

---

# **AHA – Adaptive Hypermedia for All**

---

Data design and manipulation 1.0 –  
David Smits

# Contents

Contents	2
Introduction	3
What is AHA?	4
How does AHA work?	4
<b>Object Model</b>	<b>5</b>
Data definition	6
Introducing concepts, attributes and expressions	6
About actions, propagation and relative attributes	6
Implementation	7
Concept	7
Attribute	7
AttributeType	7
Action	8
Statement	8
Profile	8
AttributeValue	8
<b>The Engine</b>	<b>10</b>
User profile updates	11
The access attribute	11
The action queue	11
The time attribute	11
Handling a request	12
Resources	12
Resource handlers	12
The handler manager	12
The 'Get' servlet	13
Writing your own handler	13
Implementation	14
ResourceType	14
Resource	14
interface ResourceHandler	14
HandlerManager	15
ActionQueue	15
ProfileUpdate	15
ProfileManager	15
Interpreter	16
<b>Data Storage</b>	<b>17</b>
Structure of storage	18
Introduction	18
The profile interface	18
The concept interface	18

Implementation	19
AHADB	19
interface ConceptDB	19
interface ProfileDB	20

# Introduction

## What is AHA?

AHA stands for Adaptive Hypermedia for All. Its purpose is to extend web-servers with transparent adaptivity functionality. This means that information provided by a web-server should be user dependant. This dependency includes the use of different colors to indicate the relevance of links to other information and the conditional inclusion of information. A profile of the user requesting information is necessary. This profile is dynamically updated depending on the interest or knowledge of the user.

## How does AHA work?

Information the user requests can be any resource. Most likely this will be a webpage of some sort, but it can also be a picture, a movie or any other resource. This resource may be stored locally in the file system or in a database or externally. If necessary this resource will be modified based on the stored user profile to suit the needs of the user. Then this resource is delivered to the user and the profile is updated.

To perform this task AHA needs particular information: the user profile, a scheme defining how this profile changes the presented information and a scheme defining how this profile is updated when accessing a particular resource. In *Part 1: AHA Data model* the information needs will be further analyzed. In *Part 3: AHA Data storage* a full description is given how this data is actually stored (in a mySQL database or in XML files) and how other means of storage can be added.

The process of actually modifying different resources and updating the user profile is discussed in *Part 2: AHA Engine*. Here you can find how new resource types can be added to the AHA system that incorporate adaptivity.

Every part has its own implementation chapter where the concepts described earlier are discussed in more detail. Java is used to implement the AHA system. This will be the language in which implementation dependant subjects are described.

Part 1

# Object Model

*Concepts, Actions & Requirements*

A user profile is used to determine the user's interest in or knowledge of a particular subject. The use of this profile lies in adapting a resource requested by the user. Since the profile is subject based, a way has to be devised in which resources relate to subjects (concepts). Furthermore the AHA system should be able to determine how suitable a particular concept is for a user based on the user profile (requirements). And after the user has requested information some changes might be made to the user model based on the resource that was requested (actions).

# Data definition

## Introducing concepts, attributes and expressions

A concept is basically a subject the user can learn things about. A particular subject may have several aspects. These aspects are stored as attributes of a concept. So a concept is a structure with a name and a set of named attributes. Different types may be used for different kinds of attributes (ie. a string, a number, a boolean, etc.). An example of an attribute may be a number indicating the knowledge of the concept. Attributes have attributes themselves, for instance they can be system attributes that should not be deleted or they might be read only. Attributes may also be non-persistent indicating that they will revert to a default value every time the user requests a resource instead of storing the value.

Because we need to determine the relevance of resources for a user, we need some means to link these resources to our concept definitions. For this purpose every concept may be linked to a resource. Not every concept needs to be linked, but if a concept is linked it may be linked to only one resource.

