Department of Informatics
University of Fribourg, Switzerland
http//diuf.unifr.ch/

Game-Master - Visual Editor

Visual Editor for creating games based on Dynamic Rules

Bachelor Thesis

Roland Plüss
November 2008

Under the supervision of

Prof. Rolf Ingold

Maurizio Rigamonti

Denis Lalanne

Prof. Béat Hirsbrunner

Fulvio Frapolli

Amos Brocco

Table of Contents

1 Introduction....4
1.1 Context and Motivation....4
1.2 Goals....4
1.3 Outline....4
2 Dynamic Rules Theory....5
2.1 Model....5
2.2 Language (DSL)....7
3 Visual Editor....10
3.1 GUI....10
3.2 Interface Components....11
3.3 Architecture....12
3.3.1 Game Model....13
3.3.2 GUI Models and Components (Model, View)....13
3.3.3 Undo System (Controller)....15
3.3.4 XML File Parsing....15
3.4 XML File Format....16
4 Conclusions and Perspectives....20
4.1 Summary....20
4.2 Extensions....21
5 References....22
5.1 Illustration Index....22
5.2 Index of Tables....22
5.3 Code Listings....22
6 Annexes....23
6.1 XML File Format Schema....23
6.2 Tutorial....27
6.2.1 Create a new game....27
6.2.2 Create models....27
6.2.3 Assign properties, relationships and appearances....27
6.2.4 Create entities....28
6.2.5 Setup layers....29
6.2.6 Update relationships / creating game topology....30

6.2.7 Creating states and behaviors..31
6.3 Developer Information..35
6.3.1 Tag Parser...35
6.3.2 Adding a new statement...35
6.4 CD-Rom Content..37

1 Introduction

1.1 Context and Motivation

Creating games is a complex and hard endeavor especially if it has to be done by hand. Providing a structure to the process of creating a game as well as providing supportive tools can help a lot to improve this process. It is usually easier to design a game in a visual way than it is to create it using programming languages or formats maintained by hand. This bachelor work provides a solution for reducing the complexity of creating a game providing a visual editor using the generic model provided by the Dynamic Rules Theory.

1.2 Goals

The main goal is to implement a visual editor application using the model defined by the Dynamic Rules Theory. The editor allows users to create a game using only visual editing features. Editing has to be simple and intuitive using the principle of the least surprise. Writing code in text form that is parsed is not required neither editing complex XML files. The editor allows editing generic games and is not limited to one particular game or game engine.

The second goal is to create an XML file format storing a game definition. The definition of this format is generic as is the editor. Editor specific informations are separated from the game definition in the XML file.

The final editor is validated and tested using an example game of Awele as found in the tutorial section in the annexes.

1.3 Outline

First the Dynamic Rules Theory is explained briefly. Then a proposal is given on how to represent and edit games according to the theory. Next the graphic user interface and architecture of the editor is explained and the created game definition file used by the editor will be explained. In the end a conclusion of the work and possible future extensions are talked about. A tutorial section as well as developer informations are provided in the annexes for further reading.

2 Dynamic Rules Theory

In simple games it happens often that rules are slightly modified by the players to adapt the game for different player skills, shifting the game into a more interesting or challenging one or creating house rules which are more fun to play. In this case hard-coding game rules are a hindrance and deny this flexibility. On the other hand in complex games it makes sense to delegate some rules to a computer. These are usually book keeping rules which are cumbersome to keep track of by the players. Relieving them of this burden improves the game experience since the players can focus on the important game mechanics instead of tedious book keeping. The Dynamic Rules Theory explores the possibility to define a game as well as redefining dynamically the rules thereof using a higher abstraction level model.

2.1 Model

The game logic is distributed amongst various game entities. Each entity knows a small subset of the entire set of rules which affect the entity in one way or the other. In addition entities know the part of the game topology surrounding them. Connections between entities are called relationships. Entities can communicate with neighbor entities if they are known through a relationship. Entities can represent any element of the game ranging form tokens on a play field over the play field itself to the players. Entities can also be objects without a physical shape.

A model defines the properties, behaviors and appearance of a group of entities. Each entity belongs to exactly one model which defines its structure. Models are comparable to rubber stamps for entities. The properties define the current state of an entity while relationships define neighbor entities. The appearance defines the visual representation of an entity. Behaviors define the actions an entity can carry out. Models can inherit from other models to reuse definitions. The following graphic shows an example of a game structured using models and entities. Models are represented in blue while entities are represented in red. Models inherit and optionally assign new values to the definitions found in their parent model. Entities inherit all definitions from their parent model. In contrary to models they can only change the value of definitions but not alter the definitions itself.

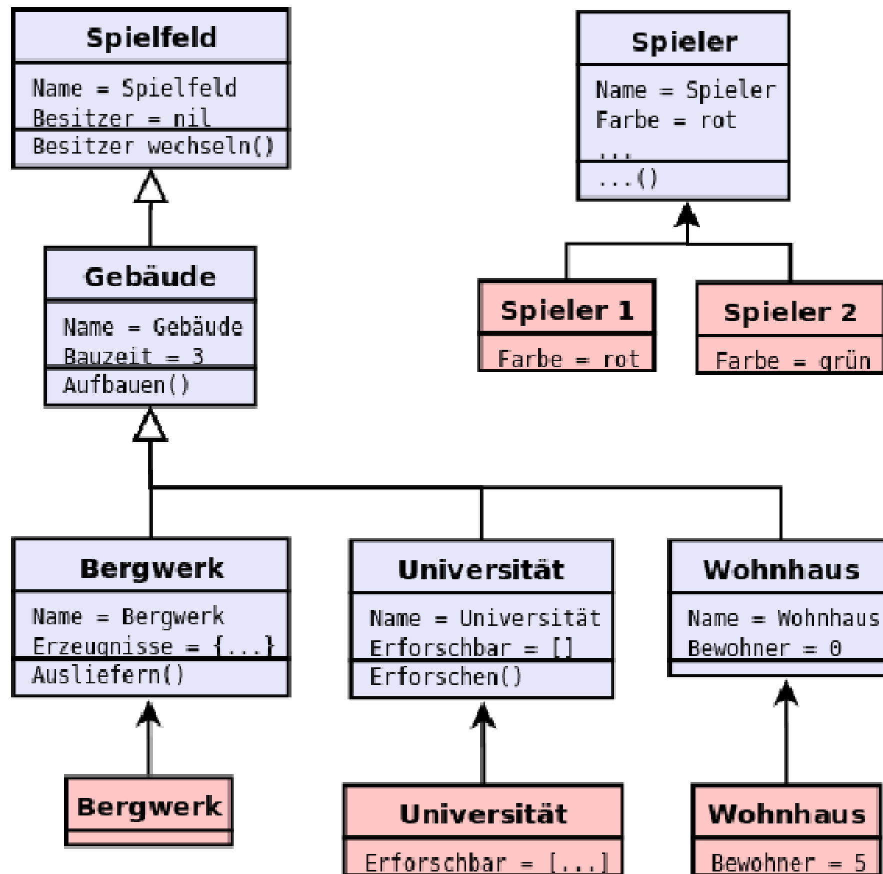


Illustration 1: Game structure using models and entities

The game is represented using a two layer view: the physical layer and the logical layer. The physical layer represents where the game physically takes place for example a chess board. The logical layer represents the game logic and is defined by the state and behavior of the entities. Entities communicate with their neighbor entities using message passing. A message is event based and can alter states on both layers. Physical properties as for example the proximity of entities can also influence the behavior of an entity.

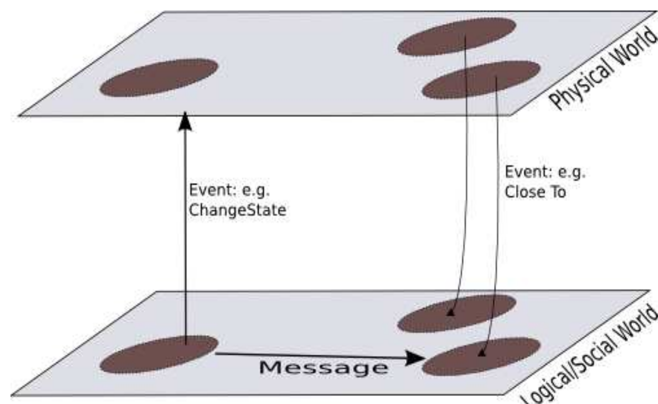


Illustration 2: Logical and Physical Layer/World

2.2 Language (DSL)

The editor is based on the language proposed by the Dynamic Rules theory. To use this language two important elements are required: Value Types and Statements. The values are stored in entity properties, relationships and appearances. A basic set of value types is provided by the language. The following table contains a rundown of the known types.

Value type	Description
Numeric	Stores a numerical value as floating point. Both integers and decimal values are represented by this type.
String	Stores a text string of variable length. The strings are 0-terminated and does not have an explicit length.
ID	Stores a reference to an entity which is the unique name of the entity. “No entity” is represented by an empty string.
List	Stores an ordered collection of values. The values inside the list can be of arbitrary type.
Dictionary	Stores an unordered associative collection of values. Each value is identified using a string key. Any kind of value can be stored.

Table 1: List of value types

The second important element are the statements. Behaviors are build out of a nested list of statements. The language contains a basic set of statements. There is a special case in terms of value types concerning the statements. Boolean values are mentioned where an ID of “nil” equals false and all other values of any type equal true. The following table lists all statements including a short description as they are used in the editor.

Statement	Description
Block	Executes an ordered list of statements in sequence.
Operator	Executes one or two statements (operands) applying a given operator. Various arithmetic, comparison and logical operators exist some of which require two operands others only one. If only one operand is required the first of the two operands is used and the second ignored. Valid operators are: *negate, add, subtract, multiply, divide, modulus, equals, not equals, less, less or equal, greater, greater or equal* and *logical and/or/not.*
Get Property	Retrieves the value of a property of an entity. Requires the name of the property and the entity (a statement).
Set Property	Sets the value of a property of an entity. Requires the name of the property, the entity (a statement) and the value (a statement).
Get Relationship	Retrieves the value of a relationship of an entity. Requires the name of the relationship and the entity (a statement).

