Progress Engine

The progress engine controls the progress of a lesson. When a learner starts a lesson or moves from activity to activity, it is the progress engine guiding their way. Without the progress engine, LAMS isn't LAMS.

This page covers some background to the progress engine, the overall system design as well as the implementation algorithm.

This page is based on the original LAMS 1.1 Progress Engine document in the lams_document project. It has been updated to reflect LAMS 2.1, but the original document makes more comparisons with LAMS 1.0, so if you want to see how the progress engine has evolved, check the original document. The UML diagrams on this page have been taken from that document (as screenshots) or are in one of the MagicDraw files in lams_common/design. MagicDraw v10 is the best version for updating these diagrams - you open the .zip file from within MagicDraw to access the diagrams.

- Background
  - Domain Object Model and Database Model
  - Lesson States
- Teacher's Perspective Use Cases
- Learner's Perspective
  - Business Class Diagram
  - Major Algorithm Explanation
    - LAMS 2.x activity page loading Mechanism
    - Move to next activity
    - Calculate the next activity at service layer
    - Mapping the Activity to the URL
    - Join a lesson with Retrieving Progress Data - Needs Updating
    - Communication between Server progress engine and Flash Progress bar
    - Non-Flash Progress Bar
- Authoring Preview
- System Activities
  - Parallel Activities
  - Optional Activities and Optional Sequences
  - Grouping
    - Object and Database Models
    - Preview
  - Gates
    - Object Model
    - Gate Algorithm
    - Preview
  - Branching
- Future Improvements
  - Lesson and Progress Data Caching
  - Making System Tools More Pluggable
  - Separate Instances for System Tools

Background

To understand the progress engine, it is necessary to review some background concepts that related to LAMS progress engine. The main purpose of this section is to let new developers to understand the e-learning concept behind the design of progress engine and what progress engine is doing from user's point of view. If you knows LAMS concept already, please feel free to skip this section.

Learning Design
The concept of learning design has been defined in the IMS specification. The LAMS learning designs are not described in the same way as learning designs in the IMS standard, but the concept is the same. From progress engine's perspective, learning design defines a sequence of collaborative learning activities and tools required to support these activities. It defines the basic navigation rules that progress engine should cope with. Learning designs are also known as **"sequences"** as main structure of a learning design is a sequence of Activities, rather than a set of unordered Activities.

Activity
An Activity is the major element of a learning design. In another words, it is main learning object that sits inside the learning design. From progress engine's point of view, activities are the nodes of the sequence that the engine needs to navigate through. Generally they are a "box" of some kind on the learning design picture.

Lesson
While a learning design is a reusable template, the lesson is one instance that using the learning design template. A lesson should have a number of participating learners (see Lesson Class), who are going to do the activities that are defined in a learning design. Therefore, data that is involved in the learning design instance of a particular lesson should not be shared across lessons. In terms of progress engine, the start of a lesson is the trigger of the progress engine. At this point, progress engine should initialise any resources of the learners.

Learner Progress
The learner progress is the data holder that records the progress status of runtime learning design instance. In other words, learner progress record where the learner is in a lesson. Therefore, learner progress has to be specific to a particular learner and a particular lesson.

Learner Progress Bar
The learner progress bar is a graphical representation of a user's progress. It appears in three forms - a vertical Flash component that sits on side of the LAMS learner GUI interface or a non-Flash tree structure that overlays the LAMS learner GUI interface, and the horizontal Flash layout used in the the LAMS monitoring GUI interface. These give a user friendly graphical representation of where a learner is in the learning design.

Lesson Class
The lesson class is the group of users who will participate in the lesson. Some users are assigned learner (or participant) status, which gives access to the Learner interface. Some users are given Staff status, which gives them access to the Monitoring interface.

Progress Engine History

The LAMS 2.0 progress engine is the third version of the algorithm of progressing the learning design since the born of LAMS software. LAMS has gained lot of lessons from previous implementation in terms of scalability and maintainability. Before documenting the current design, it is handy to review the experience and pitfalls from previous versions.

The first version of LAMS progress engine is based on JMS asynchronous messaging. This approach has been referred to a Polling. Under the polling architecture, instead of LAMS itself to figure out what is the next activity for the learner, the client side (essentially the Flash component) drive the progress engine to keep moving by keep sending message to the server. This approach works fine if the number of concurrent learners is very small. However, once we moved onto the loading testing stage of LAMS application, the overhead generated by client messaging quickly becomes the major bottleneck that halted the whole system. The high concurrency nature of progress engine decides asynchronous messaging is not a suitable technology choice for LAMS.

