AMBIENT ASSISTED LIVING JOINT PROGRAME

AAL-2013-6-039

SeniorLudens

Serious Games development platform for older workforce training and intergenerational knowledge transference

D1.3
Scenario, Task and Game Files descriptors

Workpackage	WP1 - System functional and technical requirements
Lead beneficiary	CBIM
Editor(s)	Dani Tost- CREB-UPC Ariel von Barnekow – CREB-UPC Rodolfo Nuñez – CREB-UPC Salvador Aguilar - INDRA
Contributor(s)	Dani Tost – CREB-UPC Ariel von Barnekow – CREB-UPC Rodolfo Nuñez – CREB-UPC
Reviewer(s)	Gary Honegger - YR
Release Date	08/2014
Version	V1.1
Circulation	Project Partners, AAL Control Management Unit, and National Funding Agencies.

Table of Contents

Abstract

This document contains a description of Senior Ludens Game Kit: its structure, classes and the file descriptor files that allow exporting its definitions to other components of SeniorLudens.

1- SeniorLudens Game Kit

1.1- Definition

The SeniorLudens Game Kit (from now on SLGK) is two-fold: on the one hand, it provides an abstract definition of the virtual environments and the training activities that can be performed in them, and on the other hand, it implements these definitions on top of a game engine in order to allow the creation of training games.

Figure 1 illustrates the relationship of SLGK with the others components of SeniorLudens. The creation of a training program involves 6 steps (first row of the figure): the creation of the virtual environment called *world* that includes all the objects and actions available; the configuration of scenarios using a particular set of objects of the world; the definition of the training tasks and their testing through a simulation tool; the training itself and the analysis of results. Each of these steps requires specific components of SeniorLudens (second row of the figure): the *Scenario Editor*, the *Task Editor*, the *Simulator*, the *Trainer* and the *Analysis Tool*. The third row of the figure shows the different roles of users using these components.

Three components of the system are games directly built on top of SLGK:

- The *Scenario Editor,* which is used by Scenario Designers to configure virtual environments by putting objects at precise locations;
- The *Simulator* that implements a task designed by Game Designers using the Task Editor component;
- The *Trainer*, which is the actual training game composed of different tasks.

In addition, two components: the *Task Editor* and the *Training Program Designer* use the definitions of the *world* and its components provided by the SLGK. These definitions are exported from the SLGK as XML file to feed SeniorLudens database in order to be used by the *Task Editor* and the *Training Program Designer*.

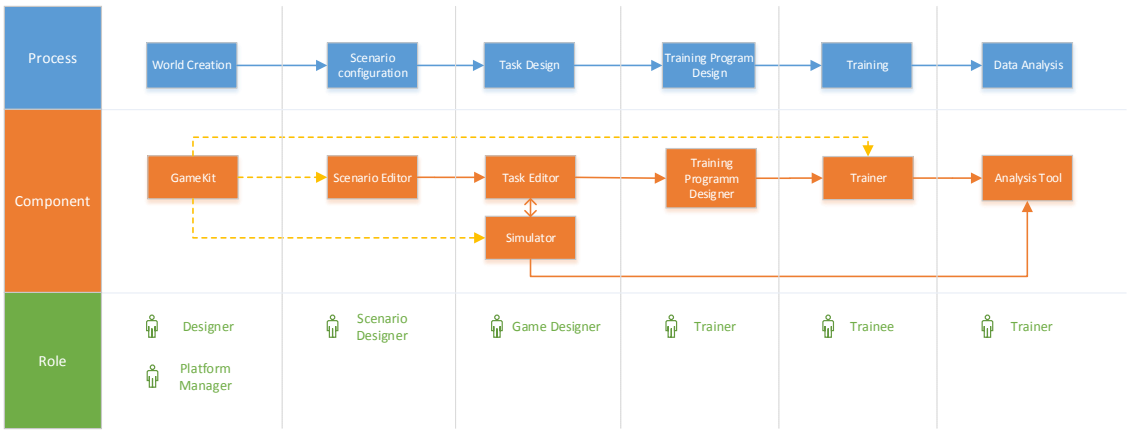

Figure 1: Senior Ludens components

1.2- Structure

The SeniorLudens Game Kit provides an abstract layer of definition of the components of a game and an interface layer on top of the game engine Unity. This architecture presents two main advantages: first, the abstract layer eases the definition of games because it does not require technical knowledge of the underlying game engine, and second, since the abstract layer is independent from the game engine, if it was necessary to change the game engine, only the interface layer would need to be reimplemented.

This architecture is shown in Figure 2: on the bottom layer the graphical API GL; above, the game engine (Unity 3D in the case of the current implementation or another); above, SLGK composed of two layers: the abstract model (*Senior Ludens Core*) and the interface to unity (*Senior Ludens Interface to Unity*); finally on top the games built on SLGK, in particular, the Scenario editor, the Simulator (test tasks) and the trainer (the training games themselves).

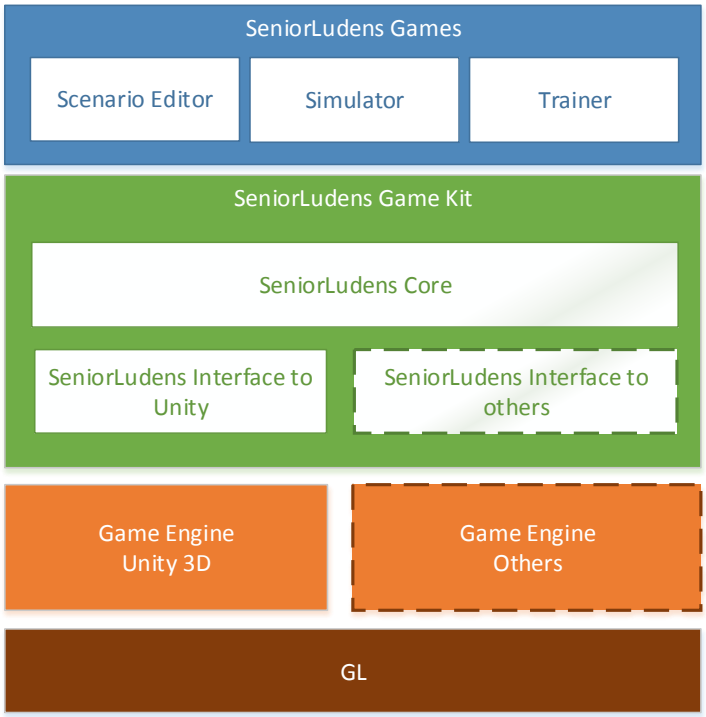

Figure 2: Structure of SeniorLudens Game Kit

In addition, the SeniorLudens Core is defined at two levels: the abstract definition classes and the specific instances of elements (scenes, objects, actions) that compose a game in run-time.

2- Components

2.1- World, Scenarios and Configurations

The aim of SeniorLudens is to be usable in a diversity of training applications. Every training case has associated a *world* that defines the *objects*, *actions* and *characters* needed for the type of training that must be done in the world. For instance, three SeniorLudens pilots address the traditional fabrication of cheeses, the rehabilitation of patients and a metaphor of project management through gardening activities. The world associated with the fabrication of cheese requires the definition of the character cheese maker that will be driven by the trainee. It needs objects such as oven, pans, spoons and actions such as moving, whipping and cooking that involve characters and actions. The world associated with rehabilitation requires various characters: the rehabilitator trainee and patients. It also needs objects such as the patient heart rate, a bicycle, a treadmill, a rehabilitation program and actions such as check heart rate or program an exercise.

Finally, for the project management pilot, the world will represent plots, vegetables, gardening tools such as pruning scissors and ravels. The characters will be the gardener, the warehouse manager, and customers. Finally, required actions will be planting, watering, cutting among others.

A scenario is a concrete virtual environment based on the definition of a world. It represents a virtual 3D space composed of objects characters and actions defined in the corresponding world. Many different scenarios can be built on the basis on the same world definition, therefore, the relation *world:scenario* is 1:n.

Some training tasks may require different training spaces. For instance, in the rehabilitation use case, it may be necessary to train in the reception room, in a cardio vascular training room and in a medical consultancy room. To support these tasks, either a unique scenario made of different spaces can be built, or each space can be built as a separate scenario. The latter approach presents the advantage of easing the re-use of scenarios, and from the point of view of training, it avoids having to navigate from one space to the other. In order to support the second approach, SeniorLudens Game Kit allows linking a training task to one or more scenarios. Scenarios that are connected to others must define actions of transition from one scenario to the other.

The ***¡Error! No se encuentra el origen de la referencia.*** describes the initial content of the corresponding virtual space: the objects that are inside, their location and state. A part of the scenario remains usually invariable, typically structural elements such as walls, doors, windows, closets and in general heavy furniture. Another part of the scenario can vary in its contents as well as in the position and orientation of the objects. For instance, a kitchen scenario can have different initial contents of the fridge object. This variable part of the scenario is called **¡Error! No se encuentra el origen de la referencia.**. For each scenario, various configurations can exist. These different configurations are defined in the Scenario Editor. Thus, the **¡Error! No se encuentra el origen de la referencia.** *Class* is the definition of the location (position and orientation) and state of the variable part of objects and characters of the corresponding scenarios. Figure 3 illustrates the definition of a *world*, a *scenario* and a *configuration*.