Statement	Description
Set Relationship	Sets the value of a relationship of an entity. Requires the name of the relationship, the entity (a statement) and the value (a statement). Only values of type ID, List and Dictionary (with nested values obeying the same rules) are allowed.
Get Appearance	Retrieves the value of an appearance of an entity. Requires the name of the appearance and the entity (a statement).
Set Appearance	Sets the value of an appearance of an entity. Requires the name of the appearance, the entity (a statement) and the value (a statement).
ChangeState	Changes the state number of an entity. Send messages only execute behaviors with matching names in the matching state.
Send Message	Sends a message to another entity. Requires the name of the message (the behavior to execute) and the entity (a statement). Optionally parameters can be send together with the message. The parameters are represented using an unordered associative map of values (a statement) identified by a name (a string).
Get Parameter	Retrieves the value stored in a local variable. Variables are local to the statement they are defined in and below.
Myself	Retrieves the current entity.
Constant	Retrieves a constant value.
If-Else	Executes statements depending on a condition. Multiple If-Cases can be specified each with a condition (a statement) and a statement to execute if the condition holds true. The condition statement is required to return a boolean value. Optionally an else statement can be specified executed if no If-Case condition holds true.
While	Executes a statement as long as a condition (a statement) holds true. Condition is required to return a boolean value.
For Each	Executes a statement for each element in a given list or dictionary. Prior to executing of a list or dictionary element a local variable of a given name is set to the element. For a list this is a value for a dictionary the key of the value.
Continue	Skips the rest of the loop statement returning to the condition test for a while statement or retrieving the next element for a for each.
Break	Stops executing a while or for each statement.
Return	Stops executing a behavior returning an optional value.
Declare Variable	Declares a local variable. The variable is local to the statement it is defined in as well as all nested child statements.

Statement	Description
Get Variable	Retrieves the value of a local variable. Requires the name of the local variable to read.
Set Variable	Sets the value of a local variable. Requires the name of the local variable to modify and the value to set.
Range	Retrieves a list value containing number values starting at a given start value up to a given end value spaced by a given step size.
Length	Retrieves the number of values stored in a list or dictionary value as a numeric value.
ListGetAt	Retrieves the value at the given position from a list value.
ListSetAt	Replaces the value at the given position in a list value with a new value.
ListAdd	Adds a value to the end of a list value.
ListRemove	Removes the value from the given position from a list value.
DictionaryGet	Retrieves the value with the given key from a dictionary value.
DictionaryPut	Adds a value to a dictionary value. If a value with the given key exists it is replaced with the new value. If not a new value with the given key is appended to the dictionary.
DictionaryRemove	Removes the value with the given key from the dictionary.
Random	Retrieves a random number value with the number located between a lower and an upper value (inclusive).

Table 2: List of statements

Defining a game using all these properties, relationships, appearances and defining the behavior is a complex task. An editor is proposed to reduce the complexity of editing a game. The editor is capable of working with both layers described in the Dynamic Rules model. It is possible to edit properties, relationships between entities, appearance and behaviors as well as working with entity models. A graphical interface is proposed for visually editing the behaviors and establishing the game topology.

3 Visual Editor

Using the Visual Editor games can be created using only visual means. Both the logical and physical layer as defined by the Dynamic Rules Theory are represented in the editor. The entities are represented as real objects visible on the physical layer using their appearance and controllable on the logical layer using their model. For editing the behaviors a visual representation of statements is used not requiring to write code explicitly. The following sections describe the functionality and design of the editor.

3.1 GUI

The editor is split into 4 major parts. The *Game Tree* lists all models and entities present in a game. This tree allows to add and remove entities and models as well as establishing the hierarchy amongst models and assigning entities to them. The *Properties Table* displays detailed information about the selected entity or model. Provided are tables for editing properties, relationships, appearances, behaviors as well as the layer list. The *Behavior Editor* shows the selected behavior using a nested view of statement panels. Editing the behavior is done using this graphical representation not requiring a scripting language to be used. The last part is the *Workspace* displaying the entities using their appearance properties. Entities can moved around and assigned as neighbors to each other in this location. The next four sections describe the mentioned editor parts in more detail.

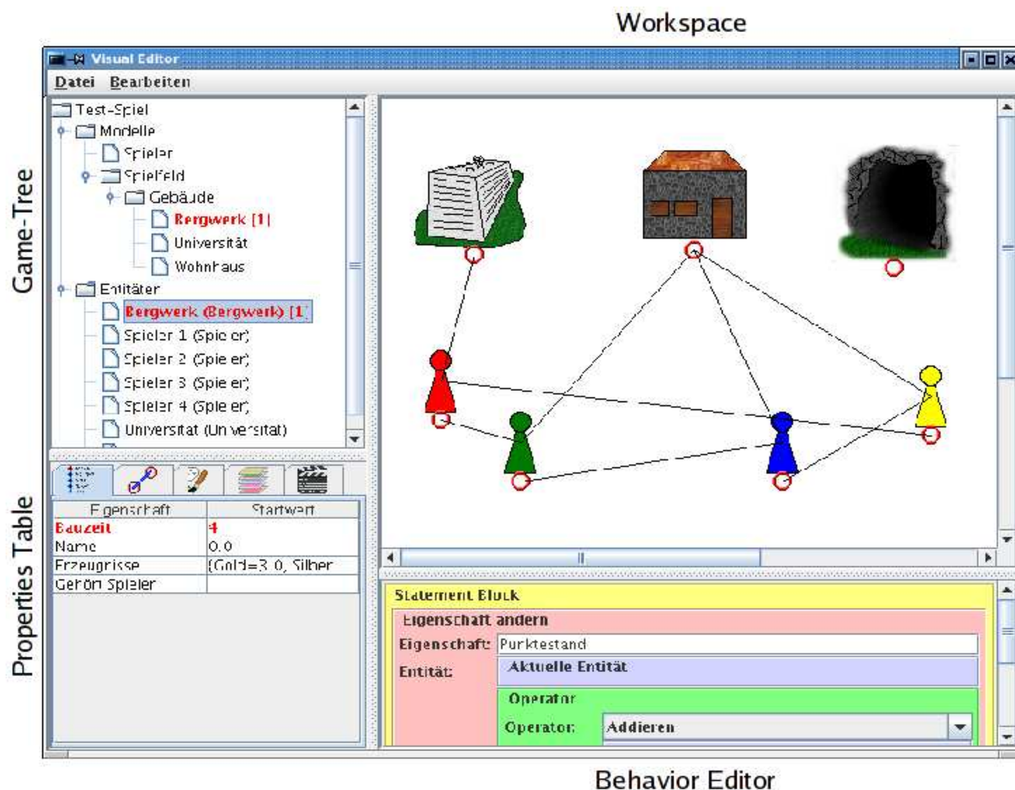


Illustration 3: The 4 editor parts:

3.2 Interface Components

The *Game Tree* allows to edit models, entities and their relationships using a context sensitive pop-up menu. Models can be added and removed, their parent-ship changed or renamed. Models are shown in a tree hierarchy showing their relationship. Models are displayed under their respective parent model or directly under "Modelle" if they have no parent model. Entities can be added and removed, assigned a model and renamed. They are shown all under "Entitäten" with the name of their parent model in round parenthesis. Entities without a parent model show no model name. If displaying of errors is enabled models and entities with errors are drawn in a red color with the number of found errors in square brackets.

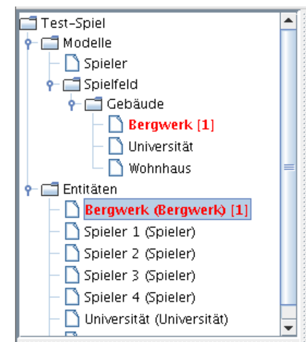


Illustration 4: Game Tree

The *Properties Table* allows to edit properties, relationships and appearances of both models and entities, states of models and visual layers for the Workspace. The name and the value of these properties can be edited right inside the table. Depending on the value type an adequate editor field is provided. List and Dictionary values pop up a separate dialog for editing. A varying font is used to provide additional informations about the modification and origin of properties. A normal font indicates properties defined in the active model. An italic font indicates properties inherited from a parent model and a bold font indicates locally modified values. Incompatible types and other errors are displayed using a red color.

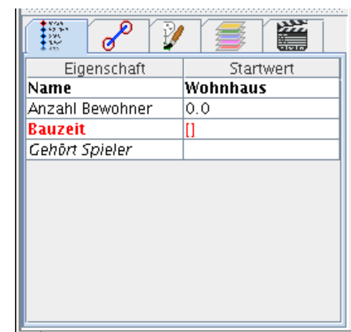


Illustration 5: Properties Table

The *Behavior Panel* allows to edit the statements of the active behavior. Each statement is displayed using a separate panel. Panels are nested and can be collapsed for increased readability. A context sensitive pop-up menu allows to add, remove and otherwise alter statements. Copy and paste is supported for individual statements and values where applicable. To further increase readability a color code is used for the statements. The following table summarizes the colors used:

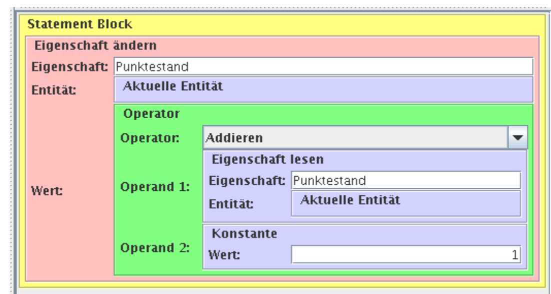


Illustration 6: Behavior Editor

Color	Meaning
	Statements controlling the flow of code including loops, conditional execution and the statement block.
	Statements acting as the source of values. These statements read a value from properties, relationships, appearances, parameters, local variables or from a constant value. These statements are always the start of a chain of statements. If the top of a statement pyramid is not a statement of this kind

Color	Meaning
	or a send message statement an error is usually present.
	Statements consuming a value and altering the state of an entity or a local variable. These statements are at the end of statement chains and change the state of the game. They are therefore marked in a reddish tone since this is the place to look for if incorrect actions occur in a game.
	A send message statement. This statement transfers the execution to a different behavior optionally on a different entity. Helps to locate places where the execution of code ventures to other entities. Can return a value and can therefore happen anywhere in a chain.
	Statements that process one or more inputs and produces a new output based on these inputs. Green statements never happen at the beginning or the end of a statement chain and are a sign of an error.
	Declaration of local variables.
	Empty statement.

Table 3: Color Code of statements

A typical behavior consist of a yellow block statement composed of a a chain of red-green-blue statements where the green statements are optional. If this color chain is not present an error is often present. Exception are send message statements which can happen anywhere.

The *Workspace* shows the graphical view of the entities and the relationships between them. The visual layers from the Properties Tables are used to govern visibility of entities and relationships as well as protecting them against modification. The context sensitive pop-up menu allows to add, remove and alter entities similar to the Game Tree as well as breaking connections. Entities can be repositioned and connections created using the mouse.