This was then replaced with an approach that completely removed the overhead result from polling architecture, which make it possible for LAMS to scale under hundreds of concurrent users. However due to the lack of proper data structure to hold the learner's progress status, the Flash learner progress bar has to request the server to go through the entire learning design to calculate the exact position for display. Profiled showed this calculation consumed 65% of the system resources. The bigger the learning design is, the more resources need to be consumed by this profiling. Even worse, this calculation needs to be done every time when each learner wants to progress to next activity. Consequently, designing a flexible data structure that holds necessary progress data becomes the major rationale we followed in LAMS 2.0 Progress engine.

Domain Object Model and Database Model

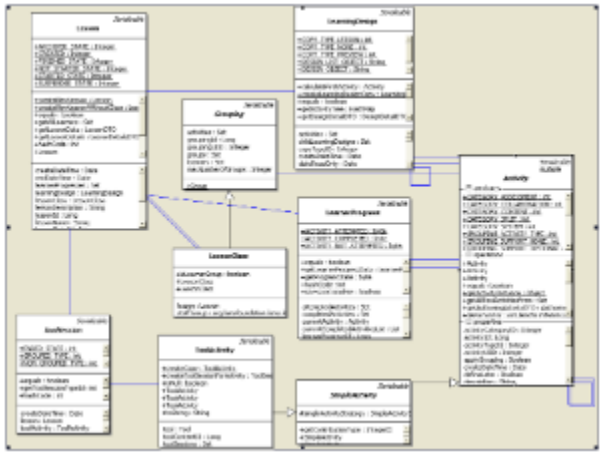

Figure 1 Lesson related domain object diagram

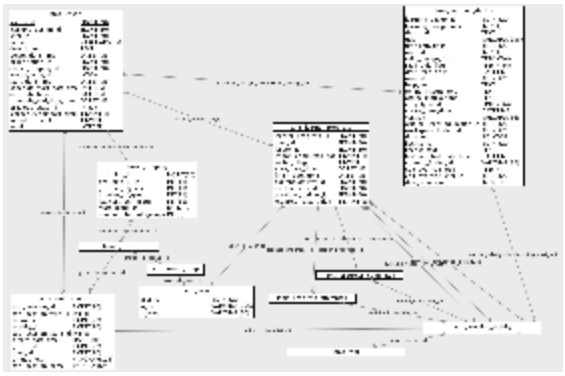

Figure 2 Lesson related database design

The above diagrams are fragments of the entire domain object diagram and database to illustrate the objects that handle lesson related learner progression. The major inheritance mapping we adopted is "Table per class hierarchy" so the database diagram doesn't distinguish between tool activities and the abstract activity class, as all activities are stored in the one table in the database.

There are four major components for a lesson:

- The **LearningDesign** defines the major learning sequence that students in a lesson should complete.
- The **LessonClass** holds all the students information who are participating a lesson. It is also a special grouping type in LAMS to perform class based learning tasks.

The relationships between the group tables looks a little confusing, as the grouping classes are used for both normal groups and the lesson class. A Lesson has a LessonClass, which is a subclass of Grouping (and hence is stored in the lams_grouping table). The staff_group_id column is only used for the LessonClass object. The LessonClass has a single group in its groups collection, and this group is all the learners in the LessonClass. Then there is a second group which is the staff group, and that group's id is stored in the staff_group_id column.

In the course of the lesson, there maybe "normal" groups set up, such using the Random Grouping. These will have a Grouping class and one or more Group classes, so one lams_grouping row and multiple lams_group rows.

- **ToolSession**. As you already know, learning design is a static aggregation of learning activities. Each activity refers to a tool session record that holds the runtime data for the activity. Moreover, the ToolSession is the means by which we assign a batch of learners to a particular instance of an activity. If the activity is being used for the whole class, then the ToolSession is associated with the LessonClass' group of learners. If the activity is grouped, then there is a separate ToolSession for each group, creating a separate instance of the ToolActivity.
- The **LearnerProgress** records the progress data for each learner. One lesson should have 0 or many learner progressions. Every learner should only have one learner progress record for a lesson.