Every concept has some expression associated with it, that gives an indication about the suitability of this concept for the user. This expression can use the various attribute values in the user profile as variables. In this way the relevance of a particular resource can be estimated by evaluating the suitability expression of the associated concept.

## About actions, propagation and relative attributes

Subjects are related to each other. For instance, learning things about Java will also mean learning something about programming, which will in turn increase your knowledge of computers. So concepts should be able to influence each other too. This is done through actions.

Each attribute can have a list of associated actions. When the attribute is changed (by another action), the other actions will be executed. Actions are conditional meaning that they are only executed if some condition is true. If so they will change the attribute of a particular concept to the value of some expression. Actions can in turn change attributes of other concepts and thus trigger new actions. To prevent this, a flag is introduced that can be used to disable the triggering of further actions.

Beside using absolute expressions, we introduce relative variables. These will prove useful in defining logical relations between concepts. The value of a relative variable is defined as the change in the underlying variable. For instance an action may be something of the form: `programming.knowledge := programming.knowledge + 0.2*[Java.knowledge]`. The meaning of this action would be that the knowledge of Java would contribute for 20% to the knowledge of programming. If this action is placed in the list of actions belonging to the `Java.knowledge` attribute, this would guarantee that `programming.knowledge` is updated properly.

# Implementation

## Concept

This class defines a single concept, which is composed of attributes and a suitability expression.

```
public Concept(String name)
    Creates a new concept with the specified name.

public Vector getAttributes()
    Returns a list (possibly empty) with attributes that are defined for this concept.

public String getName()
    Returns the name of this concept.

public void setSuitability(Expression expr)
    Sets the suitability expression.

public Expression getSuitability()
    Returns the suitability expression.
```

## Attribute

This class defines a single attribute, including the type and the actions that will be executed when this attribute is changed.

```
public Attribute(String name)
    Creates a new attribute with the specified name.

public Attribute(String name, AttributeType type)
    Creates a new attribute with the specified name and type.

public void setType(AttributeType type)
    Sets the type of this attribute.

public AttributeType getType()
    Returns the type of this attribute.

public Vector getActions()
    Returns a list (possibly empty) of actions that are to be executed if this attribute is changed.

public boolean isReadonly()
public setReadonly(boolean readonly)
public boolean isSystem()
public setSystem(boolean system)
public boolean isPersistent()
public setPersistent(boolean persistent)
    The above six methods get and set the read only, system and persistent attributes respectively.
```

## AttributeType

This class is used to indicate the type of an attribute.

```
public AttributeType(long type)
    Creates a new attribute type.

public long getType()
    Returns the value indicating the attribute type.

public toString()
    Returns a string representation of this attribute type.
```

## Action

This class defines a single action, containing a condition, a set of statements if the condition is true and a set of statements if it is false.

```
public Action()
    Creates a new action.

public void setCondition(Expression expr)
    Sets the condition part of this action.

public Expression getCondition()
    Returns the condition part of this action.

public Vector getTrueStatements()
    Returns a list of actions that should be executed if the condition is true.

public Vector getFalseStatements()
    Returns a list (possibly empty) of actions that should be executed if the condition is false.
```

## Statement

This class defines a single assignment from one expression to a variable.

```
public Statement()
    Creates a new statement.

public Statement(String variable, Expression expr)
    Creates a new statement with the specified expression that should be assigned to the specified variable.

public void setVariable(String variable)
    Sets the variable that should be assigned by this statement.

public String getVariable()
    Returns the variable that should be assigned by this statement.

public void setExpression(Expression expr)
    Sets the expression that should be assigned to a variable by this statement.

public Expression getExpression()
    Returns the expression that should be assigned to a variable by this statement.
```

## Profile

This class defines the user profile. It actually is nothing more than a large collection of name value pairs.

```
public Profile()
    Creates a new user profile.

public AttributeValue getAttributeValue(String concept, String name)
    Returns the value of the specified attribute.

public setAttributeValue(String concept, String name, AttributeValue value)
    Sets the value of the specified attribute.
```