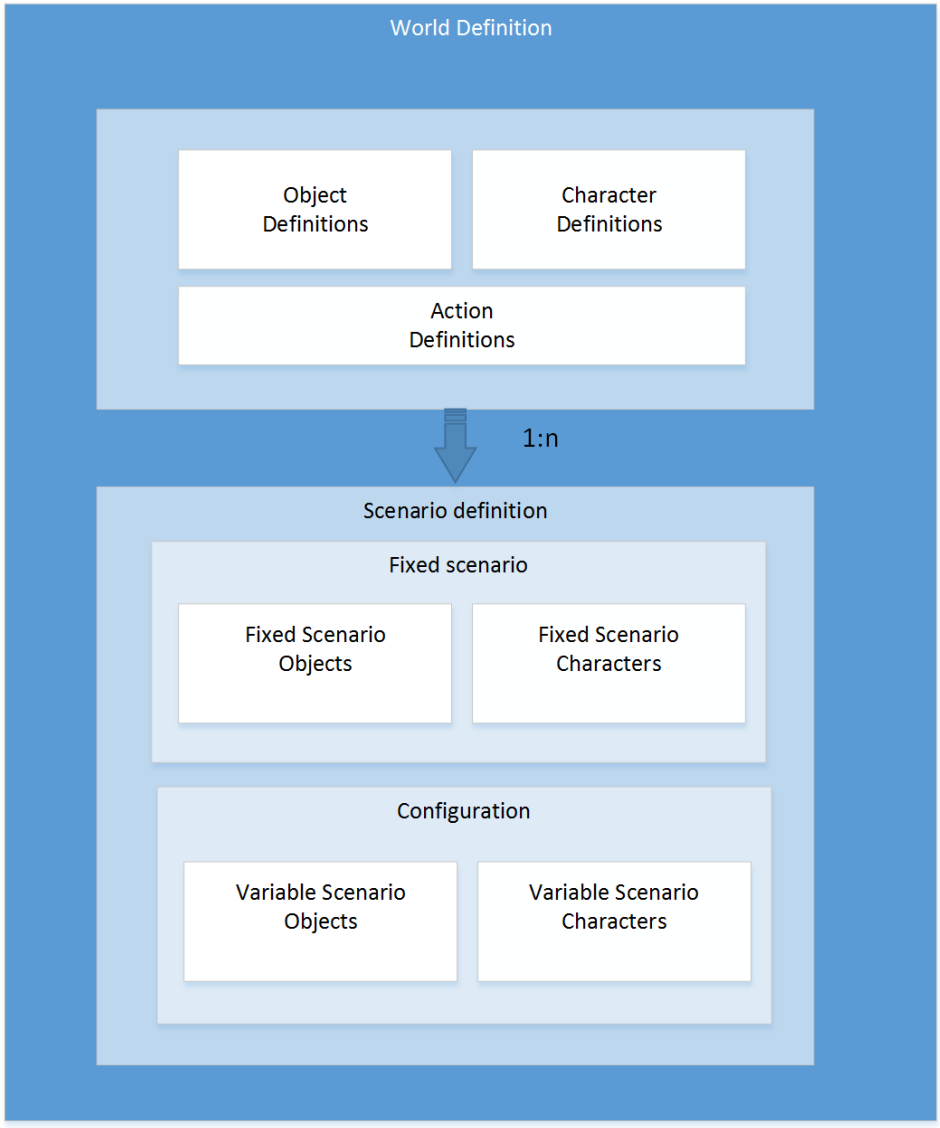

Figure 3: World, scenario and configuration

Finally, the *Scene Instance* is defined in run-time of a game. It is the object representing the actual *scenario* created with a specific *configuration*. It is composed of the set of object and character instances defined in the scenario and the set of objects and characters instances defined in the corresponding configuration (see Figure 4).

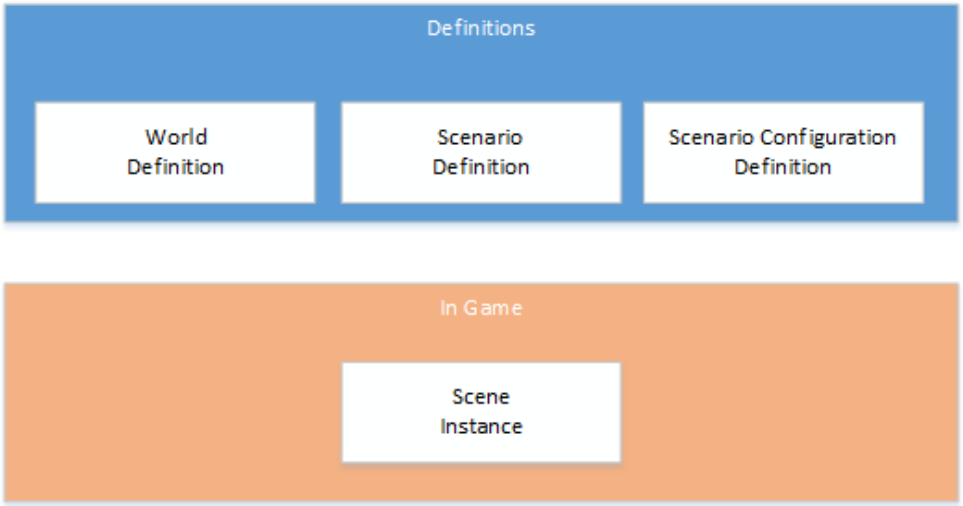

Figure 4: World definition and Scene Instance

2.2- Objects

Objects are entities of the *World* that participate in *Actions*. The *Object Definition Class* defines the characteristics of the objects and the actions in which they can participate. An *Object Instance* is a specific realization of a particular object definition. Finally, *Visual Objects* link objects to their graphical models in the corresponding game engine (Unity 3D in the current implementation). Not all objects have visual representations, some are purely abstract concept. In addition, objects may have more than a visual representation (see below). Therefore, the relationship *object:visual object* at the definition level as well as at the in-game instances level is *1:n*.

Figure 5 shows the relationship between *object definition*, *object instances* and *visual objects*.

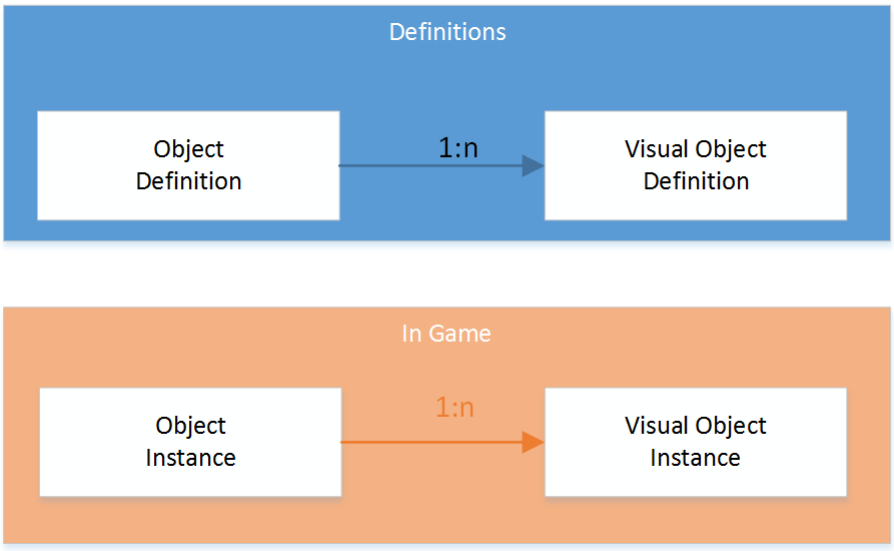

Figure 5: Object and Visual Object definition and instances

2.2.1- Objects Definition

Objects have particular attributes that can be queried, for instance the value of the temperature, or the text written on a blackboard. They are also labelled according to different classification criteria (vegetable, tool, fresh…). These attributes and labels are defined in the *Object Definition Class.*