The learner's progress object is represented as one class in Java, but the attempted and completed sets are implemented as separate tables in the database. So there are 5 direct and indirect links from the learner's progress to activity - current activity, previous activity, next activity, attempted set and completed set.

The completed and attempted sets are exclusive sets. When a learner starts an activity then the activity is recorded in the attempted set. When the activity is completed, it is removed from the attempted set and added to the completed set.

Lesson States

The lesson passes through a number of states. The states are defined in the Lesson class.

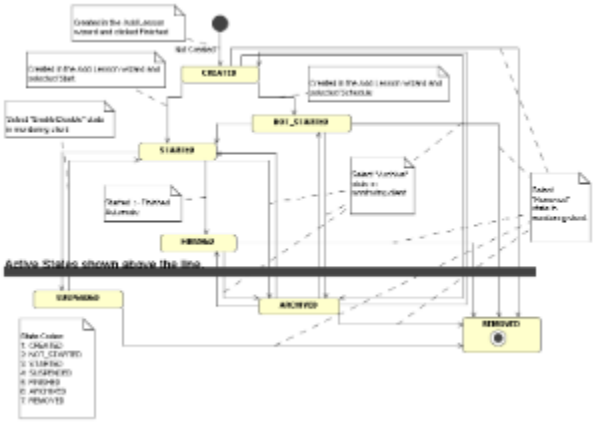

Figure 3 Normal Flow for Lesson States

As of LAMS 2.1, you will be able to move from any state to removed - you do not need to make it archived first.

Lesson State	Description
CREATED	A newly created lesson. The learning design has been copied. The lesson class may or may not have been configured. The lesson is shown on the staff interface but not on the learning interface.
NOT_STARTED_STATE	A CREATED lesson may be scheduled to start at a particular date. When it is schedule, state is set to NOT_STARTED_STATE for lessons that have been scheduled. The lesson is shown on the staff interface but not on the learning interface.
STARTED_STATE	The lesson has been started. The lesson is shown both the staff interface and learner interface. Whilst in this state, the learners can participate in the lesson.
SUSPENDED_STATE	A lesson that is in STARTED_STATE may be suspended. While suspended, the lesson can be seen on the staff interface but not on the learning interface. When the lesson is no longer suspended, it returns to STARTED_STATE.
FINISHED_STATE	The state for lessons that have been finished. Currently not used, as there is no definition of "Finished" as new learners could be added at any time.
ARCHIVED_STATE	The state for lessons which are shown as inactive on the staff interface but no longer visible to the learners.
REMOVED_STATE	The state for lessons that are removed and never can be accessed again. In reality, the records are still in the database and could be retrieved by setting the state back to STARTED_STATE directly in the database.

Teacher's Perspective Use Cases

Primary Actor	**Use Cases**

Teacher	Create and Start Lesson
Teacher	Create Lesson
Teacher	Start Lesson
Teacher	Force Complete Learner
Teacher	Live Edit

Use Case Name:
Create and Start Lesson
Actors:
Teacher
Description:
The teacher creates a lesson for a learning session.
Preconditions:
The teacher loads up the "Add Lesson" interface.
Postconditions:

1. A lesson gets created.
2. A copy of learning design for lesson gets created.
3. A copy of tool content gets created.
4. A new lesson class is created.
5. Progress engine data such as the tool sessions is initialised.

Normal Flow:

1. Teacher clicks on the Add Lesson button. Each Add Lesson button is association with an organisation and this will become the organisation used to display the list of potential users, and be associated with the lesson.
2. Teacher selects the learning design he wants to run for the new lesson and clicks Next.
3. Teacher select the users who will be in the lesson and clicks Next.
4. Teacher types in the title and description for the lesson.
5. Teacher clicks on the Start Now to tell LAMS to create lesson.
6. LAMS initialises the lesson:

- Creates a read only copy of the learning design, of type COPY_TYPE_LESSON
- LAMS notifies all tools that involved in the learning design to create a copy of their content for lesson.
- Creates a new lesson for the copied learning design (without a lesson class), associated with the selected organisation. Lesson status is set to Lesson.CREATED.

