# 3. ALAT

The new authoring environment for GALE is called ALAT. This stands for “*Adaptive Learning Authoring Tool*”. Coincidentally it is also Indonesian for “tool”. It is an entirely new authoring environment for GALE in which users can create domain and adaptation models with ease. This section describes ALAT and the important design decisions taken creating it. It will also include a description of all important features and main workflow aspects. A reference to the previous authoring environments as described in section 2 will be given where appropriate to explain to what extent features from the previous authoring tools have been recreated or redesigned.

## 3.1 The ALAT authoring interface

This section will focus on the design decisions of the ALAT interface. [APPENDIX REF] will provide a more detailed description of the user interface layout and menus.   
The goal of the ALAT authoring interface is to be easy and straight forward to use. Alloperations available in ALAT are easy to perform without having to navigate through multiple elaborate menus and interfaces. The interface is also aimed to be user friendly towards both novice adaptive hypermedia authors as well as more experienced authors. The design of the interface has been made in collaboration with the members of “De Roode Kikker” as well as a HTI (human technology interaction) expert[[1]](#footnote-1). The resulting interface design [APPENDIX REF] has been the foundation of the main ALAT screens.

Unlike both GAT and the AHA! graph author, there are no graph-like domain or adaptation model representations in ALAT. Graph interfaces tend to get cluttered quickly and make it difficult to keep track of the domain and adaptation model. Therefore the choice has been made to show all information either in lists or a hierarchical structure. By navigating these hierarchies and filtering these lists it is easy to display the desired concept information without interface cluttering.

ALAT has a main concept hierarchy which is used to navigate through domain concepts in the user interface. This hierarchy is a tree-like structure which connects concepts through parent-child relations. These relations will also be defined in the resulting GALE application in order to allow further use of this structure. Using a main hierarchy is very different from the domain construction in GAT, which did not use a default relation and displayed all non-pedagogical relations together in a single canvas. ALAT takes the AHA! graph author approach which also used a tree-like project hierarchy to enforce a maintainable project structure. It is possible to add more (pedagogical and) non-pedagogical relations between concepts, but this functionality is moved to the concept settings screen so that the main tree structure is maintained.



Figure 3.1: The step by step screen

All authoring functionalities in ALAT are divided amongst two main screens in which the domain and adaptation model can be constructed. There is a main screen which shows the current domain hierarchy and a concept settings screen which shows all concept details in a single overview.  
The screen displaying the domain hierarchy has two view modes, which can easily be switched between with the click of a button. The first view (*figure 3.1*) provides a step-by-step domain navigation, reducing the number of concepts on the screen at any given time. The other view (*figure 3.2*), shows the domain hierarchy as a collapsible list, much like the AHA! graph author did. For novice users, the step-by-step view provides an easy way to navigate the domain hierarchy without causing an overload of information by displaying too many concepts on the screen at once. The list view is suited for more advanced users as the domain hierarchy is easy and fast to navigate and edit.



Figure 3.2: The overview screen

The second screen (*figure 3.4*), as mentioned before, shows all concept information in a single view. This screen can be easily opened and closed with a single click of a button, minimizing the amount of menu navigating and context switching. It deals with adding, editing and removing concept properties, attributes, adaptation rules and non-pedagogical relations. These can all be managed using clean, straight forward controls.  
All concept attributes can be easily managed using the corresponding section in the screen. Adding, removing and editing attributes is a quick and easy process which does not require further menu navigation.

The “*relations and expressions”* section of the screen is used to manage all pedagogical and non-pedagogical rules and relations. The user can choose to inspect and add a type of rule by using a drop down menu.  
Tooltips are provided to inform the users about the function of adaptation rules and non-pedagogical relations. This section also enables the creation of new non-pedagogical relations. This is an easy process which only involves entering a name and tooltip as no GAM code is required to apply these relations. Once such a rule is created it is available in the filter list of any concept, so that it can easily be reused when needed.

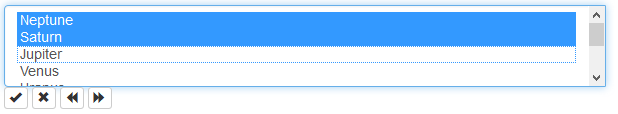


Figure 3. 3: The rule target selection controls.