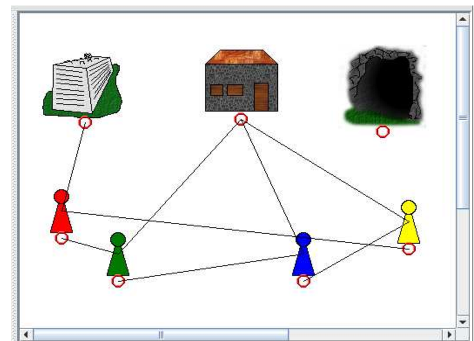


Illustration 7: Workspace

3.3 Architecture

The editor is based in general on the Model-View-Controller paradigm. The following image gives an overview of the individual parts of the editor explained in the upcoming sections.

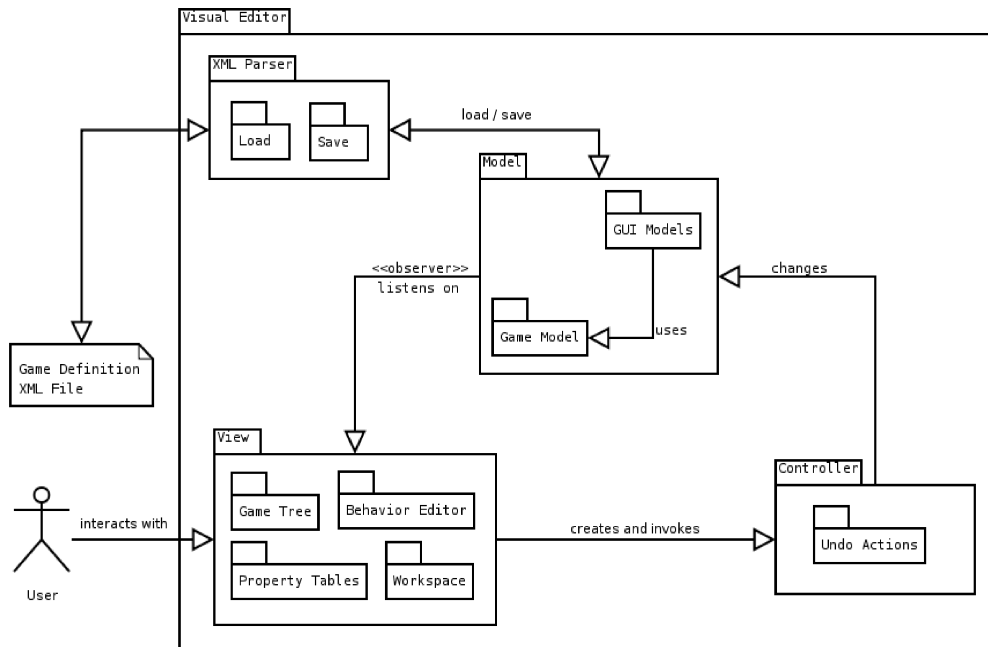


Illustration 8: Architecture Overview

3.3.1 Game Model

Two different models are used one for the game and one for the GUI. The game model defines the individual parts of the game. The Game class contains a list of models and entities both containing a back link to the game object. The models store their parent model if existing hence all models are located in the game object not distributed across models. The entities contain also a link to their model and are all located in the game object not nested under their parent model. Inside models is a list of states. Inside states is a list of behaviors. To avoid a mess of observers only one observer is used for the entire game. This way observing is simpler but the observer interface is more complex.

3.3.2 GUI Models and Components (Model, View)

For displaying informations in the GUI a couple of adapter models are used which are driven by the game observer. Some of these models are used by only one GUI component while others are reused at various places like the list of entities to be used in combo boxes. To keep things clean all models are localed in a GUI Models class. The individual GUI components can then query the required models from this central place.

The *Game Tree* uses a modified tree model containing objects derived from a common tree node model. Different types of objects in the tree use specialized tree nodes. This allows to send down the mouse click events to the tree nodes receiving per object specific actions like filling the properties tables with the right values and displaying a pop-up menu with object specific actions. Supporting new object types or adding new object actions can be done easily by modifying the appropriate node model or creating a new one. Models are displayed nested using their parency. This way the structure of the game is visible right

inside the Game Tree. The entities are shown all on the same layer. A possible extension could group them for better overview. The Game Tree keeps track of the collapsing state restoring the collapsing of models upon notifications from the game object. The Game Tree belongs to the logical layer of the Dynamic Rules model.

The *Properties Tables* (properties, relationships and appearances) use a modified table model using specialized cell renderer and cell editors. The original JTable class and DefaultTableModel have not been able to provide the complex per object editing behavior required by the editor. For this the JTable has been modified to support individual cell renderers and cell editors for each table cell instead of entire colons. This way editing properties with the matching component right inside the table is possible. For editing ID values a combo box is used filled with the names of all entities. For List and Dictionary values a button is used to bring up an editing dialog. This dialog uses the same modified table class and therefore allows also in-place editing. List and dictionary dialogs can be nested if required. The Properties Tables belong to the logical layer of the Dynamic Rules model.

For the layers table (also found inside the Properties Tables area) the conventional table model is used with a cell renderer and cell editor for each table colon. Since a renderer and editor for boolean values is lacking simplistic implementations thereof are provided. The layer table belongs to the physical layer of the Dynamic Rules model.

The *Behavior Panel* displays the statements contained in a behavior using a nested view of statements. Each statement is represented using its own component derived from a basic statement panel class. Statement panels can contain other statements resulting in a nesting of components. Compared to the other parts of the editor the Behavior Editor is rather complicated and tricky to modify and extend. The individual statement panels store the child statement panels directly instead of retrieving them from the JContainer. This is done since the layout of the components in the panels is rather tricky requiring often nested panels to do the proper layout. Furthermore each statement panel can be collapsed individually. During updates this state would be lost which would turn editing cumbersome. To avoid this partial updating is used. Therefore statement panels only modify children statement panels if they are new. Last but not least the right mouse button context menu is created most of the time telling the parent statement panel to create a context menu for us. This is required since modifying a statement in fact requires a modification of the parent statement. The Behavior Editor is different from other components in that it does not use a GUI model at all but using only the game observer. The Behavior Editor belongs to the logical layer of the Dynamic Rules model.

The *Workspace* displays the entities using their appearance. The Workspace has a given dimension inside which the entities are rendered. For rendering the entities appearances with a given name are used. The following list contains the appearance names known by the Workspace (names are case sensitive).

Appearance Name	Description
Bild X	The X coordinate of the image measured relative to the upper left corner of the workspace.
Bild Y	The Y coordinate of the image measured relative to the upper

Appearance Name	Description
	left corner of the workspace.
Bild Dateiname	The path to the image to use. Supports all images the Java runtime can read. The path is relative to the location of the game file but can be also an absolute path.
Bild Z-Order	Defines the drawing order of overlapping entities. Entities with higher number are drawn above entities with lower numbers. The order for entities with equal numbers is undefined.

These names are a proposal. Game engines can use different names. In the options class the names can be changed if required. The Workspace too uses no GUI model to display the entities using only the game observer. For entities, relationship slots and relationship links individual classes are used which are not derived from Java swing components. The Workspace stores a list of these objects traversing them upon rendering or mouse interaction. Drag and drop is supported using a simple mechanism not using the Java swing drag and drop support. The Workspace belongs to the physical layer of the Dynamic Rules model.

3.3.3 Undo System (Controller)

Controllers are realized using the undo system. Only classes derived from the Undo class do change game states of an active game object. Observing is used to notify the views about the changes. This design has been chosen since this way all doable actions also have immediately the undo-able action at hand. Using the undo system as the controller enables users to prototype easily their games. All actions can be undone on a fine grained scale which allows users to learn the interface without fear of corrupting their game definition doing something wrong. Removing the worry to mess up also prevents the need to save permanently. This makes the Undo System the prime choice for placing all the controlling needs.

3.3.4 XML File Parsing

For parsing the XML file the internal XML parsing support of Java is used. Parsing is done using a stack of tag parser derived objects sending the current XML parser action to the top tag parser. This way parsing is done using a tree structure which is easier to handle for the rather complex file format of the game definition file. Due to the structure of the parsing at a couple of places delayed initialization is used at various places in the tag parsers. Upon reaching certain tags a new tag parser is placed on the stack. The end of the tag is send to this new parser. Therefore initializing the tag content right after the end of the tag is not directly possible. This is delayed until the tag of the current tag parser ends. After removing it the original tag parser can gathered the data and store it into the objects. Saving the XML file is done using a simple method based writing.

3.4 XML File Format

The XML file format is based on a separation between the game and editor related informations. All informations concerning only the editor are located in tags named “visual”. Game engines can ignore all tags named “visual” obtaining the pure game definition. This section contains a short description of the file format in table form. See 6.1 for an XML Schema.

The main section is the “game” tag which is the root of every game definition.

Tag	Description
game	Root tag. Has no attributes.
game.name	Name of the game.

Every model is defined using a “model” tag. Models are required to be defined in the order they parent to each other. Hence if model A is the parent of model B then model A has to be defined before model B.

Tag	Description
game.model	Defines a game model. The name attribute indicates the unique name of the model.
game.model.initialStateNumber	Initial state number of the model. Number is an integer given as the CDATA.
game.model.parentModel	If specified indicates the parent model. The name of the parent model is given as the CDATA.
game.model.property	Property in the model named using the name attribute. Initial value is given by the one and only statement type tag inside.
game.model.relationship	Relationship in the model named using the name attribute. Initial value is given by the one and only statement type tag inside.
game.model.appearance	Appearance in the model named using the name attribute. Initial value is given by the one and only statement type tag inside.
game.model.state	State in the model with the number given in the number attribute. Number has to be unique.
game.model.state.behavior	Behavior named using the name attribute.
game.model.state.behavior.statements	Statement block of the behavior.

Every entity is defined using an “entity” tag. Entities have to be defined after the model they use. Therefore models are written before entities.

Tag	Description
game.entity	Entity named using the name attribute and belonging to the model with the name given by the parent attribute. If the entity has no model the parent attribute is the empty string.

Tag	Description
game.entity.property game.entity.relationship game.entity.appearance	Value of a property, relationship or appearance. The value replaces the initial value given by the model. The appropriate child tags behave the same as the game.model.* counterparts with the exception that the relationship tag does not have a visual tag.