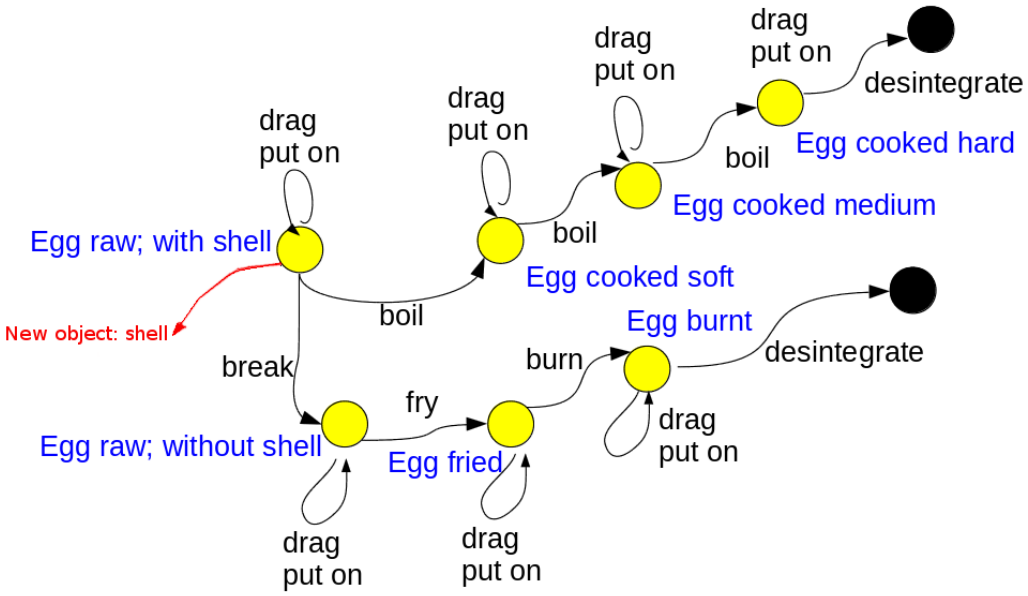

Figure 6: An example of an object's state diagram

Objects can be in different discrete states: for instance a door can be opened or closed, an egg can be raw or cooked. The *Object Definition* contains a state diagram graph in which nodes are states and edges are actions. Transformation actions that change the state of an object correspond to edges between states of this object, and actions that do not change states are loops. Object instances have a specific state at each moment of the game. Thus, if a transformation action lasts more than an iteration of the game loop, the object instance being transformed stays in the original state until the transformation has occurred.

Figure 6 illustrates the definition of an object. The object egg has 7 states: raw with shell, raw without shell, fried (without shell), burnt (without shell), cooked soft (with shell), cooked medium (with shell) and cooked hard (with shell). In every state, two self-edge actions are available: to drag with and to put on. The instruments needed to drag the egg in its different states are different, though. For instance, to drag the egg in the fried state, the instrument needed is a skimmer, whereas, a raw egg with shell can be dragged without any instrument. The action “to boil” changes the state of the egg: after a given time since the action has been launched, the egg passes from raw to soft cooked; after another while to medium; then to hard.

Objects have particular attributes, for instance the value of the temperature, or the text written on a blackboard. These attributes can be related to attribute values of other objects so that their value varies accordingly. For instance, the object thermometer has the attribute temperature value. This attribute can be connected to the attribute temperature of the oven. Thus, if the temperature of the oven increases, the temperature of the thermometer will also increase.

Finally, objects can be purely abstract concepts (time, temperature) or they can have a visual representation. In the latter case, the visual representations are classified by styles. An object can have one or more styles and each of its state is associated to as many visual representations as object's styles. These visual representations can be shared by different states. Table 1 illustrates the relationship between objects and their visual representation for the egg object. The states *raw with shell*, *soft cooked*, *medium cooked* and *hard cooked* have the same visual object. The object *egg* has two different representations styles.

States	Style 1	Style 2
Raw egg with shell Soft cooked egg Medium cooked egg Hard cooked egg		
Raw egg without shell		
Fried egg		
Burnt egg		

Table 1: An example of the relationship between objects and visual objects

Figure 7**¡Error! No se encuentra el origen de la referencia.** summarizes the main information associated to an *object definition class*.

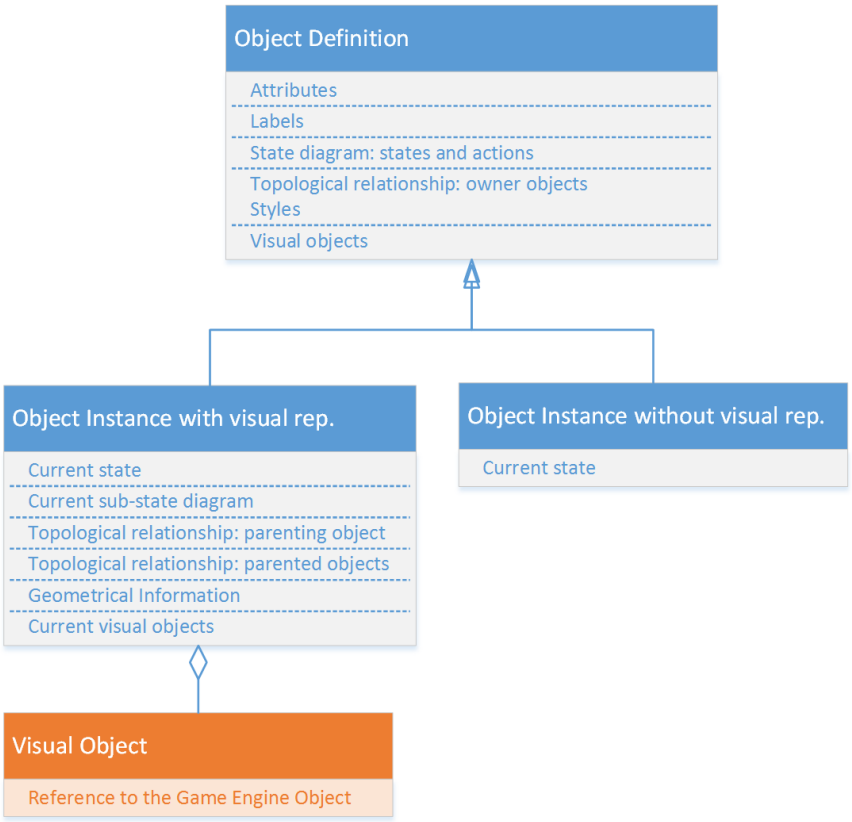

Figure 7: Simplified class diagram of the Object Definition Class, the Object Instance and the Visual Object.

2.2.2- Objects Instances

Object instances follow the pattern of the corresponding object definition. At any time, they are at a specific state of the state diagram. They are associated to a sub-state diagram of the corresponding object definition that contains only edges corresponding to actions actually available on the object at a given moment of the game. This sub-state diagram is needed to restrict some actions in order to ease the realization of tasks and in order to avoid ambiguities in the interpretation of the actions to be done. This is illustrated in Figure 8: the bottle object has been defined as having two possible states and three actions. In a particular scenario, a bottle can be instantiated with this complete behaviour (top right). In another (middle right), the bottle has only one possible state and one action. In this scenario, it will not be possible to do anything more with the bottle than to pick it. In the third case, the bottle cannot be picked but opened and closed. Since the behaviour of the objects can change during a game, the sub-state diagram of an object instance is modifiable dynamically during the game, by copying edges and states of the reference state diagram of the object definition or removing them.

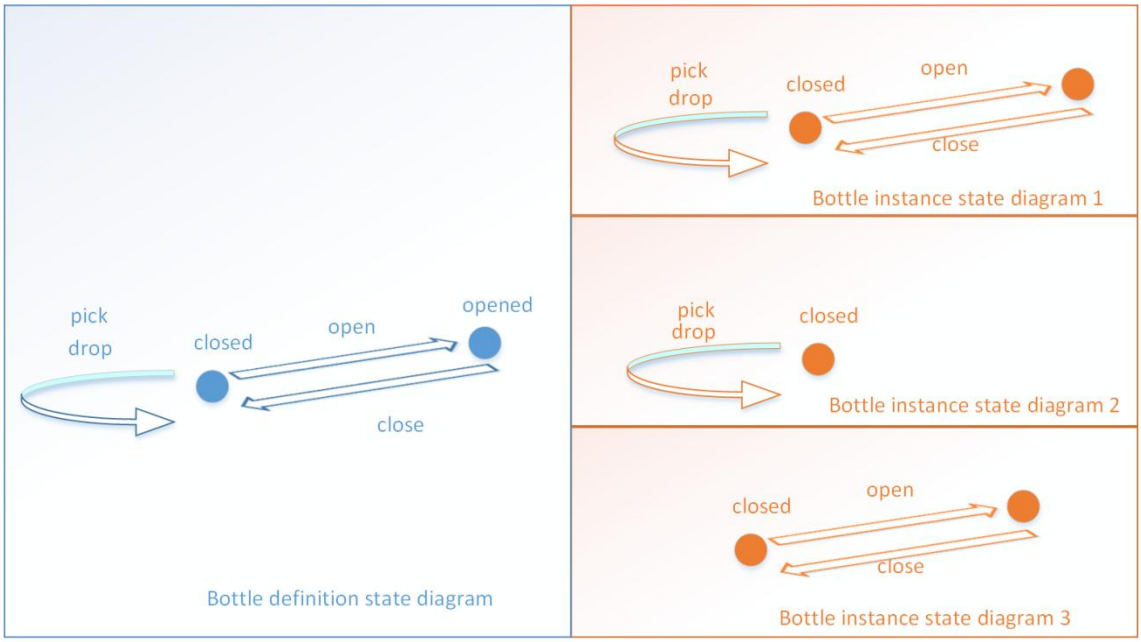

Figure 8: Complete state diagram of the object definition and three possible state diagrams of object instances

If they are visual, they are linked to as many visual representations as visual styles the object has. Moreover, they have geometrical and topological properties that they retrieved from the corresponding visual object, for instance, the position of the centre of the object in the 3D. The position is needed to evaluate the fulfilment of actions such as “put the object A at distance XX of the object B” or “stop [the trainee’s avatar] at a distance X from the object Y”. Other geometrical properties could be retrieved from the visual object information and stored as object instance attributes, if needed.