An easy set of controls has been created in order to easily select targets for any given pedagogical or non-pedagogical relation (*figure 3.3)*. These controls can be used to walk through the project hierarchy and select the desired rule targets. This solves the problems of huge lists of concepts, which was a big problem in GAT. A strong feature of GAT was the ability to apply adaptation rules on multiple concepts at once with the use of the socket interface as described in section 2. While it is no longer possible to select multiple sources for a single rule, there is the possibility to select multiple targets for any given pedagogical or non-pedagogical relation by means of multi-selection, which is enabled in the controls used to select a rule target.

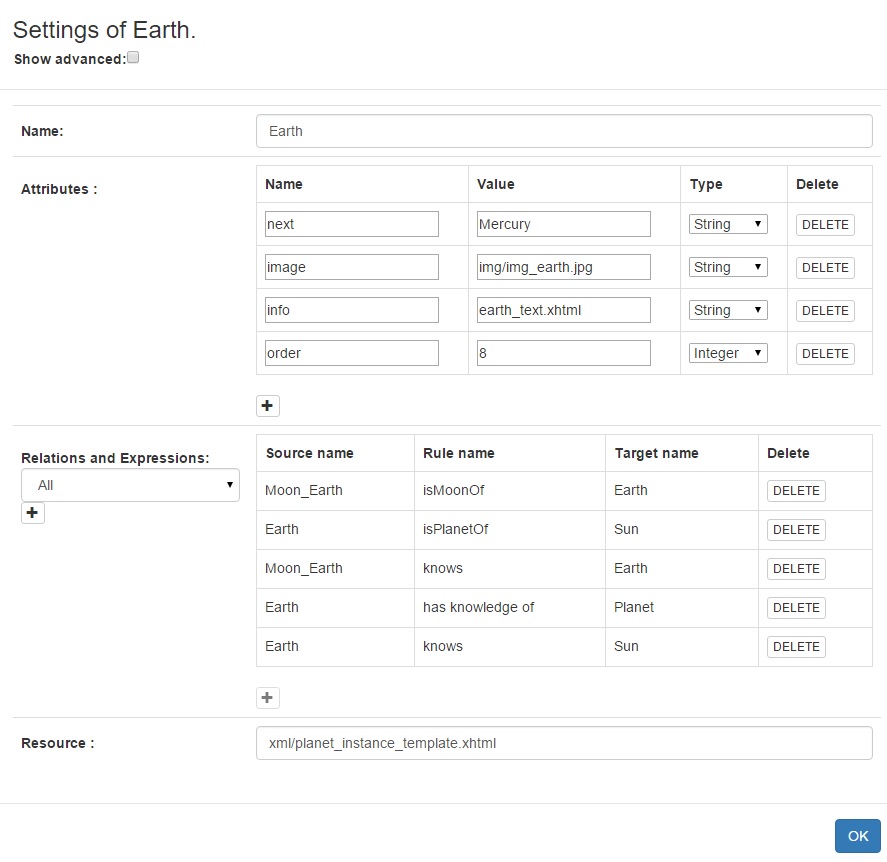
The user does not have to deal with any GALE code anywhere during the entire authoring process. As will be described in the next section, all concepts are created based on a templated blueprint. This means a selection of attributes and adaptation rules is applied when the concept is created. All data that originates from templates cannot be deleted from the concept. This is done to protect users from “breaking” the template. On top of that ALAT hides templated rules and attributes the user cannot edit. This prevents an overflow of information and confusion amongst novice users. More advanced users are able to display this hidden information by means of an “advanced mode” which can be toggled on and off.

Figure 3.4: The settings screen

## 3.2 Templating

One of the biggest improvements ALAT provides when compared to its predecessors is the ability to move the complicated scripting and technical aspects away from the user even more than the graph author and GAT. By extensively templating novice users with basic understanding of adaptive hypermedia will be able to author a course.

Templating to this extent is great for user-friendliness and workflow efficiency. It prevents having to apply standard behavior repeatedly and allows a set of attributes and concept adaptation rules to be set without confusing the author with its details.

An engineer with more advanced skills in GALE will be able to set up these templates and can edit and extend them when needed. The novice user will then be able to use these additions without having to deal with any GALE Adaptation Model (GAM) code.

