EMERGO toolkit architecture

Open Universiteit in the Netherlands
CELSTEC (Centre for Learning Sciences and Technologies)

Authors: Aad Slootmaker, Hub Kurvers

Date: august 24, 2010
Version 1.0

Centre for Learning Sciences and Technologies
celstec.org

Contents

1. Introduction 4
2. Domain model 4
  Account class 6
  Role class 6
  Component class 7
  Case class 8
  CaseRole class 9
  CaseComponent class 9
  CaseComponentRole class 9
  Run class 10
  RunGroup class 10
  RunGroupCaseComponent class 11
  RunCaseComponent class 12
  RunTeam class 12
  RunTeamCaseComponent class 12
  RunGroupCaseComponentUpdate class 13
  Blob class 13
  Sys class 14
  Mail class 14
  AccountRequest class 14
3. Application layers 14
  View layer 16
    Zul files 16
    Style 16
    Package nl.surf.emergo.view 16
    Properties files 16
  Control layer 16
    Package nl.surf.emergo.control.def 17
    Package nl.surf.emergo.control 17
    Package nl.surf.emergo.control.run 20
  Model layer 22
    Package nl.surf.emergo.domain 22
    Package nl.surf.emergo.business 22
    Package nl.surf.emergo.business.abstr 23
    Package nl.surf.emergo.business.impl 23
    Package nl.surf.emergo.zkspring 24
  Persistence layer 24
    Package nl.surf.emergo.data 24
    Package nl.surf.emergo.data.hibernate 24
  Database layer 25
    Blob submap 25
    Streaming submap 25
    Temp submap 25
4. XML use 25
  XML definitions 28
    The ‘locations’ component 28
    The ‘tasks’ component 31
    The ‘empack’ component 32
    The ‘references’ component 32
    The ‘conversations’ component 34
    The ‘mail’ component 36

The ‘googlemaps’ component 39
The ‘alerts’ component 40
The ‘note’ component 41
The ‘logbook’ component 41
The ‘items’ component 42
The ‘assessments’ component 44
The ‘scripts’ component 46
The ‘case’ component 48
XML data 49
The ‘locations’ component 49
The ‘assessments’ component 50
The ‘conversations’ component 50
The ‘scripts’ component 51
The ‘case’ component 54
XML progress data 56
The ‘locations’ component 56
The ‘assessments’ component 57
The ‘conversations’ component 58
The ‘scripts’ component 58
The ‘case’ component 59
The ‘references’ component 59
XML progress update data 61
XML import and export for a case 61
XML import for accounts and run group accounts 62
5. Adding a new EMERGO component 63
Appendix 1: Overview XML elements 64
Elements within XML definitions 65
Elements within XML data 66
Elements within XML progress data 66
Elements within XML progress update data 67
Appendix 2: EMERGO SQL 67

Figures

Figure 1: Domain model 5
Figure 2: Application layers 15
Figure 3. Illustration of XML use 28

1. Introduction

This document is meant for whom who wants to know how the EMERGO toolkit is build and how it can be adjusted or extended.

The toolkit is meant for the web-based development and delivery of scenario-based educational games, called cases. For this it contains administrator, author, run management, tutor and student environments. A player environment is developed for playing the case. These environments are described shortly within this document. For a more extensive description we refer to documents ‘Installation and administrator manual’, ‘Author manual’ and ‘User manual’.
The toolkit is developed such that components can be added rather easily, without having to change the domain model. And even without having to change most of the environments. The player environment has to be extended with a player for the added component. And sometimes the author environment has to be adjusted to be able to enter specific content.
Having said components can be added rather easily, what do we mean by component en content?
By component we mean a software component that is used to enter a specific kind of case content and to play it. So it is always related to case functionality. Within the toolkit every component has a definition which defines a part of this functionality.
By content we mean plain content (assets), structure of the content, relations between content and rules that operate on the content. So it is all that’s entered within the (case) author environment to make the case work.
Component definition, content and end user progress are saved as XML data (see paragraph 4 ‘XML use’).
The most important component is the ‘scripts’ component which allows authors to enter rules, which will fire based on user interaction, timers or other fired rules.
The toolkit is developed in Java using Eclipse. We used two frameworks, ZK (http://www.zkoss.org/) and Spring (http://www.springsource.org/), to build it. MySQL is used for the database. See further document ‘Installation and administrator manual’.

We best illustrate all kind of aspects of the toolkit architecture and features by describing the domain model, see figure 1.
After the domain model we describe the different application layers with their own features. It concerns mainly describing all Java packages we created.
Then we describe where we used XML within the toolkit. By using XML we could simplify the domain model and make adding new components rather easy. But then the XML structure is also a kind of domain model. That’s why we describe it here extensively.
And at last we describe what steps to take to add a new EMERGO component, so to extend the toolkit.

In two appendices we give an overview of all XML Elements used and the initial EMERGO SQL script, to be used to initialise a new database.

2. Domain model

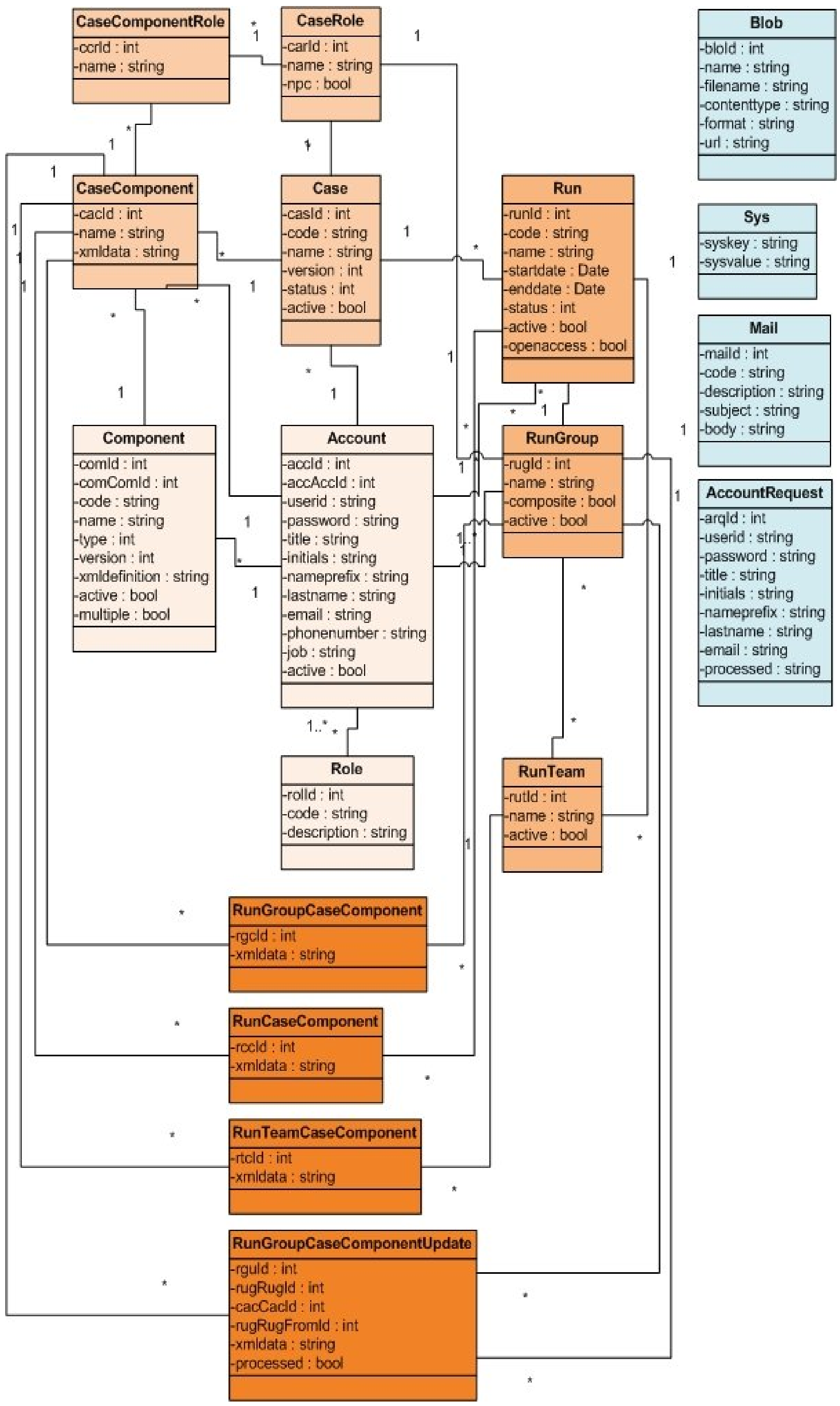

CaseComponentRole
-ccrId : int
-name : string
CaseRole
-carId : int
-name : string
-npc : bool
Blob
-blobId : int
-name : string
-filename : string
-contenttype : string
-format : string
-url : string
CaseComponent
-cacId : int
-name : string
-xmldata : string
Case
-casId : int
-code : string
-name : string
-version : int
-status : int
-active : bool
Run
-runId : int
-code : string
-name : string
-startdate : Date
-enddate : Date
-status : int
-active : bool
-openaccess : bool
Sys
-syskey : string
-sysvalue : string
Mail
-mailId : int
-code : string
-description : string
-subject : string
-body : string
Component
-comId : int
-comComId : int
-code : string
-name : string
-type : int
-version : int
-xmldefinition : string
-active : bool
-multiple : bool
Account
-accId : int
-accAccId : int
-userid : string
-password : string
-title : string
-initials : string
-nameprefix : string
-lastname : string
-email : string
-phonenumber : string
-job : string
-active : bool
RunGroup
-rugId : int
-name : string
-composite : bool
-active : bool
AccountRequest
-arqId : int
-userid : string
-password : string
-title : string
-initials : string
-nameprefix : string
-lastname : string
-email : string
-processed : string
Role
-rolId : int
-code : string
-description : string
RunTeam
-rutId : int
-name : string
-active : bool
RunGroupCaseComponent
-rgcId : int
-xmldata : string
RunCaseComponent
-rccId : int
-xmldata : string
RunTeamCaseComponent
-rtcId : int
-xmldata : string
RunGroupCaseComponentUpdate
-rguId : int
-rugRugId : int
-cacCacId : int
-rugRugFromId : int
-xmldata : string
-processed : bool

Figure 1: Domain model. Background color indicates role responsible for maintaining data of this type. From light to darker background respectively administrator, case developer, case run manager and student. Tutor only has inspecting role. The four classes to the right are general classes.

The domain model reflects the data model. So for each domain class you'll find a database table. Some domain classes aren't visible in the model although they have an associated table. It concerns cross tables needed to enable nXm relations, but which don't have extra fields.
We describe the domain classes in order of depency, so more or less in a logical order. For instance before you enter a case you have to have an account, so the class Case depends on class Account.
See Java package nl.surf.emergo.domain for the Java code of the domain classes. All domain classes are Java Beans containing setters and getters for properties. All classes have properties creationdate and lastupdatedate. These are left out of the domain model. Also properties not used anymore are left out.

Account class

Central in the model are <u>accounts</u>, the users of the EMERGO toolkit. Apart from a userid and password an account has trivial properties like for instance lastname and email. Further an account can be active or not, meaning it has access to the toolkit of not (anymore). Never delete an account, because all of its data within the EMERGO database will be deleted too. Better set active to false to prevent a user accessing the toolkit.
Further an account has a property accaccid which indicates the account id of the administrator account who created it and will manage it.

Role class

Within the toolkit an account can have multiple <u>roles</u>. There exists an nXm relation between accounts and roles implemented by a class AccountRole.
Currently there are five roles: administrator, case developer, case run manager, tutor and student. Roles have a code which is used within the Java code to distinguish them: respectively adm, cde, crm, tut and stu. This code is also used to get the right full name of the role out of the language specific property files (see section 'View layer' within paragraph 3 'Application layers').

We first shortly describe the different roles. We need this to understand the other domain classes. Don't bother if you don't fully understand the description of roles beneath. It will become clear further on.

- In his environment an administrator manages all accounts and their roles. He also manages components and their XML definitions. Further he can help accounts who are stuck within an EMERGO case, by changing their progress status and/or starting the player environment read-only using their progress status, to see where they are stuck. And at last he can deactivate cases that are not used anymore, and per case he can upload streaming files to the server.
- In the author environment a case developer can define new cases. Per case he can define case roles and case components. Case roles are roles which can be played by playing characters or are non-playing characters. He defines a case component by choosing one of the existing component templates and

assigning an author to it (he himself or another account with the role case developer). Per case component he enters the content (saved as XML data). During authoring he can preview the case within the player environment.

- In the run management environment a case run manager can manage runs of cases developed by case developers. He can assign accounts with roles tutor and/or student to a run. Further he can define run teams if appropriate.
- In the tutor environment a tutor can view the progress of students for cases he is assigned to by the case run manager. Further he can interfere within the case scenario (by sending e-messages to students assigned to the case) if this is defined by a case developer.
- In the student environment a student gets an overview of cases he can 'play' (assigned to him by the case run manager) and he can start the player environment with a chosen case.

Component class

An administrator account can manage multiple components. He is owner of the component indicated by property accaccid. Only an administrator account can be owner and editor of a component. Components have a code which is used in the Java code to distinguish them. This code indicates the purpose of the component. The code is also used to get the right full name of the component out of the language specific property files.

A component can be a functional or system component indicated by the type property. Functional components can be chosen by case developers to be used and filled with content within their case. System components are hidden for case developers and are used beneath the surface. There is only one system component. Its code is 'case' indicating it is used for overall case purpose. A 'case' case component is automatically created if a case developer creates a case. All relations between content of other case components are saved as content of the 'case' case component, so it functions as a sort of 'cross table'. Further it is used to save global case properties like case time.

A component has a property comcomid which indicates its 'parent' component within the player environment. For instance all components which appear on the empack component have this component as parent.

The multiple property indicates if the component can be used multiple times by a case developer within one case. Restriction to one instance is mainly due to interaction design of the player environment. For instance there is only one 'empack' button, so there can be only one instance of this component within a case. Multiple is also false for the 'mail' component, because it would be strange for a user to have two functionally the same mail components.

The most import property is xmldefinition. It contains the XML definition of the component (see section 'XML definitions' of paragraph 4 'XML use').
It defines which component properties and type of content elements can be entered by a case developer and in which way the content elements are structured so the author environment can render a component specific editor. A component property is for instance if the component is accessible by a student account within the player environment. Content elements are for instance all maps and sources used within the 'references' component.

The XML definition also defines which component property and content element (property) changes are saved (including a timestamp) for a student account within the player environment and which can be read and changed by rules within a 'scripts' component.
Properties are simple boolean or integer values (saved as strings). For instance opened="true" or score="10". Content elements only contain strings.
Only such simple properties can be checked and set by rules within a 'scripts' component. Script can only be defined on existing other component properties or content element properties.
See paragraph 4 'XML use' for a description of all XML definitions and for a description of properties to be used within a 'scripts' component.

Some component definitions allow a case developer to define if user generated content of a student account is saved privately or is shared by all other students or just shared by students in a team. There are three domain classes to take care of this: RunGroupCaseComponent, RunCaseComponent and RunTeamCaseComponent (see further). The three types are not mixed so a case component will always be defined as one of these types.
Also a case developer can define if user generated data is owned by student accounts, meaning they are only allowed to change it, or everyone can change it.

Further a component can be active or not, meaning it can be used by case developer accounts or not (anymore). Never delete a component, because all of its data within the EMERGO database will be deleted too. Better set active to false to prevent case developers using the component.

A component has a version to be able to distinguish between different versions. For instance an old version could be deactivated.

Case class

A case developer account can manage multiple cases. He is owner of each case he defines. Only a case developer account can be owner and editor of a case indicated by property accaccid (for case components other case developers can be authors too). A case can be given a name and a code. Code can be used for instance to indicate a course the case belongs to. Name and code are not multilingual, so if one wants to create a case in two languages using the same scenario, two cases will have to be created.