In addition, object instances store information of all the instances of objects that are on top of them or inside them *(parented objects).* They also store information of the object instances on top or inside which they are located *(parenting objects).* These relationships allow querying for the position of objects instances in relation to others, which is need to accomplish or validate actions such as “to put on”, “to put inside”, “to hang on”.

2.2.3- Visual Objects

Visual objects are implementations of visual representations. In SeniorLudens GameKit, they are implemented in the interface layer to Unity. They are a link to the corresponding object of the Game Engine, specifically, in the current implementation to a Unity Object.

2.3- Actions

Actions are events that can occur in a scenario yielding to changes in the objects: change of attribute values, state transformations, geometrical transformations, etc. At the abstract level, action definitions express precisely the conditions needed for an action to be executed: who can realize it, on which objects, for whom and in which circumstances.

During a game, users interact with the objects of the scenario in order to launch actions. For instance, they click on an apple to pick it, or click on the treadmill to stop it. Thus, the game must interpret from the input which action the user wants to be done. This action is called *User*

Intended Action. The process of identifying the *User Intended Action* involves an analysis of the situation in which the user input has been done (clicked object, user avatar's position and held objects) in relation to the actions defined on the objects involved in the input action.

Similarly to objects, actions can be visual or not: a modification of an attribute, for instance may not be reflected graphically in the environment, but moving an object is visual. Therefore, some actions are associated to *Visual Action*s that are in charge of executing the corresponding Game Engine events.

Finally, in a training game, at each moment of the game, the user is expected to do specific actions. The actions that the user is expected to do at a given instant of the game are called *Task Actions.*

Figure 9 illustrates the process followed in a game from a user input to the execution of an action. The user input information is parsed with the *Action Definitions* in order to determine the *User Intended Action*. Then, if the action is visual, a feasibility analysis is performed to determine if it possible to realize the action graphically. For instance, if the *User Intended Actio*n is to drop an object on the table but the table is full, the action is rejected. If the action is feasible, the action is compared to the expected *Task Actions* defined by the trainer. If the *User Intended Action* matches one of the *Task Actions*, it is executed and scored accordingly. Otherwise, it is executed or not depending on if the level of difficulty of the tasks allows trainee to realize incorrect actions.

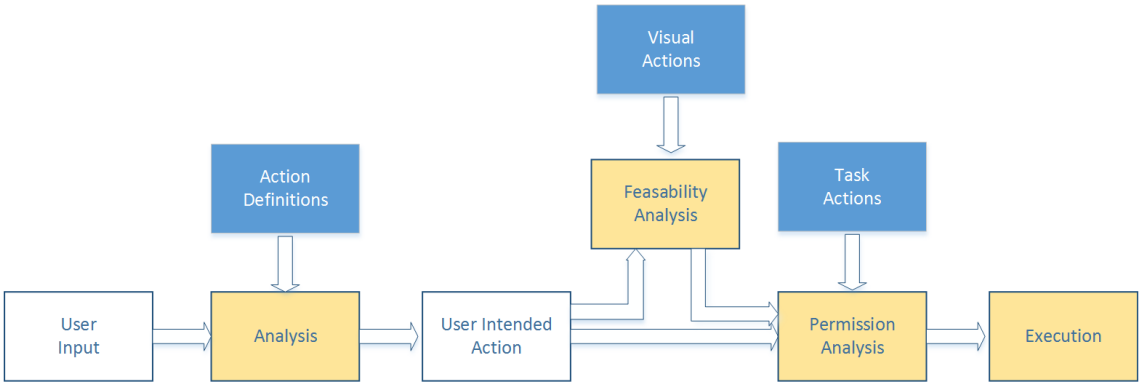

Figure 9: Differences between Actions Definition, Visual Action and User Intended Actions and how user input is processed during the game execution to yield to actions

2.3.1- Action Definitions

Actions are defined on the basis of classical grammar analysis. Specifically, actions represent complete propositions composed of five grammatical components:

```
<subject> <verb> <direct object> <indirect object> <circumstantial
complements>
```

Where:

- **subject** is the characters or objects that can launch the action (for instance: *trainee* picks an apple);
- **verb** is the verb that defines the action; It is also the identifier of an action;
- **direct object** is any object of the scenario that receives the action (for instance: trainee picks an *apple*)

- **indirect object**, i.e. any character of the scenario to or for whom the action is performed (for instance: trainee attaches the blood pressure sensor *to the patient*);
- **complements** give precisions on the action. They can be of three types:
  - **instrument**: trainee pick the cheese *with a fork*
  - **place**: trainee drop the notebook *on the table*
  - **time:** trainee move the cheese *fast; trainee go fast to the table*

In the object definition state diagram, the actions associated to edges of the graph store information on the role of the object in the corresponding action (subject, object or complement).

Actions that have the same definition except for the verb and the same implementation are called *alias*. They are used to enrich the training scenario allowing the use of real life verbs even though in the virtual world they have the same implementation. Examples of alias actions are those corresponding to the verbs to chop and to cut that are absolutely identical in the virtual world:

```
<subject: any character> <verb: to chop/to cut> <direct object: vegetable|fruits> <instrument: knife>
```

Actions that have the same definition except for the verb but different implementation are called *Equivalent Actions.* Equivalent Actions can produce ambiguities in the identification of the User Intended Action. Examples of *Equivalent Actions* are those corresponding to the verbs to cut and to peel that produce different results in the virtual world:

```
<subject: any character> <verb: to peel/to cut> <direct object: vegetable|fruits> <instrument: knife>
```

2.3.2- Implementation of actions

The implementation of actions is based on the following processes:

- Creation of a new instance of an object (e.g. *trainee breaks an egg* yields to the creation of two new instances of a shell object; *system creates a panel* yields to the creation of a new instance of the object panel)
- Destruction of an object instance (e.g. *trainee throw a piece of paper in the garbage* yields to the removal of the piece of paper instance)
- Creation and removal of a parental relationship (e.g. *trainee picks an apple from the plate* yields to the destruction of the parenting relationship of the apple with the plate and the creation of a parenting relationship of the trainee's avatar and the apple)
- Change object state (e.g. *trainee opens the door*, yield to a change of state of the door)
- Change object attributes (e.g. *trainee resets timer* yields to assigning the value 0 to the attribute time of the object timer)

In order to promote the re-use of the code, the implementation of actions can be defined as compositions (composite actions) of other simpler actions (basic actions). For instance, *trainee breaks an egg against a pan* is the composition of:

- the action "*break egg"* , which is in fact a state transition from raw+full to egg_shell,
- the action *"create egg instance"* , in state "*raw without shell"*
- the action "*drop egg in the pan"*, which consists of an animation and the creation of a parental relationship between the pan and the egg.

The composition of atomic actions is based onto two schemes: sequential composition in which atomic actions are done one after the other and parallel composition in which the atomic actions are done at the same time.

.

2.3.3- User intended actions

The retrieval of the requested action from the user input is based on the analysis of the circumstances of the input. Specifically, the user input is a triplet: <clicked object, user avatar, input frequency>. The clicked object provides information on its state and thus on the actions in which it can be involved. It also provides information on its position. The user avatar provides data on its position, the objects that he/she is carrying virtually, their current state and allowed actions and the previous action that he has done. From this input at most one unique *user intended action* must be found.

The solution of the analysis is not unique, because *equivalent actions* can be found. As an example a user click on an apple with a knife could yield to two equivalent actions to cut and to peel. In this case, the analysis stage chooses between the equivalent actions according to a priority system.

Depending on this analysis, the clicked object can be considered as:

- Direct object: e.g. trainee clicks on pencil corresponds to:
  <subject: trainee> <verb: picks> <direct object: the pencil>
- Indirect object: e.g. trainee clicks on a patient avatar with a rope corresponds to:
  <subject: trainee> <verb: gives > <direct object: a rope> <indirect object: to the patient>
- Complement of place: e.g. the trainee clicks on the door from far away, corresponds to:
  <subject: trainee> <verb: goes > <place complement: to the door>.
  Observe that if the door is within the user's avatar reach, the same interaction will be interpreted as:
  <subject: trainee> <verb: opens> <direct object: the door>.