The templates the tool uses cover a wide range of functionalities in order to support as much extendibility as possible. The templated data is divided in two JSON files; a concept blueprints file and pedagogical relation definitions file.

### 3.2.1 Format and structure

An appropriate storage format is necessary to store the templated information. This format must be able to efficiently store all required data. It must also be easy to parse by the authoring tool and easy to read for humans. Lastly it is important to choose an existing well-known format. This is preferable over a custom format as existing formats are more likely to be familiar to new users and are extensively documented.  The main candidate formats have been XML and JSON, both of which are W3C[[2]](#footnote-2) standards. XML stands for eXtensible Markup Language and is a language designed to be read by both humans and machines. It is structured to store and transport data.

JSON stands for JavaScript Object Notation. It is a format built for storing and exchanging data and is described as an easier to use alternative to XML.   
JSON has been chosen over XML as it is easy to read and write. Its integration in rich web applications is intuitive and parses faster on servers than XML [NuAL09]. It is structured by declaring objects and arrays which can be read by ALAT without parsing.

### 3.2.2 In-application templating versus file editing

As described in the previous section, all templates are defined in JSON and are divided amongst two template files. The design decision has been made to craft these templates by hand as opposed to providing in-application support. This approach has been taken based on the evaluation of the AHA! graph author and GAT as described in chapter 2. GAT has a user interface for creating new adaptation rules, which is practical when a skilled author/engineer wants to add a new, non-existing rule on the fly. But this user interface approach is not practical when creating concept blueprints, or when creating an adaptation rule template setup. Elementary actions available in text editors, such as copying and pasting or searching and replacing are not possible. These operations would be very useful when performing operations such as creating variants of existing rules for example. It is simply faster and easier to just edit the properties file instead of having a graphical user interface. That is why the AHA! graph author approach has been taken and all templating has been moved outside of the application. The creation and maintenance of templates is efficient and fast because of the use of JSON, which is easy to understand and edit.

### 3.2.3 Blueprints

The concept blueprints are a set of basic concept types which are selected as a basic structure for new concepts. These types contain a set of concept attributes and adaptation rules. The blueprints file also contains a set of standard attributes which is applied to all concept blueprints. This is an extension to the extent of templating in the AHA! graph author. The reason for the standard attributes is that GALE makes use of adaptive views, such as the “static-tree-view” or the “next-view” [SmBr11] which use these standard variables for their adaptive behavior. These views are usually part of a layout in which concepts are presented.

By having these attributes apply to all concept blueprints it is much easier to use multiple concept blueprints within the same view. The remainder of this section will cover the template format and content.

An example blueprint template is shown in *figure 3.4*. Two main objects are declared in this template file; “*defaultAttributes*” and “*conceptTypes*”. These contain the default attributes and the concept blueprints respectively. The attributes consist out of an attribute name, data type (Boolean, Integer, Double or String) and a default value. Attributes with the Double or Integer type contain an extra parameter which is used to resolve multiple adaptation rules applied to this attribute. The current version of ALAT supports “SUM” and “AVG” to either sum up the outcome of all expressions or take the average of the resulting adaptation expression values. The source code is set up for future expansions such as using a minimum (MIN) or maximum (MAX) value.

The object containing the concept types in this case contains the definition of a single concept blueprint. It consists out of a template name and the sets of concept attributes and adaptation rules as mentioned before. The concept attributes are defined much like the default. The main difference is that these attributes do not support multiple rules to be applied to them. The reason for this lies in the way adaptation rules are applied. The attributes targeted by adaptation rules do not need to be declared separately in the concept blueprint. Even though this limits the possibility of applying multiple rules to a non-default attribute, it prevents having to declare every attribute twice. This would have made creating and editing template data confusing and complicated.  
These concept attributes are meant to be used in page templates[[3]](#footnote-3) or to be used as parameters for adaptation rules (for example, a minimum amount of right answers required to pass a test).

Figure 3.5: An example blueprints template.