The status property can have four values: under construction, template, runnable and locked. The idea was a case first is under construction and can be based on a template case. If it is then published, status will become runnable. Locked would be used to lock a case temporarily.
In practice only status runnable is used, so a case is immediately runnable when created. It's the responsibility of the case run manager when to use a case or not. There are no template cases yet but this could be the case in the future. Locked could be used to prevent case developers editing the case while it's being used within active runs.

Further a case can be active or not. Active means a case run manager can create runs of it. Never delete a case, because all of its data within the EMERGO database will be deleted too. Better set active to false to prevent case run managers using the case. Only adminstrator accounts can change active.

A case has a version to be able to distinguish between different versions. For instance an old version could be deactivated.

CaseRole class

A case owner can add multiple case roles to a case indicated by property cascasid. These are roles that are relevant within the case scenario, so while playing the case within the player environment, as a student account. These roles are in no way connected to the account roles described before.
There are two types of case role (indicated by the property npc): playing characters (pc's) and non playing characters (npc's). Pc's are real persons. A case run manager assigns student accounts to pc's. Npc's are virtual persons within the case. For instance persons represented in video which pc's can ask questions.

CaseComponent class

A case owner can add multiple case components to a case indicated by property cascasid. Default the case owner will be author of the case component properties and content given by property xmldata, but he can assign another case developer account as author, indicated by property accaccid.
The case component is based on a component indicated by property comcomid.

The most important property is xmldata (see section 'XML data' of paragraph 4 'XML use'). It contains the case component properties and content added by the case component author. It is saved as XML. Properties and content are based on the XML definition within the associated component. XML definition and XML data are used to render the case component within the author and player environment.

Within the XML, references can be made to urls, shared files on the server (uploaded by administrator) and private files on the server (uploaded by case developer). For these references a separate class Blob exists, see further.

CaseComponentRole class

A case owner can assign case components to case roles, but only to case roles of type pc (playing character) because only pc's use case components. Npc's don't use case components. Npc's can be seen as decoration objects which don't interfere with other objects.

The purpose of this class is to make it possible to define one case component for multiple case roles, but also to define one for one case role. For instance a 'references' case component could be identical for two case roles, but a 'scripts' case component could be different, so you would create two 'scripts' case components, one for each case role.

The name property default gets the name of the related case component, but can be overruled by the case owner. This name is visible to student accounts within the player environment.

Run class

A case run manager account can define multiple runs of a case indicated by property cascasid. He will be the owner of the run indicated by property accaccid. Only case run manager accounts can be owner. A run can be given a name and a code.

A run has a startdate and an enddate property. Default start date and end date are set to current date and time.
Before the start date the run isn't available to student accounts. After the start date they can choose it to play the related case within the player environment.
The end date is an indication for student accounts how much time they have to play the case. After the end date the run still is available for them so they still have access to the case./

The status property can have four values: under construction, template, runnable and test. The idea was a run first is under construction and can be based on a template run. If it is then published status will become runnable. Test would be used for test runs.
In practice only status runnable and test are used, so a run is immediately runnable when created. Start date determines when it is available to student accounts. Status test is used for the preview option within the author environment. There are no template runs yet but this could be the case in the future.

Further a run can be active or not, default is true, meaning it is accessible for student accounts. Never delete a run, because all of its data within the EMERGO database will be deleted too. Better set active to false. No account role can change active yet.

The openaccess property indicates if student accounts have direct access to a case run without being assigned to it by a case run manager account. Default is false so the case run manager should explicitly assign student accounts to a run.
Tutor accounts are always assigned by the case run manager, whatever the value of the openaccess property.

RunGroup class

A run group stands for one or more student and/or tutor accounts which are playing one case role, indicated by property carcarid, within a run, indicated by property runrunid. There exists an nXm relation between run groups and accounts implemented by a class RunGroupAccount.
The class run group was necessary to allow multiple accounts to act as one player, for instance for a group assignment.
The case run manager account assigns student and/or tutor accounts to a run. In his environment he doesn't have the option yet to assign multiple accounts to one run group. An administrator account which has access to the database will be able to do this.

The name property contains the name of the run group. If it has one account only associated to it, the name is generated out of the account's data. Otherwise a group name should be given.

The property composite indicates if a rungroup has one (false) or more (true) accounts associated to it.

Further a run group can be active or not. A case run manager account can (de)activate a run group. A run group is never deleted, because all of its data within the EMERGO database will be deleted too.

RunGroupCaseComponent class

This class is used to save a student account's progress per case component. This is done automatically within the player environment. The progress is private so no other student accounts can access it.

Progress is saved as XML within property xmldata (see section 'XML progress data' of paragraph 4 'XML use').
The XML definition of the associated component defines which case component property and content element (property) changes are/can be saved (including a timestamp).
Some components allow student accounts to generate content within the player environment or even to adjust content generated by case component authors.
There are five types of progress data:

- Case component property changes. For instance if a certain case component like a 'references' component is opened.
- Case component content element property changes. For instance if a certain source or map within a 'references' component is opened. This can also can be a user generated source or map.
- Case component content element adjustments. For instance the adjustment of a map title within a 'references' component or the removal of a certain source. This can also can be a user generated source or map.
- Case component content element extensions: adding new content elements. For instance adding a source within a 'references' component. For this type the context is relevant. So in which map it is added and before which other source. This data is also saved, to be able to rerender it correctly.
- Case component content element extensions based on a template: adding new content elements based on a template. For instance sending a mail within a 'mail' component. Only mails entered by a case developer are allowed to be send. One mail can be send multiple times with another body or subject string, so a version number is saved.

See paragraph 4 'XML use' for examples of XML progress data.

Within the XML progress data, references can be made to urls, shared files on the server (uploaded by administrator) and private files on the server (uploaded by student account, for instance a mail attachment). For these references a separate class Blob exists, see further.

Component XML definition, case component XML data and XML progress data are used to render the case component within the player environment. Saved XML progress data overrules case component XML data. Before rendering both are merged.

Changes in progress data caused by user interaction, timers or a fired rule within a 'scripts' component, can fire a(nother) 'scripts' component rule.

Script rules only apply to changes in progress data of a run group case component. Not for run case component or run team case component: see further.

RunCaseComponent class

This class is used to save progress of all student accounts within a run per case component. This is done automatically within the player environment.
Much of what is mentioned for class RunGroupCaseComponent applies to this class too.
Differences are:

- Progress data for all student accounts is saved in one XML string but not all progress data is shared. For instance if another student acount adds a source within a 'references' component, the player environment will show it to you too. But if he opens a source the player environment doesn't show this, because this probably isn't relevant for you. It depends on the type of component what you should see from other student accounts.
- Because progress is saved in one XML string, for every change also the id of student account is saved.
- Script rules don't apply for changes in run case component progress data.

Because progress must be shared among multiple student accounts, the player environment has a mechanism, using a timer, to update the environment for all other student accounts (that is if they are 'on line'), if one of them changes something. This is done by using the application memory, which is shared by all EMERGO toolkit users.

RunTeam class

A run team stands for one or more run groups which have to work together in a run, indicated by property runrunid. In a run team run groups can participate with different or the same case roles or a mixture. There exists an nXm relation between run teams and run groups implemented by a class RunTeamRunGroup.
The case run manager account assigns run groups to a run team.

The name property contains the name of the run team given by the case run manager.

Further a run team can be active or not. The case run manager environment does not allow him to set active. Instead he deletes the run team if he wants to deactivate it. This means all of its data within the EMERGO database will be deleted too.

RunTeamCaseComponent class

This class is used to save progress of student accounts within a run team per case component. This is done automatically within the player environment.
Progress only is shared by run team members.
Further all that is mentioned for class RunCaseComponent applies to this class too.

RunGroupCaseComponentUpdate class

This class is used to update a student account's private progress handled by class RunGroupCaseComponent.
There are three situations for which this class is used:

- One student account sends a mail to another one using the 'mail' component. The sent mail has to be saved in the outbox of the sender and a copy has to be made in the inbox of the receiver. For the latter this class is used.
- A tutor account sends a mail to a student account using the 'mail' component. The sent mail doesn't have to be saved in the outbox of the sender because the tutor is using the student account to accomplish this. But the mail has to be put in the inbox of the receiver. For the latter this class is used.
- An administrator can help out student accounts if they are stuck in a case run. He does this by changing the progress data of a student account. This class is used for this.

The rugrugid property indicates the run group whose progress data should be updated.
The rugrugfromid property indicates the run group which caused the update.
The xmldata property contains the update in XML.
See paragraph 4 'XML use' for examples of XML progress update data.
The property processed indicates if the update is processed already (so run group progress data is updated accordingly) or not.

The player environment has a mechanism, using a timer, to update the progress data and environment for student accounts who's progress data should be updated (that is if they are 'on line'). This is done by polling the table rungroupcasecomponentupdates.

Within the XML progress update data, references can be made to urls, shared files on the server (uploaded by administrator) and privately uploaded files on the server. For these references a separate class Blob exists, see further.

Blob class

Within all XML data entered by case developers or student accounts, references can exist to blobs. The blob class is used for this. References are made using the bloid property of the blob class. So blobs are related to other classes, but not directly within the domain model, but by XML data references.
There are three types of blobs (the type is part of the XML definition for a component):

- External urls, other urls than the EMERGO toolkit url.
- Internal urls, pointing to shared files for a case on the EMERGO toolkit server. These files, mostly large video files but also other types, are uploaded by an administrator account.
- Privately uploaded files on the server. Uploaded by case developers or student accounts. For instance a mail attachment.

The name property is filled with the url or the filename pointed to.
The filename, contenttype and format type only are used for privately uploaded files.
The url property only is used for external and internal urls.

Sys class

There are some configuration values that have to be stored in the database.
This sys class is used to read them. Property syskey contains the configuration key and property sysvalue the associated value.

Mail class

Certain notification mails (real mails) are sent to tutor or student accounts. For instance, when student accounts send a mail using the ‘mail’ component within the player environment, a real notification mail is sent to a tutor.
This mail class is used to read templates for these (real) mails.
The code property is an indicator for the type of mail.
The description property describes the type.
The subject and body properties are templates for the mail, and contain tags. Before the mail is send, these tags are replaced by the appropriate current values.

AccountRequest class

This class is used to store data for accounts who want to get access to runs of type openaccess=true within the EMERGO toolkit.
Normally an account is added by an administrator account and the account applicant gets his login data from him. And then this account has to be added to a case run by a case run manager account, before he can play a case. So normally a case run isn’t publicly available.
But accounts will have to have access to public runs (openaccess=true), without interference of an administrator or case run manager account. Still an account has to be made for saving progress data. But the account applicant has to be able to sign in himself.
This class has the same properties as the Account class plus one extra, processed, which indicates if the account request to get access is processed or not.

3. Application layers

The EMERGO toolkit is built using different application layers, see figure 2.
We have view, control, model, persistence and database layers.
The idea of using different layers is that you can relatively easy exchange the implementation of one layer by another. And you have separation of concerns. For instance the model layer doesn’t have to ‘know’ how data is saved within the database. That’s the concern of the persistence layer.
The domain model classes described in the previous section are part of the model layer.
Due to the use of XML only the control and view layer have to be adjusted if a new component is added by an administrator account. And this only accounts for the player environment. For the other environments only the view layer has to be adjusted. Only if the administrator account uses a new type of tag within the XML definition, the control layer of the author environment has to be adjusted slightly, to render the new tag type and handle input for it. If a tree doesn’t satisfy as an author input mechanism (as is the case for the Google Maps component), more adjustments have to be made to the control layer of the author environment.

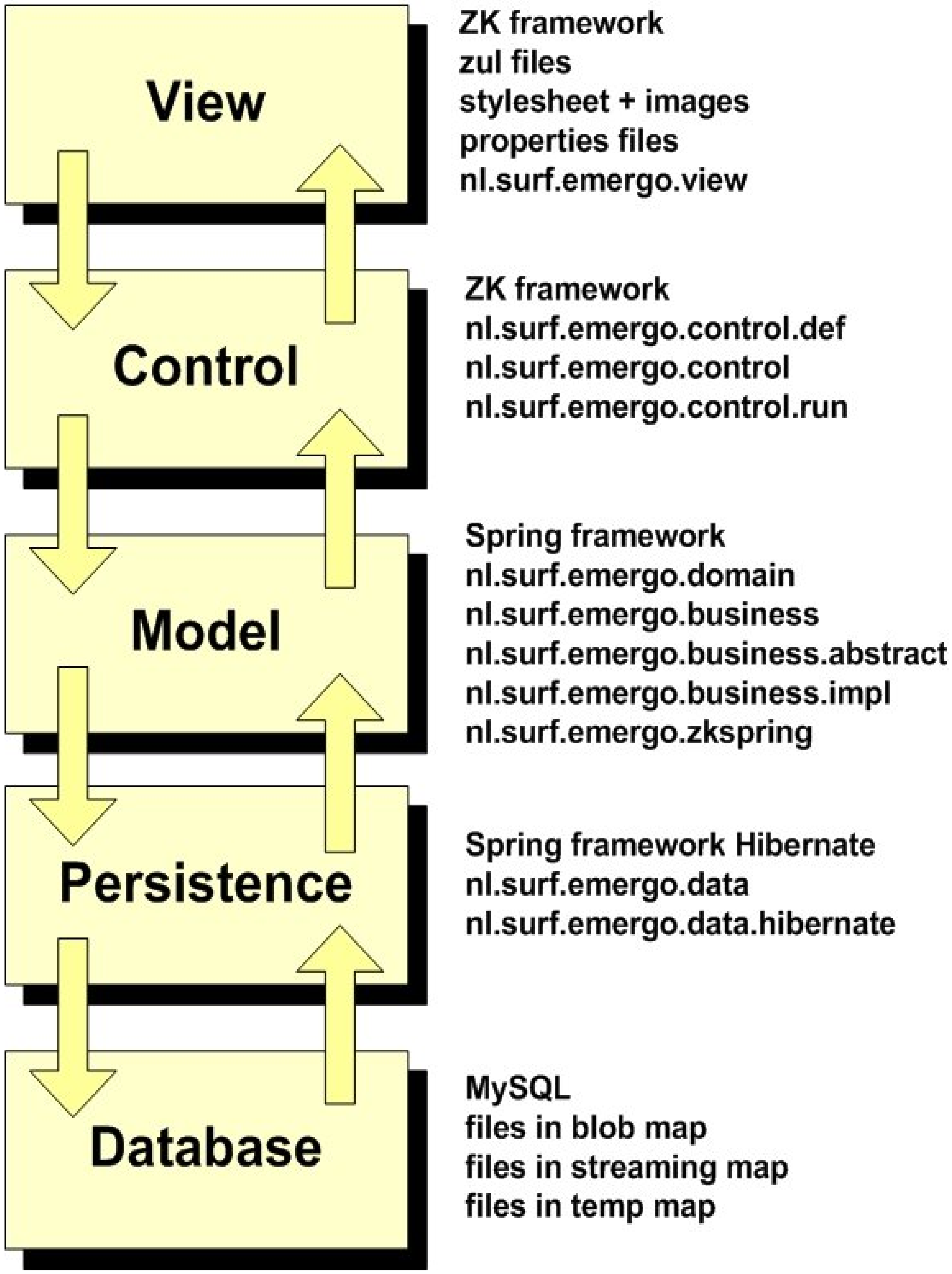


Figure 2: Application layers. Per layer is shown which OS software we used to build it, which packages we build and which other features were needed.

View layer

This layer exposes an application to its users. We used the ZK framework (http://www.zkoss.org) to implement this layer. This Ajax based Java framework offers a lot of interface components. These components are defined and used in so called zul files.

Zul files

The zul files are found in the root map of the application.
Each account role has its own specific zul files starting with the account role code, but there are also general zul files, like the login.zul file. The player environment has its own zul files starting with 'run'.
There are zul files which correspond with functional browser pages. But there also are zul files which are used for modal popup windows on a browser page, or for parts of the browser page. The latter mostly have 's_' in their name.

Style

Another part of the view layer is formed by the style sheet and images used in the toolkit. These files are found in the submap 'style'.

