the current concept. The specified object is loaded, its content send through the same processor structure and then inserted in the XML structure.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}handler, {http://gale.tue.nl/adaptation}plugin -> nl.tue.gale.ae.processor.xmlmodule.PluginModule</td></tr>
<tr><td><pre><gale:plugin>
  <gale:name>logout</gale:name>
  <gale:linkdescription>Logout
  </gale:linkdescription>
</gale:plugin>

<gale:plugin>
  <gale:name>export</gale:name>
  <gale:linkdescription>Export concept
  </gale:linkdescription>
  <gale:param name="root"
    value="gale://gale.tue.nl/tutorial"/>
</gale:plugin></pre></td><td colspan="2">This module will replace the tag with a link to a plug-in. The link will have a description as specified in the 'linkdescription' sub element. Extra parameters to the plug-in can be specified using the 'param' sub element.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}variable -> nl.tue.gale.ae.processor.xmlmodule.VariableModule</td></tr>
<tr><td rowspan="3"><pre><gale:variable name="#visited"/>

<gale:variable name="->(parent)?type"/>

<gale:variable
  expr="gale.request().getRequestURL()"/>

<gale:variable expr=
  "${->(parent)[0]}.getEventCode()"/></pre></td><td>expr</td><td>a GALE expression</td></tr>
<tr><td>name</td><td>a relative URI resulting in an attribute URI</td></tr>
<tr><td colspan="2">If 'name' is specified it is resolved against the current concept and the resulting attribute value is returned as a string. Otherwise, if 'expr' is defined, the expression in 'expr' is evaluated. It's result is returned as a string.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}for -> nl.tue.gale.ae.processor.xmlmodule.ForModule</td></tr>
<tr><td rowspan="3"><pre><gale:for var="concept"
          expr="${<-(parent)}">
  <gale:variable
      expr="${%concept?title}"/>
  <br/>
</gale:for>

<gale:for var="text"
    expr="new String[] {"h1"}">
  <gale:variable
      expr=""%text""/>
</gale:for></pre></td><td>expr</td><td>a GALE expression resulting in an Iterable object</td></tr>
<tr><td>var</td><td>the placeholder variable that can be referred to in the body of the for.</td></tr>
<tr><td colspan="2">Replaces the for tag by a repetition of the content of the for tag, for each element in the Iterable object that is the result of the expression. In each iteration an occurrence of the literal string '%var' inside an attribute, where var is replaced by the string specified in the var attribute, is replaced by an object in the Iterable object.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}count -> nl.tue.gale.ae.processor.xmlmodule.CountModule</td></tr>
<tr><td rowspan="2"><pre><gale:count
  uri="gale://gale.tue.nl/tutorial"</pre></td><td>uri</td><td>the absolute URI of a concept</td></tr>
<tr><td>method</td><td>the string "todo" or "done"</td></tr>
</table>

<table>
<tr><td>method="todo"/></td><td colspan="2">Replaces the count tag by a number indicating the number of read or still to be read pages, depending on the method used. This number is calculated by obtaining all concepts whose URI begins with the uri specified. All concepts whose visited attribute is 0 are counted toward pages still to be read, otherwise they are counted towards pages read.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}attr-variable -> nl.tue.gale.ae.processor.xmlmodule.AttrVariableModule</td></tr>
<tr><td rowspan="3"><gale:a><br><gale:attr-variable name="href"<br>expr="${->(parent)[0]}.getUri()"/><br><gale:variable<br>expr="${->(parent)?title}"/><br></gale:a></td><td>name</td><td>the name of the attribute to replace in the parent element</td></tr>
<tr><td>expr</td><td>a GALE expression</td></tr>
<tr><td colspan="2">Replaces the attribute called name in the parent element of the attr-variable element with the result of the expression.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}view -> nl.tue.gale.ae.processor.xmlmodule.ViewModule</td></tr>
<tr><td rowspan="3"><gale:view name="file-view"<br>file="gale:/tutorial/header.xhtml"/><br><gale:view name="static-tree-view"/><br><gale:view name="file-view"<br>content="${#description}"/></td><td>name</td><td>the name of the view to show as defined in galeconfig.xml</td></tr>
<tr><td>*</td><td>redirected as parameter to the specified view</td></tr>
<tr><td colspan="2">Replaces the view tag with the content generated by the specified view, a subclass of nl.tue.gale.ae.processor.view.LayoutView. A view is obtained by getting configuration key 'gale://gale.tue.nl/config/presentation#view-name', where name is replaced with the actual view name. By default this will return a view as specified in galeconfig.xml.</td></tr>
<tr><td colspan="3">{http://gale.tue.nl/adaptation}test -> nl.tue.gale.ae.processor.xmlmodule.TestModule</td></tr>
<tr><td>for an example see 6.3</td><td colspan="2">Allows the insertion of an adaptive test that updates the user model, based on its result.</td></tr>
<tr><td colspan="3">html -> nl.tue.gale.ae.processor.xmlmodule.HTMLModule</td></tr>
<tr><td>works on every html tag, hence no example</td><td colspan="2">6.2.1.1.1.1.1.1.1 Adds an appropriate head section that links to the selected style sheet and includes a proper base URL for the browser to use, in order to find relative resources (like images) specified in the page.</td></tr>
</table>