{    
   **"defaultAttributes"**:[    
      {    
         **"name"**:"suitability",  
         **"type"**:"Boolean",  
         **"value"**:"true"  
      },  
      {    
         **"name"**:"knowledge",  
         **"type"**:"Double",  
         **"value"**:"0",  
         **"operator"**:"AVG"  
      }  
   ],  
   **"conceptTypes"**:[    
      {    
         **"name"**:"text-topic",  
         **"default\_attributes"**:[    
            {    
               **"name"**:"info",  
               **"type"**:"String",  
               **"value"**:"This is some basic concept information text!"  
            }  
         ],  
         **"default\_rules"**:[    
            "own knowledge update",  
            "visited",  
            "knowledge\_update"  
         ]  
      }  
   ]  
}

As can be seen in the template example, the adaptation rules section of the concept blueprint does not contain actual adaptation rule definitions. All rule and relation definitions are stored in a separate template. This makes all adaptation rules reusable and prevents redundant rule declarations.

Having Blueprints takes care of standard concept structures without having to set these manually and without having to know their details. On top of that it prevents redundant work in setting up this structure multiple times for each new concept. In other words; the concept blueprints benefit both the barrier-of-entry as well as the authoring workflow efficiency. The standard attributes can be edited in order to take in account possible future updates to GALE views as new views and layouts might use a different set of default properties for their adaptive behavior. The concept blueprints template ensures the tool remains usable and can be brought up to date as new content is created for GALE.

### 3.2.4 Relation Definitions

The pedagogical relation definitions templates define all rules and relations that exist within ALAT. The purpose of the authoring tool is to have the user author adaptive hypermedia content without having to deal with any code or definition files. It is also important that rule definitions can be edited, extended and created. For this reason all rules and relations are defined within a template. Though moving all code and definitions away from the user limits the versatility and variety of possible rules within ALAT, it provides users with rules which are easy to comprehend and which are even easier to apply.

An example of how a rules and relations template is structured is provided in *figure 3.6*.   
Three types of rules and relations are defined in this example template. These can either be applied manually by adding them to concepts in the authoring interface, or they can be applied automatically by being included in a concept blueprint template. The distinction between different types of rules and relations can be made based on their application.

Figure 3.6: An example pedagogical relation definitions template file

{

**"def\_att\_rules"**:[    
      {    
         **"name"**:"knowsOR",  
         **"type"**:"binary",  
         **"target"**:"suitability",  
         **"tooltip"**:"Target concept must be learned before source is recommended.",  
         **"code"**:"${%target%#knowledge} > 0.8",  
         **"operator"**:"and"  
      }  
   ],  
   **"persistent\_att\_rules"**:[    
      {    
         **"name"**:"visited",  
         **"type"**:"unary",  
         **"properties"**:[    
            {    
               **"name"**:"visited",  
               **"type"**:"Integer",  
               **"defval"**:""  
            }  
         ],  
         **"tooltip"**:"stores number of concept visits in ‘visited’",  
         **"code"**:"#[visited]:Integer event +`if (${#suitability}) { ${#visited}++;}`"  
      }  
   ],  
   **"def\_relations"**:[    
      {    
         **"name"**:"rotatesAround",  
         **"tooltip"**:"source concept rotates around the target object."  
      }  
   ]  
}

First there are rules that apply to the default attributes defined in the blueprint template. These rules are different from others because multiple conditions and rules can be applied to a single default attribute. They usually consist out of a name, type, target, tooltip and the rule GAM code and are defined within the “*def\_att\_rules*” object. The rule type can either be “*unary*” or “*binary*”. ALAT uses this parameter to know whether a rule targets another concept or not. This is necessary because ALAT does not interpret the GAM expressions set in the template. By having this type distinction ALAT will have the user select a target concept for binary rules. This will be applied by replacing *“%target%”* in the gam expression by the actual target concept name.

The “*target*” attribute refers to the name of the default attribute this rule is targeting.  
Rules targeting Boolean default attributes get another parameter called “*operator*”. This is used to indicate whether this rule should be declared in an “*AND*” or an “*OR*” clause. This can be used for special kinds of prerequisites, for example. Multiple “*OR*” and “*AND*” clauses can be applied to the same standard attribute.

**Example:** We take three rules called “*OrRule1*”, “*OrRule2*” and “*AndRule*” targeting Boolean attribute “*suitability”*. “*AndRule*” has the “AND” operator applied to it whereas “*OrRule1”* and “*OrRule2”* have the “OR” rule apply to them. When applying these three rules the resulting GAM code would be:   
“*#suitability:Boolean =* `*true && AndRule && (false | OrRule1 | OrRule2) `*”. This combines all three rules into a single GAM-expression. Now “*suitability”* will have a “*true”* value if “*AndRule”* values true and at least one of the two “OR” rules is true.