## AttributeValue

This class holds the value of a particular attribute.

```
public AttributeValue(AttributeType type)
    Creates a new attribute value based on the specified type.

public AttributeValue(AttributeType type, String value)
```



Creates a new attribute value based on the specified type and initially equal to the specified value.

```
public AttributeType getType()
```

Returns the type this attribute value.

```
public void setValue(String value)
```

Sets the value of this attribute.

```
public String getValue()
```

Returns the value of this attribute.

```
public boolean isNew()
```

Returns whether this is a new attribute value.

```
public boolean isUpdated()
```

Returns whether this is an updated attribute value.

```
public void clearNew()
```

Clears the 'new' flag.

```
public void clearUpdated()
```

Clears the 'updated' flag.

## Part 2

# The Engine

### *Action queuing & Handlers*

In this part we discuss in more detail the process of actually changing the user profile by means of actions. We will take a look at the concept of an action queue that is used to process all actions that are generated when a user requests a resource. Furthermore a framework is defined for handling different types of resources. This framework makes it possible to write separate modules for every different type of resource that requires some form of processing to make it more suitable for the user.

# User profile updates

## The access attribute

We stated earlier that actions can trigger other actions, meaning that the change of a particular attribute's value can lead to new changes that should be made to other attributes. But what actions get triggered first? Where does the process begin?

The user profile should be updated after the user requests a resource. So somehow we need to be able to define a list of actions that have a direct relationship with a resource. We already have the possibility to define such a list for an attribute and we have a means of linking resources to concepts (that in turn contain attributes). So the most obvious solution is to define a dedicated (or system) attribute where we can store our list of initial actions.

We combine this with another desirable property of a concept. That is the desire to make expressions (and thus actions) dependant on the currently requested resource (or concept). For this purpose and the need for a set of initial actions we introduce the 'access' attribute. This is a non-persistent attribute that is true only for the concept whose associated resource is currently requested by the user. So when this attribute is set to true, the associated set of actions is executed and thus this is a good place to store the initial actions.

## The action queue

Because of termination constraints a breadth first type of approach is chosen instead of a depth first approach. This means that a simple recursive procedure to handle the triggering of other actions is not an option in the process of handling actions. Instead we choose to use a queue in which all actions are stored that should still be executed.

The initial change of an attribute – setting the 'access' attribute – will load the first set of actions in the queue. The action at the beginning of the queue is removed and processed. If this action changes another attribute for which new actions are defined and this action's 'triggered' attribute is true, then the new actions will be loaded and placed at the end of the queue. This process will continue until no more actions remain on the queue.

## The time attribute

Beside the *access* attribute that is present for every concept, there also is a *time* attribute. While the access attribute is read-only, this attribute is not. The *time* attribute is a number representing the time (in seconds) that the user read material about this concept. The engine will only update *time* attributes of concepts that are associated with a resource. Further actions can be defined however that can ensure that a concept not directly associated can still have a useful value in his *time* attribute.

For example: for the attribute `Java.time` you might define the action: `programming.time := programming.time + [Java.time]`. All time a user spends on reading things about Java will then be added to the time he has spend on reading things about programming.

# Handling a request

## Resources

A resource is everything that can be served by a computer. It can be a document (HTML, XML, PDF, etc.), a sound, a picture, etc. There are two things the engine needs to know about resources: where is it and what is it.

Resources are identified by their Unique Resource Locator (URL). Java has built in functionality for using these URLs. A URL may indicate that a resource can be found in the local file system, on an FTP server, via HTTP, etc.

After the engine has located the resource by means of a URL, it determines the type of the resource. This is basically done by looking at the MIME-type of a resource. For some basic types it may be desired to define a sub type. An XML document for example, may have the name of its used DTD as its subtype. So every resource has a sufficiently detailed type description.