## 6.3 Adaptive multiple choice tests

Using the gale:test element you can include multiple choice tests that can update the user model based on the result. The test questions and answers can be adaptive. Here we explain the format in which these tests can be included in the page or domain model.

We start by giving a 'small' example:

```
<gale:test
    title="Test for the Introduction"
    action="#{testintro#knowledge,value}; #{testintro#done,true};"
    expr="!${testintro#done}"
    ask="2"
    alt="[You are not allowed to repeat this test.]"
    verbose="true">
  <gale:question
      answers="2"
```

```
      right="1">
    True or False:
    According to the definition given in this course, this course text is a
    hypertext.
    <gale:answer correct="true">
      False
      <gale:explain>
        The course text is indeed not a hypertext but a hyperdocument, because
        a hypertext is the document plus the software, not the document alone.
      </gale:explain>
    </gale:answer>
    <gale:answer correct="false">
      True
      <gale:explain>
        The course text is a hyperdocument, not a hypertext, because a
        hypertext is the document plus the software, not the document alone.
      </gale:explain>
    </gale:answer>
  </gale:question>
  <gale:question
      answers="4"
      right="1">
    What makes it difficult to offer a search facility for this course text?
    <gale:answer correct="true">
      The course text is adaptive.
      <gale:explain>
        Indeed, because the course text is adaptive the information that is
        shown on a page depends on what you have studied already. A search that
        takes into account what would be shown on each page, at the time the
        search is performed, is very difficult to implement.
      </gale:explain>
    </gale:answer>
    <gale:answer correct="false">
      The course is only available after a login procedure.
      <gale:explain>
        The login procedure may prevent a search engine like Google from
        reaching the course, but does not prevent us from creating our own
        search facility. However, because the course text is adaptive the
        information that is shown on a page depends on what you have studied
        already. A search that takes into account what would be shown on each
        page, at the time the search is performed, is very difficult to
        implement. So the adaptive aspect of the course text was the correct
        answer.
      </gale:explain>
    </gale:answer>
    <gale:answer correct="false">
      Some information is hidden in images.
      <gale:explain>
        In general it is a problem that information may only be available in
        images. However, in this course the images only serve as additional
        illustrations. Everything is explained in the text, so searching just
        the text is sufficient. However, because the course text is adaptive
        the information that is shown on a page depends on what you have
        studied already. A search that takes into account what would be shown
        on each page, at the time the search is performed, is very difficult
        to implement. So the adaptive aspect of the course text was the correct
        answer.
      </gale:explain>
    </gale:answer>
    <gale:answer correct="false">
      The pages are not available as a whole, but only accessible through links.
      <gale:explain>
        By following links to retrieve all the pages one by one all the
        information can be "harvested" in order to create a search database.
        This is exactly what search engines like Google have to do. However, in
        this course the images only serve as additional illustrations.
        Everything is explained in the text, so searching just the text is
        sufficient. However, because the course text is adaptive the
        information that is shown on a page depends on what you have studied
        already. A search that takes into account what would be shown on each
```

```
        page, at the time the search is performed, is very difficult to
        implement. So the adaptive aspect of the course text was the correct
        answer.
      </gale:explain>
    </gale:answer>
  </gale:question>
  <gale:result>
    Your score was <gale:variable expr="${testintro#knowledge}.intValue()" />.
    <br />
    <gale:if expr="${testintro#knowledge}&lt;100">
      <gale:then>
        <br />
        <br />
        You are not allowed to repeat this test, but you can now continue to
        the <gale:a href="welcome">chapter index</gale:a> to choose a(nother)
        chapter to study.
      </gale:then>
      <gale:else>
        You can now continue to the <gale:a href="welcome">chapter index
        </gale:a> to choose a(nother) chapter to study.
      </gale:else>
    </gale:if>
  </gale:result>
</gale:test>
```

This example illustrates all aspects of adaptive tests in GALE. The gale:test element is bound to the actual MCModule xml module. It replaces the test element by an html form containing part of the test as questions. The form's result is posted to the 'mc' plugin for processing.