The second type or rules within the template consists of attributes which are defined by a single GAM expression. They are contained within the “*persistent\_att\_rules”* object. These rules can thus not use the “operator” parameter as only a single rule can be applies to the attribute targeted. Instead these rules do have a set of attributes which are defined in the rule “properties” array. These attributes should include the target attribute the rule expresses as well as all other attributes which are used as parameters in this rule. The attributes declared here consist out of a name, type and default value. As ALAT does not interpret the GAM code, the default value of the targeted attribute should remain blank. This tells ALAT that the value of this attribute is defined by the GAM code.

Lastly there are the non-pedagogical relations. These are defined within the “*def\_relations”* object. These relations do not have any particular adaptation behavior attached to them, but are labeled relations between two concepts (as described in section 1. [CHAPTER REF]). They consist out of a name and tooltip. No other parameters are required as no attributes can be targeted by these rules and they are binary by default.

The implementation of this template in the authoring environment provides easy and clear rules which can be added by the click of a button without any GALE code knowledge required. All rules and relations are supplied with tooltips in order to prevent misinterpretation and to make them easier to understand.

3.3 The ALAT system architecture

This section deals with the ALAT system architecture and technical aspects. ALAT is a browser-based web application with HTML and Javascript running in a browser screen. This is a lightweight, well-documented and platform-independent solution which is up to current web standards. The AngularJS framework has been chosen in order to develop dynamic and extensible Javascript code. AngularJS is a framework which provides the tools necessary to create dynamic views in which rapidly changing data can be displayed with ease.

AngularJS also provides the tools for a layered object orientated approach which aids in the creation of maintainable extensible code. The ALAT client side application layer consists out of a 3-tier model (*figure 3.7*): A controller layer in order to manage the views displayed in the corresponding HTML page, a service layer for data storage and manipulation and a data layer which communicates with the server-side of the ALAT system. A more elaborate version of this architecture can be found in [APPENDIX REF].

Figure 3. 7: The ALAT client-side Application Architecture.

The server-side of ALAT (*figure 3.8*) uses PHP to manage database and storage operations which are used to manage user account and project information as well as the actual project data. The user account and project information is stored in a MySQL database. The project data is stored in JSON files.



Figure 3. 8: The ALAT Back-end Architecture

## 3.3 Generating GALE applications

After a GALE application has been designed using the graphical user interface it is ready to be generated and deployed. In ALAT this is done by using a “generate” button. This single-click operation starts both the generation of the GALE application by generating a resulting “concepts.gam” file as well as the deployment of this application on the GALE server. This makes the creation of GALE applications in ALAT very easy and helps the user deploy his application without requiring any knowledge of the GALE server workings or the deployment procedure.

### 3.3.1 Generating a gam file

Before the GALE application can be deployed, its GAM file first has to be generated. This section will deal with all relevant steps ALAT takes when generating this file. Unfortunately, this process is more complicated than simply printing out the data structure created in ALAT and a series of steps has to be taken in order to create the right GAM format containing the appropriate information.

The first thing that needs to be taken care of is printing the concept templates. The list of concept blueprints is first filtered by checking which of them are used within the ALAT project. There is no need to print concepts that are not used and doing so would result in unsanitary and obsolete GAM code.  
After this selection is made the used concept blueprints are translated to GAM. When this is completed all selected project concepts are outputted. The ALAT interface has the option to include or exclude parts of a concept tree. *Figure 3.9* shows a project in which only the concept hierarchy of “*Plants”* and “*Animals”* would be generated and deployed. This enables applications with a select part of the designed ALAT project to be generated and deployed.