## Resource handlers

When a resource is found and its type is determined, the engine might have to change the resource in some way, so it will produce an output that is more suitable for the user. This processing of a resource is of course dependent of the type of the resource. A picture for instance, may not need any modification at all and may thus directly be served to the client. But an HTML document may need some processing.

For this purpose handlers are introduced. A handler is an independent subcomponent of the engine, which takes a resource and a user profile as parameters and then creates a new resource (that is more suitable for the specified user). Handlers are type specific, meaning that one handler can specifically be designed to handle XML documents (with some specified sub type) and another handler may deal with HTML documents. Based on the type of the resource an appropriate handler is chosen by the engine. After one handler is finished with a resource, another handler may still need to process it further. This may result in a chain of handlers, until finally the resource is ready for the client.

After the resource is processed and ready, the engine updates the user profile as described above. Then the resource is served to the client.

## The handler manager

This component manages all handlers that are known to the system. New handlers should be registered here (by using `addHandler`). Handlers are chosen based on a method that is implemented by every handler called `handlesResource`. This method takes one parameter, the type of a resource, and returns if it can handle that type. The order in which handlers are registered to the manager is important, because handlers that are added later are queried earlier. For instance: the first handler that is added when a new manager class is created is the pass-through handler. This handler handles all types and simply returns the same resource as it received. This ensures that all resource can be 'handled'. Any handler that will be registered later will end up earlier in the list of handlers and the first handler that handles the specified type is chosen.

## The 'Get' servlet

This is the main component of the engine. This servlet processes all requests made by users. The servlet has a single parameter: the URL of the requested resource. It uses the built-in functionality of Java to locate the resource and then it uses the handler manager to process the resource until it is ready for the client. After that another component (the *profile manager*) is called to update the user model. When updating has finished, the resource is ready to be served.

The get servlet also uses the http-session, to retrieve information about the previously requested resource and when it was requested, to update the *time* attribute of the associated concept. After that the currently requested URL and time are stored in the session.

## Writing your own handler

A handler is nothing more than a class that implements the Java `ResourceHandler` interface. The interface has two methods: `handlesResource` and `processResource`. The first method returns a boolean indicating whether it can handle a resource of the specified type. The second produces the new modified resource.

A handler's `processResource` method receives two useful parameters in modifying a resource. The first is the user profile and the second is a reference to the database where all concept definitions are stored. A handler may use the *profile manager* to obtain specific information from one concept, for instance its suitability for a specified user (by calculating the suitability expression).

For further information about the classes described above, see the respective sections in the *Implementation* chapters.

# Implementation

## ResourceType

The class `ResourceType` is used to store the type of a particular resource. A resource's type can be simply the MIME-type of the resource or it can contain some additional information. This additional information may be used to identify different kinds of XML files for instance.

```
public ResourceType(String mime)
    Creates a new resource type with the specified MIME type.

public ResourceType(String mime, String subtype)
    Creates a new resource type with the specified MIME type and subtype.

public String getMime()
    Returns the MIME type of this resource type.

public String getSubtype()
    Returns the subtype of this resource type.

public boolean equals(ResourceType type)
    Compares this resource type with another.
```

## Resource

The class `Resource` contains the data of a resource and a definition of the format of the resource. Furthermore the class has methods that can be used to determine if this resource is ready to be sent to the client. If it is not ready to be sent to the client then this resource will need further processing.

```
public Resource(InputStream data, ResourceType type)
    Creates a new resource with the specified data and type.

public boolean isReady()
    Returns if this resource can be sent to the client or not.

public void setReady(boolean ready)
    Sets if this resource can be sent to the client or not.

public void setURL(URL original)
    Sets the original URL of this resource.

public URL getURL()
    Returns the original URL of this resource.

public ResourceType getType()
    Returns the type of this resource.

public InputStream getInputStream()
    Returns an inputstream that returns the data for this resource.
```