Similarly, the carried object can be considered as:

- Direct object: e.g. when the user clicks on the table with a knife the corresponding user intended action is:
  <subject: trainee> <verb: drops> <direct object: *the knife*> <place complement: *on the plate*>
- Instrument complements: e.g. when the user clicks on a fried egg with a skimmer the corresponding requested action is:
  <subject: trainee> <verb: *picks*> <direct object: *the egg*> <instrument complement: *with the skimmer*>.

When the object carried by the user hosts another objects, this object can also intervene in the interpretation of *user intended action*:

- Direct object, e.g. when the user clicks on the pan with a skimmer carrying an egg, the corresponding user intended action is:
  <subject: *trainee*> <verb: *drops*> <direct object: *the egg*> <instrument complement: *with the skimmer*> <place complement: *on the plate*>

Observe that the latter example, could have been interpreted as <subject: *trainee*> <verb: *drops*> <direct object: *the skimmer*> <place complement: *on the plate*>. Because of the parenting relationship, the skimmer would have been dropped with the egg on top of it. It is a clear example of *equivalent actions*. The analysis process will choose one or the other according to a disambiguation mechanism based on priorities.

2.3.4- Visual Actions

Visual actions are simple launchers of the graphical actions executed by the Game Engine. These graphical actions are compositions of:

- Animations
- Geometrical transformations (grab and rotate)
- Change of texture or colour
- Change of graphical model
- Removal of graphical objects
- Creation, update and removal of the topological relationship of hosting.

2.4- The task

The task represents the narrative in the environment. It is expressed in terms of expected actions. It expresses the desired behaviour of objects and characters, the required trainee actions and the reaction of objects and characters to these actions. It is also associated with the scene configurations in which it works.

Figure 10 shows a diagram of the structure and components of a task.

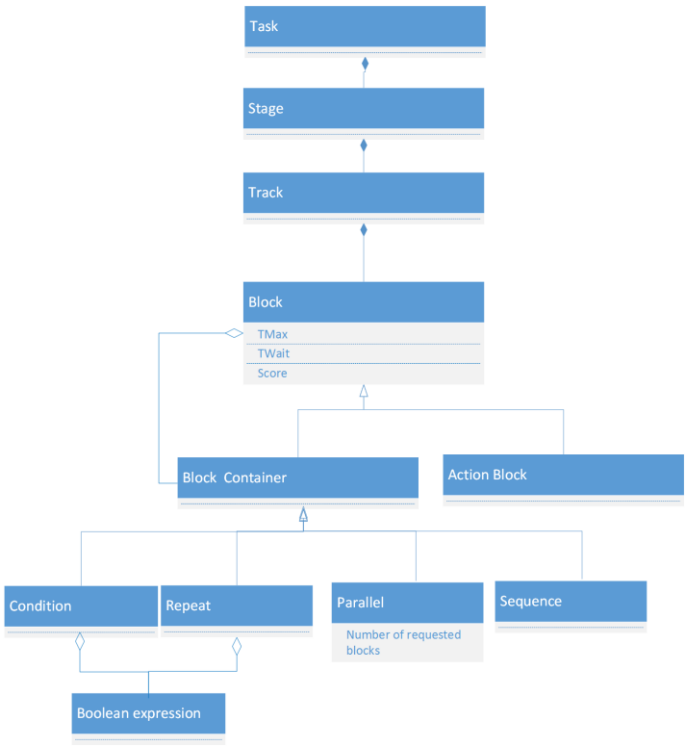

Figure 10: Structure of a task

A task is divided into three stages: *introduction*, *development* and *conclusion*. The introduction stage is devoted to the initial setting of the scene and the introduction of the game, generally with welcome and instruction messages; the development is the game itself and the conclusions usually shows results or evaluation messages.Each stage is divided into *tracks* composed of independent nested *blocks*.

The tracks are structured in blocks. There are two types of blocks: *container blocks* and *action blocks*. The former ones nest other blocks according to different types of structure, whereas the latter ones are atomic and represent actions to be done.

Blocks express a relationship of temporal dependency between the inner blocks. There are four types of blocks:

- Sequence: blocks into a sequential block must be performed one after the other. User actions that do not follow the sequential order are considered incorrect.
- Parallel (N, M): a number M of the N blocks of the parallel block (M<=N) must be performed, no matter in which order.
- Repeat (B): blocks in the repeat block must be repeated while the Boolean expression B is fulfilled.
- Conditional (B): blocks into the conditional blocks must be performed if the Boolean expression is true.

The Boolean expressions required in the *Repeat* and *Conditional* blocks are composed of Boolean sub-expressions operated with the Boolean operators *and*, *or* and *not*. The sub-expressions are made of relationship expressions on the object and character attribute values with the relational operators >, <, ≤, ≥, =, ≠.

In theintroduction stage the initial setting of the scene is performed. Typically, in this stage, the attribute values of some objects are set, some objects are put in a given state and a message or video is shown. This stage contains only actions performed by the system. The development stage contains a description of the task itself. It is composed of various independent tracks, one of which expresses the expected user actions. Finally, the last stage devoted to a summary of the game results contains only system actions. Typically, it consists on posting results summary messages and system actions needed to finish the task. As in the introduction stage, there are no user actions in the conclusion track.

Figure 11: Example of a task structure shows a simplified example of a task structure in the rehabilitation pilot case. Training occurs in a gym with only one patient and treadmill. In the introductory stage the system shows an introductory message. Then, the patient goes to the treadmill and gets on it. The development track is composed of two tracks. Track 1 is devoted to the patient that expresses that the patient will run whenever the treadmill is on. Track 2 represents the expected trainee behaviour: first the trainee should turn on the treadmill. Then, if the patient heart rate is higher than 100, he must stop the treadmill and say good bye to the patient. Otherwise, the trainee will wait until the treadmill will stop at the end of the program and say good bye to the patient. Finally the conclusion stage consists of a unique system action, showing a message.

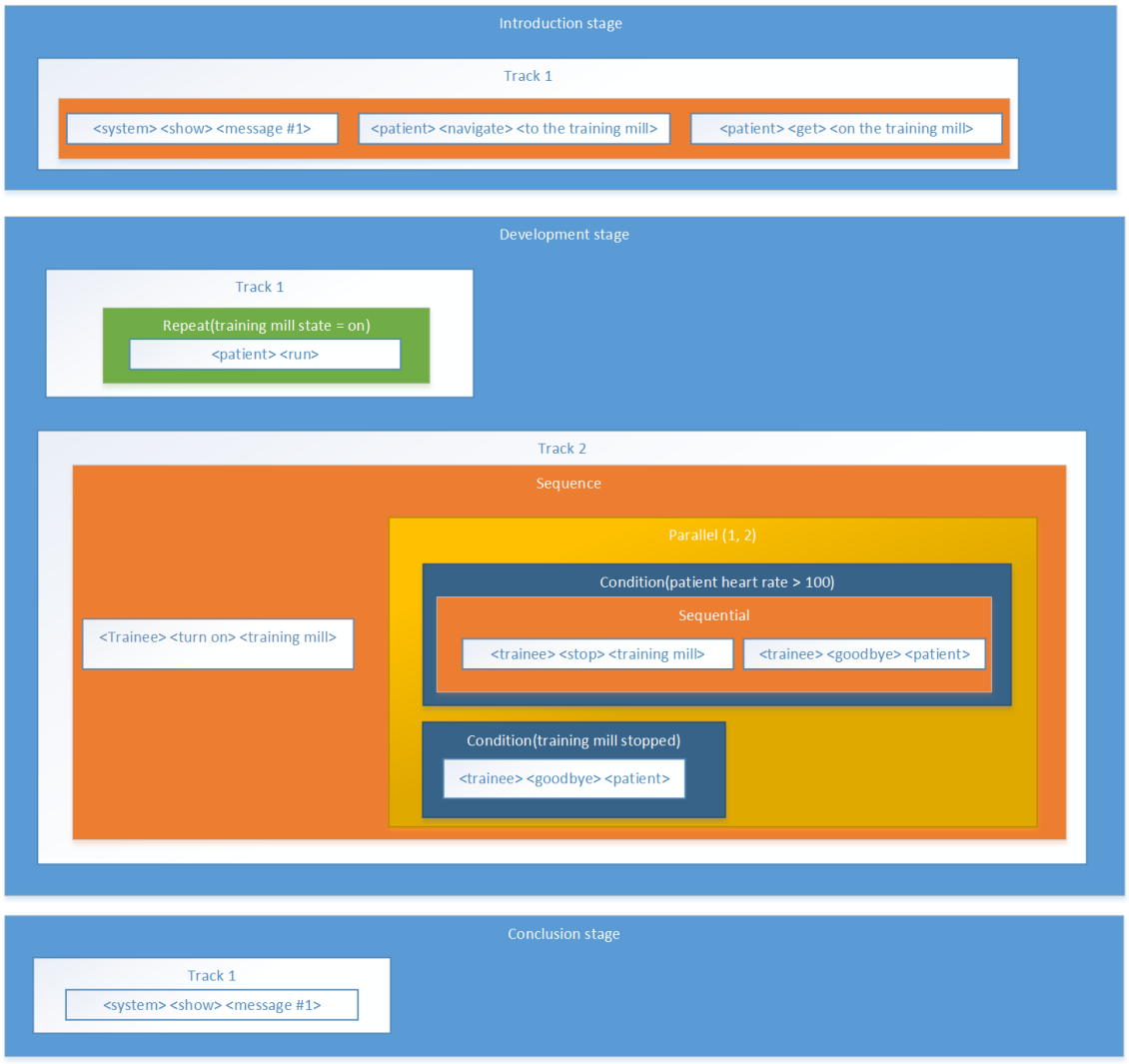

Figure 11: Example of a task structure

2.5- Training program

A training program is a composition of tasks. It is defined as nested blocks containing programmed tasks and levels of difficulty (see Figure 12). The levels of difficulty are defined as a combination of different factors: time multiplier to increase the allowed time, type of feedback, speed of navigation, allowed actions and so on. The levels of difficulty are defines as a list of tuples, label, value.