1. LAMS creates the LessonClass, based on the list of users supplied by the GUI, and associates it with the lesson.
2. LAMS starts the lesson

- Goes through all the non-grouped tool activities and initialises the tool sessions.
- Goes through all the scheduled based activities and start scheduler of each task if necessary.
- Goes through all the branching activities and assigns it a grouping (if one doesn't exist) so that the users in the branch can be tracked.
- Marks each activity as initialised. This is needed later for Live Edit.
- Sets lesson status to Lesson.STARTED_STATE

Use Case Name:
Create Lesson
Actors:
Teacher
Description:
The teacher creates a lesson for a learning session and starts it ready for use.
Preconditions:
The teacher loads up the "Add Lesson" interface.
Postconditions:

1. A lesson gets created.
2. A copy of learning design for lesson gets created.
3. A copy of tool content gets created.
4. A new lesson class is created.

Normal Flow:

1. Teacher clicks on the Add Lesson button. Each Add Lesson button is association with an organisation and this will become the organisation used to display the list of potential users, and be associated with the lesson.
2. Teacher selects the learning design he wants to run for the new lesson and clicks Next.
3. Teacher select the users who will be in the lesson and clicks Next.
4. Teacher types in the title and description for the lesson.
5. Teacher selects on the Schedule or Start In Monitor to tell LAMS to create lesson but not start the lesson.
6. LAMS initialises the lesson:

- Creates a read only copy of the learning design, of type COPY_TYPE_LESSON
- LAMS notifies all tools that involved in the learning design to create a copy of their content for lesson.

- Creates a new lesson for the copied learning design (without a lesson class), associated with the selected organisation. Lesson status is set to Lesson.CREATED.

1. LAMS creates the LessonClass, based on the list of users supplied by the GUI, and associates it with the lesson.
2. If the teacher selected Start In Monitor, the lesson is left as Lesson.CREATED. If the teacher scheduled the lesson, then a scheduler task is set up to start the lesson automatically and the lesson status is Lesson.NOT_STARTED.

Use Case Name:
Start Lesson in Monitoring
Actors:
Teacher
Description:
The teacher starts a lesson that was created previously.
Preconditions:
The teacher loads up the "Monitor" interface.
Postconditions:

1. Progress engine data such as the tool sessions is initialised.

Normal Flow:

1. Teacher clicks on the Start Now button.
2. LAMS starts the lesson

- Goes through all the non-grouped tool activities and initialises the tool sessions.
- Goes through all the scheduled based activities and start scheduler of each task if necessary.
- Goes through all the branching activities and assigns it a grouping (if one doesn't exist) so that the users in the branch can be tracked.
- Marks each activity as initialised. This is needed later for Live Edit.
- Sets lesson status to Lesson.STARTED_STATE

Similarly to the previous test case, if the lesson is scheduled, LAMS will do the same "start the lesson" process automatically at the correct time. The teacher may also go into the Monitoring interface and do "Start Now" on a scheduled lesson, or set up scheduling on a non-started lesson.

Use Case Name:
Force Complete
Actors:
Teacher
Description:
The teacher moves a learner from one part of the lesson to another part of the lesson.
Preconditions:
The teacher loads up the "Monitor" interface and brings up the sequence tab. The learner is partway through the lesson.
Postconditions:

1. Learner's progress is updated

Normal Flow:

1. Teacher finds the learner's icon on the learning design diagram, then clicks and drags the learner's icon to a later activity, or to the "finished" bar at the end of the screen.
2. If the teacher dragged to the icon to a later activity, the call to the server passes the activity id of the activity just proceeding the selected activity (ie the "last" activity to be marked as completed"). LAMS then processes the learner's path through the design to the nominated activity using the progress engine to determine the next task. Along the way, LAMS will create tool sessions and do branching calculations as required. Once the "last" activity is reached, the learner is started on the next activity. This gives the affect of moving the user to the nominated activity.
3. If the teacher dragged the icon to the finished bar, the call to the server does not pass any activity id. LAMS will process the learner's path to the end of the learning design.
4. In either case, LAMS obeys the normal "stop" mechanisms. For example if the user reaches a permission gate that is shut, or a chosen grouping but has not been grouped, then the processing will stop and LAMS will return the current activity id of the activity where the user is stopped. Also, LAMS only marks the user's progress as completed - it does not call the tool and tell the tool that this user has completed the activity. Therefore if the learner returns to the activity later they will complete the activity the same as normal (e.g. in MCQ they will get to answer the questions).