## interface ResourceHandler

The `ResourceHandler` interface defines the functionality of a class that can process a resource and create another resource, which is better suited to the client. If the `ResourceHandler` determines that the resource is ready to be sent to the client and needs no further processing, it can use the `Resource`'s `setReady` method. A `ResourceHandler` should be able to indicate if it can handle a specific resource (by using its resource type).

```
public boolean handlesResource(ResourceType type)
    Returns if this handler can process the given resource.
```

```
public Resource processResource(Resource resource, Profile profile, AHADB db)
throws ProcessorException
    Processes the resource using the user profile and database and returns the new resource.
```

## HandlerManager

This class is used in finding a resource given a URL and in finding a suitable handler for the resource.

```
public HandlerManager(AHADB db)
    Creates a new manager.

public void addHandler(ResourceHandler handler)
    Adds the specified handler to the beginning of the list.

public void removeHandler(ResourceHandler handler)
    Removes the specified handler.

public Resource processResource(Resource resource, Profile profile)
throws ProcessorException, NoHandlerException
    Processes the resource using the user model and database and returns the new resource.

public Resource locateResource(URL resourceURL)
throws ResourceNotFoundException
    Returns the resource that belongs to the given URL.
```

## ActionQueue

This is the queue where all new actions are stored in and the next to be executed action can be retrieved.

```
public ActionQueue()
    Creates a new queue.

public addActions(Vector actions)
    Adds the actions to the end of the list.

public boolean isEmpty()
    Returns whether the queue is empty.

public Action next()
    Returns the current action and removes it from the list.
```

## ProfileUpdate

This class performs a single profile update. It uses an `ActionQueue` to store the actions that have to be performed.

```
public ProfileUpdate(Profile profile, ProfileManager manager)
    Creates a new profile update for the specified user. This method makes an in-memory copy of the user model to determine relative updates.

public void UpdateAttribute(String concept, String attribute, AttributeValue
value)
    Updates the specified attribute to the new value. This may produce new actions in the queue.

public Profile getProfile()
    Returns the current user profile.
```

## ProfileManager

This class is used in updating the user profile and in calculating the suitability of concepts, based on a user's profile.

```
public ProfileManager (AHADB db)
    Creates a new profile manager based on the specified database.

public void accessedResource (Profile profile, URL resourceURL)
    This method is called by the get servlet to update the user model. It creates an instance of the ProfileUpdate class and then updates the
    access attribute.

public int getSuitability (URL resourceURL)
    This method returns the suitability of the resource (if it is associated with a concept).

public AHADB getAHADB ()
    Returns a reference to the database interface.
```

## Interpreter

This class is used to evaluate the expressions in AHA.

```
public Interpreter ()
    Creates a new interpreter.

public void setSymTable (Hashtable sym)
    Sets the symbol table containing all the variables that can be found in an expression.

public String getString (String expr)
    Returns the string value of the evaluated expression.

public int getInteger (String expr)
    Returns the integer value of the evaluated expression.

public boolean getBoolean (String expr)
    Returns the boolean value of the evaluated expression.

public Object getObject (String expr)
    Returns the value of the evaluated expression as an object.
```



## Part 3

# Data Storage

*User profiles, mySQL & XML*

As seen in the previous two parts, the need for AHA exists to store information. This information consists mainly of user profiles and concept definitions. While storing in a database is a better solution when AHA is employed as a web server, we still want to give users the possibility to download a package and, for instance, study an AHA based course offline. In the latter case, it is inconvenient (and inefficient) for users to install whatever DBMS might be chosen for AHA. This is why data storage should be as flexible as possible, allowing for the storage of all data to different structures (based on the needs) without having to recompile all code.

# Structure of storage

## Introduction

Because of the needs described earlier, the general storage component of AHA is basically a large interface describing the methods needed to access all information in the database. The actual implementation of this interface is specific to the means of storage used.

The total interface of methods is split in two parts: one describing the interface for accessing profiles and another to do the same for concepts. There is one class definition to access these two interfaces.