The test element has several attributes with the following semantics:

title, added as header for the test

ask, an integer indicating the number of randomly chosen questions to ask

expr, a GALE boolean expression to decide whether the test is shown or the 'alt' text is shown

alt, the alternative text that is shown when 'expr' returns false

action, GALE statements to execute when the test is done (the 'value' variable can be used to refer to the test score (from 0-100).

verbose, a boolean to indicate whether explanations should be shown upon completion of the test

The test element contains a list of questions, each question contains a list of answers and each answer can optionally contain an 'explain' element. There can be more questions than the number indicated by 'ask'. Questions will then be chosen randomly. A question has the attributes 'answers' and 'right', indicating how many answers to display and how many of those answers should be correct answers. If there are more answers available they are again picked at random.

Each answer has a single attribute 'correct', indicating whether the answer is correct. The optional 'explain' block is shown as comment upon completion of the test, if the 'verbose' flag of the test is 'true'.

The last child element of the test element can be a 'result' element. Its content is shown upon completion of the test if 'verbose' is true.

## 6.4 Using plugins and views

GALE can be extended in various ways. To add adaptation functionality for xml files, additional xml modules can be written (section 6.2). Additional processors can be written to perform adaptation on any file format served by GALE. Some xml modules refer to views and plugins as objects that GALE uses. Views can produce any x(ht)ml based on the current state of processing (the resource). This xml is inserted into the resource's in-memory xml model at the point where the view was called (see ViewModule in 6.2). Plugins can take over the entire processing and write directly to the HttpResponse of the web server. Examples are the Password and Logout plugin.

By default there are two views. The 'static-tree-view' shows a tree structure based on the 'parent' relationship. It uses the attributes 'order' (Integer) and 'hierarchy' (Boolean) respectively to order the tree and possibly omit elements. The other view is the 'file-view'. It displays the content of a file specified directly ('file' parameter) or indirectly through a GALE expression ('expr' parameter). You could also use content from the DM that would be interpreted as XML ‘content’ parameter. For examples see section 6.2.

The plugins that are useful for the author are 'logout' and 'password'. They both take no parameters and a link to them can be created using the 'plugin' tag mentioned in 6.2. There are some other plugins that need to be present in galeconfig.xml for various reasons. The 'mc' plugin interprets a finished adaptive test, the 'exec' plugin performs the instructions specified in the 'exec' part of an adaptive link and the 'export' plugin can export domain model information to an xml file (gdom format).

Writing a new view or plugin is fairly straightforward. Create a java class that implements either `nl.tue.gale.ae.processor.view.LayoutView` or `nl.tue.gale.ae.processor.plugin.Plugin` respectively (in both cases there is one or two methods to implement). Update galeconfig.xml to include a reference to your class. Now you can use your new view or plugin in the same way as those included with GALE.

## 6.5 GALE statements and expressions

Any GALE expression or statement is basically a piece of Java code extended with a special notation for using user model variables and domain model concepts and attributes. You also have access to the current GaleContext or current resource that GALE is working on from within your code. There is a reserved variable available called ‘gale’ that is of the type `nl.tue.gale.ae.GaleContext`. This is the wrapper around the current resource and includes utility functions to retrieve everything GALE has access to at the moment.

The special notation to access user model and domain model is often referred to as pseudo code. `${ <pseudo> }` is used to *access* UM and DM and `#{ <pseudo> , … };` is used to *assign to* UM variables, where <pseudo> refers to a special expression that can be resolved to a URI. The URI can refer to concepts, attributes, and their properties. A pseudo expression has the following grammar:

```
pseudo:
        pseudo-part
        pseudo-part pseudo
        relative-uri
pseudo-part:
        relational-expr
        relative-uri relational-expr
relational-expr:
        ->(name)
        ->(name)[number]
        <-(name)
        <-(name)[number]
```

‘relative-uri’ can be any relative or absolute uri that can be resolved to a concept, an attribute, or any of its properties. ‘name’ is a string that follows the Java identifier rules. ‘number’ is a constant integer.

The semantics are as follows. The pseudo expression has a notion of ‘current URI’ and starts with the current concept URI. The ‘relational-expr’ and ‘relative-uri’ blocks are processed step by step as they are encountered in the pseudo expression.