Use Case Name:
Live Edit
Actors:
Teacher
Description:
The teacher makes a small change to the learning design of an existing lesson. Note: this is only designed for quick changes to the learning design structure, not large scale changes. If only the content of an activity is to be changed, it should be changed via the activity's screen in monitoring.
Preconditions:
The teacher loads up the "Monitor" interface and brings up the sequence tab.
Postconditions:

1. The learning design attached to the lesson is updated.

Normal Flow:

1. Teacher clicks Live Edit, and confirms that they wish to edit the learning design.

2. LAMS sets up the design ready for editing. As authoring will not allow a user to edit a read only design, edit override flags are set. To stop users moving through parts of the sequence that are being edited, a temporary stop gate is added after the last activity attempted by a learner.
3. The monitoring client is "replaced" by a cut down version of authoring. The teacher is able to change any part of the design that the learner's have not yet reached (ie after the temporary stop gate). The teacher makes their changes then clicks Apply Changes.
4. LAMS updates the runtime copy of the learning design (not the original version created in authoring). The stop gate is removed. Any new activities are initialised (hence the initialisation flag set when the lesson was created). If any of the learners had completed the lesson, then their completion flags are cleared so that next time they go into the lesson, they new activities will be displayed.
5. The teacher is taken back to the normal monitoring screen.

Note: If the teacher closes the monitoring/editing screen before completing the editing, then the lesson is left in editing mode. If that teacher returns to monitoring they will be taken back into Live Edit. If another teacher attempts to access monitoring, then they will be blocked from using monitoring and will be told who is editing the lesson.

A more detailed explanation of how Live Edit works is given elsewhere.

Learner's Perspective

Business Class Diagram

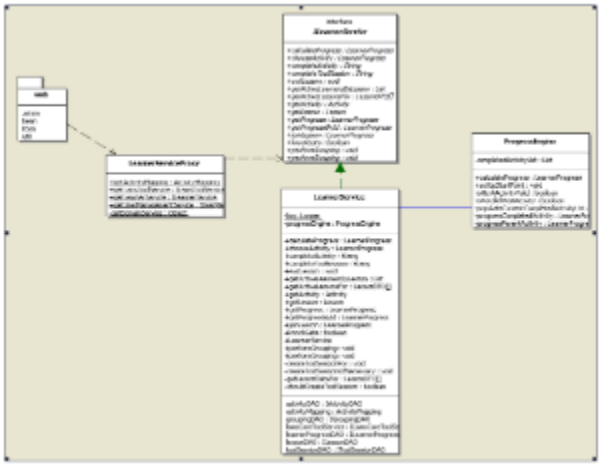

Figure 4 Business Class Diagram - Learning

Figure 4 shows the service layer class diagram for lams_learning module. Apart from standard service façade pattern we followed like the other modules. We have factored out the progress engine related calculation into the class "ProgressEngine". The algorithm that implemented in this class will be explained in detail with sequence diagram and activity diagram in the later sections.

The other layer to the Progress Engine is the web layer. The web layer defines all the screens for the system activities as well as the action classes that are used to display the tool pages. It also includes the ActivityMapping class that determines the URL to be called.

In general the LearnerService and the ProgressEngine are responsible for maintaining the state of the Lesson (opening gates, adding users to lessons, updating a learners's progress) and the web layer interprets the learner's progress to determine which screen to display. The aim is to keep the learner's progress describing the current state of the progress, rather than what the screen looks like. The field parallelWaiting (which implies that one of the parallel activities is waiting for the other to finish) may break this principle a little but it was hard to avoid this field but we should avoid flags like this in the future if we can.

Major Algorithm Explanation

This section explains the algorithms implemented underneath to achieve the functionalities described in the above sections. To explain the algorithms, sequence diagrams, activity diagrams and code snippets will be provided wherever necessary.

LAMS 2.x activity page loading Mechanism

To fully understand how progress engine display the JSP of the next activity, we have to explain the activity page loading mechanism first. As you might know, the LAMS activities are classified into two big categories - complex activity and simple activity.

From the perspective of displaying these activities using JSP, we would like re-classify these activities into three major categories - Single frame JSP activity, multi-frame JSP activity and LAMS system JSP activity. These three categories and corresponding page loading strategy are explained as follows:

- Single framed JSP activity - A single frame loading JSP is used to display the content of this type of activity. Referring back to the activity hierarchy designed in the object model, all simple activities as well as sequence activity falls into this category.
- Multi-framed JSP activity - When we are dealing with parallel activity, single frame is no longer enough to display two or more activities on the same screen. It is therefore multi-frame needs to be initialized before the actual activity contents can be loaded into the JSP.