Package nl.surf.emergo.view

The view layer uses one class, VView, which is found in the Java package nl.surf.emergo.view. It contains logical constants for almost all zul pages, so the Java code doesn't reference to the zul pages directly. Further the class has some general methods useful for the view layer.
The VView class is used within other classes and by a file view.zs found in the submap 'view'. The extension zs stands for zscript, Java script interpreted by the zk framework. You can write your Java code in zscript but then you cannot debug it and the performance is less. When we use zscript we can include the view.zs file to access the VView class.

Properties files

The properties files are found in the submap 'WEBINF'. The file i3-label.properties contains Dutch label texts used by the toolkit. File i3-label_en_UK.properties contains English label texts. Adding another language can be done by putting another properties file with the same keys in the submap and adjusting the file login.zul, by adding a language option.

Control layer

This layer functions as an interface between the view and model layer. For this layer we also used the ZK framework. We extended the framework classes with our own classes. From within a zul file you can define a ZK component to use such a class. Or you can create the component completely in Java code, without defining it in a zul file.

The classes are arranged in a number of packages. We describe the packages beneath. For documentation on the classes itself we refer to the Java documentation.

Package nl.surf.emergo.control.def

This package contains classes extending ZK classes we used. Purpose was to be able to operate our own root classes if necessary, without having to change ZK code. All other control classes extend from these classes.

The package also contains two interfaces and one class used to implement the Observer design pattern. Several control classes use them to handle notifications: for instance notification of a button click to a container class, so upward. It isn't used the other way around (downward), for instance for notifying the button something happened outside it. Then an appropriate button method is called.

Package nl.surf.emergo.control

This package contains a lot of classes. It all are classes which are used within the environments for each account role: administrator, author, run manager, tutor and student environment. You recognize these classes because their name contains the role code. The player environment classes are found in package nl.surf.emergo.control.run.
Not all classes have the role code included in the name. There are some classes which are used for login and choosing an account role, so these classes are independent of role. Other classes are also used in the player environment or are used for more roles or possibly could be used by more roles.
The following important (groups of) classes can be recognized.

CControl class
It contains logical constants for the control layer and it has some general methods useful for the control layer.
The CControl class is used within other classes and by a file control.zs found in the submap 'control'. When we use zscript we can include the control.zs file to access the CControl class.

CLoginBtn class
This class is used to handle login.

CLoginAccount classes*
These classes are used to handle users who want to get a login account for public runs.

CAccountRole classes*
These classes are used for showing/choosing account roles.

CForbidden classes*
Used to redirect users to an error zul page if they try to access a zul page while they aren't logged in or a zul page meant for an account role they aren't assigned to.

CAdmTask classes*
Used for showing/choosing administrator tasks: manage accounts, components, cases or runs.

CAdmAccount classes*
Used for account management (crud). Accounts also can be imported using an XML file (see paragraph 4 ‘XML use’).

CAdmComponent classes*
Used for component management (crud).

CAdmCase classes*
Used for case management. Administrator account only can (de)activate a case or upload large (streaming) files for a case. Further a case developer account is responsible for case management.

CAdmRun classes*
Used for showing/choosing all runs.

CAdmRunGroupAccount classes*
Used for showing accounts participating in a run, adjusting case progress data for an account or starting player environment read-only for an account.

CCdeCase classes*
Used for case management (crud). Cases also can be copied, exported or imported. They are exported and imported as an IMS content package (see paragraph 4 ‘XML use’).

CCdeCaseRole classes*
Used for case role management (crud).

CCdeCaseComponent classes*
Used for case component management (crud). Case components also can be copied within the same case.

CXmlHelper class
This class is the ancestor of all XML helper classes. Helper classes are used to render XML content data within ZK components.

CCdeComponent classes*
Used for case component xmldata content (element) management (crud).

CContent classes*
These classes are used for rendering XML data content elements (CContentHelper), for rendering a content element edit window (CContentItemHelper), and for showing/handling popup menus for crud operations on elements and previewing content. Important classes are CContentItemMenuPopup, used for popup menus for all content elements, and CContentItemOkBtn, used to handle creation/adjustment of all content elements.
Class CContentItemHelper has to be adjusted, if an administrator account adds a new type of tag within the XML definition. This tag type has to be rendered and input for it has to be handled. See paragraph 5 ‘Adding a new EMERGO component’.

CCaseHelper class
This class is used for manipulating XML data of the ‘case’ case component, which holds all cross references between XML data of other case components.

CTree classes*
Almost all XML data content elements are rendered in a tree.

CGmaps and CGmarker classes
An exception is the CGmaps class. It is used to render XML data content elements as markers on Google Maps.

CCdeChooseFile classes*
Used for showing/choosing files.

CUpload classes*
Used for uploading different file types.

CCdeScriptMethodWnd class
This class is used to show a popup window for creating/adjusting a script condition, action, timer or counter.

CScript class
Used for ‘script’ case component, so within author environment, but also within environments like the playing environment.

CScript classes*
Used for creating/adjusting a part of a script condition or action. Conditions and actions can be built using logical operators.

CCdePreviewItem classes*
Used for creating a preview popup window containing preview item management (crud) and handling preview of the current case within the player environment.

CCrmRun classes*
Used for run management (crud).

CCrmRunGroupAccount classes*
Used for run group account management (crd). Run group accounts also can be imported using an XML file (see paragraph 4 ‘XML use’).

CCrmRunTeam classes*
Used for run team management (crud).

CCrmRunTeamRunGroup classes*
Used for run team run group management (crd).

CTutRun classes*
Used for showing/choosing runs a tutor account is assigned to.

CTutRunGroupAccount classes*
Used for showing student accounts participating in a run or for starting player environment read-only for a student account.

CTutRunGroupAccountEmails classes*
Used for showing all mails sent (within the player environment, no real mails) by student accounts participating in a run.

CTutRunGroupAccountTasks classes*
Used for showing all finished tasks for student accounts participating in a run.

CStuRunGroupAccount classes*
Used for showing/choosing runs a student account is assigned to. Choosing a run opens the player environment.

Package nl.surf.emergo.control.run

This package contains all classes used by the player environment.
A lot of classes extend from classes within the package nl.surf.emero.control.
In the class names you can recognize names of components which can be defined by case developers, such as locations, references or conversations, so these components have their own classes to render and interact with users. Mostly you see three classes: one container class with title, interaction area and close button, extending from CRunComponent, one tree class extending from CRunTree, and one helper class, extending from CRunComponentHelper, for rendering the tree.
The following important (groups of) classes can be recognized.

CRunWnd class
This is the main window of the player environment. It reads request parameters, for instance which student account starts the player environment, creates all initial components and restores progress data.
It also functions as a switchboard by handling notifications from interface elements, like buttons, to show other components.
It uses class CRunMainTimer to save progress regularly and to update the environment according to fired script rules, actions of other student accounts, or tutor or administrator interventions.

CRunMainTimer class
Saves case time within ‘case’ case component progress data, checks if timers defined in script should fire, checks pending actions, checks pending environment updates and saves pending progress data.

CRunTitleArea class
This class is used to show a title.

CRunChoiceArea class
Used for showing/handling choices. An area in which you can choose what to see in the run view area, see CRunViewArea. In this area you can choose which location to show, which empack component to show or which conversation question answer to play. The area is also used to show alerts and to make notes.

CRunLocationBtns class
This class is used to show all location hover buttons in the run choice area.

CRunEmpackActionBtns class
This class is used to show all empack component hover buttons in the run choice area.

CRunViewArea class

This class is used to show content of different components. Content cannot be shown simultaneously for multiple components.

CRunLocationView class
This class is used to show a location within the run view area.

CRunComponentView class
This class is used to show a component within the run view area.

CRunLocationAction classes*
Used to show a list of actions on a location, a user can choose from. If there is only one action it will automatically start when a student account enters a location. Otherwise a list of actions is shown. It is used for actions which aren't available in the run choice area, for instance playing a conversation or opening a component which isn't available on the empack.

CRunHover classes*
Used to render hover buttons and handle clicking on them.

CRunComponent and CRunComponentHelper classes
CRunComponent is the ancestor of all components shown within the run view area and of the 'tasks' component (shown in a popup window). CRunComponentHelper is used to help rendering the content of the component.

CRunContent classes*
These classes are used for rendering XML data content elements (CRunContentHelper), for rendering a content element edit window (CRunContentItemHelper), and for showing/handling popup menu for crud operations on elements. Important classes are CRunContentItemMenuPopup, used for popup menus for all content elements, and CRunContentItemOkBtn, used to handle creation/adjustment of content elements.
XML data is content created by a case developer merged with progress data.

CRunTree class
This class is the ancestor of all tree components within the player environment.

CRunGmaps classes*
These classes are used to render XML data content elements as markers on Google Maps.

CRunAlert class
This class is used to show an alert in the run choice area.

CRunAssessments classes*
These classes are used to show the assessments component in the run view area.

CRunConversation classes*
These classes are used to show a conversation video or text in the run view area and/or to show a question tree in the run choice area.

CRunQuestionsTree class
This class is used to show a question tree in the run choice area.

CRunGoogleMaps classes*
These classes are used to show the google maps component in the run view area.

CRunLogbook classes*
These classes are used to show the logbook component in the run view area.

CRunMail classes*
These classes are used to show the mail component in the run view area.

CRunNewMail classes*
These classes are used to show a new mail popup window on top of the run view area.

CRunNote class
This class is used to show/edit a contextualized note in the run choice area

CRunReferences classes*
These classes are used to show the references component in the run view area.

CRunTasks classes*
These classes are used to show the tasks component in a popup window on top of the run view area.

Model layer

This layer contains the business logic of an application. We used the Spring framework (http://www.springsource.org) to implement this layer. This Java framework is a so-called meta framework, integrating all kinds of other frameworks, for instance Hibernate (used within the persistence layer). It uses dependency injection. This means you can switch classes within a configuration file, without having to rebuild the application. Classes are injected in the application at runtime.

The classes are arranged in a number of packages. We describe the packages beneath. For documentation on the classes itself we refer to the Java documentation.

Package nl.surf.emergo.domain

This package contains all domain classes described in paragraph 2 ‘Domain model’. Every class has its interface class starting with ‘IE’. Domain classes start with ‘E’.

Package nl.surf.emergo.business

This package contains all interfaces for implemented manager classes, see package nl.surf.emergo.business.impl. Almost every domain class has its own manager interface.
Other interfaces are: IAppManager, IFileManager, IXmlManager, IXMLTag and IXMLTree. IAppManager contains application constants used throughout the toolkit. Further see package nl.surf.emergo.business.impl for implementations of these interfaces.

Package nl.surf.emergo.business.abstr

This package contains abstract classes which are needed to enable dependency injection by the Spring framework. You see an abstract class for every manager interface within package nl.surf.emergo.business.

Package nl.surf.emergo.business.impl

This package contains all manager classes, and XMLTree and XMLTag.
The manager classes are injected by the Spring framework using a configuration file applicationContext.xml, which is found in the 'WEB-INF' submap.

The following (groups of) classes can be recognized.

Domain manager classes
Manager classes manage instances of domain classes so do crud operations on these instances (using the persistence layer).
When an instance is deleted all referencing instances are deleted too using database cascade within MySQL (see 'Database layer'). So the manager class doesn't have to take care for this.
But we use XML data and from within XML data references can be made to blob instances. These blob instances aren't deleted in the database cascade, so we have to do it ourselves. That's why you find a method deleteBlobs in almost every manager class. It is called when an instance of a domain class is deleted.

Manager classes also do validation of a new or updated instance of a domain class by returning an error list, for instance an empty or not unique value. So this validation doesn't have to take place in the view or control layer. These layers take care of showing an appropriate error using returned error codes.

For performance reasons we decided to keep case components not only in the database but also in memory. So in some manager classes you find code to update the memory.

The BlobManager class doesn't only do crud operations on blob instances, but creates and deletes related files too, using the FileManager class.

AppManager class
This class contains some methods useful for the whole toolkit.

Filemanager class
This class does crd operations on files. It is used for blob files, streaming files and temporary files.

XmlManager class
This is a large class containing all kinds of methods for managing XML trees. Important to notice is that in a number of methods an XML definition is used to define allowed operations on the tree. See paragraph 4 'XML use' for XML definitions of all EMERGO components.

XMLTree class

This class is used to convert an XML string (at a certain url) to a tree of instances of XMLTag, where an XML definition can be used to populate XMLTag instances with their appropriate XML definition tag, see XMLTag class.
The class also is used to convert an XML tree back to an XML string.

XMLTag class
This class is used to store an XML tag. It has obvious properties as name, value and attributes, and methods to get and set them. But it also contains a reference to the parent XMLTag, a list of references to child XMLTags and a reference to a definition XMLTag, and methods to get and set them. The definition tag is useful because in it is defined which attributes and child types are possible, and default values for attributes. See paragraph 4 ‘XML use’ for XML definitions of all EMERGO components.

Package nl.surf.emergo.zkspring

This package only contains one class SSpring. As the name of the package indicates it is responsible for the interface between the ZK and the Spring framework. Through this class the control layer can access the model layer, using the manager classes described before.

This very large class is used by all toolkit environments, from adminstrator to player environment and maybe better should be divided into smaller classes. It is for instance responsible for reading and saving XML progress data, merging XML author and XML progress data, and checking and firing script rules.

This class also buffers a lot of player environment data as properties (so in memory), for better performance. The player window (instance of CRunWnd) uses one instance of the SSpring class, so if the player window closes, this memory is released again.

Persistence layer

This layer functions as an interface between the model and database layer. For this layer we used the Hibernate framework included in the Spring framework. We extended the Hibernate classes with our own classes to do crud operations on the database.

The classes are arranged in a number of packages. We describe the packages beneath. For documentation on the classes themselves we refer to the Java documentation.

Package nl.surf.emergo.data

This package contains all DAO (Data Access Object) interfaces for implemented DAO classes, see package nl.surf.emergo.data.hibernate. Every database table has its own DAO interface.

Package nl.surf.emergo.data.hibernate

The DAO classes manage instances of domain classes so do crud operations on these instances. They are implemented using Hibernate. Beneath the surface Hibernate synchronizes the database according to these crud operations.

When an instance is deleted all referencing instances are deleted too using database cascade implemented by MySQL (see ‘Database layer’). So the DAO class doesn’t have to take care for this.

The domain classes are mapped on the database through mapping files, found in submap ‘WEB-INF/classes’. For every domain class there is one mapping file mapping it to the database. These mapping files are used in the Hibernate configuration file applicationContext-hibernate.xml in submap ‘WEB-INF’. Within the same configuration file the DAO classes are injected by the Spring framework.

Database layer

MySQL is used for the database. The database connection is found in applicationContext-db.xml in the submap ‘WEB-INF’. See appendix 2 ‘EMERGO SQL’ for information about the SQL dump of the initial database.
All foreign keys within the database are defined as ‘ON DELETE CASCADE’, so when a record is deleted, MySQL deletes all referencing records too. So the application isn’t responsible for deleting them. Records in table blob aren’t referenced within the database but within XML data, the model layer takes care for deleting referenced blobs.

Not all data used by the toolkit is saved in the database. A part is saved as files on the application server, in submaps.

Blob submap

This map contains all privately uploaded files, either by case developer accounts or by student accounts.
Initially blob files were saved in the database as blob type field, that’s why they got there name. The idea was files were private within the database because they could not be accessed by other users. But every time a file was accessed it had to be created temporarily on the server. For performance reasons we decided to put them on the server. Every blob file is saved in a submap with a name equal to the blob id given to it in the database.

Streaming submap

This map contains all video and other large files like powerpoints. An administrator has to put them there. If put in this map all case developer accounts can use the files in their cases, the files are shared. But if the files are put in a submap with a name equal to the case id, the files can only be used within one case.