Whenever a ‘relative-uri’ block is encountered, it is resolved to the current URI, and this becomes the new current URI.

Whenever a ‘relational-expr’ block is encountered, the current URI is expected to refer to a concept (it is an error otherwise) and the specified relation is followed (-> meaning outgoing relations and <- meaning incoming relations), which leads to a list of concepts. If the ‘relational-expr’ contains a number index, the current URI becomes the indexed concept URI from the list. If the number is not specified there are two options. If this ‘relational-expr’ is the last block in the pseudo expression, the list of concepts is the result of the entire pseudo expression and is returned as a Java array of Concept objects; which ends processing of the pseudo expression. If this ‘relational-expr’ is not the last block, the index is assumed to be 0, and the current URI is updated accordingly.

The pseudo expression can thus be resolved to a URI (the last current URI). This URI is resolved to the object it refers to. If the object refers to an attribute, it’s corresponding UM value is returned. In the resulting

URI the fragment part is used to refer to attributes of a particular concept and the query part is used to refer to properties.

Here are some examples of pseudo code in expressions and statements:

`${#suitability}`
: the value of the suitability attribute of the current concept

`${#image?title}`
: the value of the “title” property of the “image” attribute of the current concept

`${->(parent)?type}`
: the value of the type property of the (first) parent concept

`${->(parent)?type}.equals("page")`
: whether the value of the type property of the (first) parent concept is equals to "page".

`${->(parent)<-(parent)}.length`
: the number of siblings of the current concept, based on the ‘parent’ relationship

`${details<-(related)}`
: an array of concepts that have a ‘related’ relation to the ‘details’ relative concept

  (if the current concept would be ‘gale://gale.tue.nl/welcome’, the details concept would be ‘gale://gale.tue.nl/details’)

`${gale://gale.tue.nl/personal#email}`
: the value of the email attribute of the ‘gale://gale.tue.nl/personal’ concept

`#{->(parent)#visited, ${->(parent)#visited}+1};`
: increments the UM variable ‘visited’ of the parent concept

`#{gale://gale.tue.nl/personal#history, gale.conceptUri()};`
: store the current concept URI in a variable called ‘history’ in the concept ‘gale://gale.tue.nl/personal’

All GALE expressions and statements are pre-parsed into real Java code and then compiled to Java bytecode. This bytecode is reused whenever possible.

GALE expressions can be used in various places in the domain model, like a concept’s event code and an attribute’s event and default code. They can also be used in (x)html pages written by the author, whenever he uses an ‘if’ or ‘variable’ tag. We are aware that this poses a security risk that needs to be studied and solved before GALE can be considered final.

The GaleContext variable ‘gale’ is only valid within the adaptation engine that runs the processors. Expressions that appear in pages (in “if” and “variable” tags for instance) are all evaluated by the adaptation engine. However, the ‘default code’ and ‘event code’ of attributes is not evaluated inside the adaptation engine, but evaluated in the UM service. This code that appears in DM (domain *and adaptation* model) cannot make use of the ‘gale’ variable, only known by the adaptation engine. Instead, these adaptation rules can make use of a special ‘dm’ variable, which is a reference to a DM cache object.

# 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., Nejdl, W. Adaptivity in the KBS Hyperbook System. 2nd Workshop on Adaptive Systems and User Modeling on the WWW, workshop held in conjunction the World Wide Web Conference (WWW8) and the International Conference on User Modeling, 1999.
6. Knutov, E., De Bra, P., Pechenizkiy, M., AH 12 years later: a comprehensive survey of adaptive hypermedia methods and techniques. New Review of Hypermedia and Multimedia, 15(1), pp. 5-38, 2009.

# Appendix A: Backward Compatibility

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). 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 "commit to AHA!" button in this Graph Author tool thus actually commits to GALE.
   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 'gale:/tutorial/xml/welcome.xhtml' to refer to resources located in (under) the $GALE_HOME directory (see Section 6.1 for a description of $GALE_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 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 $GALE_HOME/config directory as its source of domain model information. The AHA! 3 domain model service allows testing of GALE before the GRAPPLE authoring tools are available, and without the need to use the Graph Author tool.
   The AHA! 3 domain model service sends an 'updatedm' event for a particular application whenever its underlying .aha file changes (it monitors the file system).
3. **Sending resources to GALE**
   GALE currently supports the Application Management tool, adapted from AHA! version 3. The AMt-tool can be opened by following a link on the main GALE starting page (index.html in GALE's root directory) and the graph author can be loaded from within the AMt-tool. All author files are located on the server in the directory '$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 who 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.