- LAMS system JSP activity - A single framed JSP is also used for system activities. The difference between this category and single framed JSP activity is that the actually activity content won't be displayed directly. Instead, a LAMS page will be loaded to represent activity's content. Options activity is a good example - LAMS option page will be shown with all the sub-activities outlines and instructions rather than the contents of all optional activities.

Why did we make this classification? We want all activities contents are loaded by LAMS loading page rather than shown directly from Struts action forward. Why LAMS loading page is important? LAMS loading page has to exist because of two major reasons. Firstly, the loading page is a clean solution to add extra frame and clean unnecessary frame whenever required. Without using an intermediate loading JSP, we could not find a solution to achieve the same functionality.

Secondly, the loading page is a good place to send update progress bar message to the Flash client. In LAMS 1.0 we had the entry JSP page of each tool doing updates of the Flash progress bar. This resulted in a tight dependency between tool and the Flash progress bar, duplicate code and consequently a maintainability nightmare.

Isn't the loading page is extra overhead and makes LAMS very "chatty" (ie a lot of calls to the server)? Yes, but for the benefits it has brought use, it was worth the overhead. If usage and/or profiling show an issue in this area then we will optimise the code however the overall methodology appears sound at this stage and there are other areas we can look at for performance improvements.

Move to next activity

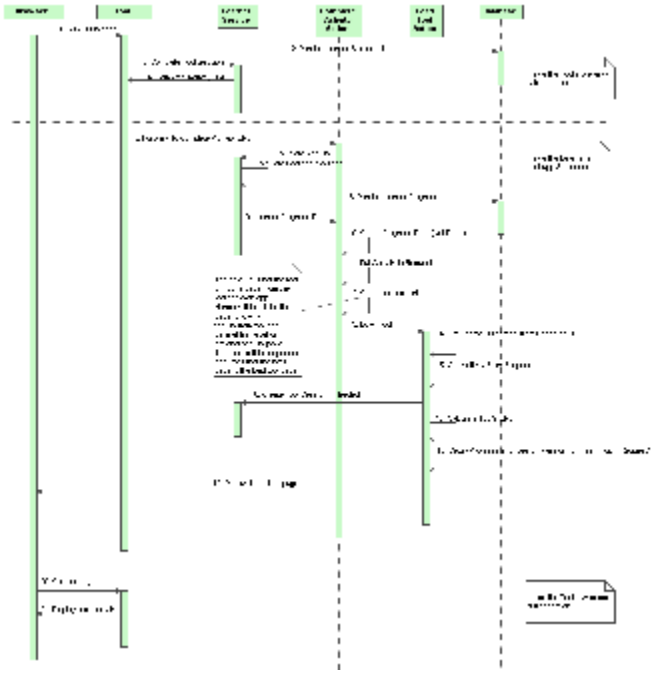

Figure 5 Move to Next Activity - Sequence Diagram

Above diagram shows the overall algorithm used to complete the moving from one activity to the other. This process usually triggered by clicking on the "Finished" button of a particular tool page. Originally we did Step 2 to Step 8 in one transaction so as to ensure there is no inconsistent state between tool and LAMS server. However when we implemented Live Edit, there was an increased chance that calculating the learner progress would fail due to some case we hadn't considered (setting up Live Edit while allowing the learners to proceed and not putting in synchronisation points which may slow down the progress engine wasn't easy). So we split the update into two transactions, so that the tool is always able to persist their data, and then it forwards to a URL in the learning web app to do all of the progress updates.

It is important to notice that LAMS travel from tool web application to lams_learning web application via http URL calls. This means that any calls to the core services that are NOT in the tool's Spring contexts are available to the learning logic. The tool's do not include all the context.xml files for the core, so some of the service beans are not available.

The lams_learning is responsible for calculating the URL for next activity and forward to it. Two major algorithms, Calculate the next activity at service layer and Mapping the Activity to the URL, are hidden in above diagram, which will be explained in following sections.

In the diagram above, many of the calls such as Calculate Learner Progress, Put Activity in Request, etc, are really calls to other classes, but the other classes have to be hidden or the diagram would be impossible to read.