Values are defined using one of the value tags. They can be defined at every place a value tag is valid (see 6.1 XML File Format Schema).

Tag	Description
<value>	One of the possible values. If the value is part of a dictionary value the key attribute is required storing the string key of the value.
number	Number value. CDATA contains the float number.
string	String value. CDATA contains the string.
id	Entity ID. CDATA contains the name of the entity or the empty string if nil.
list	List value.
dictionary	Dictionary value.

Statements are defined using one of the statement tags. They can be defined at every place a statement tag is valid (see 6.1 XML File Format Schema).

Tag	Description
block	Block statement which contains a list of statements.
operator	Operator statement. The op attribute defines the operator to use and can be any value from the following list: • negate => Negates the first operand • add => Adds the second operand to the first one • subtract => Subtracts the second operand from the first one • multiply => Multiplies the first operand with the second one • division => Divides the first operand through the second one • modulus => Remainder of the division of the operands • equal => True if both operands are equal • nequal => True if both operands are not equal • less => True if the first operand is less than the second one • lequal => True if the first operand is less or equal to the second one • greater => True if the first operand is greater than the second one • gequal => True if the first operand is greater or equal to the second • not => True if the first operand is false • and => True if both operands are true • or => True if one or both of the operands are true
operator.operand{1|2}	First respectively second operand. Requires one statement child tag.
getProperty	Get property statement. Retrieves the property with the name stored in the property attribute.
getRelationship	Relationship statement. Retrieves the relationship with the name

Tag	Description
	stored in the relationship attribute.
getAppearance	Appearance statements. Retrieves the appearance with the name stored in the appearance attribute.
*.entity	For all statements having an entity tag this defines the entity the statement operates upon. Has to contain one child statement returning an ID value.
setProperty	Set property statement. Changes the value of the property with the name stored in the name attribute of the entity given by the entity tag to the value given by the value tag.
setRelationship	Set relationship statement. Changes the value of the relationship with the name stored in the name attribute of the entity given by the entity tag to the value given by the value tag.
setAppearance	Set appearance statement. Changes the value of the appearance with the name stored in the name attribute of the entity given by the entity tag to the value given by the value tag.
*.value	For all statements having a value tag this defines the value to use. Has to contain one child statement returning a value of the matching type.
sendMessage	Send message statement. Sends the message with the name stored in the the message attribute to the entity given by the entity tag.
sendMessage.parameter	Parameter to send with a message. The parameter has the unique name defined in the name attribute and the value of the child statement.
myself	Myself statement returning the current entity.
constant	Constant statement. Returns the value given by the child value.
ifElse	If-else statement.
ifElse.if	If-case in an if-else statement.
*.condition	For all statements having a condition tag this defines the statement to evaluate as the statement condition.
*.statement	For all statements having a statement tag this defines the statement to execute if the condition holds true.
ifElse.else	Statement to execute if no if-case condition matches.
continue	Continue statement. Advances to the next loop run.
break	Break statement. Leaves a loop.
while	While statement. Loops while condition is true.
return	Return statement with an optional value given by the child value tag.
getParameter	Get parameter statement. Retrieves the value of the message parameter with the name given by the name attribute.
declareVariable	Declare local variable statement. Defines a variable with the name given by the name attribute.
declareVariable.value	Value to initialize a local variable with.
getVariable	Get variable statement. Retrieves the value of the local variable with the name given by the name attribute.

Tag	Description
setVariable	Set variable statement. Sets the value of the local variable with the name given by the name attribute.
setVariable.value	Value to set the local variable to.
forEach	For-each statement. Defines a local variable with the name stored in the variable attribute set to the iterated value for each run.
forEach.list	List value to iterator over.
range	Range statement. Returns a list value of number values.
range.from	First number value in the list given by the child statement tag.
range.to	Last number value in the list given by the child statement tag.
range.step	Steps between number values in the list given by the child statement tag.
length	Length statement. Returns a number value.
length.element	Element to retrieve length of given by the child value.
listGetAt	Retrieves the value at the given position from a list value.
listSetAt	Changed the value at the given position in a list value.
listAdd	Adds a value to a list.
listInsertAt	Inserts a value into a list at the given position.
listRemoveFrom	Remove the value from the given position from a list value.
list*.list	The list the statement operates on.
list*.index	Index (number) of the value to operator on.
list*.value	Value to add/set.
dictionaryGet	Retrieves the value with the given key from the a dictionary.
dictionaryPut	Adds/Changes the value with the given key in a dictionary.
dictionaryRemove	Removes the value with the given key from the dictionary.
dictionary*.dictionary	Dictionary to operate upon.
dictionary*.key	Key (string) of the value.
dictionary*.value	Value to put in the dictionary.
random	Retrieves a random number value from a range.
random.minimum	Minimum value of the returned random number.
random.maximum	Maximum value of the returned random number.

All the above tags are used to define a game. For the visual representation in the editor some additional editor only tags are used. Game engines can ignore those safely.

Tag	Description
game.visual	Visual informations used by the editor.
game.visual.layer	A layer for the workspace named using the name attribute.
game.visual.layer.visible	Sets the layer visibility (1=visible, 0=invisible)

Tag	Description
game.visual.layer.protected	Sets if the layer is protected (1=protected, 0=editable)
game.model.relationship.visual	Visual informations for a relationship. Editor only informations that can be ignored by engines looking for the pure game definition.
game.model.relationship.visual.lane	Defines the edge around the entities the relationship slot is placed. Valid CDATA values are *top*, *left*, *right* and *bottom*.
game.model.relationship.visual.layer	The layer the relationship is located on. CDATA contains the layer name.
game.entity.visual	Visual informations for an entity. Editor only informations that can be ignored by engines looking for the pure game definition.
game.entity.visual.layer	The layer the entity is located on. CDATA contains the layer name.

4 Conclusions and Perspectives

4.1 Summary

In this document the Dynamic Rules Theory has been explained and a solution proposed on how to represent and edit games visually. The graphic user interface and architecture of the editor has been explained and the used game definition file shown. A tutorial as well as developer informations can be found in the annexes.

The choice of the 4-Panel layout for the editor worked well. The context sensitive pop-up menus at various places help a lot to work quick with the editor. In doubt the user can always call upon the pop-up menu to achieve his goals. The small undo actions also help a lot to achieve good prototyping behavior. Users can learn the interface quickly and safely since errors are easily undone in small steps not requiring to save before critical changes.

The choice of Java as programming language worked well in most cases. Some language specific problems in the domain of GUI design required some workarounds but they are functional and working.

There has been more statements in the end as first expected implementing the Dynamic Rules Theory model. The available cast of statements should though be enough for all kinds of games.

The XML file format is rather clean except a few tags which could be possibly remove. The separation between logical and physical layer as well as editor specific informations turns the file format well structured. It is also possible to parse the file into separate files for logical and physical layers if required this way.

The editor has been validated using a test implementation of the game Awele. The game could be defined fast and simple. The steps have been documented in a tutorial and should be easy to accomplish also for somebody without programming knowledge. Larger games should therefore be also simple to create. There are a couple of possible extensions to improve the editor.

4.2 Extensions

The following list contains a few possible extensions .

- *Behavior verification and input helpers.* The editor does not do any error checking inside statements. In the code statement types have already been included to some degree but they are unused. Most basic checks would be to see if source, pipe and sink statements are used in the right place (always a sink as the last statement or a source as the first). Also for local variables, properties, behaviors or other elements it would be possible to give the user upon request a list of possible values. For example sending a message possible behaviors could be proposed to avoid typing them by hand.
- *More level of connections in the Workspace.* In the editor only connections on the first level (not nested inside lists or dictionaries) is used. It would be possible to add support to create and show connections also of deeper levels.
- *Entity grouping.* In the Game Tree entities are all showed on the same tree level. For games with a large amount of entities this could get a bit untidy. Groups could be introduced to tidy entities up (for example all player entities all hole entities and so forth) for better overview.
- *Improved Workspace.* The workspace displays entities as 2D images. Additional appearance names could be defined to alter the rendering of entities like tinting or transparency. Also a 3D view would be a possibility.
- *Refactoring in the Game-Tree.* It would be possible for larger projects or derived work to have refactoring for models to push up properties, relationships, appearances, states or behaviors to parent models. This would reduce editing work for the user.

5 References

5.1 Illustration Index

Illustration 1: Game structure using models and entities....6
Illustration 2: Logical and Physical Layer/World....6
Illustration 3: The 4 editor parts:....10
Illustration 4: Game Tree....11
Illustration 5: Properties Table....11
Illustration 6: Behavior Editor....11
Illustration 7: Workspace....12
Illustration 8: Architecture Overview....13
Illustration 9: Awele game tree after adding some entities....28
Illustration 10: Workspace with one relationship layer visible....31
Illustration 11: Hole behaviors....31
Illustration 12: Behavior "Kugel hinzufügen"....32
Illustration 13: Behavior "Nächstes Loch"....33
Illustration 14: Behavior "Kugeln verteilen"....34