## The profile interface

The general life cycle of a profile object is that it is created and retrieved when the user logs on, stored whenever the profile is updated and destroyed when the user logs off. This means that the only need lies basically in storing and retrieving entire profiles. Of course you would also need to create a new profile and search for existing profiles with specific attributes. These four methods form the profile interface.

## The concept interface

With concepts the life cycle is a little different. Loading all concepts in memory prior to the actual start of the engine, may take up a lot of memory. So concept information should be retrieved directly from the database. While the system is running the need rarely exists to retrieve an entire concept. Moreover you would want to retrieve just the suitability expression of a concept, a single attribute definition of a concept or check what resource is linked to a concept. So we need some specific methods for retrieving and storing parts of a concept.

Sometimes however you will have the need for the entire concept. This may be the case in authoring tools for concept definitions where you want to allow the author to edit an entire concept. We saw methods to allow the retrieval and storage (and creation and search) of entire objects in the profile interface. We need similar methods for concepts.

One could argue (especially in case the implementation is something like XML) that providing a general interface with only the four methods defined also for profiles is enough to provide access to every element of a concept. Though this may be virtually as efficient as specific methods when the implementation is XML, this is obviously not the case if storage is done in a database. In XML the entire file has to be loaded and parsed, but in a database every record has to be retrieved.

The above argument resulted in the design of the concept interface that contains both specific (retrieving and storing single aspects of objects) and general (retrieving and storing entire objects) methods. The different implementations may use either use the specific methods to build the general methods (a database implementation) or use the general methods to build the specific methods (a XML implementation).

# Implementation

## AHADB

This is the main class that can be used to access the database. It contains all the methods to retrieve and store information and to register means of storage.

```
public AHADB(ConceptDB concept, ProfileDB profile)
    Creates a new AHADB class and registers the specified components.

public registerConceptDB(ConceptDB concept)
    Registers another ConceptDB class.

public registerProfileDB(ProfileDB profile)
    Registers another UserDB class.

public ProfileDB getProfileDB()
    Returns the registered UserDB.

public ConceptDB getConceptDB()
    Returns the registered ConceptDB.
```

## interface ConceptDB

This interface must be implemented by a class to function as a concept storage means.

```
public long createConcept(String name)
    Creates a concept with the specified name and returns the id.

public long findConcept(String name)
    Returns the id of the specified concept.

public void setSuitability(long id, String expr)
    Sets the suitability expression.

public String getSuitability(long id)
    Returns the suitability expression.

public void linkResource(long id, String resource)
    Links the specified concept to the specified resource. Concepts and resources may be linked only once.

public void unlinkResource(long id)
    Removes a link from the specified concept to a resource.

public String getLinkedResource(long id)
    Returns the resource that is linked to the specified concept.

public String getLinkedConcept(String resource)
    Returns the concept that is linked to the specified resource.

public void setAttribute(long id, String name, Attribute attr)
    Sets the specified attribute.

public Attribute getAttribute(long id, String name)
    Returns the specified attribute.

public void removeAttribute(long id, String name)
    Removes the specified attribute.

public Vector getAttributeNames(long id)
    Returns the list of names of all attributes of this concept.

public Vector getAttributes(long id)
    Returns all attributes.

public setAttributes(long id, Vector attrs)
    Sets all attributes.

public void setConcept(long id, Concept concept)
```

Saves a concept with the specified id.

```
public Concept getConcept(long id)
```

Loads a concept with the specified id.

## interface ProfileDB

This interface must be implemented by a class to function as a user profile storage means.

```
public long createProfile()
```

Creates a new user profile and returns the id.

```
public Profile getProfile(long id)
```

Loads a user profile with the specified id.

```
public void setProfile(long id, Profile profile)
```

Saves a user profile with the specified id.

```
public Vector findProfile(String concept, String attribute, String value)
```

Finds all profiles that match the criteria and returns a list of their id's.