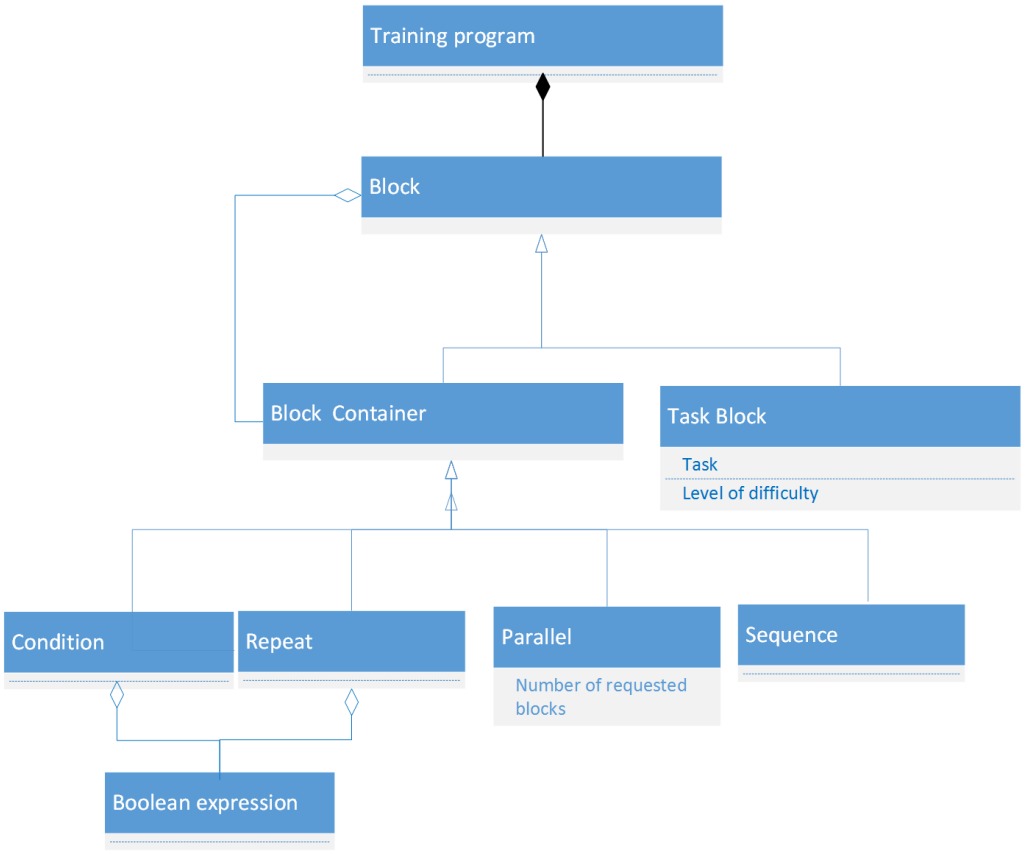

Figure 12: Simplified class diagram of the training program

3- File Descriptors

3.1- Overview

This section describes the files used to share information between components. In the previous sections we described the different data classes, and now we will introduce where each data class is created and which components use it.

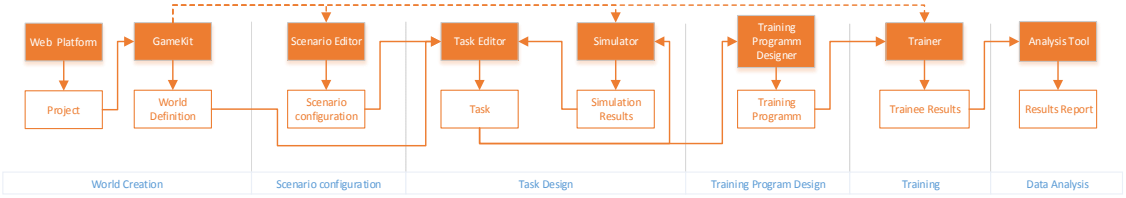

Figure 13 - File Descriptors overview

This figure represents the components created by the game kit (Scenario Editor, Simulator, Trainer) using dash-arrows, the file descriptors using white boxes (project, world definition, scenario configuration, and so on), who creates the descriptor with an arrow from the component to the descriptor (Game Kit → World Definition), and the descriptors used by each component with and arrow from the descriptor to the component (World Definition → Task Editor).

3.2- Project File Descriptor

To create new world the first thing that the developer/manager will need to do is create a new game entry in the platform, this will generate the *Project File* which contains the project name, description, version and a unique identifier…

This information will be used as input for the Game Kit during the world creation process to associate the generated files with the game.

Example:

```
<?xml version="1.0" encoding="utf-8"?>
<project xmlns="http://movibio.lsi.upc.edu/seniorludens"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <meta>
  <name>D1.3 Project Example</name>
  <copyright>SeniorLudens</copyright>
 </meta>
 <id>D1.3</id>
 <version>0.1</version>
 <key>QWERTY123124332412341</key>
 <secret>TOPSECRETTOKEN</secret>
</project>
```

3.3- World Descriptor

Once the game has been designed all the information about the objects, actions, scenes, needs to be stored to be used latter by the Task Editor.

This descriptor will be generated by the Game Kit and will be included on the generated components: Scenario Editor, Simulator and Trainer.

The next example represents a subset of the Figure 8: Complete state diagram of the object definition and three possible state diagrams of object instances:

```
<?xml version="1.0" encoding="utf-8"?>
<world
 version="0.1"
 app="d1.3"
 xmlns="http://movibio.lsi.upc.edu/seniorludens"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 >
 <!-- Actions -->
 <actiondef id="put_on">
  <meta>
   <name>Put on</name>
   <description>Puts the object on a surface.</description>
  </meta>
 </actiondef>
 <actiondef id="boil">
  <meta>
   <name>Boil</name>
   <description>Boils an object.</description>
  </meta>
  <paramdef name="time" type="int"/>
 </actiondef>
 <actiondef id="fry">
  <meta>
   <name>Fry</name>
   <description>Fries an object.</description>
  </meta>
  <paramdef name="time" type="int"/>
 </actiondef>
 <actiondef id="destroy">
  <meta>
   <name>Destroy</name>
   <description>Removes the object from the scene</description>
  </meta>
 </actiondef>
 <!-- Objects -->
 <objectdef id="egg">
  <meta>
   <name>Egg</name>
   <description>Chicken egg</description>
  </meta>
  <propertydef name="temperature" type="int">0</propertydef>
  <propertydef name="heat_temp" type="int">0</propertydef>
  <state name="raw">
   <meta>
    <name>Decoration</name>
   </meta>
   <visual style="basic">
    <param name="model">egg_shell</param>
   </visual>
   <action name="click"/>
  </state>
  <style name="basic">
   <name>Basic</name>
   <description>Basic</description>
  </style>
 </objectdef>
 <!-- Scenes -->
 <scenedef id="kitchen">
  <meta>
   <name>Kitchen</name>
  </meta>
  <instance name="egg.01" definition="egg" state="raw"/>
  <instance name="egg.02" definition="egg" state="shell"/>
  <instance name="egg.03" definition="egg" state="fried"/>
 </scenedef>
</world>
```

3.4- Scenario Configuration Descriptor

After the creation of the world the Scenario Designer should create a scenario configuration. They will use the Scenario Editor to place objects in the different scenes and change the state and properties of them or some of the objects that appear by default on each scene.

The next example file shows a scenario configuration with an egg shell (on the marble) and an egg raw (on the pan). Also changes some objects: the default state of the kitchen door to make it appear opened by default and the temperature to display in the thermometer.

```
<?xml version="1.0" encoding="utf-8"?>
<scenario
 xmlns="http://movibio.lsi.upc.edu/seniorludens"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 id="sceneario1"
 world="d1.3v01"
 version="0.1">
 <meta>
  <name>Kitchen defaults collocation</name>
 </meta>
 <sceneconfig scene="scene1">
  <collocation action="put" name="egg.01" definition="egg" state="shell"
parent="marble"/>
  <collocation action="put" name="egg.02" definition="egg" state="raw"
parent="pan">
   <property name="coocked">20</property>
  </collocation>
  <configure instance="kitchendoor" state="opened"/>
  <configure instance="thermomether.01">
   <property name="temperature">22</property>
  </configure>
 </sceneconfig>
</scenario>
```

3.5- Task Descriptor

The task editor is the tool used by the trainer to design the reference task for the trainee and define the different roles of the characters.