5.2 Index of Tables

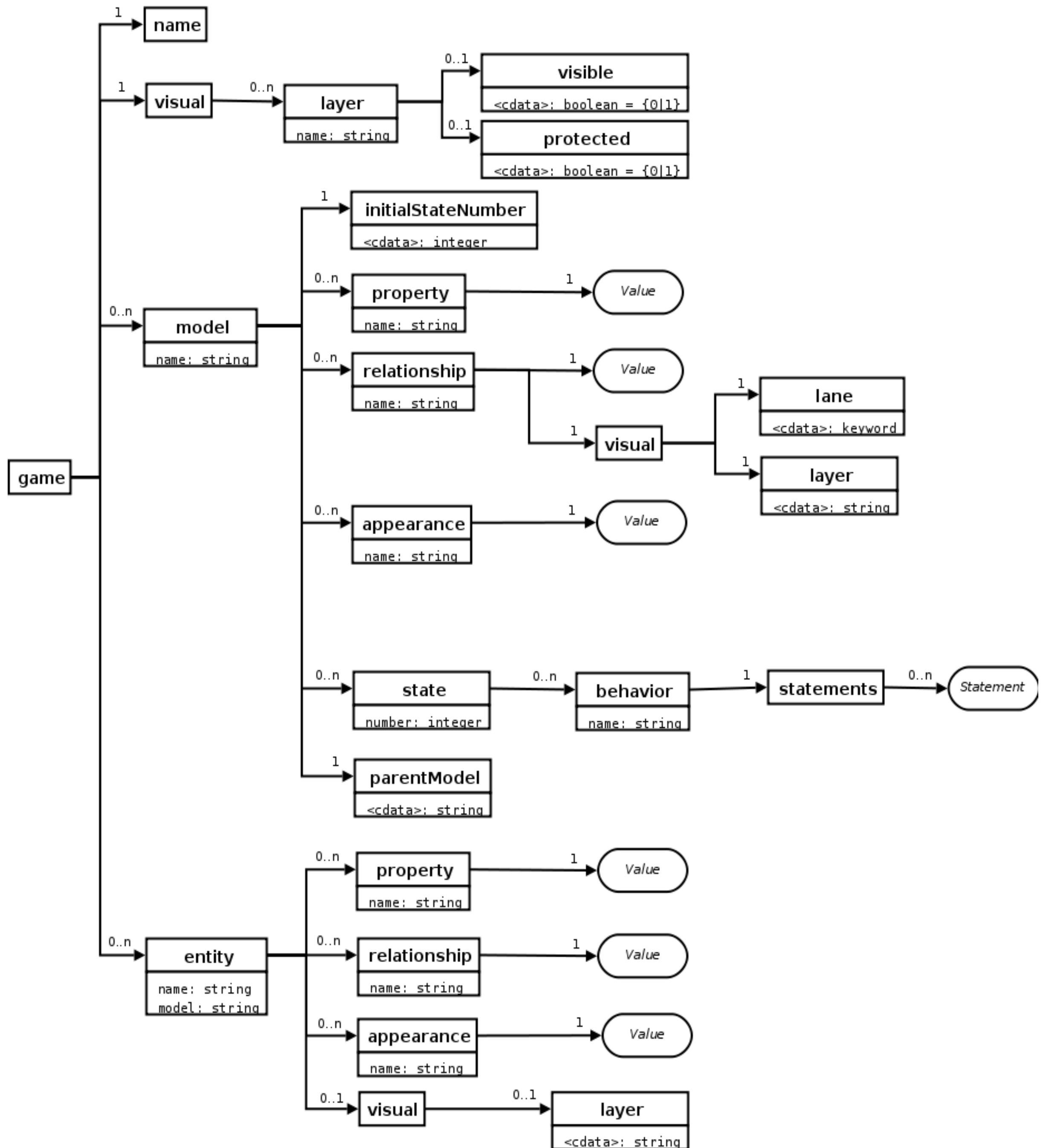
Table 1: List of value types....7
Table 2: List of statements....9
Table 3: Color Code of statements....12

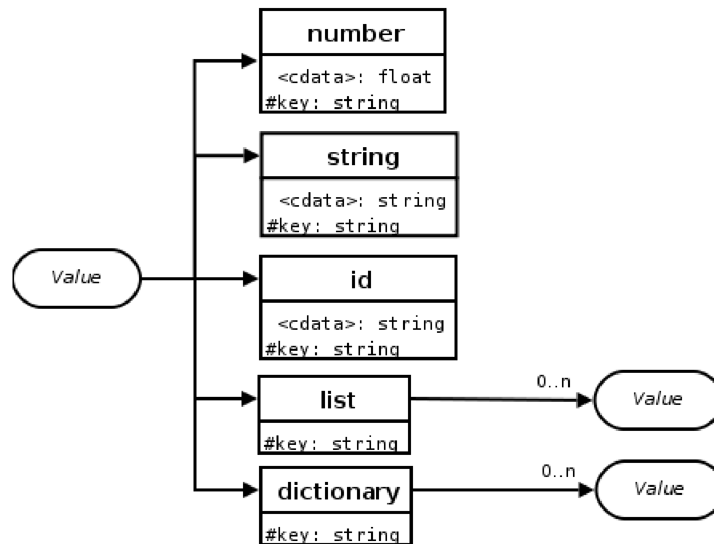
5.3 Code Listings

Code 1: Pseudo-Code "Kugeln hinzufügen"....32
Code 2: Pseudo-Code "Nächstes Loch"....33
Code 3: Pseudo-Code "Kugeln verteilen"....33