Temp submap

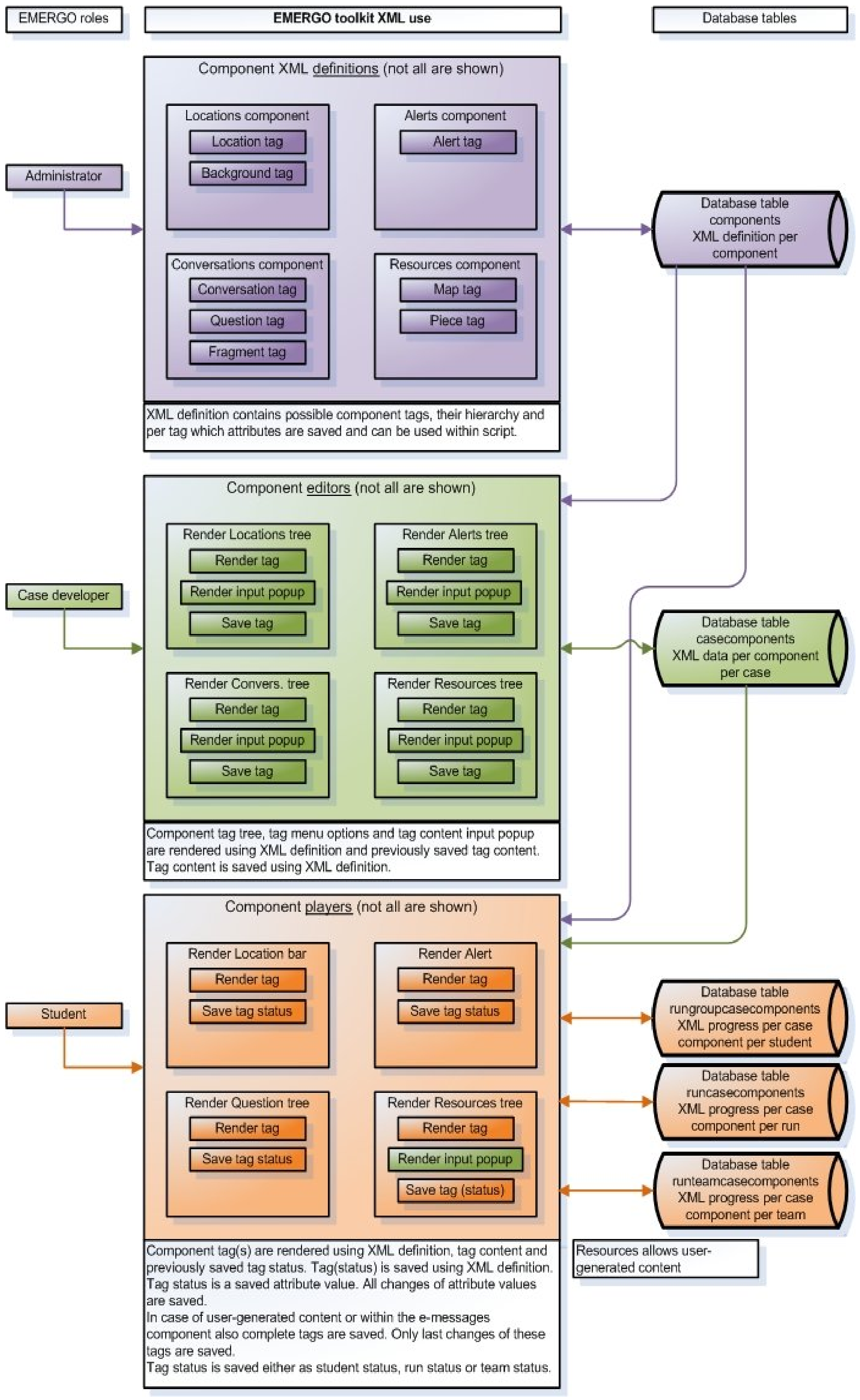
This map is used for temporary files, for instance when previewing a file. Files should be deleted regularly, files aren’t deleted automatically.

4. XML use

The EMERGO toolkit makes extensive use of XML.
It is used to:

- Define definitions for all EMERGO components, used by administrator account.
- Store content of these components, added by case developer accounts.
- Store progress data of student accounts using these components.
- Adjusting progress data of student accounts by other account types.
- Importing and exporting a case, used by a case developer account.
- Importing accounts and run group accounts, used by administrator and case run manager account.

The first three uses are illustrated in figure 3.



XML definition contains possible component tags, their hierarchy and per tag which attributes are saved and can be used within script.

Component tag tree, tag menu options and tag content input popup are rendered using XML definition and previously saved tag content. Tag content is saved using XML definition.

Component tag(s) are rendered using XML definition, tag content and previously saved tag status. Tag(status) is saved using XML definition. Tag status is a saved attribute value. All changes of attribute values are saved.
In case of user-generated content or within the e-messages component also complete tags are saved. Only last changes of these tags are saved.
Tag status is saved either as student status, run status or team status.

Resources allows user-generated content

Figure 3. Illustration of XML use by administrator, case developer and student roles. XML is used for XML definitions, data and progress. Administrator role is responsible for XML definitions. Case developer role is responsible for XML data. XML definitions are used to render the component editors. Student role is responsible for XML progress. XML definitions and XML data are used to render the component players.

XML definitions

Every EMERGO component is defined by an XML definition. It is saved in the database in table 'components'.
The definition defines which component properties and type of content elements can be entered by a case developer and in which way the content elements are structured so the author environment can render a component specific editor. A component property is for instance if the component is accessible by a student account within the player environment. Content elements are for instance all maps and pieces used within the 'references' component.
The XML definition also defines which component property and content element (property) changes are saved (including a timestamp), for a student account within the player environment and which can be read and changed by rules within a 'scripts' component.
Properties are simple boolean or integer values. For instance opened=true or score=10. Content elements only contain strings.
We don't have an XML schema yet for the XML definitions.

We start describing the definitions format with the definition of the 'locations' component because it is a simple component and the first component a case developer account will have to fill.
Afterwards we describe the other components more or less in filling order and increasing complexity. We only describe component aspects which are not described before.

The 'locations' component

Within this component is defined which locations a student account can visit within the player environment.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible"/>
    <status present="true" accessible="true" selected="false" opened="false"/>
  </component>
  <content type="root" childnodes="location" maxid="" preview="true">
    <location id="" type="node" childnodes="background" key="pid" singleopen="true" preview="true">
      <pid type="line" private="true"/>
      <name type="line" notempty="true"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,accessible,opened"/>
      <status present="true" accessible="true" opened="false" selected="false"/>
    </location>
    <background id="" type="node" childnodes="" key="pid" singleopen="true">
      <pid type="line" private="true"/>
      <picture type="picture"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="opened"/>
      <status opened="false"/>
    </background>
  </content>
</data>
```

The <u>data</u> element is the root element for all XML data within EMERGO.
Within the XML definitions the data element always has two child elements: <u>component</u> and <u>content</u>.
The <u>component</u> element defines possible component properties.
The <u>content</u> element defines possible content elements, their hierarchy, their possible content child elements and their possible properties.

The component element
This element is of <u>type</u> “root” to distinguish it from its child elements.
The <u>initialstatus</u> element defines which component properties can be set by a component author. In this case attributes present and accessible. See status tag for an explanation of these attributes.
The <u>status</u> tag defines which component property changes are saved (including a timestamp) for a student account within the player environment, and which initial values they have. These values can be partly overrruled by the component author (see initialstatus element).
<u>Present</u> means a component is used within the player environment. Present=”false” means a student account doesn’t see an option to open the component. (For instance for a ‘scripts’ component, it means the component isn’t used (temporarily)).
<u>Accessible</u> means the option to open the component isn’t ‘grey’, so it is clickable. Accessible=”false” means the option is visible but not clickable.
<u>Selected</u> means a student account has selected the option to open the component.
<u>Opened</u> means the component is opened by the student account. Of course selecting the option directly leads to opening the component. We made a distinction because selected only can become true, while opened can get true or false (when something is closed).

All properties are simple booleans or integers. Using the ‘scripts’ component these properties can be checked and set.
See appendix 1 ‘Overview XML elements’ for a complete list of possible properties.

The content element
This element is of type “root” to distinguish it from its child elements.
The <u>childnodes</u> attribute defines all possible child elements of type ‘node’, comma separated. In this case ‘location’ elements are the only possible ‘node’ child elements of the content element, meaning an author can only add this kind of child elements to the content element.
The <u>maxid</u> attribute is used to hold the id of the last added ‘node’ child of the whole XML tree. Every new ‘node’ child, added by a component author, gets an id of (maxid + 1). This guarantees every ‘node’ child gets his unique id within the XML data entered by the component author.
The <u>preview</u> attribute defines if a component can be previewed by a component author, meaning the component is shown within the player environment. Not all components can be previewed. For instance the ‘scripts’ component has no counterpart within the player environment, so cannot be previewed.

The location element

The location element defines a location a student account can visit within the player environment. It is of type 'node' distinguishing it from elements of type 'root' and child elements of other types.
Every 'node' element gets a unique id, een integer, indicated by attribute id.
The childnodes attribute defines all possible 'node' child elements of the location element, comma separated. In this case 'background' elements (see further) are the only possible 'node' childs of the location element, meaning an author can only add these childs to the location element.
The key attribute indicates which child elements, comma separated, of the location element are functioning as a unique key for the element. Why using a key if we already have an id attribute? The key is a readable string which can be used, for instance within the 'scripts' component, to show to the author when he is creating relations between 'node' elements of different components.
The singleopen attribute indicates that only one 'location' element can have state opened="true". A component author can only set one location opened, meaning a student default enters this location, when starting the player environment. And a student account can be only at one location at the same time. If he enters another location, the opened state of the previous location is set to "false".
The preview attribute again defines that the component can be previewed, but then with the selected 'location' element preopened.

The child elements of the 'location' element define which content a component author can add to the element. The author environment uses these tags to render an appropriate edit popup for the 'location' element.
The pid element means 'private id'. It is meant to create a readable id for the 'location' element. It has type 'line' meaning it is a single line string. See for an overview of all types appendix 1 'Overview XML elements' (If a new component definition requires a new type, the author environment has to be adjusted accordingly. See paragraph 5 'Adding a new EMERGO component'.) Attribute private indicates if element content is only visible for a component author. Private="false" means a student account can see the content.
The name element is the name of the location visible to the student account. It cannot be empty as indicated by notempty="true". Two locations can have the same name but always have a different pid (because it used as key).
The explanation element is meant for a component author to make private notes about the 'location' element. It is of type 'text' meaning a multiline plain text string can be entered. Every 'node' element definition has an 'explanation' child element.

The initialstatus element defines which location properties can be set by a component author. In this case attributes present, accessible and opened. See status tag for an explanation of these attributes.
The status tag defines which location property changes are saved for a student account within the player environment, and which initial values they have. These values can be partly overrruled by the component author (see initialstatus element).
Present means a student account sees an option to open the location, for instance a button. Present="false" means the button is absent.
Accessible means the option to open the location isn't 'grey', so the button is clickable. Accessible="false" means the button is 'grey'.
Selected means a student account has selected the option to open the location.
Opened means the location is opened by the student account. It can be opened by default, if a case author has given it a value "true".

The background element

The background element defines a background (picture) of a location which a student account sees within the player environment. There can be multiple backgrounds. Using the ‘scripts’ component a case developer account can switch a background by setting property ‘opened’.
The childnodes attribute is empty. Backgrounds don’t have ‘node’ child elements.
The preview attribute misses. Backgrounds are part of a location element and cannot be previewed separately.

A new child element is picture. This element is used to upload a background picture. Type is ‘picture’. The uploaded picture is shown in the edit popup for the background element. The picture has a relation with a record within the database table blobs. The value of the picture element is set to the id of this record (bloid).

The status tag only defines an opened property. A background element doesn’t have present, accessible and selected properties, because a student account cannot select it directly. It is indirectly selected when a location element is selected.
Opened means the background is shown to the student account if he opens the corresponding location. It can be opened by default, if a case author has given it a value “true”.

The ‘tasks’ component

This component can be used as a case consists of tasks. It functions as a task overview to student accounts. They can check tasks themselves or tasks can be checked using a ‘scripts’ component.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible,taskscheckable"/>
    <status present="true" accessible="true" taskscheckable="false" selected="false" opened="false" />
  </component>
  <content type="root" childnodes="task" maxid="" preview="true">
    <task id="" type="node" childnodes="task" key="name">
      <name type="line"/>
      <hovertext type="simplerichtext"/>
      <time type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="required,present,accessible,outfolded"/>
      <status required="true" present="true" accessible="true" outfolded="false" outfoldable="true"
selected="false" opened="false" finished="false" canceled="false"/>
    </task>
  </content>
</data>
```

The component element

A new property is taskscheckable indicating if a student account can check tasks himself. If not, tasks have to be checked using a ‘scripts’ component.

The task element

The task element defines a task within the ‘tasks’ component.
The childnodes attribute indicates the tasks can have subtasks as ‘node’ child elements.
A new child element is hovertext. It is meant to enter a hover text shown above the task. Its type is ‘simplerichtext’, so simple markup is possible.

Another new child element is time. It's an indication for student accounts how long they approximately will be preoccupied with the task.
New attributes within the status element are required, finished and canceled.
Required is used to indicate to student accounts if a task is required or optional.
Finished is used when a task is checked.
And canceled is used when a student account checks a task, but a 'scripts' component finds out it isn't finished yet. Canceled is then set by the 'scripts' component to save within progress data this event happened.
Because tasks can have subtasks new attributes outfolded and outfoldable are available.
Outfolded indicates if a map is outfolded or not.
Outfoldable means a map can be outfolded or not.

The 'empack' component

This component has to be added to a case if empack components, like the 'references' component should be used.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible"/>
    <status present="true" accessible="true" selected="false" opened="false"/>
  </component>
  <content type="root" childnodes="" maxid="" preview="true">
  </content>
</data>
```

The content element

The content element doesn't have any childnodes, so no content elements can be added. Which components show up on the empack component is managed by the administrator account: their 'parent' is set to the empack component.

The 'references' component

Using this component, references can be made available to student accounts. This component by default is situated on the 'empack' component.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible,extendable,ownership,shared,sharedbyteam"/>
    <status present="true" accessible="true" extendable="false" shared="false" ownership="false"
sharedbyteam="false" selected="false" opened="false" started="false"/>
  </component>
  <content type="root" childnodes="map,piece" maxid="" preview="true">
      <reflocations type="ref" reftype="locationtags_referencesroot" refcomp="locations" reftag="location"
multiple="true"/>
    <map id="" type="node" childnodes="map,piece" key="name">
      <name type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,outfolded"/>
      <status present="true" outfolded="false" outfoldable="true" selected="false" opened="false" />
    </map>
    <piece id="" type="node" childnodes="" key="name">
```

```
      <name type="line"/>
      <blob type="blob" blobtypes="database,int_url,ext_url"
mediatypes="document,picture,audio,video,url,news,note,other" blobtype="" mediatype=""/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,accessible"/>
      <status present="true" accessible="true" selected="false" opened="false" finished="false"
deleted="false"/>
    </piece>
  </content>
</data>
```

The component element

New properties are extendable, ownership, shared and sharedbyteam. These properties are available to certain components, for which they are relevant. For a ‘references’ component it is relevant to add one’s own content and to possibly share it with others.
Extendable means a student account can extend the content of the component with his content (user generated content) within the player environment. He even can adjust content entered by a component author.
Ownership means one can only change its own added content within the player environment. So component author content cannot be changed.
Shared means content is shared by all student accounts within the current run, so they can see each other extensions. If ownership=”false” they can change each others content, otherwise only their own content.
Sharedbyteam means content is shared by all student accounts within the current run team.
A component cannot be both shared and sharedbyteam so these options exclude each other.

The content element

The childnodes attribute indicates the content element can have maps and/or pieces as ‘node’ child elements.