Calculate the next activity at service layer

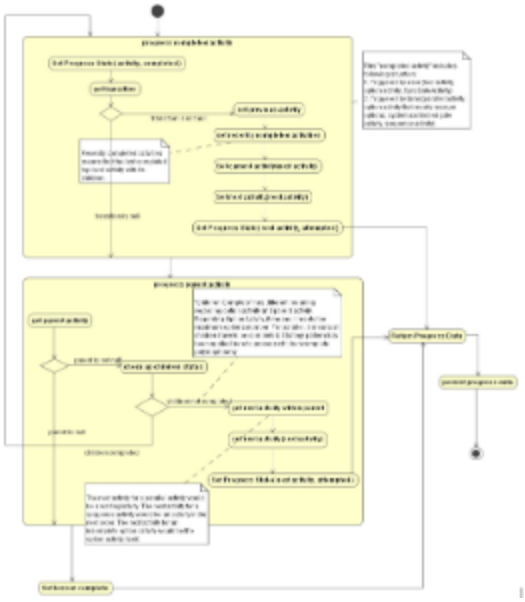

Figure 6 Calculate the Next Activity - Activity Diagram

The first major functionality of progress engine is to be able to figure out what is the next activity in the learning design when one activity is completed. Figure 6 shows the detail logic of calculating next activity at service layer. The trigger of this procedure is the completed activity that is fed either by the web layer or the calculation procedure itself.

According to the learning design structure we defined for LAMS 2.x, transitions are used to identify the next activity either within a Sequence Activity (in an Optional Sequence or a Branch) or at the top level activities. All children activities within Optional Activities and Parallel Activities are structured using order id. For Optional Sequences, the Sequence activities within the Optional Sequence activity are ordered by order id, but the activities within the Sequence activities are ordered by transitions.

In this context, transition has been used as the major condition to identify the completion of a node. In other words, we assume we've found the next activity if there is transition following the completed activity. Furthermore, a completed activity without transition also indicates it might have a parent activity or it is the last activity in a learning design.

Whenever a parent activity is identified, the algorithm looks through all children activities belong to that parent. If all children are completed, it recursively feed this parent activity into "process completed activity" logic to figure out the proper next activity. The algorithm is meant to be able to walk through the whole activities tree until a transition is identified. If incomplete children exist, the algorithm should pick up the right next child activity according to the type of this parent and return the learner progress back to the client. The actual calculation of whether a complex activity is completed is done in the activity's strategy class (e.g. BranchingActivityStrategy, ParallelActivityStrategy). Therefore the progress engine doesn't need to know what makes a complex activity completed.

Similarly, the web layer part of the complex activities combined with logic in the Strategy class, determines which child of a complex activity is shown first.

There are a number of complications to the algorithm that are not shown on the diagram.

- The **"Stop After Activity"** flag on an activity indicates an immediate end to the learning design, even if the parent activities would lead the user on to a further activity. This is used in Branching to stop a learner at the end of a branch, rather than continuing with the activities after a branch. So before it attempts to navigate to the next activity via transitions or parent activities, it checks this flag and if it is true then sets the learner's progress to **LESSON_IN_DESIGN_COMPLETE**. If the learner reaches the normal end of the design, then the learner's progress is set to **LESSON_END_OF_DESIGN_COMPLETE**.
- We have different value for the end of design flag to support Live Edit. When a Live Edit is completed, LAMS finds all the learners that are LESSON_END_OF_DESIGN_COMPLETE and sets them to LESSON_NOT_COMPLETE, so the next time they join/resume a lesson, the progress engine recalculates their next activity (which would be a new activity at the end of the learning design). However if a learner stopped at the end of the branch we do not want them to be set back to incomplete, as they are still completed. Hence the two values.
- Live Edit puts a stop gate in the design to stop users going passed the current editing point. However it wasn't possible to guarantee that users could sneak past this gate if they happened to be going to the next activity just as the gate is being set up, without adding synchronisation that could cause a bottleneck. So before assigning the next activity to a learner, we call LearnerProgress.canDoActivity() to check that if Live Edit is in progress then learners can only do activities that are already read only (ie an activity that someone else has done). There is also a catch all method LearnerProgress.clearProgressNowhereToGoNotCompleted() if there was a problem setting up the user's next activity - this is only likely to occur due to some unlucky combination of factors due to Live Edit. This method clears the current and next activity and displays an error message. Next time the learner tries to join the lesson, the progress engine will recalculate the learner's position and it should work its way out of the problem situation.