The next example will show the task descriptor file for the task of the Figure 11: Example of a task structure:

```
<?xml version="1.0" encoding="utf-8"?>
<task
 xmlns="http://movibio.lsi.upc.edu/seniorludens"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 id="example1">
 <meta>
  <name>A demo task</name>
 </meta>
 <introduction>
  <track>
   <sequence>
    <action>
     <subject>system</subject>
     <verb>show</verb>
     <directobject>message #1</directobject>
    </action>
    <action>
     <subject>patient</subject>
     <verb>navigate</verb>
     <directobject>treadmill</directobject>
    </action>
    <action>
```

```
      <subject>patient</subject>
      <verb>get</verb>
      <directobject>treadmill</directobject>
     </action>
    </sequence>
   </track>
  </introduction>
  <development>
   <track>
    <repeat>
     <invariant>
      <equals>
       <currentstate>treadmill</currentstate>
       <state from="treadmill">on</state>
      </equals>
     </invariant>
     <action>
      <subject>patient</subject>
      <verb>run</verb>
     </action>
    </repeat>
   </track>
   <track>
    <sequence>
     <action>
      <subject>trainee</subject>
      <verb>turn on</verb>
      <directobject>treadmill</directobject>
      <place>pan</place>
     </action>
     <parallel requested="1">
      <condition>
       <expression>
        <gt>
         <propertyfrom name="heart_rate">patient</propertyfrom>
         <value type="int">100</value>
        </gt>
       </expression>
       <sequence>
        <action>
         <subject>trainee</subject>
         <verb>stop</verb>
         <directobject>treadmill</directobject>
        </action>
        <action>
         <subject>trainee</subject>
         <verb>goodbye</verb>
         <indirectobject>patient</indirectobject>
        </action>
       </sequence>
      </condition>
      <condition>
       <expression>
        <propertyfrom name="treadmill">stopped</propertyfrom>
       </expression>
       <action>
        <subject>trainee</subject>
        <verb>goodbye</verb>
        <indirectobject>patient</indirectobject>
       </action>
      </condition>
     </parallel>
    </sequence>
   </track>
  </development>
  <conclusion>
   <track>
    <action>
     <subject>system</subject>
     <verb>show</verb>
     <directobject>message #2</directobject>
    </action>
```

```
  </track>
 </conclusion>
 <valid>scenario1</valid>
</task>
```

3.6- Training Program Descriptor

The trainer creates customized training programs on the basis of the existing tasks.

The next example is a training program in which two difficulty levels are defined (slow, feedback). The first task is the demo task with feedback and then the trainee will do four times the demo task with different configurations and in difficulty mode slow.

```
<?xml version="1.0" encoding="utf-8"?>
<trainingprogram
 xmlns="http://movibio.lsi.upc.edu/seniorludens"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 id="example1">
 <meta>
  <name>Training program 1</name>
 </meta>
 <level id="slow">
  <param name="timemultiplier">4</param>
 </level>
 <level id="feedback">
  <param name="objecthighlight">true</param>
 </level>
 <track>
  <sequence>
   <taskblock level="feedback" task="demotask1">
    <configuration>sceneconfig0</configuration>
   </taskblock>
   <repeat>
    <times>4</times>
    <taskblock level="slow" task="demotask1">
     <configuration>sceneconfig1</configuration>
     <configuration>sceneconfig2</configuration>
     <configuration>sceneconfig3</configuration>
    </taskblock>
   </repeat>
  </sequence>
 </track>
</trainingprogram>
```

3.7- XML Schemas

The previous examples follow the specification defined using XML Schemas. The current definition is represented using 5 different files. The basic elements, the world, the scenario configuration, the task and the training program.

3.7.1- Base Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://movibio.lsi.upc.edu/seniorludens"
  elementFormDefault="qualified"
  xmlns="http://movibio.lsi.upc.edu/seniorludens"
  xmlns:mstns="http://movibio.lsi.upc.edu/seniorludens/base.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
 <xs:simpleType name="type">
  <xs:restriction base="xs:string">
   <xs:enumeration value="string"/>
   <xs:enumeration value="float"/>
   <xs:enumeration value="int"/>
   <xs:enumeration value="vector"/>
```

```
  <xs:enumeration value="bool"/>
 </xs:restriction>
</xs:simpleType>

<xs:simpleType name="function">
 <xs:restriction base="xs:string">
  <xs:enumeration value="directobject"/>
  <xs:enumeration value="place"/>
  <xs:enumeration value="indirectobject"/>
  <xs:enumeration value="subject"/>
  <xs:enumeration value="instrument"/>
 </xs:restriction>
</xs:simpleType>

<xs:complexType name="metadata">
 <xs:sequence>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="description" type="xs:string" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="extmetadata">
 <xs:complexContent>
  <xs:extension base="metadata">
   <xs:sequence>
    <xs:element name="copyright" type="xs:string" minOccurs="0"/>
    <xs:element name="publishdate" type="xs:date" minOccurs="0"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="param">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="name" use="required" type="xs:string"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="property">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="name" use="required" type="xs:string"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="instance">
 <xs:sequence>
  <xs:element name="property" type="property" minOccurs="0"/>
 </xs:sequence>
 <xs:attribute type="xs:ID" name="name"/>
 <xs:attribute type="xs:IDREF" name="definition"/>
 <xs:attribute type="xs:string" name="state"/>
 <xs:attribute type="xs:IDREF" name="parent"/>
</xs:complexType>


<xs:group name="expressionelements">
 <xs:choice>
  <xs:element name="value" type="value"/>
  <xs:element name="propertyfrom" type="property"/>
  <xs:element name="gt" type="pair"/>
  <xs:element name="equals" type="pair"/>
  <xs:element name="currentstate" type="xs:string"/>
  <xs:element name="state" type="from"/>
 </xs:choice>
</xs:group>

<xs:complexType name="expression">
 <xs:group ref="expressionelements" maxOccurs="1"/>
```

```
</xs:complexType>

<xs:complexType name="value">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="type" type="type" use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="from">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute name="from" use="required"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>


<xs:complexType name="pair">
  <xs:group ref="expressionelements" maxOccurs="2"/>
</xs:complexType>

<xs:element name="project">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="meta" type="extmetadata"/>
   <xs:element name="id" type="xs:ID"/>
   <xs:element name="version" type="xs:float"/>
   <xs:element name="key" type="xs:string"/>
   <xs:element name="secret" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

3.7.2- World Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://movibio.lsi.upc.edu/seniorludens"
  elementFormDefault="qualified"
  xmlns="http://movibio.lsi.upc.edu/seniorludens"
  xmlns:mstns="http://movibio.lsi.upc.edu/seniorludens/world.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  version="0.1"
>
 <xs:include schemaLocation="base.xsd"/>

 <xs:complexType name="paramdef">
  <xs:attribute name="name" use="required"/>
  <xs:attribute name="type" default="string" type="xs:string">
  </xs:attribute>
 </xs:complexType>

 <xs:complexType name="propertydef">
  <xs:simpleContent>
   <xs:extension base="xs:string">
    <xs:attribute name="name" use="required"/>
    <xs:attribute name="type" default="string" type="type"/>
   </xs:extension >
  </xs:simpleContent>
 </xs:complexType>

<xs:complexType name="actiondef">
   <xs:sequence>
    <xs:element name="meta" type="extmetadata"/>
    <xs:element name="paramdef" type="paramdef" minOccurs="0" maxOccurs="unbounded"/>
                </xs:sequence>
  <xs:attribute name="id" use="required" type="xs:ID"/>
  <xs:attribute name="type" type="xs:string" default="sv"/>
```

```
  </xs:complexType>

<xs:complexType name="state">
 <xs:sequence>
  <xs:element name="meta" type="metadata"/>
  <xs:element name="visual" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="param" type="param" maxOccurs="unbounded"/>
     </xs:sequence>
     <xs:attribute name="style" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="action" maxOccurs="unbounded">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="param" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
       <xs:simpleContent>
        <xs:extension base="xs:string">
         <xs:attribute name="name" type="xs:string"/>
        </xs:extension>
       </xs:simpleContent>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string"/>
    <xs:attribute name="function" type="function"/>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="name" type="xs:string"/>
</xs:complexType>

<xs:complexType name="objectdef">
 <xs:sequence>
  <xs:element name="meta" type="extmetadata"/>
  <xs:element name="propertydef" type="propertydef" maxOccurs="unbounded"
minOccurs="0"/>
  <xs:element name="state" maxOccurs="unbounded" type="state"/>
  <xs:element name="partfrom" type="xs:IDREF" minOccurs="0"/>
  <xs:element name="style" maxOccurs="unbounded">
   <xs:complexType>
    <xs:complexContent>
     <xs:extension base="metadata">
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="default" type="xs:boolean" default="false"/>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="id" use="required" type="xs:ID"/>
</xs:complexType>