The reflocations element

The reflocations element defines a ‘references’ component to be only available on certain locations. Default this component is available on the ‘empack’ component, but there are situations where you only want it to be available at one or more locations. If one or more locations are defined, the ‘references’ component isn’t available anymore on the ‘empack’ component, so the ‘parent’ relation between components is overruled.
The element is of type ref meaning it defines a relation between content of two different components. The relation is saved within the content of the ‘case’ component.
The reftype attribute defines the type of relation. In this case a number of location tags (elements) within the ‘locations’ component are related to the root (content) element of the ‘references’ component. Reftype always indicates a 1x1 or nx1 relation. So here it has value ‘locationtags_referencesroot’, indicating multiple locations related to one root. The reftype is saved within the content of the ‘case’ component, and functions as a sort of cross table id. See section ‘XML within the ‘case’ component’.
The refcomp attribute defines to which component the current component is related, in this case a ‘locations’ component. If there are more components of a certain type, the component author has to choose one.
The reftag attribute defines to which type of tag (element) the relation is made, in this case to ‘location’ elements.

Multiple defines if the relation is multiple, so if multiple locations can be chosen or only one.

The map element
The map element defines a map which can contain pieces and/or submaps.
The childnodes attribute indicates the map element can have maps and/or pieces as 'node' child elements.
The preview attribute misses. Maps are not previewed separately but always shown within the 'references' component.
For a map attributes outfolded and outfoldable are relevant, so can be set.
The accessible attribute is missing within the status element because the content of a map can be made inaccessible.

The piece element
The piece element defines a piece within the 'references' component.
The childnodes attribute is empty. Pieces don't have 'node' child elements.
The preview attribute misses. Pieces are not previewed separately but always shown within the 'references' component.

A new child element is blob. It defines a reference to a blob. Type is 'blob'. The blob can be previewed within the edit popup for the blob element. The blob has a relation with a record within the database table blobs. The value of the blob element is set to the id of this record (bloid).
There are three types of blob indicated by attribute blobtypes. See the 'Blobs' section within paragraph 2 'Domain model'. A component author has to choose one type. The chosen type is saved in attribute blobtype.
The attribute mediatypes is used to type the piece so student accounts see the kind of piece. The chosen type is saved as attribute mediatype. If not chosen mediatype 'other' is shown to student accounts.

A new attribute within the status element is deleted. It is used when a student account deletes a piece in a 'references' component which can be extended with user generated content. The piece isn't removed from progress data to be able to see what student accounts exactly have done. Another reason for not deleting is, is that the context of user generated content has to be preserved, to be able to show it correctly.
The finished attribute within the status element isn't set because within the player environment we cannot detect closing a piece.

The 'conversations' component

This component is used to define conversations within a case. Conversations can be a simple video shown on some location, but also can be interviews where student accounts can ask questions and subquestions and a appropriate video is shown.
A conversation takes place on one or more locations. When a student account enters a location the conversation will start automatically. If there are more conversations available or other so called location actions, a popup window is shown to choose one location action.
If a conversation is closed it can be reopened by clicking on the location background.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
```

```
  <component type="root">
    <initialstatus attributes="present"/>
    <status present="true" selected="false" opened="false" status=""/>
  </component>
  <content type="root" childnodes="conversation" maxid="">
    <conversation id="" type="node" childnodes="background,fragment,map,question" key="pid"
preview="true">
      <pid type="line" private="true"/>
      <name type="line"/>
      <reflocations type="ref" reftype="locationtags_conversationtags" refcomp="locations"
reftag="location" multiple="true"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status present="true" selected="false" opened="false" finished="false" started="false"/>
    </conversation>
    <background id="" type="node" childnodes="" key="pid" singleopen="true">
      <pid type="line" private="true"/>
      <picture type="picture"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="opened"/>
      <status opened="false"/>
    </background>
    <map id="" type="node" childnodes="map,question" key="name">
      <name type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,outfolded"/>
      <status present="true" outfoldable="true" outfolded="false" selected="false" opened="false"/>
    </map>
    <question id="" type="node" childnodes="fragment" key="text">
      <text type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,outfoldable"/>
      <status present="true" outfoldable="false" outfolded="true" selected="false" opened="false"/>
    </question>
    <fragment id="" type="node" childnodes="map,question" key="pid" singleopen="true">
      <pid type="line" private="true"/>
      <blob type="blob" blobtypes="database,int_url,ext_url" blobtype=""/>
      <description type="simplerichtext" private="true"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="opened"/>
      <status present="true" selected="false" opened="false" finished="false"/>
    </fragment>
  </content>
</data>
```

The component element

Present="false" means the component isn't used within the player environment (temporarily).

The conversation element

The conversation element defines a conversation within the 'conversations' component.
The childnodes attribute indicates a conversation can have backgrounds, fragments, maps and/or questions as 'node' child elements.
Backgrounds are used to show an image when no video runs. Just like within the 'locations' component there can be more backgrounds, and one of them will be active (can be changed using 'scripts' component).
Fragments define the video (or other formats) fragments. If defined directly as child of the conversation element, these fragments function as start fragments, meaning one of them will play if the conversation is opened. There can be multiple start fragments, and one of them will be active (can be changed using 'scripts' component).

Maps can be used to group questions in categories.
And questions are used to ask questions. If there is no start fragment, the questions are presented to student accounts when a conversation is opened. Otherwise a student account has an option to show them, when he is ready.

The reflocations was handled before, see the ‘references’ component. It defines on which locations a conversation is available. The only difference is the value of reftype. Now it indicates a number of location tags (elements) within the ‘locations’ component are related to conversation tags (elements) of the ‘conversations’ component.

A new attribute within the status element is <u>started</u>. It is set when a conversation is started by a rule within the ‘scripts’ component.
Finished is set when a student account closes the conversation.

The background element
This element is used to thow a background image, when no video is playing. It has the same definition as within the ‘locations’ component.

The map element
This element is used to group questions.
It almost has the same definition as within the ‘references’ component.
The value of the childnodes attribute is different. Other map elements and question elements are allowed as ‘node’ child elements.

The question element
The <u>question</u> element is used to show a question in plain text. A question can only have a fragment as ‘node’ child element. Attributes outfolded and outfoldable are present because a question can have follow-up questions. These are child elements of the fragment element played when the question is asked.

The fragment element
The <u>fragment</u> element is used to play a fragment. It can have maps and/or questions (ie follow-up questions) as ‘node’ child elements
The blob child element accepts all kind of blobs, so you can even use the fragment to play a flash animation or game, or to show an image, a website or a youtube movie.
A new child element is <u>description</u> of type simplerichtext. It can be used to show a text when no blob is available yet. So the case can be tested before video has been shot. Or a fragment can be added afterwards without having to shoot video.
The finished attribute within the status element isn’t set because within the player environment we cannot detect closing a fragment.

The ‘mail’ component

This component is used to define predefined mails which are sent to student accounts using the ‘scripts’ component. The predefined mails can also be sent be a tutor account to interfere in the case scenario.
The component is also used to define mail templates for mails to be sent by student acounts.
The component is situated on the ‘empack’ component.

XML definition:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<data>
  <component type="root">
    <initialstatus attributes="present,accessible"/>
    <status present="true" accessible="true" selected="false" opened="false" started="false"/>
  </component>
  <content type="root" childnodes="map" maxid="" preview="true">
    <map id="" type="node" childnodes="map,outmailpredef,outmailhelp,inmailpredef,inmailhelp"
key="name">
      <name type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,outfolded"/>
      <status present="true" accessible="true" outfoldable="true" outfolded="false" selected="false"
opened="false"/>
    </map>
    <inmailpredef id="" type="node" childnodes="attachment" key="title" multiple="true">
      <title type="line"/>
      <refsendernpc type="ref" reftype="caseroles_npc_mailtags" refcomp="" reftag="" npc="true"/>
      <richtext type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status present="true" selected="false" opened="false" sent="false" version="0"/>
    </inmailpredef>
    <inmailhelp id="" type="node" childnodes="" key="pid" multiple="true">
      <pid type="line" private="true"/>
      <refsendernpc type="ref" reftype="caseroles_npc_mailtags" refcomp="" reftag="" npc="true"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status childnodes="attachment" present="true" selected="false" opened="false" sent="false"
version="0">
        <title type="line"/>
        <richtext type="simplerichtext"/>
      </status>
    </inmailhelp>
    <outmailpredef id="" type="node" childnodes="" key="title" multiple="true">
      <title type="line"/>
      <refreceiversnpc type="ref" reftype="caseroles_npc_mailtags" refcomp="" reftag="" multiple="true"
npc="true" choiceempty="true"/>
      <refreceiverspc type="ref" reftype="caseroles_pc_mailtags" refcomp="" reftag="" multiple="true"
npc="false" choiceempty="true"/>
      <inbox type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status childnodes="attachment" present="true" selected="false" opened="false" sent="false"
version="0" deleted="false">
        <richtext type="simplerichtext"/>
      </status>
    </outmailpredef>
    <outmailhelp id="" type="node" childnodes="" key="pid" multiple="true">
      <pid type="line" private="true"/>
      <refreceiversnpc type="ref" reftype="caseroles_npc_mailtags" refcomp="" reftag="" multiple="true"
npc="true" choiceempty="true"/>
      <refreceiverspc type="ref" reftype="caseroles_pc_mailtags" refcomp="" reftag="" multiple="true"
npc="false" choiceempty="true"/>
      <inbox type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status childnodes="attachment" present="true" selected="false" opened="false" sent="false"
version="0" deleted="false">
        <title type="line"/>
        <richtext type="simplerichtext"/>
      </status>
    </outmailhelp>
    <attachment id="" type="node" childnodes="" key="name">
      <name type="line"/>
      <blob type="blob" blobtypes="database,int_url,ext_url"
mediatypes="document,picture,audio,video,url,news,note,other" blobtype="" mediatype=""/>
```

```
      <explanation type="text" private="true"/>
      <initialstatus/>
      <status opened="false"/>
    </attachment >
  </content>
</data>
```

The component element
A started attribute within the status element is set when the ‘mail’ component is started by a rule within the ‘scripts’ component.

The content element
Notice only maps are allowed as ‘node’ child elements, so all mails must be defined within maps. Normally one inbox and one outbox map will do.

The map element
This element is used to group mails.
It almost has the same definition as within the ‘references’ component.
The value of the childnodes attribute is different. Other map elements and mail elements are allowed as ‘node’ child elements.

The inmailpredef element
The inmailpredef element is used to define a mail which could be sent to a student account either by a script rule or a tutor account. The tutor account can adjust the body text.
As you see the childnodes attribute indicates attachments can be defined. The multiple attribute indicates there can be multiple instances in progress data for this (template) mail. The same mail can be sent multiple times.

A new child element is refsendernpc of type ref. Because this type of mail is sent by ‘the system’, a reference to one non player character (npc) case role, which sent the mail, has to be made. Reftype indicates this relation. Refcomp and reftag are empty because a case role is referenced and these aren’t defined within XML. Attribute npc indicates the relation is with npc case roles.
Another new child element is richtext of type simplerichtext. It is meant to enter the body text of the mail.

New attributes within the status element are sent and version.
Sent is set when the mail is sent. Version indicates a version number for the mail. When the mail is sent the second time, version will be 2 within the progress data. Every mail instance is saved separately within progress data.

The inmailhelp element
The inmailhelp element is used to define a mail which could be sent to a student account by a tutor account. It has no title (that’s why the pid child element is necessary) or body. These have to be filled in by the tutor account. Also childnodes attribute is empty because attachments have to be added by the tutor account.
It is sufficient to define one inmailhelp element for all help mails, because all mail data has to be filled in.

New is that we see the status element allows ‘node’ child elements: in this case, attachments which will be added by a tutor account.
Also the status element has two child elements, title and richtext, indicating the tutor account has to fill in these data.

The outmailpredef element

The outmailpredef element is used to define mails that can be sent by a student account. Mail title is fixed (for instance 'finished report on …' or 'have interviewed …'), but body and attachtments can be added by a student account. So you see childnodes attribute is empty. Mails can be sent multiple times.

New child elements are refreceiversnpc and refreceiverspc. The mail can be sent to multiple non playing or playing characters which are defined within these elements. A student account thus cannot choose the receivers, but has to see who is getting the mail. A new attribute is choiceempty, indicates a choice isn't required.
Another new child element is inbox. It is used to fill in the name of the map in which the mail has to appear for playing character receivers. If the map exists within the 'mail' component it is put in there. Otherwise a map with the specified name is added under the root element.

Again we see the status element allows child attachments, to be added by a student account.
The deleted attribute is present, but a student account isn't yet able to delete a mail.
The status element has one child element, richtext, the body to be added by a student account.

The outmailhelp element

The outmailhelp element is used to define a mail which could be sent by a student account. It has no title (that's why the pid child element is necessary) or body. These have to be filled in by the student account. Also childnodes attribute is empty because attachments have to be added by the student account.
It is sufficient to define one outmailhelp element for all help mails, because all mail data has to be filled in.

The 'googlemaps' component

The googlemaps component is used to add markers to Google Maps and make them available to student accounts. The markers can contain text and a reference to a sourde. This component is situated on the 'empack' component.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible,extendable,ownership,shared,sharedbyteam"/>
    <status present="true" accessible="true" extendable="false" shared="false" ownership="false"
sharedbyteam="false" selected="false" opened="false" started="false" status=""/>
  </component>
  <content type="root" childnodes="piece" maxid="" preview="true">
    <piece id="" type="node" childnodes="" key="name">
      <name type="line"/>
      <blob type="blob" blobtypes="database,int_url,ext_url" blobtype=""/>
      <exlatitude type="line" hidden="true"/>
      <exlongitude type="line" hidden="true"/>
      <description type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,accessible"/>
      <status present="true" accessible="true" selected="false" opened="false" finished="false"
status=""/>
    </piece>
  </content>
</data>
```

The component element
Properties extendable, ownership, shared and sharedbyteam are avaible, just like for the ‘references’ component.

The content element
The childnodes attribute indicates the content element can only have pieces as ‘node’ child elements. Of course there are no maps containing pieces on Google Maps.

The piece element
The piece element defines a piece within the ‘googlemaps’ component.
It has almost the same definition as the piece element within the ‘references’ component. There are three extra child components.

Two new child elements are <u>exlatitude</u> and <u>exlongitude</u>, defining the GPS location of the piece element (the Google Maps marker). Attribute <u>hidden</u> is new. It indicates GPS location isn’t shown to student accounts directly.
Another child element is <u>description</u> of type simplerichtext, containing the text shown when the marker is clicked. Above this text the name of the piece is shown and if a blob is added, this name is a link to this blob.

The ‘alerts’ component

The alerts component is used to show textual alerts to student accounts. Alerts can for instance be hints to hurry up or to go somewhere. Student accounts cannot open the alerts component themselves. Alerts are shown to them using the ‘scripts’ component.

XML definition:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes=""/>
    <status present="true" selected="false" opened="false"/>
  </component>
  <content type="root" childnodes="alert" maxid="">
    <alert id="" type="node" childnodes="" key="pid">
      <pid type="line" private="true"/>
      <modal type="boolean">true</modal>
      <richtext type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status present="true" selected="false" opened="false" sent="false" version="0"/>
    </alert>
  </content>
</data>
```

The component element
Because alerts are sent by script rules it doesn’t make sense to let component authors change the present property.

The content element
The childnodes attribute indicates the content element can only have alerts as ‘node’ child elements. Of course there are no maps containing alerts.

The alert element
The alert element defines an alert.
There is one new child element <u>modal</u>. It isn't used anymore. All alerts are non-modal.
Notice the status element contains attributes sent and version, just as mail elements. Alerts are sent and can be sent multiple times.

The 'note' component

The note component is used to let a student account make contextualised notes, meaning the context in which the note was made is saved as a string within progress data. A component author could add some notes already available to a student account. For instance in his logbook (see logbook component).