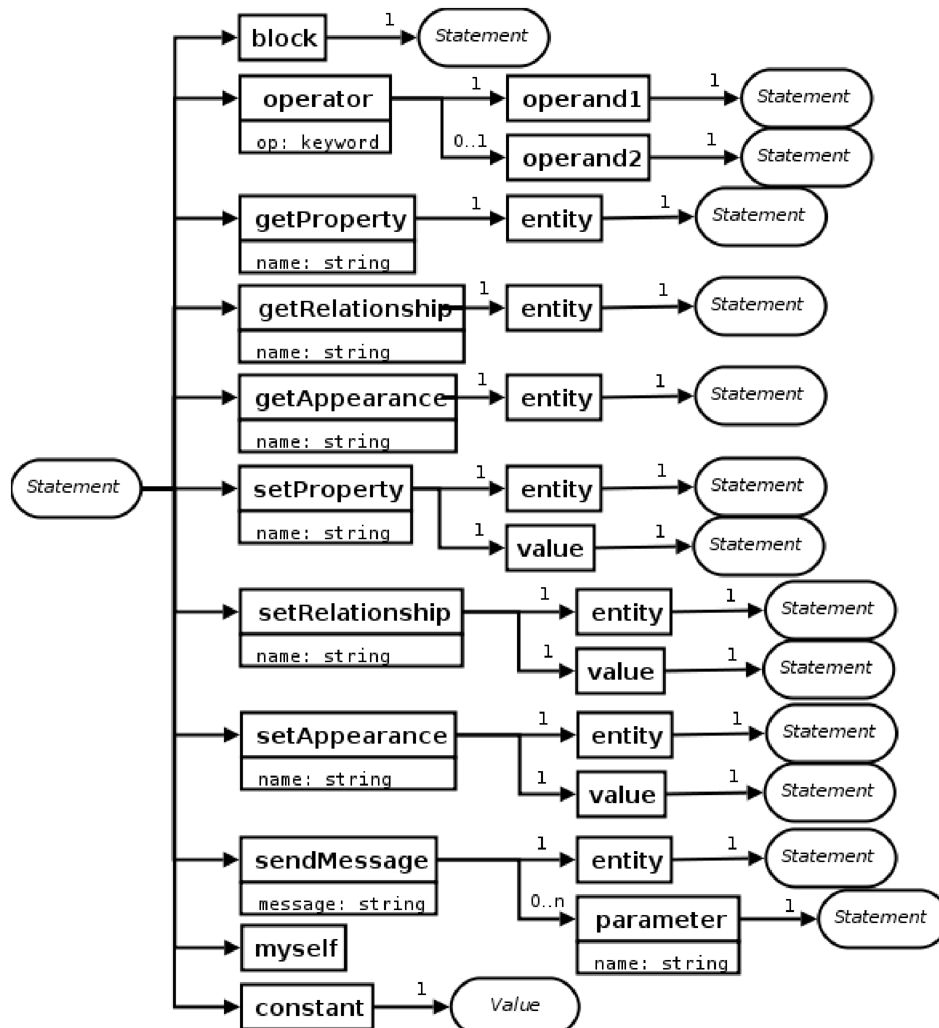
6 Annexes

6.1 XML File Format Schema

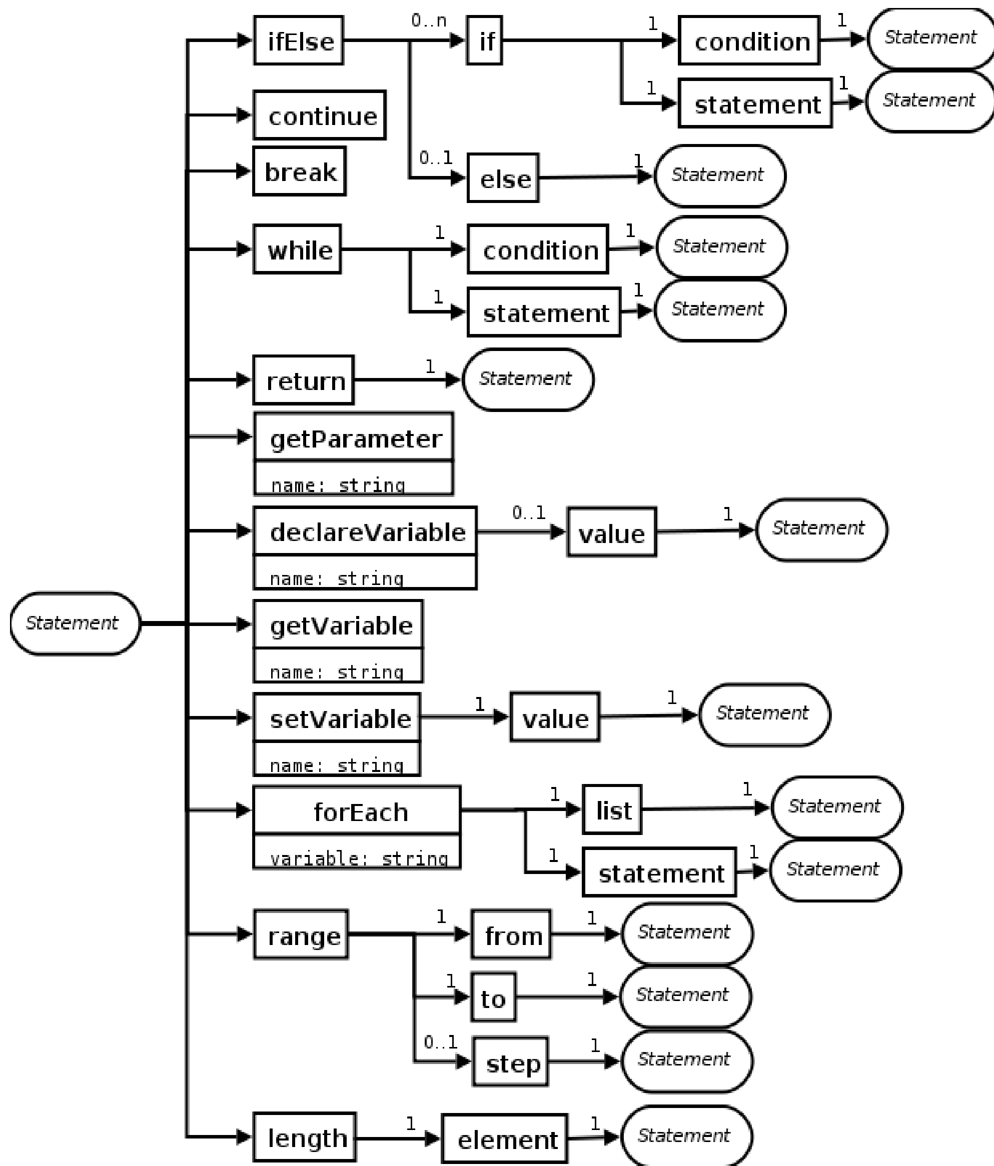


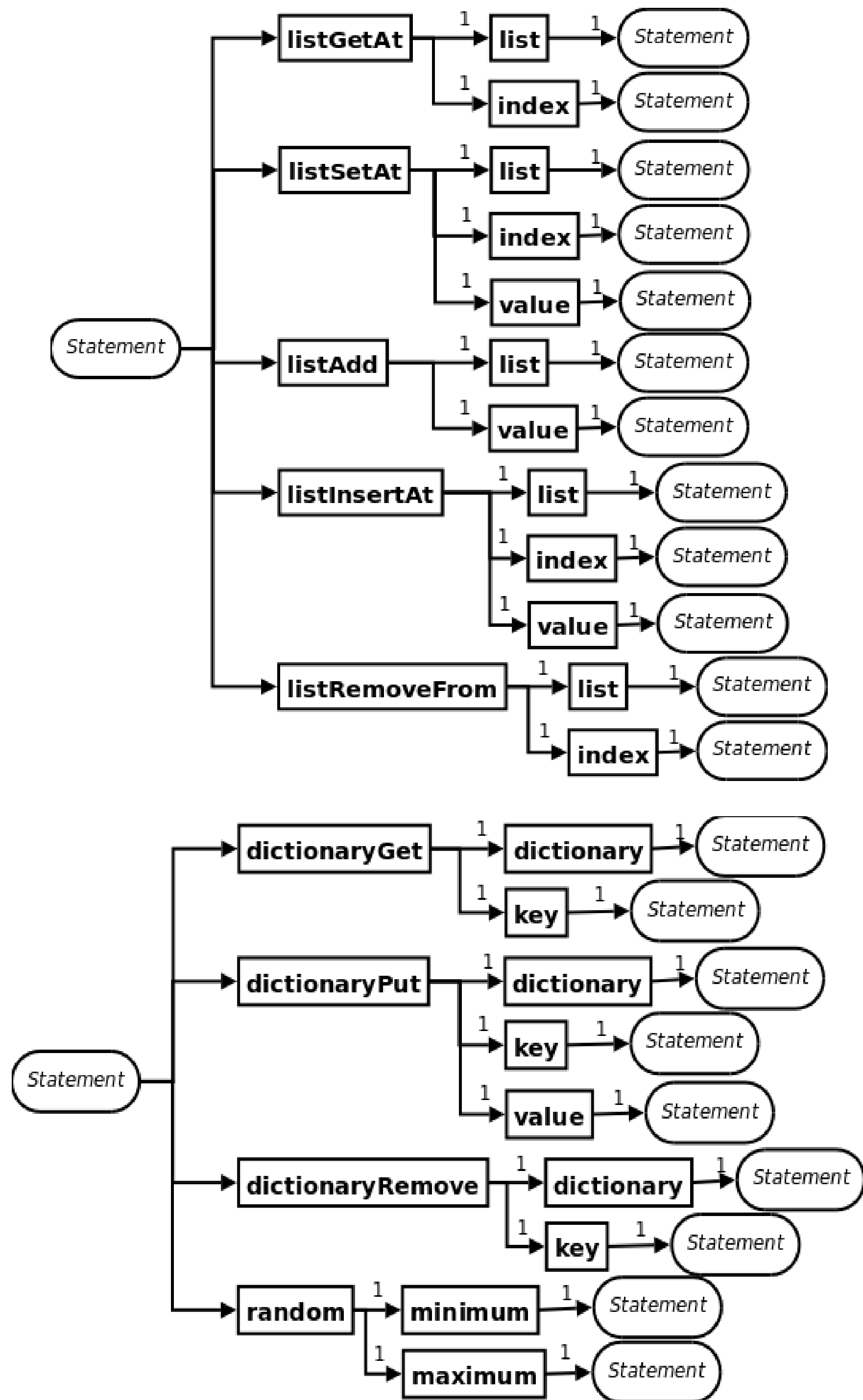


Value
number
<cdata>: float
#key: string
string
<cdata>: string
#key: string
id
<cdata>: string
#key: string
list
#key: string
0..n
Value
dictionary
#key: string
0..n
Value



Statement
block
1
Statement
operator
op: keyword
1
operand1
1
Statement
0..1
operand2
1
Statement
getProperty
name: string
1
entity
1
Statement
getRelationship
name: string
1
entity
1
Statement
getAppearance
name: string
1
entity
1
Statement
setProperty
name: string
1
entity
1
Statement
1
value
1
Statement
setRelationship
name: string
1
entity
1
Statement
1
value
1
Statement
setAppearance
name: string
1
entity
1
Statement
1
value
1
Statement
sendMessage
message: string
1
entity
1
Statement
0..n
parameter
name: string
1
Statement
myself
constant
1
Value


Statement
ifElse
0..n
if
1
condition
1
Statement
1
statement
1
Statement
0..1
else
1
Statement
continue
break
while
1
condition
1
Statement
1
statement
1
Statement
return
1
Statement
getParameter
name: string
declareVariable
name: string
0..1
value
1
Statement
getVariable
name: string
setVariable
name: string
1
value
1
Statement
forEach
variable: string
1
list
1
Statement
1
statement
1
Statement
range
1
from
1
Statement
1
to
1
Statement
0..1
step
1
Statement
length
1
element
1
Statement


Statement
listGetAt
list
index
listSetAt
list
index
value
listAdd
list
value
listInsertAt
list
index
value
listRemoveFrom
list
index
Statement
dictionaryGet
dictionary
key
dictionaryPut
dictionary
key
value
dictionaryRemove
dictionary
key
random
minimum
maximum
1

6.2 Tutorial

This tutorial shows how to implementing a small example game Awele. This is a simple game pitching two players against each other. 12 holes are located in between the players aligned in 2 rows with each row belonging to one of the players. Pebbles are placed in the holes. Each player can then remove pebbles from one hole distributing them across the neighboring holes. To create this simple game the following tasks have to be done:

- Create 2 player and 12 hole entities
- Holes have a pebble count and belong to 1 player
- Holes have a behavior to distribute the pebbles according to the game rule
- Holes have a next and previous hole neighbor

6.2.1 Create a new game

Starting the editor a new game is created. To assign images easily afterwards it's a good idea to save the new game right at the start. For this example we use “awele.gdf.xml” as filename. Now we can start working.

6.2.2 Create models

We need to create two models. The model “Spieler” represents players in the game and the model “Loch” a hole. To create the models use the Game Tree. Right click on “Modelle” and click on “Model hinzufügen”. Name the model “Spieler” and hit “OK”. Now the model “Spieler” is created and visible if the “Modelle” branch is unfolded. Do the same to create a model “Loch”.

For this example game there is no need for a hierarchy of models. For a more complex game though where for example there are different kinds of buildings players can build certain models are specializations of other models. Hence for example a model “Building” would be the generalization of the models “House” or “University” inheriting the properties defined in the parent model and adding their own special features. To achieve this you have to click with the right mouse button on the model you would like to use as the parent of your new model and then select “Model hinzufügen”. The new model is now a specialization of the chosen model instead of being a top level model without a parent. Changing the parent model can be done using the “Model Vater ändern” menu entry while clicking with the right mouse button on the model to re-parent.

6.2.3 Assign properties, relationships and appearances

Unfold the “Modelle” branch in the Game Tree and select the “Spieler” model. Now the Properties Tables below show the definition of this model. For players we have no properties but relationships and appearances. Click on the “Beziehungen” tab (second from the left) to bring up the model relationships. Click with the right mouse button on the

table background and click “Hinzufügen” and “ID” to add a relationship named “Nächster Spieler” which is of typ ID. This relationship links player entities to the next player in turn. Each entity is uniquely identified by an ID hence this type is suitable. We also need a relationship named “Löcher” of type List (click on “List” instead of “ID” in the above example). This relationship links a player entity to its 6 holes. The list is going to contain only IDs which we are going to assigned later on.

Now click on the “Aussehen” tab (third tab from the left) to create appearances the same way as you did with relationships. Here though we can use all types. Create the appearances “Bild X” with type Number (x coordinate of image in the Workspace), “Bild Y” with type Number (y coordinate), “Bild Dateiname” with type String (filename of the image) and “Bild Z-Order” with type Number (used to define drawing order if images do overlap where higher values are drawn above images with lower values). The model “Spieler” has now all properties, relationships and appearances defined.

Now select the “Loch” model from the Game Tree to start working on it. Go to the properties tables (first tab from the left) since this model does have a property to keep track of the number of pebbles in the hole. Create a property named “Anzahl Kugeln” with type Number. Go to the relationships tables and create the relationships “Nächstes Loch” with type ID (next hole), “Vorheriges Loch” with type ID (previous hole) and “Spieler” with type ID (which player the hole belongs to). In the appearances tables create the same appearances as you did for the “Spieler” model. This time we put a default image since all holes have to use a given image by default. For this double click on the empty text right to the “Bild Dateiname” appearance and enter the name of the image file to use. For this example the image is named “loch.png” and is supposed to be located in the same directory as the game file resides in. The filename can be an absolute path or relative to the game file. Hence you can enter simply “loch.png” in this case as the value of the appearance. Now the model “Loch” is also ready and we can move on to create the entities.

6.2.4 Create entities

To create entities there exist two ways. Either click with the right mouse button on the branch “Entitäten” in the Game Tree or right click on the Workspace. In both cases select “Entität hinzufügen” and name the new entity “Spieler 1”. This is going to be the first player. Now select the model this entity uses which is “Spieler”. The new entity is shown under the “Entitäten” branch of the Game Tree (unfold to view them) and in the Workspace using a placeholder image. Drag the image to the top center position. Dragging an entity in the Workspace updates the appropriate appearances of the entity for you. If you want to you can also edit the values manually using the appearances tables for example to align entities precisely. On the top side of the entity a small circles are visible. These represent the relationships defined by the model of the entity. We get back to them later. Now our player needs an

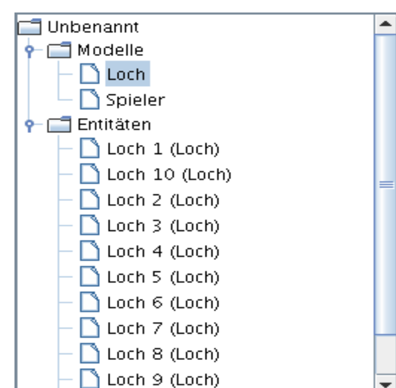


Illustration 9: Awele game tree after adding some entities

image. For this click with the right mouse button on the image of the entity and select “Bild wählen”. The editor shows now the directory the game file is located in allowing to select png image files. Select the “spieler1.png” image. The relative path of the image (relative to the game file) is entered into the appropriate appearance and the entity drawn now using the selected image. Do the same to create a second player named “Spieler 2” located on the bottom center selecting “spieler2.png” as the image to use.