<xs:complexType name="scenedef">
 <xs:sequence>
  <xs:element name="meta" type="extmetadata"/>
  <xs:element name="instance" type="instance" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="id" use="required" type="xs:ID"/>
</xs:complexType>


 <xs:element name="world">
  <xs:complexType>
  <xs:sequence>
   <xs:element name="actiondef" type="actiondef" maxOccurs="unbounded"/>
   <xs:element name="objectdef" type="objectdef" maxOccurs="unbounded"/>
   <xs:element name="scenedef" type="scenedef" maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="version" type="xs:string" use="required"/>
```

```
    <xs:attribute name="app" type="xs:string" use="required"/>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

3.7.3- Scenario Configuration Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
 targetNamespace="http://movibio.lsi.upc.edu/seniorludens"
  elementFormDefault="qualified"
  xmlns="http://movibio.lsi.upc.edu/seniorludens"
  xmlns:mstns="http://movibio.lsi.upc.edu/seniorludens/scenario.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
<xs:include schemaLocation="base.xsd"/>

 <xs:complexType name="collocation">
  <xs:complexContent>
   <xs:extension base="instance">
    <xs:sequence>
     <xs:element type="param" name="param" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="action" type="xs:string"/>
   </xs:extension>
 </xs:complexContent>
 </xs:complexType>

 <xs:element name="scenario">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="meta" type="metadata"/>
    <xs:element name="sceneconfig">
     <xs:complexType>
     <xs:sequence>
      <xs:element name="collocation" type="collocation" maxOccurs="unbounded"/>
      <xs:element name="configure" maxOccurs="unbounded">
       <xs:complexType>
        <xs:sequence>
         <xs:element type="property" name="property" maxOccurs="unbounded"
minOccurs="0"/>
        </xs:sequence>
        <xs:attribute use="required" name="instance" type="xs:IDREF"/>
        <xs:attribute name="state" type="xs:string"/>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
     <xs:attribute name="scene" type="xs:string"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="id" type="xs:string" use="required"/>
   <xs:attribute name="version" type="xs:string" use="required"/>
   <xs:attribute name="world" type="xs:string" use="required"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

3.7.4- Task Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
 targetNamespace="http://movibio.lsi.upc.edu/seniorludens"
  elementFormDefault="qualified"
  xmlns="http://movibio.lsi.upc.edu/seniorludens"
  xmlns:mstns="http://movibio.lsi.upc.edu/seniorludens/task.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
 <xs:include schemaLocation="base.xsd"/>
```

```
<xs:complexType name="message">
 <xs:sequence>
  <xs:element name="text"/>
  <xs:element name="audio" minOccurs="0"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:ID" use="required"/>
</xs:complexType>

<xs:complexType name="action">
 <xs:sequence>
  <xs:element name="subject" type="xs:string"/>
  <xs:element name="verb" type="xs:string"/>
  <xs:element name="directobject" type="xs:string" minOccurs="0"/>
  <xs:element name="indirectobject" type="xs:string" minOccurs="0"/>
  <xs:element name="instrument" type="xs:string" minOccurs="0"/>
  <xs:element name="place" type="xs:string" minOccurs="0"/>
  <xs:element name="time" type="xs:string" minOccurs="0"/>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="condition">
 <xs:sequence>
  <xs:element name="expression" type="expression"/>
  <xs:choice>
   <xs:element name="sequence" type="sequence"/>
   <xs:element name="parallel" type="parallel"/>
   <xs:element name="action" type="action"/>
   <xs:element name="repeat" type="repeat"/>
  </xs:choice>
 </xs:sequence>
</xs:complexType>

<xs:group name="blocks">
 <xs:choice>
  <xs:element name="sequence" type="parallel"/>
  <xs:element name="parallel" type="parallel"/>
  <xs:element name="condition" type="condition"/>
  <xs:element name="action" type="action"/>
  <xs:element name="repeat" type="repeat"/>
 </xs:choice>
</xs:group>

<xs:complexType name="sequence">
 <xs:group ref="blocks" minOccurs="1" maxOccurs="unbounded"/>
</xs:complexType>

<xs:complexType name="repeat">
 <xs:sequence>
  <xs:choice>
   <xs:element name="invariant" type="expression"/>
   <xs:element name="times" type="xs:int"/>
  </xs:choice>
  <xs:choice>
   <xs:element name="sequence" type="sequence" maxOccurs="unbounded"/>
   <xs:element name="parallel" type="parallel" maxOccurs="unbounded"/>
   <xs:element name="condition" type="condition"/>
   <xs:element name="action" type="action" maxOccurs="unbounded"/>
  </xs:choice>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="parallel">
 <xs:group ref="blocks" minOccurs="2" maxOccurs="unbounded"/>
 <xs:attribute name="requested" use="required" type="xs:int"/>
</xs:complexType>

<xs:complexType name="track">
 <xs:choice>
  <xs:element name="sequence" type="sequence"/>
  <xs:element name="parallel" type="parallel"/>
  <xs:element name="condition" type="condition"/>
  <xs:element name="action" type="action"/>
```

```
    <xs:element name="repeat" type="repeat"/>
   </xs:choice>
  </xs:complexType>

  <xs:complexType name="stage">
   <xs:sequence>
    <xs:element name="track" type="track" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>

  <xs:element name="task">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="meta" type="metadata"/>
     <xs:element name="message" type="message" maxOccurs="unbounded"/>
     <xs:element name="introduction" type="stage" minOccurs="0"/>
     <xs:element name="development" type="stage"/>
     <xs:element name="conclusion" type="stage" minOccurs="0"/>
     <xs:element name="valid" type="xs:IDREF" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string" use="required"/>
   </xs:complexType>
  </xs:element>


 </xs:schema>
```

3.7.5- Training Program Schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema
 targetNamespace="http://movibio.lsi.upc.edu/seniorludens"
  elementFormDefault="qualified"
  xmlns="http://movibio.lsi.upc.edu/seniorludens"
  xmlns:mstns="http://movibio.lsi.upc.edu/seniorludens/trainingprogram.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
 <xs:include schemaLocation="base.xsd"/>

 <xs:complexType name="taskblock">
  <xs:sequence>
   <xs:element name="configuration" type="xs:IDREF" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="level" type="xs:IDREF" use="required"/>
  <xs:attribute name="task" type="xs:IDREF" use="required"/>
 </xs:complexType>

 <xs:complexType name="tr_condition">
  <xs:sequence>
   <xs:element name="expression" type="expression"/>
   <xs:choice>
    <xs:element name="sequence" type="tr_sequence"/>
    <xs:element name="parallel" type="tr_parallel"/>
    <xs:element name="taskblock" type="taskblock"/>
    <xs:element name="repeat" type="tr_repeat"/>
   </xs:choice>
  </xs:sequence>
 </xs:complexType>

 <xs:group name="trainingblocks">
  <xs:choice>
   <xs:element name="sequence" type="tr_sequence"/>
   <xs:element name="parallel" type="tr_parallel"/>
   <xs:element name="condition" type="tr_condition"/>
   <xs:element name="taskblock" type="taskblock"/>
   <xs:element name="repeat" type="tr_repeat"/>
  </xs:choice>
 </xs:group>

 <xs:complexType name="tr_sequence">
  <xs:group ref="trainingblocks" minOccurs="1" maxOccurs="unbounded"/>
```

```
</xs:complexType>

<xs:complexType name="tr_repeat">
 <xs:sequence>
  <xs:choice>
   <xs:element name="invariant" type="expression"/>
   <xs:element name="times" type="xs:int"/>
  </xs:choice>
  <xs:choice>
   <xs:element name="sequence" type="tr_sequence" maxOccurs="unbounded"/>
   <xs:element name="parallel" type="tr_parallel" maxOccurs="unbounded"/>
   <xs:element name="taskblock" type="taskblock" maxOccurs="unbounded"/>
  </xs:choice>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="tr_parallel">
 <xs:group ref="trainingblocks" minOccurs="2" maxOccurs="unbounded"/>
</xs:complexType>

<xs:complexType name="trainingtrack">
 <xs:group ref="trainingblocks" />
</xs:complexType>

<xs:element name="trainingprogram">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="meta" type="metadata"/>
   <xs:element name="level" maxOccurs="unbounded" minOccurs="1">
    <xs:complexType>
     <xs:all>
      <xs:element name="param" type="param"/>
     </xs:all>
     <xs:attribute name="id" type="xs:ID" use="required"/>
    </xs:complexType>
   </xs:element>
   <xs:element name="track" type="trainingtrack"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>

</xs:schema>
```

Figures and tables

Acronyms

Acronym	Explanation
SLGK	SeniorLudens Game Kit
XML	eXtensible Markup Language