XML definition:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible"/>
    <status present="true" accessible="true" selected="false" opened="false"/>
  </component>
  <content type="root" childnodes="" maxid="" preview="true">
    <note id="" type="node" childnodes="" key="pid">
      <pid type="line" private="true"/>
      <text type="text"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status present="true" selected="false" opened="false"/>
    </note>
  </content>
</data>
```

The component element
Because the note component is opened by a student account it contains present and accessible attributes.

The content element
The childnodes attribute indicates the content element can only have notes as 'node' child elements.

The note element
The <u>note</u> element defines the structure of a note.
Child element pid is used to store the context of the note and child element text for the note text.

The 'logbook' component

The logbook component is used to show an overview of all notes taken using the note component. So the two components cannot live without one another, so to speak. This component is situated on the 'empack' component.

XML definition:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
```

```
    <initialstatus attributes="present,accessible"/>
    <status present="true" accessible="true" selected="false" opened="false" started="false" status=""/>
  </component>
  <content type="root" childnodes="" maxid="" preview="true">
     <refnote type="ref" reftype="notecomponent_logbookroot" refcomp="note" reftag=””/>
  </content>
</data>
```

The content element

The childnodes attribute is empty because the logbook component gets its content from the note component.
Therefore it has a new child element refnote. This defines which note component is used by the logbook component, as indicated by reftype, refcomp and reftag.

The ‘items’ component

This component is used to define assessment items to be used by the ‘assessments’ component. Items are only visible to student accounts through an assessments component. The items component cannot be opened by a student account. It functions as an item container.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus/>
    <status/>
  </component>
  <content type="root" childnodes="item" maxid="">
    <item id="" type="node" childnodes="alternative,feedbackcondition,piece,refpiece" key="pid">
      <pid type="line" private="true"/>
      <type type="singleselect" items="mkea" selecteditem=""/>
      <richtext type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus/>
      <status/>
    </item>
    <alternative id="" type="node" childnodes="piece,refpiece" key="pid">
      <pid type="line" private="true"/>
      <richtext type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status present="true" selected="false" opened="false"/>
    </alternative>
    <feedbackcondition id="" type="node" childnodes="piece,refpiece" key="pid">
      <pid type="line" private="true"/>
      <conditionstring type="condition" subtype="itemfeedback"/>
      <correct type="boolean"/>
      <score type="singleselect" items="0,1,2,3,4,5,6,7,8,9,10" selecteditem="">0</score>
      <richtext type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status present="true" selected="false" opened="false"/>
    </feedbackcondition>
    <piece id="" type="node" childnodes="" key="name">
      <name type="line"/>
      <blob type="blob" blobtypes="database,int_url,ext_url"
mediatypes="document,picture,audio,video,url,news,note,other" blobtype="" mediatype=""/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status present="true" selected="false" opened="false" finished="false"/>
    </piece>
    <refpiece id="" type="node" key="name">
      <name type="line"/>
      <ref type="ref" reftype="piecetag_itemstag" refcomp="references" reftag="piece"/>
      <initialstatus/>
      <status selected="false" opened="false"/>
    </refpiece>
  </content>
</data>
```

The component element
Because the items component is opened by a student account or script it has no status element attributes.

The content element
Only possible 'node' child elements are items.

The item element
The item element is used to enter a question item. Possible 'node' child elements are alternative, feedbackcondition, piece and refpiece.
Alternatives of course are alternatives to choose.
Feedback conditions are rules which determine if an answer is correct and can show an appropriate feedback.
Pieces and references to pieces are shown as hyperlinks when the item text (given by richtext element) is shown.

A new child element is type of type 'singleselect'. It is meant to select the item type. Attribute items contains the item types to select, comma separated. Only one type (mkea=multiple choice) is available yet. Attribute selecteditem is used to store the selected item type.

The status element doesn't have any attributes, item attributes are saved within refitem 'node' element within the assessments component.

The alternative element
The alternative element is used to define a choosable alternative. Alternatives can have 'node' child elements piece and refpiece, so every alternative can have its own list of hyperlinks to pieces.

The status element does have attributes, because they can be changed using script.

The feedbackcondition element
The feedbackcondition element is used to define a condition for a given set of answers. For instance if you have three alternatives and one is correct and the other two not, you can define one condition for the correct alternative and one for the two others, so two conditions will do. The condition has an indication if the given answer is correct, a score and a feedback text to show.
Feedback conditions can have 'node' child elements piece and refpiece, so every feedback can have its own list of hyperlinks to pieces.

New child elements are conditionstring, correct and score. Richtext is used for the feedback text.
The element conditionstring is used to define when the feedback condition is valid. For instance alternative a or alternative b is chosen. As you see composite conditions can be made. You can even create a condition like: if alternative a is chosen and

location x is visited. The conditionstring element has type condition. It is the same element as used within the condition element within the ‘scripts’ component, see there. A new attribute is subtype. It is meant to distinguish this conditionstring from the one used within the ‘scripts’ component.
The element correct is used to indicate if the chosen alternative(s) is(are) correct.
The element score indicates the corresponding score. A correct alternative could have a score 1, the others score 0. A score can be chose from 0..10. Default value is 0.

The piece element
The piece element defines a piece, belonging to an item, an alternative or a feedback condition, within the ‘items’ component.
It has same definition as the piece element within the ‘references’ component.

The refpiece element
The refpiece element defines a reference to a piece within the ‘references’ component, as indicated by attributes reftype, refcomp and reftag.

The ‘assessments’ component

Using this component, assessments can be made available to student accounts. This component by default is situated on the ‘empack’ component.

XML definition:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present,accessible,directfeedback"/>
    <status present="true" accessible="true" directfeedback="true" selected="false" opened="false"
started="false"/>
  </component>
  <content type="root" childnodes="assessment" maxid="" preview="true">
      <reflocations type="ref" reftype="locationtags_assessmentsroot" refcomp="locations"
reftag="location" multiple="true"/>
    <assessment id="" type="node" childnodes="refitem,feedbackcondition" key="name">
      <name type="line"/>
      <hovertext type="simplerichtext"/>
      <numberofitems type="line"/>
      <duration type="line"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,showfeedback,autostart"/>
      <status present="true" showfeedback="false" autostart="false" selected="false" opened="false"
started="false" finished="false" score="0"/>
    </assessment >
    <refitem id="" type="node" key="pid">
      <pid type="line" private="true"/>
      <ref type="ref" reftype="itemtag_assessmenttag" refcomp="items" reftag="item"/>
      <weighting type="singleselect" items="0,1,2,3,4,5,6,7,8,9,10" selecteditem="">1</weighting>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present,accessible"/>
      <status present="true" accessible="true" selected="false" opened="false" answer=""
feedbackconditionid="" score="0" correct="false"/>
    </refitem>
    <feedbackcondition id="" type="node" childnodes="" key="pid">
      <pid type="line" private="true"/>
      <conditionstring type="condition" subtype="assessmentfeedback"/>
      <richtext type="simplerichtext"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes=""/>
      <status selected="false" opened="false"/>
    </feedbackcondition>
  </content>
</data>
```

The component element
A new property is directfeedback. It indicates feedback should be shown directly after an answer is given. If false, a student account has to choose an interface option to see the feedback.

The content element
Only possible 'node' child elements are assessments.

The assessment element
The assessment element is used to enter an assessment. Possible 'node' child elements are refitem and feedbackcondition.
Refitems are references to items in a 'items' component.
Feedback conditions are rules meant to present an appropriate feedback text for the whole assessment. They can be filled in, but aren't checked yet by the toolkit, so no assessment feedback is shown yet.

A new child element is numberofitems. It is used to indicate the number of items to show to a student account. For instance if the assessment contains 80 items and the number of items is 10, these 10 items are randomly chosen from the 80, when an assessment is started. If numberofitems is left blank, all 80 items are shown.
The new child element duration is meant to give a student account an indication how long the assessment will take.

New attributes within the status element are showfeedback and autostart.
Showfeedback indicates feedback should be shown if present. If it is false no feedback is shown. It could be you don't want to show feedback the first time an assessment is made, but the second time you will, to help the student account.
Autostart indicates an assessment should start automatically. Normally a student account has an option to start the assessment and to end it when he thinks he's ready. When he is ready an assessment score is presented. If autostart is true he cannot start or end the assessment himself and no score is presented. This option can be used if you just want to ask a student account some questions.
As you see attributes started and finished are present to save in progress data when an assessment is started or ended. Attribute score is used to store the calculated score at the end of the assessment.

The refitem element
The refitem element is used to define a reference to one item within an 'items' component. This item then is used within the assessment. The ref child element is used to define such a relation, as indicated by reftype, refcomp and reftag.

The new child element weighting is used to give a weight to the item within the assessment. It can have a value in the range 0..10 and default value is 1. The assessment score is calculated by multiplying the item score with the refitem weighting and adding up these values for all answered refitems.

The status element has new attributes answer and feedbackconditionid.
Answer is used to store the given answer (for mc questions it is the id of the chosen alternative, but for open questions it could be the entered answer) and

feedbackconditionid to store the id of the feedbackcondition which was true for the given answer.
Score and correct are used to store the values of the fired feedbackcondition.

The feedbackcondition element
The feedbackcondition element is meant to define a feedback for the assessment as a whole. It isn't checked yet.
It's almost the same as the feedbackcondition element within the 'items' component. Only child elements correct and score are missing, because they aren't relevant.

The 'scripts' component

Using this component, script rules can be entered that fire if they are triggered by certain events.
Also timers and counters can be defined. This component has no counterpart within the player environment.
It's important to notice that conditions aren't checked on a timely basis but only are triggered by events. An exception is made for conditions which check timers. Timers are checked regularly if they must fire. If a timer fires all conditions refering to this timer are checked.

XML definition:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes="present"/>
    <status present="true" selected="false" opened="false"/>
  </component>
  <content type="root" childnodes="condition" maxid="">
    <condition id="" type="node" childnodes="condition,action,timer,counter" key="pid">
      <pid type="line" private="true"/>
      <conditionstring type="condition"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status present="true" opened="false"/>
    </condition>
    <action id="" type="node" childnodes="" key="pid">
      <pid type="line" private="true"/>
      <actionstring type="action"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status present="true" opened="false"/>
    </action>
    <timer id="" type="node" childnodes="" key="pid">
      <pid type="line" private="true"/>
      <realtime type="boolean"/>
      <fromstartofrun type="boolean"/>
      <delay type="line" private="true"/>
      <repeats type="boolean"/>
      <repeatcount type="line" private="true"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status present="true" opened="false" started="false" finished="false" realtime="0"/>
    </timer>
    <counter id="" type="node" childnodes="" key="pid">
      <pid type="line" private="true"/>
      <explanation type="text" private="true"/>
      <initialstatus attributes="present"/>
      <status present="true" opened="false" value="0"/>
    </counter>
  </content>
</data>
```

The content element
Only possible ‘node’ child elements are conditions.

The condition element
The condition element is used to define a condition which as it fires generates some actions. For instance you can check in a condition if a certain location is visited for the first time.
Conditions can have ‘node’ child elements condition, action, timer and counter. So a condition can have sub conditions, which are checked if the condition fires. If the condition fires the child actions are conducted, possible child timers are started and possible child counters are initialised.

The child element conditionstring is used to build a condition. Conditionstring can be composite by using logical operators. The name conditionstring refers to the condition being saved as a string representation, as value of the element. This value still is saved, but isn’t used for validation of the condition. The condition itself is saved as child elements of the conditionstring element. These child elements are no part of the XML definition (yet) and are handled in section ‘XML data’. And because conditions can refer to content of all component types these references are saved within the content of the ‘case’ component, see section ‘XML data’.

Notice the status element has attributes present and opened. Present can be set to false (using another script rule) if you (temporarily) want to prevent a condition from firing. Opened is set when the condition fires.

The action element
The action element is used to define an action which will be conducted if the corresponding condition fires. For instance an action could be to send a mail to a student account.
Actions cannot have any ‘node’ child elements.

The child element actionstring is used to build an action. Actionstring cannot be composite, so only one action can be defined. The name actionstring refers to the action being saved as a string representation, as value of the element. This value still is saved, but isn’t used for conducting the action. The action itself is saved as child elements of the actionstring element. These child elements are no part of the XML definition (yet) and are handled in section ‘XML data’. And because actions can refer to content of all component types these references are saved within the content of the ‘case’ component, see section ‘XML data’.

Notice the status element has attributes present and opened. Present can be set to false (using another script rule) if you (temporarily) want to prevent an action being conducted. Opened is set when the action is conducted.

The timer element
The timer element is used to define a timer. A condition is used to check if a timer fires. Timers cannot have any ‘node’ child elements.

The child element realtime is used to indicate the timer to be realtime or not. Normally realtime is false and a timer counts case time, that is the time a student

account operates within the player environment. If this environment is closed, case time stops ticking. It starts ticking again if the environment is opened. If realtime is true real time is measured from the moment the timer is started and real time ticks even if the player environment is closed.
The child element fromstartofrun is used to relate the timer to the real start date and time of a run. This is meant to be able to react on a cohort of students starting a case at the same start date and which has to finish the case at a certain end date. Realtime will have to be set true too, to accomplish this.
The child element delay is used for the timer delay in seconds. Notice timers aren't checked every second, due to performance, so the delay should be at least 5 seconds.
The child element repeats indicates if the timer should repeatedly fire after every delay.
If repeat is set to true, child element repeatcount can limit the number of firing events to a certain value.

Notice the status element has attribute present. Present can be set to false (using another script rule) if you (temporarily) want to prevent the timer starting. Started and finished are used to store starting and firing of a timer. A new attribute is realtime, to store the realtime when the timer is started.

The counter element
The counter element is used to define a counter. Counters cannot have any 'node' child elements.

The status element has a new attribute value. This is used to store the value of the timer. Default value is 0. Present can be set to false (using another script rule) if you (temporarily) want to prevent the counter being initialised.

The 'case' component

This component is used to store case properties such as starttime and currenttime. It is also used to store all references between content of different components.

XML definition:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component type="root">
    <initialstatus attributes=""/>
    <status selected="false" opened="false" starttime="" currenttime=""/>
  </component>
  <content type="root" childnodes="" maxid="">
  </content>
</data>
```

The component element
New component properties are starttime and currenttime. Values of msecs since januari 1, 1970 are stored as given by the Java (new Date()).getTime() method.
The starttime is the time the player environment is started the first time by a student account. When the player is started a next time by the same student account starttime is increase by the amount of milliseconds elapsed since the player was closed the last time, given by currenttime. In this way currenttime minus starttime equals the total time the player was open summarised over all sessions, which is called case time.

The currenttime is the current time. It is increased while the player is open.

The content element
Within the definition there are no possible ‘node’ child elements. But all references between content of different components are saved as child elements of the content element. These child elements are no part of the XML definition (yet) and are handled in section ‘XML data’.

XML data

All data entered by a case component author is saved as XML data, based on the XML definition of the component. This XML data is saved in the database in table ‘casecomponents’.
The XML definition defines default values for component and content element properties. These can be overruled by the case component author, but don’t have to.

We will not show examples for all components, but will highlight some components for illustration purposes. In principal you don’t have to know how XML data is stored, because you don’t have to enter it yourself. Component editors take care for entering XML data and prevent you from having to see and understand it.

The ‘locations’ component

XML data example:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status></status>
  </component>
  <content id="1" max_id="3">
    <location id="2" type="node">
      <pid>villa</pid>
      <name>villa</name>
      <explanation></explanation>
      <status present="true" accessible="true" opened="true"></status>
      <background id="3" type="node">
        <pid>villa</pid>
        <picture>5783</picture>
        <explanation></explanation>
        <status opened="true"></status>
      </background>
    </location>
  </content>
</data>
```

The component element
The status tag has no attributes, indicating the component author has not overwritten default values defined in the XML definition.

The content element
You see this element gets id 1 and max_id is increased to reflect the number of ‘node’ elements within the content element. Content is also a ‘node’ element but a special one namely ‘root’.

The location element

The location element has the attribute type. This isn’t really necessary, because the type already is within the XML definition, but it makes the XML more readable.
You see the component author has filled in pid and name, and left explanation blank. Within the status element some attributes are set, indicating the component author has overwritten their values. Opened is true indicating this is the default location.

The background element
You see the component author has uploaded a picture. A picture is saved as a blob. The value of the picture element is set to the id of the blob within the database. Within the status element, opened is true indicating this is the default background.

The ‘assessments’ component

XML data example:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status></status>
  </component>
  <content id="1" max_id="3">
    <assessment id="2" type="node">
      <name>assessment</name>
      <hovertext>&lt;p&gt;Please answer a few questions.&lt;/p&gt;</hovertext>
      <numberofitems></numberofitems>
      <duration></duration>
      <explanation></explanation>
      <status showfeedback="false" present="true" autostart="true"></status>
      <refitem id="3" type="node">
        <pid>item 0</pid>
        <ref></ref>
        <weighting>1</weighting>
        <explanation></explanation>
        <status accessable="true" present="true"></status>
      </refitem>
    </assessment>
  </content>
</data>
```

The assessment element
The value of the hovertext element contains richtext. All element values are escaped before being saved in XML. That’s why you see ‘&lt;’ and so on. When element values are read they are unescaped.