Now we create 12 holes. For this create the entities “Loch #” with # matching 1-12 for the individual holes using the model “Loch”. Since we defined an image already in the model this time the new holes already have the right image from the word go on. Align the holes in two rows with the top row containing the holes (from left to right) 1-2-3-4-5-6 and the bottom row containing the holes (left to right) 12-11-10-9-8-7. The entities are now created and ready to be linked to each other to create the game topology.

6.2.5 Setup layers

Creating the game topology already now would result in a large mess of connection lines in the Workspace. To avoid such this layers can be used to organize entities and relationships. For this use the layers table (fourth-ed tab from the left). To create a layer click the right mouse button on the empty table and select “Ebene hinzufüen”. Name the layer “Spieler”. This layer is therefore going to host all player entities. Create now the additional layers “Nächster Spieler” (for the next player relationship connections), “Spieler -> Löcher” (for the connections from players to their respective holes), “Löcher” (for the holes), “Vorheriges Loch” (for connections from holes to their previous hole in order), “Nächstes Loch” (for connections from holes to their next hole in order) and “Loch -> Spieler” (for the connections from holes to their owning player). Each layer can now be visible/invisible and protected/unprotected. If visible all entities and relationship connections belonging this this layer are visible otherwise invisible. If a layer is protected entities and relationship connections on this layer are prevented from being modified in the Workspace (you can still modify them manually in the properties tables).

Now we can assign the entities and relationships to their respective layers. For this click with the right mouse button on each entity selecting “Ebene wechseln” and place the player entities in the “Spieler” layer and the hole entities in the “Loch” layer. Try turning those layers invisible and the entities in question vanish and reappear. Relationships work a bit different. They are manipulated for the models the entities are using instead of each entity itself. Therefore you need to edit a relationship only once and the changes are valid for all entities based on the model. You can not only place relationships in a layer as with entities but you can also indicate where the relationship slot (the red circle around entities) is located. This is a visual aid only and has no influence on the game later on. To select the placement of a relationship slot click with the right mouse button on the corresponding red circle and select “Ankerpunkt #” where # can be “Oben” (top side), “Unten” (bottom side), “Rechts” (right side) or “Left” (left side) indicating along which side of entities the slot is displayed. Reposition them the way you like and where they are least obstructive. To assign a relationship slot to a layer select “Ebene wechseln” from the pop-up menu and select the matching layer from the list you created earlier. Toggling now the visibility property of these

layers causes slots to vanish and reappear. With those connections also vanish or disappear. Now everything is ready to connect entities.

6.2.6 Update relationships / creating game topology

Entities can be connected using two ways. You can click on an entity or select it from the Game Tree to show the entity relationships. For ID based relationships you can select the entity from the drop down menu right next to the relationship of interest. For lists a click on the value right next to the relationship name brings up a list editing dialog where you can add entities using the “Hinzufügen” button and the “ID” popup menu entry. The same drop down control is used to enter the IDs. Once finished you can leave the dialog using “Bestätigen” or “Verwerfen” to reject the changes. Using the Workspace though the process of linking entities is easier and faster. Here the layers we setup earlier are useful Disable all relationship layers except the “Nächster Spieler” one. We want to connect now players to their following player (in terms of turn taking order). Only the red circles of the “Nächster Spieler” relationship are visible now. Drag the red circle from the first player (“Spieler 1”) to the second player (“Spieler 2”). Now a line appears from the circle to the second player. This link indicates that player one has player two as his next player in order. Do the same to assign player one as the next player of player 2. The two links now cross each other and we can move in circles .

Now disable the layer and enable the “Nächstes Loch” (next hole) layer. We connect the 12 holes in the same way we did with the players. Therefore hole 1 has hole 2 as its next hole and so forth (with hole 12 having hole 1 as next hole). Another circle is formed leading us around all the holes in turn. Do the same now for the “Vorheriges Loch” (previous hole) layer connecting the holes in the opposite order. This is the second loop running around the holes in the opposite order. This way we can now venture from one hole to any other hole. For the mentioned behaviors the second loop is not required but can be useful otherwise. Only thing missing now are the connection between the holes and their respective players. For this enable the “Loch -> Spieler” (hole to player) layer. Now drag the red circle from the top row holes to the player 1 assigning this player as the owner of the respective hole. Do the same for the bottom row holes using the second player. Now we can do the final connections telling each player what holes they own. Do this using the “Spieler -> Loch” (player to hole) layer. Up to now we worked with ID based relationships where you can assign one entity for each slot. Now the relationship in question is a list instead hence we can assign multiple holes to one player. For this drag the red circle one by one to each of the holes belonging to the player. Once done 6 connections run from the relationship slot to each of the holes belonging to the player. Do so also for the second player and with this the topology is fully set up.

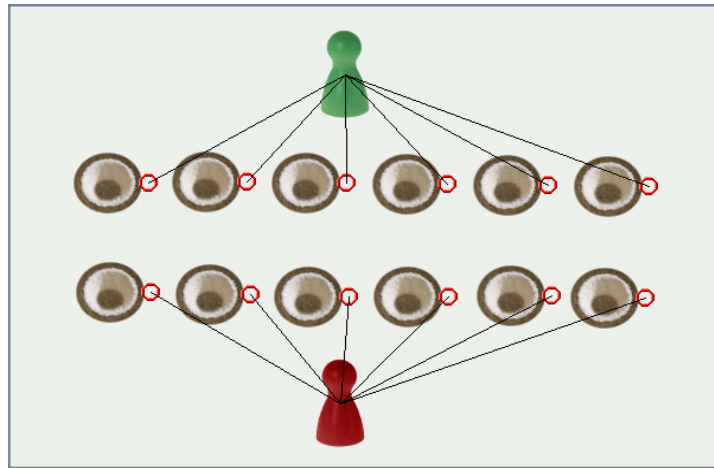


Illustration 10: Workspace with one relationship layer visible

If we would have used a dictionary instead of a list creating connections would ask for a key to identify the connection. Otherwise creating connections works the same. In case of a mistake connections can be broken by clicking with the right mouse button on the connection and selecting “Verbindung lösen” to break it.

In addition to the context sensitive menu clicking with the left mouse button on an entity selects the given entity in the Game Tree and Property Tables. Holding the shift key while clicking selects the model of the entity.

Now that the topology is all set up we can create the behaviors.

6.2.7 Creating states and behaviors

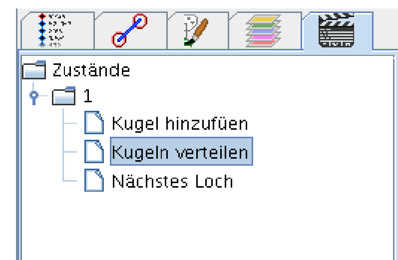


Illustration 11: Hole behaviors

For this simple game one state is enough. Select first the “Loch” model and go to the states tree (tab on the far right). Click with the right mouse button on the “Zustände” branch and select “Zustand hinzufügen”. Enter 1 as the number of the state. Now we can add behaviors to this state. Do so by clicking with the right mouse button on the create state and select “Verhalten hinzufügen”. Enter as the name “Kugel hinzufügen” which is going to simply add 1 to the number of pebbles in this hole. Once create the new behavior can be selected which displays the statements of the behavior in the Behavior Panel. Each statement is represented with a panel. Nested statements result in nested panels. By default the behavior has a block statement assigned. This executes a list of statements in the order they are defined. In general you can click with the right mouse button on any panel to bring up a pop-up menu containing actions suitable for the chosen statement. For block statements in addition new statements can be inserted before or after any given statement. We would like to create the statements for the following pseudo-code:

```
self."Anzahl Kugeln" = self."Anzahl Kugeln" + 1
```

Code 1: Pseudo-Code "Kugeln hinzufügen"

Click with the right mouse button on the block statement and select "Anweisung hinzufügen" (add statement) and then "Eigenschaft ändern" (change property). See the list of statements in 2.2 Language (DSL) to learn about the possible statements to use. The change property statement changes the value of the property with a given name of a chosen entity. All statement panels show the structure of the statement in a row by row fashion. Here you can enter the property name, the entity and the value. Both the entity and the value are statement panels themselves. Use the pop-up menu to change the statement in one of those two slots. The gray statement indicates no statement and acts as a placeholder. Enter for the property name "Anzahl Kugeln" which corresponds with the property we want to modify. The entity is by default set to "Aktuelle Entität" (current entity) which represents the entity the behavior runs on. This is already what we need and what is needed most of the time. For the value we need an operator which adds one to the number of pebbles in the hole. Click with the right mouse button on the placeholder statement and select "Anweisung ersetzen mit" (replace statement with) and then "Operator". This replaces the empty statement with an operator statement. This statement in turn contains an operator drop down box and two operands. Select "Addieren" (add) as the operator which adds the result of the second operand to the one of the first. Use the replace statement action to replace the statement of the first operand with a "Eigenschaft lesen" (read property) statement with the name "Anzahl Kugeln". This reads the current value of the property with the given name. Replace the second operand with a "Konstante" (constant) statement. This statement delivers always the same constant value which can be any value one can put into properties using the property table. By default a number value is used. Change the value into 1. If you want to change the value into a different type you can click with the right mouse button on the constant statement panel, select "Wert ersetzen" (replace value) and select the type of the value you need. You should have now the same result as shown in the screen-shot below.

Illustration 12: Behavior "Kugel hinzufügen"

This is the visual representation of the pseudo-code Code 1. To increase readability the individual statement panels can be collapsed and expanded clicking with the left mouse button on the panel title. For an explanation of the color code see 3.2 Interface Components.

Copying and pasting statements is supported too using the context specific pop-up menu. All actions can be undone. Values of constants can be copied too. Moving statements up or down in a statement block is not support directly using a menu action but can easily be performed using a copy followed by a delete and a paste. A direct moving action could be provided as an extension.