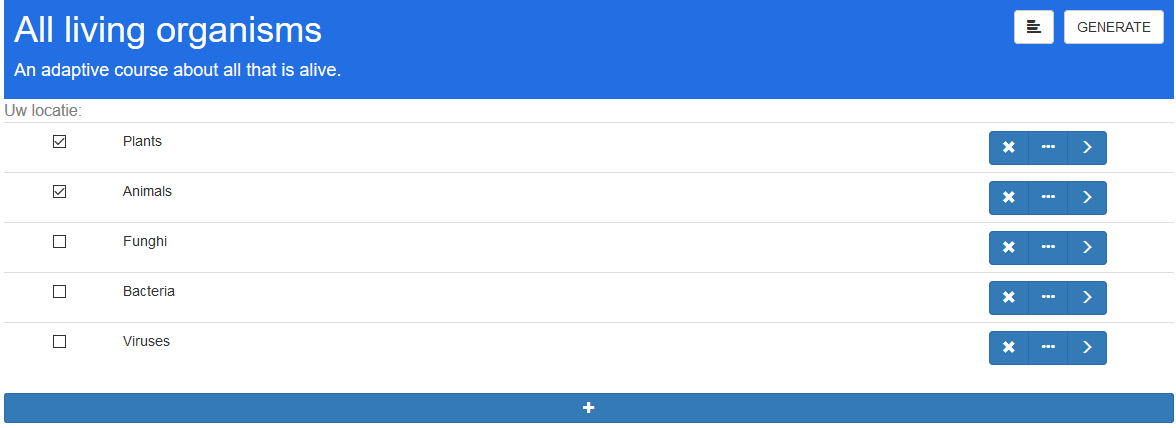


Figure 3. 9: An ALAT project set to be only partially deployed.

The concept blueprint behavior is applied to these concepts by means of inheritance. In GAM this inheritance is created by applying   
“*->(extends)blueprintName”* to the concept code, where “*blueprintName”*  is the name of the concept blueprint selected for the concept. This is a relation which enables the inheritance of concept behavior. The project structure as created in the ALAT user interface is enforced by applying relations with a “parent” label to all concepts.

Attributes, parameters and simple attribute expressions defined within both the concept blueprints as well as within the concepts themselves are pretty straightforward to translate to translate to GAM. As there are no dependencies on them, they can simply be transformed to GAM syntax. Printing the standard attributes however, is more complicated.

Standard attributes, as mentioned before, are attributes which can have multiple rules and relations applied to them. A list of all these applied rules has to be made before their GAM code can be created. When this list is constructed, all operators as described in section 3.2.3 and 3.2.4 can be applied and the GAM output can be created.   
Creating GAM code for concepts other than the used blueprints takes an extra step. Attribute relations and rules inherited from the concept blueprints can be overwritten by declaring the attribute again in the current concept. Therefore when a concept extends the rules on standard attributes as declared in its blueprint, it is required for all inherited rules to be applied to the attribute once more. This has to be done in order to maintain the blueprinted behavior.

When the resulting GAM code has been created, it is sent to the data access layer on the server.

### 3.3.2 Deploying GALE applications to the server

After the ALAT project has been translated to a GAM file it is passed through to the data access layer by means of AJAX requests. This layer creates a unique folder within the GALE application directory using the project and author name. The data layer writes the GAM code into a file named “concepts.gam” which is then deployed to this directory. When the directory already exists, the existing “concepts.gam” is replaced with the new version.

A reference to a default layout appropriately named “layout.xhtml” as well as a placeholder file named “placeholder.xhtml” which is used for concepts without any declared content resource are also stored within the data layer. After the “concepts.gam” file is created a copy of the layout and the placeholder are placed in this folder as well. This results in a deployed GALE application in which it is easy for experts to extend and edit both the GALE application itself by editing the GAM code as well as the default layout and placeholder by editing their appropriate files. All these files are conveniently located within a single directory.

[NuAL09]

Nurseitov, Nurzhan, et al. "Comparison of JSON and XML Data Interchange Formats: A Case Study." *Caine* 9 (2009): 157-162.

[Sm11]

Smits, D. David. *Towards a generic distributed adaptive hypermedia environment*. Diss. Technische Universiteit Eindhoven, 2012.

[SmBr11]  
Smits, David, and Paul De Bra. "GALE: a highly extensible adaptive hypermedia engine." *Proceedings of the 22nd ACM conference on Hypertext and hypermedia*. ACM, 2011.

1. Yuexu Chen. She was active as an interaction designer at TU/e during the development of the ALAT project. [↑](#footnote-ref-1)
2. http://www.w3.org/ [↑](#footnote-ref-2)
3. The use of page templates will not further be covered in this Thesis. [SmBr11] provides an in-depth description. [↑](#footnote-ref-3)