The refitem element
The value of the ref element is empty. This doesn’t mean no reference is made to an item within an ‘items’ component. This relation namely is saved within the XML data of the ‘case’ component.

The ‘conversations’ component

XML data example:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status></status>
  </component>
  <content id="1" max_id="6">
    <conversation id="2" type="node">
```

```
      <pid>introduction by Watson</pid>
      <name>introduction by Watson</name>
      <reflocations></reflocations>
      <explanation></explanation>
      <status present="false"></status>
      <background id="3" type="node">
        <pid>background</pid>
        <picture>5707</picture>
        <explanation></explanation>
        <status opened="true"></status>
      </background>
      <fragment id="4" type="node">
        <pid>introduction video</pid>
        <blob mediatype="" blobtype="int_url">5704</blob>
        <explanation></explanation>
        <status opened="true"></status>
      </fragment>
      <question id="5" type="node">
        <text>What time did we get the alert?</text>
        <explanation></explanation>
        <status present="true" outfoldable="false"></status>
        <fragment id="6" type="node">
          <pid>answer W1</pid>
          <blob mediatype="" blobtype="int_url">5708</blob>
          <explanation></explanation>
          <status opened="true"></status>
        </fragment>
      </question>
    </conversation>
  </content>
</data>
```

The conversation element
The value of the reflocations element is empty. This doesn't mean no reference is made to locations within the 'locations' component. This relation namely is saved within the XML data of the 'case' component.
Within the status element you see the component author has set present to false, indicating the conversation will not be opened. Present can be set true using a 'scripts' component.
You see the conversation has one background and one start fragment which starts playing when the conversation is opened. Further you see one question element with a fragment element being played as answer. The question can be asked if a student account chooses a option to show the questions.

The fragment elements
You see the component author has defined internal urls (blobtype attribute) within the blob element, meaning a file is played which is located within the 'streaming' map on the EMERGO server. The value of the blob element is set to the id of the blob within the database.

The 'scripts' component

Not all XML elements within the XML data are defined within the XML definition (yet). The condition and action element do have child elements which aren't defined. We will describe them in this section.

XML data example:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
```

```
  <component>
    <status></status>
  </component>
  <content id="1" max_id="3">
    <condition id="2" type="node">
      <pid>if conversation in villa and conversation intro are finished</pid>
      <conditionstring>
        <evalrungrouptagstatus>
          <cacid>563                          [id of conversations case component]
            <tagid>2</tagid>                  [id of first conversation]
            <tagid>3</tagid>                  [id of second conversation]
          </cacid>
          <carid>159</carid>                  [id of case role]
          <tagname>conversation</tagname>             [name of elements to evaluate]
          <statusid>10</statusid>             [id of element property change to evaluate (finished)]
          <functionid>0</functionid>          [id of function to apply to element property (no function)]
          <operatorid>1                       [id of operator to be used ('=')]
            <operatorvalue>1</operatorvalue>          [value to check element property against]
          </operatorid>
        </evalrungrouptagstatus>
      </conditionstring>
      <explanation></explanation>
      <status present="true"></status>
      <action id="3" type="node">
        <pid>make location salon and location kitchen accessible</pid>
        <actionstring>
          <setrungrouptagstatus>
            <cacid>568                        [id of locations case component]
              <tagid>2</tagid>                [id of first location]
              <tagid>3</tagid>                [id of second location]
            </cacid>
            <carid>159</carid>                [id of case role]
            <tagname>location</tagname>       [name of elements to set value of]
            <statusid>4</statusid>            [id of element property to set (accessible)]
            <operatorvalue>1</operatorvalue>          [value to set property to]
          </setrungrouptagstatus>
        </actionstring>
        <explanation></explanation>
        <status present="true"></status>
      </action>
    </condition>
  </content>
</data>
```

The condition element
The pid child element indicates the condition should check if two conversations are finished.

You see a new child element evalrungrouptagstatus. It is meant to evaluate if a property of one or more elements (tags), with the same name, has a certain value. This value can be defined within the XML definition, can be overwritten be a component author or can be overwritten again due to a student account action within the player environment. That's why the element has 'rungroup' in its name. A student account is represented by a rungroup within the player.
The first child element of evalrungrouptagstatus is cacid. Its value is equal to the database cacid, the id of a case component being evaluated, in this case a 'conversations' component. The cacid element has two child elements, called tagid. Their values are equal to the id of the content elements (tags) being evaluated, in this case two conversations.

The second child element of evalrungrouptagstatus is carid. Its value is equal to the database carid, the id of a case role for which the condition should be evaluated. There could be more carid elements.
The next child element tagname contains the name of the elements (tag) to be evaluated, in this case conversation.
The child element statusid contains the id of the content element property to be evaluated. 10 represents finished as can be found within Java interface IAppManager. The conversations have to be finished, both.
The child element functionid contains the id of the function to be applied on the property value. 0 indicates there is no function applied. For other values see IAppManager. We have three functions. One to count the number of times a property has gotten a certain value, for instance to count the number of times a location was visited. Using other two functions you can increment or decrement an integer property, like the value of a counter.
Child element operatorid contains the id of the operator to be used to evaluate the property value. 1 means '='. For other values see IAppManager. We also have '>', '>=', '<', '<=', [a..b] (interval) and [a,c,e] (collection). Notice all these operators can be used on a count function too. The operatorid element has a child element operatorvalue. Because the finished property is of type boolean, 1 indicates true, so the finished value should be true. For the interval operator there are two operatorvalue child elements, for the collection operator, multiple.
Cacid, tagids and carid are also saved within the 'case' component data but then related to the condition element of the 'scripts component given by its own cacid, tagid and carid, and also related to the statusid (called trigger). Why save the same relation twice? Because it isn't quite the same relation. A condition can be composite, so there can be multiple evalrungrouptagstatus child elements. So cacid, tagids and carid are then related to an evalrungrouptagstatus element. Within the 'case' component there is a relation with a condition element, indicating when the condition has to be evaluated when it is triggered by a content element property change.

Another possible child element is evalrungroupcomponentstatus. It is meant to evaluate if a property of one component has a certain value. It has almost the same structure as evalrungrouptagstatus. Only tagid and tagname elements are missing. For instance you could evaluate if the 'references' component is opened or how many times.

Other possible child elements are logical operators and, or, not, parenthesisopen and parenthesisclose. This allows for building composite conditions for instance:

```
<evalrungrouptagstatus/>
<or/>
<parenthesisopen/>
<evalrungrouptagstatus/>
<and/>
<evalrungroupcomponentstatus/>
<parenthesisclose/>
```

This represents:

```
evalrungrouptagstatus() or (evalrungrouptagstatus() and evalrungroupcomponentstatus())
```

The action element
The pid child element indicates the action should set accessible for two locations to true.

You see a new child element setrungrouptagstatus. It is meant to set a property of one or more elements (tags), with the same name, to a certain value. This changed

value is saved as student account progress data. That's why the element has 'rungroup' in its name. A student account is represented by a rungroup within the player environment.
The first child element of setrungrouptagstatus, cacid, is handled in the previous seciont. It's the id of a case component who's progress data is changed, in this case the 'locations' component. The tagid element values are equal to the id of the content elements (tags) being set, in this case two locations.
The second child element is carid. Its value is equal to the id of the case role for which the action should be conducted. There could be more carid elements.
Tagname contains the name of the elements (tag) to be set, in this case location.
Statusid contains the id of the content element property to be set. 4 represents accessible. The locations have to be accessible, both.
Operatorvalue is equal to 1. Because the accessible property is of type boolean, 1 indicates true, so the accessible value should be set true.
Notice there are no function and operator child elements, because function isn't relevant and operator always is '='.

Another possible child element is setrungroupcomponentstatus. It is meant to set a property of one component to a certain value. It has almost the same structure as setrungrouptagstatus. Only tagid and tagname elements are missing. For instance you could set the 'references' component to be accessible.

The 'case' component

No XML child elements of the content element within XML data are defined within the XML definition (yet). We will describe them in this section.
The 'case' XML data is used to store relations between XML data of other components.

XML data example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status></status>
  </component>
  <content id="1" max_id="1">
    <locationtags_conversationtags>
      <trigger>6                          [two locations are related to one conversation]
        <car>174                          [example only has one case role, so always same number]
          <cac>597                        [id of locations case component]
            <tag>2                        [id of first location]
              <refcar>174
                <refcac>599               [id of conversations case component]
                  <reftag>2</reftag>      [id of conversation]
                </refcac>
              </refcar>
            </tag>
            <tag>3                        [id of second location]
              <refcar>174
                <refcac>599               [id of conversations case component, same as before]
                  <reftag>2</reftag>      [id of conversation, same as before]
                </refcac>
              </refcar>
            </tag>
          </cac>
        </car>
      </trigger>
    </locationtags_conversationtags>
```

```
   <all_scripttags>
     <trigger>5                            [two locations are checked on opened by one condition]
       <car>174
         <cac>597                          [id of locations case component]
           <tag>2                          [id of first location]
             <refcar>174
               <refcac>598                 [id of scripts case component]
                 <reftag>2</reftag>        [id of condition]
               </refcac>
             </refcar>
           </tag>
           <tag>3                          [id of second location]
             <refcar>174
               <refcac>598                 [id of scripts case component, same as before]
                 <reftag>2</reftag>        [id of condition, same as before]
               </refcac>
             </refcar>
           </tag>
         </cac>
       </car>
     </trigger>
     <trigger>4                            [two locations are set accessible by one action]
       <car>174
         <cac>597                          [id of locations case component]
           <tag>2                          [id of first location]
             <refcar>174
               <refcac>598                 [id of scripts case component]
                 <reftag>2</reftag>        [id of action]
               </refcac>
             </refcar>
           </tag>
           <tag>3                          [id of second location]
             <refcar>174
               <refcac>598                 [id of scripts case component, same as before]
                 <reftag>2</reftag>        [id of action, same as before]
               </refcac>
             </refcar>
           </tag>
         </cac>
       </car>
     </trigger>
   </all_scripttags>
  </content>
</data>
```

The content element
Possible child elements (in first grade) are:
locationtags_referencesroot: used to store on which locations, a references component is available.
locationtags_assessmentsroot: used to store on which locations, a assessments component is available.
locationtags_conversationtags: used to store on which locations, conversations are available.
caseroles_npc_mailtags: used to store npc receivers (defined as case roles) for mails to be send.
caseroles_pc_mailtags: used to store pc receivers (defined as case roles) for mails to be send.
notecomponent_logbookroot: used to store which note component content is shown on a logbook component.
piecetag_itemstag: used to store which pieces (of a references component) are avaible for which items (or alternatives, or feedbackconditions).

itemtag_assessmenttag: used to store which items (of a items component) are avaible for which assessments.
all_scripttags: used to store relations between other components or content elements used within condition and/or action elements of ‘scripts’ components.
These child elements function as a sort of cross table elements. If more component defintions will be added, there will probably be more of these child elements.
Within the example XML data we see locationtags_conversationtags and all_scripttags elements.

The trigger element always is the first child element of a ‘crosstable’ element.
For all ‘cross table’ elements apart from all_scripttags its value is 6, meaning selected. For instance if a location is selected a reference component should be available.
For all_scripttags the value is equal to the property id chosen within a condition or action. For instance 5 means opened and 4 accessible. See IAppManager.

Within the trigger element relations are stored using car, cac, tag, refcar, refcac and reftag elements. Values of these elements are equal to data base case role id or case component id or content element (tag) id. For instance for the locationtags_conversationtags relations, the locations are inditified by car, cac and tag elements, and the conversations by refcar, refcac and reftag elements. Tag or reftag elements are missing or have value 0 for a component within a relation.
Case role is part of the relation because there can be multiple case roles and relations could be valid only for certain case roles.

XML progress data

All progress data for student acounts is saved as XML data, based on the XML definition of a component. User generated content is also part of progress data. The XML progress data is saved in the database in table ‘rungroupcasecomponents’, ‘runcasecomponents’ or ‘runteamcasecomponents’. The last two tables are used for shared progress data.

We will not show examples for all components, but will highlight some components for illustration purposes. In principal you don’t have to know how XML progress data is stored, because the player environment takes care of storing and retrieving it.

The ‘locations’ component

XML progress data example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status selected="true,0,true,133,true,291" opened="true,0,true,133,true,291"></status>
  </component>
  <content id="1" max_id="2">
    <location refdataid="12" type="node" id="2">
      <status selected="true,3" opened="true,3,false,12" accessible="false,19"> </status>
    </location>
  </content>
</data>
```

The component element

You see the component is selected and opened three times. There are three entries within the value of selected and opened. Every entry also has a timestamp, the case time in seconds when the component property was changed. In this way every property value change is saved. It is added at the end of the property value, so you see the 'history' of the property.
The component cannot be actively closed by a student account, so the opened property value doesn't contain false.
Recall the difference between selected and opened. See XML definition of 'locations' component.

The content element
You see progress data also works with max_id. Every child of the content element gets its own id. All progress is saved as first grade 'node' child elements of the content element. So 'node' child elements are not nested in progress data. Their context is part of the progress data.

The location element
The location element has a new attribute refdataid. It's the id of the XML data element (entered by a component author) for which something has changed within the player environment.
In the status element you see the location was selected and opened after 3 seconds. After 12 seconds the student account left the location, indicated by the value false. False is set, because only one location can be visited at a time. The moment opened for this location was set to false, it was set true for another location (not visible in example). You see that after 19 seconds the location was set inaccessible (accessible false). This was cause by a script rule which fired, because student account actions don't cause this property value to be changed. The same accounts for instance for the present property.

The 'assessments' component

XML progress data example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status selected="true,294,true,306" opened="true,294,false,300,true,306,false,324"></status>
  </component>
  <content id="1" max_id="3">
    <assessment refdataid="2" type="node" id="2">
      <status started="true,294" outfolded="true,300"></status>
    </assessment>
    <refitem refdataid="3" type="node" id="3">
      <status selected="true,297" opened="true,297" feedbackconditionid="10,300" correct="true,300"
answer="3,300" score="1,300"></status>
    </refitem>
  </content>
</data>
```

The component element
You see the component is selected two times and and opened and closed two times. A student account can close the 'assessment' component actively or by choosing another component. If the latter is the case the false value is missing.

The assessment element
You see the assessment is started and outfolded to see its items.

The refitem element
You see the item was selected and opened. Three seconds later the answer was given and a feedbackcondition fired.

The ‘conversations’ component

XML progress data example:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status selected="true,3" opened="true,3,false,30"></status>
  </component>
  <content id="1" max_id="5">
    <conversation refdataid="2" type="node" id="2">
      <status selected="true,3" opened="true,3,false,30" finished="true,30"></status>
    </conversation>
    <fragment refdataid="3" type="node" id="3">
      <status opened="true,3"></status>
    </fragment>
    <question refdataid="117" type="node" id="4">
      <status selected="true,24" opened="true,24" outfoldable="true,24"></status>
    </question>
    <fragment refdataid="133" type="node" id="5">
      <status opened="true,24"></status>
    </fragment>
  </content>
</data>
```

The component element
The component is selected and opened indirectly because a conversation is opened.

The conversation element
You see the conversation is opened and closed. Finished is set when it was closed.