To finish the example game two more behaviors are required which can be created in a similar way. One is “Nächstes Loch” (next hole) which retrieves the next hole assigned to the given hole. The pseudo code looks like this:

```
return self.”Nächstes Loch”
```

Code 2: Pseudo-Code "Nächstes Loch"

Here we need a return statement. Add one using “Anweisung hinzufügen” and “Verhalten beenden” (end behavior). This statement takes an optional value. If no value is specified the behavior ends after this statement without returning a value. Otherwise the given value is returned. In this case we simply return the value of the relationship “Nächstes Loch”. This works similar to previous example just with “Beziehung lesen” instead of “Eigenschaft lesen”. The result should look like this:

Illustration 13: Behavior "Nächstes Loch"

The last behavior required to play the game is the one applying the important game rule. This rule states that a player picks up all pebbles in one hole (from one those belong to him) and distributes them to the neighboring holes one pebble at the time in the order of the holes (along the “Nächstes Loch” loop). This rule can be represented in pseudo-code like this:

```
var Loch = self.”Nächstes Loch”
while Loch != self and self.”Anzahl Kugeln” > 0 do
      Loch.”Kugel hinzufügen”
      self.”Anzahl Kugeln” = self.”Anzahl Kugeln” - 1
      Loch = Loch.”Nächstes Loch”
end
```

Code 3: Pseudo-Code "Kugeln verteilen"

This example contains now a loop, a local variable and some send messages. The behavior is named “Kugeln verteilen” (distribute pebbles). A local variable can be declared using “Variable definieren” from the add statement menu. The name of the variable is allowed to coincident with existing property, relationship or appearance names since explicit statements are used to access the individual elements. Upon declaring a variable an initial value can be specified which in this case is the value stored in the “Nächstes Loch” relationship. Add next a “Schleife” (loop) statement which is a conditional loop. The

condition requires a nested boolean comparison. For this add an operator statement with a “Logisches Und” (logical and) operator and the actual two operator statements as the first and second operand. The while statement expects a boolean result as condition. The editor does not check for this to be true. The body of the while statement is by default a block. If we need only one statement we could get away without a block statement. Add a “Nachricht senden” (send message) statement to the while body. This statement tries to execute a behavior with the given name on the specified entity. Optionally a list of named parameters can be specified. Here we do not need any parameters. If they would be required they can be added by clicking with the right mouse button on the send message panel selecting “Parameter einfügen”. The parameter name is send along with the message as well as their actual value. Replace the entity with a “Variable lesen” (read variable) changing the name to “Loch” to get the value from the local variable. The next statement is a “Eigenschaft ändern” (change property) with a nested operator to subtract 1 from the number of pebbles in the hole. The last statement sends another message “Nächstes Loch” asking for the next hole of the current hole. This loop continues until we arrive at the same hole we started with (in which case the remaining pebbles stay in the hole) or we run out of pebbles. If all has been setup properly the behavior should look like this:

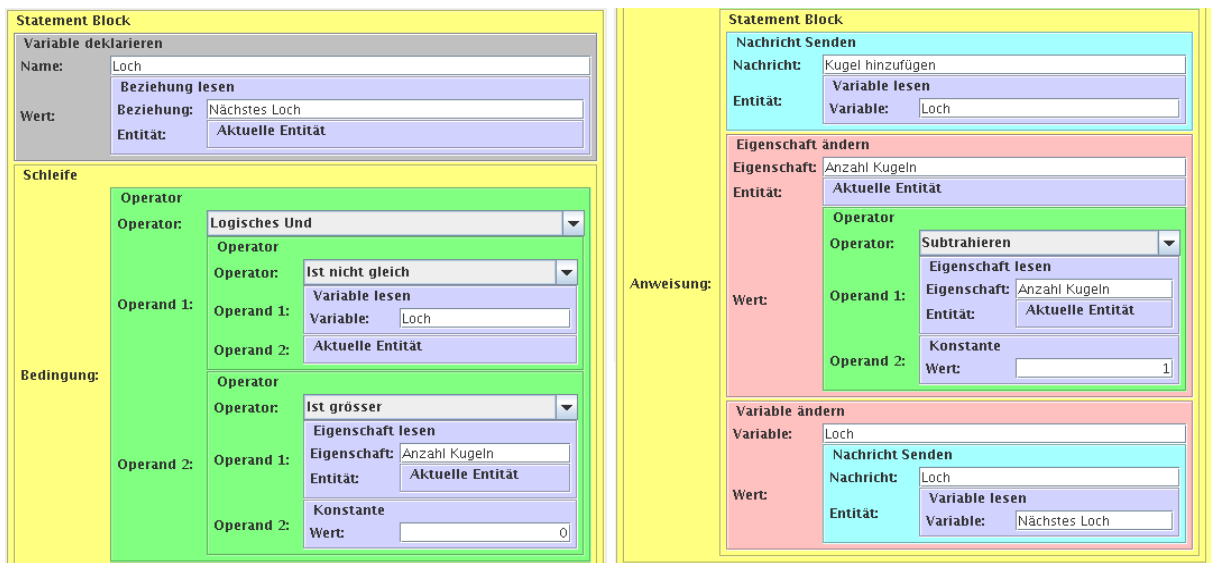


Illustration 14: Behavior "Kugeln verteilen"

With this last behavior the simple game is done. We have now defined models to serve as the base for entities. We added various entities for players and holes. We defined the visual appearance of the entities and arranged them on the workspace. We set up the connections between entities using layers to keep the workspace clean. We added various behaviors representing the actions players can do in a game.

The editor follows in general the principle of the least surprise. If you want to apply a change to an element in your game simply click with the right mouse button on the element and a context specific menu pops up with actions related to the element. Any action can be undone so don't be afraid to mess with the game.

6.3 Developer Information

There are a couple of points to consider for people interested in working with the code side of the editor. Since the entire system is rather complex there are a lot of dependencies as shown by Illustration 8. The comments inside the classes should be informative by themselves but some parts are too complicated to explain in comments alone. This section contains some pointers on successfully changing these parts.

6.3.1 Tag Parser

For parsing the XML file the Java internal XML parser is used. The tree structure of the game definition is produced using a stack of tag parsers. This works by pushing objects derived from the TagParser class onto a stack. The events from the Java XML parser are send to the top most tag parser on this stack. Care has to be taken to push and pop the tag parser properly as otherwise it is tricky to figure out why parsing suddenly fails. Each tag parser has to pop itself from the stack during the endTag method call. Due to the way the system is implemented parsing of tags is a bit particular. For example if a tag requires a child tag X storing a string in its CDATA you have to first create a buffer during the beginTag call to accumulate the string using a TPString tag parser. Once endTag is called those buffers can be examined and the changes applied. To help using this system various basic tag parsers are provided.

- TPNull. Consumes all children tags disregarding their content. Kind of a NOP
- TPString. Consumes all CDATA concatenating them into a StringBuffer.
- TPStatement. Parses one statement tag of any type and stores the assembled Statement object into the provided StatementBuffer object which simply acts as a container to carry one Statement object.

For parsing values the system works slightly different. The value of the required type is created and added to the right place. Then a TPValue* tag parser is created with the created value object. This fills in then the informations found in the child tag into the given value object.

For statements this system works slightly different. A couple of tags parse multiple statements or parse a single statement directly. To support all this without lots of duplicate code a class TPStatements has been created which allows to parse multiple statement tags. The hook method addStament is used to add a finished tag. Important to know here is only that using this system adding a new statement parser only requires to add the appropriate branch to the if-else construction in TPStatements.beginTag . This covers the reading of a new statement tag. For writing it is only required to add a new parseXYZ method to the LoadSaveGDF class and adding a new if-else branch to the LoadSaveGDF.saveStatement method. This way saving of a new statement is also done.

6.3.2 Adding a new statement

If a new statement is required a couple of changes have to be done. This section gives a

brief list of the required changes.

- Add a new method visitXYZ to StatementVisitor for visiting.
- Add an empty implementation visitXYZ to DefaultStatementVisitor.
- Modify the IdentifyStatementVisitor. This visitor simply assigns the individual statement types a unique number. This has been done outside the statements class hierarchy to maintain a clean visiting pattern. Add a new static field for the new statement with the next number. Add methods isXYZ, castToXYZ and visitXYZ. isXYZ determines if the visited statement is of a given type. castToXYZ casts the visited statement to a given statement subclass. This is a sort of safe casting since calling this method on a mismatching statement throws an exception. visitXYZ eventually is responsible to store the statement as well as the type. Once done the statement is ready in the Game Model part.
- Add a new tag parser TPStatementXYZ. Just copy an existing TPStatement* tag parser as the structure should be self explanatory including the informations from the previous section.
- Modify the TPStatements and LoadSaveGDF classes as mentioned in the previous sections. Once done the statement is ready in the Load/Save part.
- Add a PBStatementXYZ class to the behavior panel. Again just copy another PBStatement* class as the structure should be easy to get. See the Behavior Panel section in 3.3.2 for mode details on the workings of such panels.
- Add visitXYZ to PBCreateStatement. This visitor creates a PBStatement* class matching the type of the visited statement.
- In the getMenuStatementList method in the PBStatement class add a new line with the IdentifyStatementVisitor identifier of your new statement. This method is used by all the statement panels to display the appropriate menu entries for inserting new child statements.
- In the ActionStatementReplace class in the method getStatementName add a new if branch for your new statement. This name is displayed in the menu.
- In the same class in the method createStatement add a new if branch to create the statement. In here you have also to add default values to statements if appropriate. For example the entity of the StatementGetProperty is set to StatementMyself since this is most of the time what the user is looking for creating such a statement.
- To finish adding the statement a couple of house keeping classes are required especially inside the Controller part. Create copies of the classes ActionS*{Paste|Remove|Replace}Statement which are used to manipulate statements in the statement panel of the new statement. Eventually UndoS* classes contain the actual controller actions. One for each non-statement parameter you can change in the statement is enough as well as one for all statement based parameters. The last one can be split up into multiple classes but this keeps the number of classes lower. Pay attention to the import declarations in these classes. There are a couple of them required but once set up everything should work well.

After having fought yourself through this list the new statement should be fully working in the editor.

6.4 CD-Rom Content

The accompanying CD-Rom contains the editor jar including the source files, this document and the example Awele game as well as a testing game for showing better the hierarchies of models.

/distribution/visual_editor.jar => Jar of the visual editor

/sources/src => Source code of the editor

/sources/share => Shared resources to be included in the jar

/sources/lib => Required libraries to be included in the jar

/samples => Sample Awele game and testing game including image files

/document => Location of this document