The fragment elements
Opened indicates the fragment was played.

The question element
The question was selected and opened, causing the a fragment to be played.

The ‘scripts’ component

XML progress data example:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status></status>
  </component>
  <content id="1" max_id="5">
    <condition refdataid="2" type="node" id="2">
      <status opened="true,9"></status>
    </condition>
    <action refdataid="3" type="node" id="3">
      <status opened="true,9"></status>
    </action>
    <timer refdataid="6" type="node" id="4">
      <status opened="true,18" realtime="1282219904,18" started="true,18,false,78" finished="true,78"
></status>
```

```
    </timer>
    <counter refdataid="175" type="node" id="5">
      <status opened="true,300" value="1,321"></status>
    </counter>
  </content>
</data>
```

The component element
The ‘scripts’ component isn’t used directly by a student account, so no status attributes are saved.

The content element
You see condition and action are fired, opened is set to true.

The timer element
Opened indicates the timer action was fired. The timer was started after 18 seconds and finished after 78 seconds, indicated by started and finished. Realtime was set when the timer started.

The counter element
Opened indicates the counter action was fired. Its value was changed after 21 seconds due to a script rule.

The ‘case’ component

XML progress data example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status currenttime="1282220216428,342" starttime="1282219874288,0"></status>
  </component>
  <content id="1" max_id="1"></content>
</data>
```

The component element
The ‘case’ component isn’t used directly by a student account, so status attributes selected and opened aren’t saved. Starttime and currenttime are saved.

The content element
The ‘case’ component has no progress data.

The ‘references’ component

We use this component to show three examples of different progress data:

- A ‘normal’ example, where an existing piece (entered by a component author) is opened by a student account.
- An example where the existing piece is adjusted by a student account.
- An example where a new peace is added by a student account.

1. XML progress data example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status selected="true,154" opened="true,154,false,170"></status>
  </component>
  <content id="1" max_id="2">
```

```
    <piece refdataid="3" type="node" id="2">
      <status selected="true,157" opened="true,157"></status>
    </piece>
  </content>
</data>
```

The piece element
A piece is selected and opened.

2. XML progress data example, existing piece is adjusted:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status selected="true,154" opened="true,154,false,170"></status>
  </component>
  <content id="1" max_id="2">
    <piece refdataid="3" type="node" id="2">
      <status selected="true,157" opened="true,157">
        <name>new name</name>
      </status>
    </piece>
  </content>
</data>
```

The piece element
You see the name of an existing piece (given by refdataid) is changed by a student account. The change is saved within the status element.

3. XML progress data example, with user generated content:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component>
    <status selected="true,198" opened="true,198"></status>
  </component>]
  <content id="1" max_id="2">
    <piece dataparentid="1" databeforeid="2" owner_rgaid="286" id="2" extended="true" type="node">
      <status present="true,216" accessable="true,216">
        <name>test</name>
        <blob mediatype="" blobtype="ext_url">5798</blob>
        <explanation></explanation>
      </status>
    </piece>
  </content>
</data>
```

The piece element
The refdataid attribute is missing because piece is added by a student account, so doesn’t have a reference to content added by a component author.
You see this element has four new properties.
Dataparentid and databeforeid define the context of the user generated piece. Dataparentid is the id of the parent element, so the new piece is placed within an existing map. Databeforeid is the id of the element before which the new piece is added. It can be empty, if the new piece is the first one in the map. There could also be properties statusparentid and statusbeforeid (not visible in example). It could be new piece is added within a user generated map and/or before a user generated element.
Owner_rgaid gives the id of the run group account who added the new piece.
Extended indicates the new piece is user generated content.

The elements that within the definition of a piece are placed within the piece element, are now placed withint the status element, to indicate it is progress data.

XML progress update data

There are some situations where a student account progress data has to be updated by other accounts. See RunGroupCaseComponentUpdate class of the domain model. This situations require their own XML format.

XML progress update data example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<data>
  <component></component>
  <content max_id="2" id="1">
    <inmailhelp update_statuskey="sent" update_statusvalue="true" update_checkscript="true"
refdataid="" type="node" id="2">
      <status version="2" sent="true,2720">
        <sendername>manager</sendername>
        <receivername>student</receivername>
        <inbox>inbox</inbox>
        <title>Welcome to our company</title>
        <richtext>&lt;p&gt;Hope you will feel at home soon.&lt;/p&gt;</richtext>
      </status>
    </inmailhelp>
  </content>
</data>
```

The component element

In this example no status attributes are set, but this could be the case in another example. For instance an administrator account changes accessible for a reference component.

The content element

The content gets only one child element, so all updates get their own record in table rungroupcasecomponentupdates. Within this record is saved which component progress data should be updated.

The inmailhelp element

In this example inmailhelp is used, but it could also be another element.
A playing character ‘manager’ sends a mail to another playing character ‘student’.
Update_statuskey indicates the element property to be set, in this case sent.
Update_statusvalue indicates the value the property should get.
Update_checkscript indicates if script rules should be checked after sending the mail.
Refdataid is empty, because it is a user generated mail and no predefined mail. It is not empty if for instance an administrator account sets accessible to true of an existing map within a references component.
Within the status element you see the content which is needed to show the mail correctly for the receiver.

XML import and export for a case

Cases can be imported an exported. They are exported as an IMS content package. A case is exported including its blob files, but streaming files are not exported! Likewise importing includes blobs, but no streaming media files!

This is to prevent long waiting times while exporting/importing and overload of the server (disk space). Streaming files have to be copied by an administrator account which has access to the server. They also can be uploaded within the administrator environment, but this can take a lot of time, if files are large or there are a lot of files.

The content package contains the following files:
- Case.xml: the case record saved as XML.
- Caseroles.xml: all case role records for the case.
- Casecomponents.xml: all case component records for the case.
- Casecomponentroles.xml: all case component role records for the case.
- Blobs.xml: all blob records for the case.
- All blob files, each within a map, with a name equal to the original blob id.

During exporting, all references to case id, case component ids, case role ids, case component role ids and blob ids, are replaced by world wide unique ids. We have to do this, because if the case is imported on another server running the EMERGO toolkit, database ids are different.
During importing, new database records are created, and all world wide unique ids are replaced by the new database ids.
Tag ids don't have to be replaced by world wide unique ids, because they are part of a case component, so are identified by the case component id.
Component ids, within case component records, aren't replaced either. While there is only one party creating new components, this is not a problem. New components can be imported within another EMERGO toolkit database. But if other parties also create new components, component ids get mixed up. This has to be fixed still, by adding a world wide unique id to the component records and exporting components too. During importing is checked if components already exists, or have to be added.

XML import for accounts and run group accounts

An administrator can import accounts.
The XML format to use is:
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<accounts>
  <account
    id="ou_894234514"                     [must be unique if new account]
    password="pqr5uv"                     [if left blank toolkit generates password if new account]
    update=""                             [if true an existing account is updated]
    title=""
    initials="E.L."
    nameprefix="van"
    lastname="Halen"
    email="van.halen@warner.com"
    phonenumber=""
    job="musician"
    roles="stu"                           [EMERGO toolkit roles for account, comma separated]
    active="true">                        [if false, account no longer has access to toolkit]
  </account>
</accounts>
```
Id, lastname, roles and active are required, so cannot be left blank. Of course multiple account elements can be used.

A case run manager can import run group accounts. He first chooses a run and then imports. Imported accounts are added to the chosen run.
The XML format to use is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<accounts>
  <account
    id="ou_894234514"                    [id of account]
    caseroles="'student'"                [case for account, comma separated]
  </account>
</accounts>
```
You see only caseroles are needed because accounts are added to the chosen run. You can also use one file for importing accounts <u>and</u> run group accounts, by adding the caseroles attribute within the first XML format within this section.

5. Adding a new EMERGO component

We describe how you can add a new EMERGO component and what you have to do to get it up and running.
What you have to do depends on a couple of aspects:

- Does your component require new input element types in the author environment? Current types are found in appendix 1 'Overview XML elements'. For instance line, text or picture. These types are used to enter XML content for one element.
- Does your component require another input structure than a tree? All content is structured in a tree structure (except for Google Maps). Do you need another structure?
- Does your component require another presentation structure than a tree? Most content in the player environment is presented within a tree. Do you need another structure?

We describe steps to be taken. The least you have to do are steps 1, 2 and 6.

1. Add a component
Add a component within the administrator environment. Enter the XML definition. You can use another XML definition as an example.

2. Change properties files
The least you have to do is to add the component code to all properties files, so case author accounts can see and choose the new component.

If you use new 'node' elements or child elements of them, you have to add them to the properties files too. You best search for an existing 'node' element, to see what you have to add. The same accounts for new attributes or new values for attribute 'type'.

3. Possibly adjust class CContentItemHelper
This class is part of the control layer of the author environment.
If you introduce new element types, you have to adjust this class to render the new type (method renderRow) within the 'node' element edit window, and to handle input from it (method xmlRowsToNode).

4. Possibly add a new input structure class and adjust class CContentHelper
If CTree does not satisfy as input structure, add a new class. You have to adjust class CContentHelper too, to be able to render the new structure. CContentHelper is part of the control layer of the author environment.

5. Possibly add a new presentation structure class and adjust class CRunContentItemMenuPopup

If you've added a new input structure class you probably will have to add a new presentation structure class too, to be able to render the new structure. This class is part of the control layer of the player environment. Within class CRunContentItemMenuPopup you will have to adjust methods getContentHelper() and getContentItemHelper(), to be able to handle user generated content.

6. Add the component in the player environment

To use the component within the player environment you have to add at least three classes:

- A class extending from CRunComponent, for instance CRunReferences.
- A class extending from CRunTree, or your own added presentation structure class, for instance CRunReferencesTree.
- A class extending from CRunComponentHelper, for instance CRunReferencesHelper.

Further you have to possibly adjust the following classes:

- CRunComponentView. If your component must be (partly) visible within the run view area, this class has to be adjusted. For instance for references component.
- CRunChoiceArea. If your component must be (partly) visible within the run choice area, this class has to be adjusted. For instance for questions to ask during conversation.
- CRunNote. If your component requires another handling of contextualised notes than default, this class has to be adjusted. For instance for conversation questions.
- CRunWnd. If your component specific action handling method onAction has to adjusted. If your component needs partly to be rerendered when a script rule fires method checkStatusChange has to be adjusted. If your component isn't presented on the empack, but must be present on certain locations, methods getLocationActions and getCaseComponentLocationActions have to be adjusted.

You have to add your components style to the stylesheet. And you possibly have to add some images for a button on the empack, or for the component itself. You possibly have to change the properties files, if you component requires some new labels.

Appendix 1: Overview XML elements

We just give an overview and don't describe elements. For this you can search the document for the first underlined occurrence of the element (or property).

We give four overviews:

- elements within XML definitions
- elements within XML data
- elements within XML progress data
- elements within XML progress data update

Elements within XML definitions

You find these elements too within data, progress data or progress data update, but they are part of the XML definitions.

Data

Root element of all XML data within EMERGO toolkit.

Component

Element meant to define component properties.

Posible attributes
Type (value is ‘root’).

Possible child elements
Inititialstatus, Status.

Possible Status child element attributes
Present, Accessible, Selected, Opened, Taskscheckable, Extendable, Ownership, Shared, Sharedbyteam, Directfeedback, Starttime, Currenttime.
These represent all possible component properties.
Starttime and Currenttime are integers, the rest of them booleans.

Content

Element meant to define content of component.

Possible Attributes
Type (value is ‘root’), Id (value is always ‘1’), Childnodes, Maxid, Preview.
Id and Maxid are integers, Preview is a boolean, the rest of them are strings.
Childnodes can be comma separated.

Elements of type ‘node’

Location, Background, Task, Map, Piece, Conversation, Question, Fragment, Inmailpredef, Inmailhelp, Outmailpredef, Outmailhelp, Alert, Note, Item, Alternative, Feedbackcondition, Refpiece, Assessment, Refitem, Condition, Action, Timer, Counter.
These elements represent all possible content elements.
Some of them are found within different XML definitions. Sometimes the definition is the same, for instance for Background, sometimes the definition is slightly different, for instance for Piece: within the ‘googlemaps’ component the Piece also has GPS coordinates.

Possible attributes
Type (value is ‘node’), Id, Childnodes, Key, Singleopen, Preview, Reftype, Refcomp, Reftag, Multiple.
Id is an integer, Singleopen, Preview and Multiple are booleans, the rest of them are strings. Childnodes and Key can be comma separated.

Non node child elements of node elements

Pid, Name, Explanation, Picture, Hovertext, Time, Reflocations, Blob, Description, Refsendernpc, Richtext, Refreceiversnpc, Refreceiverspc, Inbox, Exlatitude, Exlongitude, Modal, Refnote, Type, Conditionstring, Correct, Score, Numberofitems,

Duration, Weighting, Actionstring, Realtime, Fromstartofrun, Delay, Repeats, Repeatcount.
These elements are meant to describe parts of node elements.

Possible attributes
Type (see all possible types beneath), Private, Notempty, Blobtypes, Blobtype, Mediatypes, Mediatype, Choiceempty, Hidden, Items, Selecteditem, Subtype.
Private, Notempty, Choiceempty and Hidden are booleans, the rest of them are strings. Blobtypes, Mediatypes and Items can be comma separated.

Possible values for type attribute
Line, Text, Picture, Simplerichtext, Ref, Blob, Singleselect, Condition, Action.
These values represent all possible input elements used within the toolkit. With these types we can render all component editors now available. It could be that new components require new types of input elements.

Status child elements of node elements
Status, Inititialstatus.
These elements are meant to define node element properties and which of them can initially be set by a component author.

Possible attributes
Present, Accessible, Selected, Opened, Required, Finished, Canceled, Outfolded, Outfoldable, Deleted, Started, Sent, Version, Showfeedback, Autostart, Answer, Feedbackconditionid, Score, Realtime, Value.
Version, Feedbackconditionid, Score, Realtime and Value are integers, Answer is a string, the rest of them are booleans.

Elements within XML data

There are elements saved within XML data which are not (yet) part of the XML definitions.

Non node child elements of non node elements conditionstring and actionstring
Evalrungrouptagstatus, Evalrungroupcomponentstatus, Setrungrouptagstatus, Setrungroupcomponentstatus, And, Or, Not, Parenthesisopen, Parenthesisclose, Tagid, CarId, Tagname, Statusid, Functionid, Operatorid, Operatorvalue.
These elements are used within the ‘scripts’ component and ‘items’ component content.

Non node child elements of content element
Locationtags_referencesroot, Locationtags_assessmentsroot, Locationtags_conversationtags, Caseroles_npc_mailtags, Caseroles_pc_mailtags , Notecomponent_logbookroot, Piecetag_itemstag, Itemtag_assessmenttag, All_scripttags, Trigger, Car, Cac, Tag, Refcar, Refcac, Reftag.
These elements are used within the ‘case’ component content.

Elements within XML progress data

There are attributes saved within progress data which are not (yet) part of the XML definitions. Like attributes id and type they can be applied to all node elements.

Possible node element attributes
Refdataid, Dataparentid, Databeforeid, Statusparentid, Statusbeforeid, Owner_rgaid, Extended.
Refdataid is used if properties or content of node elements entered by component authors is adjusted. The other attributes are used for user generated node elements.

Elements within XML progress update data

There are attributes saved within progress update data which are not (yet) part of the XML definitions. Like attributes id and type they can be applied to all node elements.

Possible node element attributes
Update_statuskey, Update_statusvalue, Update_checkscript.

Appendix 2: EMERGO SQL

The file emergo.sql in submap ‘sql’ contains SQL script. It is a dump of the initial database of the EMERGO toolkit. It contains all database tables and some data:

- All five EMERGO roles in table roles.
- An administrator account in table accounts. Password should be changed after installation.
- Five entries, for all five roles, in table accountroles for the administrator.
- All EMERGO components in table components.
- Notification mail templates in table mails.
- Configuration key-value pairs in table sys. Values should be overwritten by locally valid values after installation.