Note: The "recently completed activities" list referred to in the diagram doesn't seem to be used - potentially this could be dropped. The functionality is implemented by the method LearnerProgress.populateCurrentCompletedActivityList() so we could try dropping this method and test to ensure that the progress engine is still working correctly.

Mapping the Activity to the URL

Figure 7 Calculate the URL from Learner Progress- Activity Diagram

Once the service layer progress engine calculated the next activity to move on to and returned the learner progress, the web layer progress engine needs to figure out what is the proper URL for the next activity. Figure 7 illustrates the algorithm to fulfill this functionality.

Some calls do redirects, some do forwards. The intention is that in moving to the next activity, something should redirect to ensure that we don't accidently go back to the last screen. Normally the redirect is done via the displayTool.jsp but in other cases it is handled by the Struts "forward", which can be setup to be either a forward or a redirect. One day it would be nice to document exact at which point in the processed a redirect is done, so that we can ensure all "next activity" calls do a redirect but don't do a whole pile of redirects.

Join a lesson with Retrieving Progress Data - Needs Updating

Figure 8 Join lesson with retrieving progress data - Sequence Diagram

Based on the understanding of major progress engine algorithm and page loading mechanism, we can further the explanation of the joining lesson algorithm. The explanations of join lesson and retrieving progress data are combined because these two use cases usually happen consecutively. Once LAMS finishes joining the learner to the lesson and send the wddx acknowledgement message with the URL of loading page calculation struts action back to Flash, Flash sends the request to the LAMS server to grab entire learner progress structure to build up learner progress bar. The progress data request from Flash has to be sent after the Flash client receives the "Joined Lesson" message from LAMS because it needs to load the latest progress data.

Communication between Server progress engine and Flash Progress bar

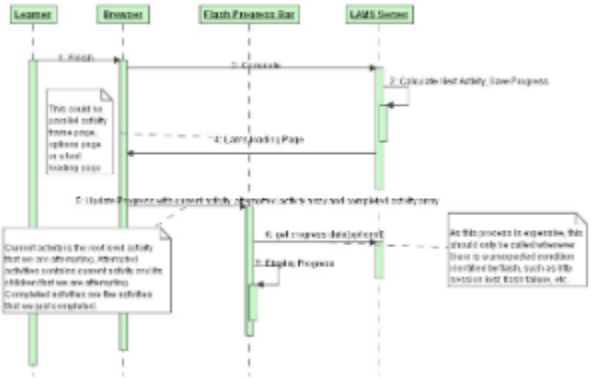

Figure 9 Server and Flash Communication - Move to next activity

We have already explained the algorithm of moving to next task on the LAMS server side. How do we achieve the visual indication of the movement on the Flash progress bar? Figure 9 explains the detail logic behind the scene. A Flash movie update script is embedded on the LAMS loading page (and the parallel and optional pages) and this movie opens a Flash pipe which passes the IDs for the current, attempted and completed activities. The code for the Flash swf is in the lams:Passon tag, so inclusion in a few jsps is easy.

Flash then updates the progress bar with the activity data, based on the design details that it received when the learner window was first opened. All the data has been calculated on the server side as part of the "move to next task". The data sent back to Flash is only small, which minimises the Flash - Java communication cost. As a safe guard, Flash can send the request to server to restore the whole progress bar structure if there is an issue on the Flash side.

One of the parameters passed to Flash is the design version. This is updated whenever a Live Edit is done, triggering Flash to get a new copy of the design when the version changes. This allows learners to see an updated design without having to exit/restart learner.

The code that actually moves to the next activity (on the "Loading Next Activity screen") is embedded in Javascript. If we are using the Flash progress bar then Flash calls the javascript method to advance to the next activity. This was done to eliminate timing issues we were getting between Flash updates and the JSP advancing. For this reason, the Flash progress bar had to be used in LAMS 2.0.

Non-Flash Progress Bar

In LAMS 2.1 we have introduced the Flashless learner interface (apart from Chat). If the learner is not using the Flash progress bar then the jsp calls the javascript to do the page redirect when the page is loaded. There is no need to update the non-Flash progress bar as that is generated only when the learner requests the progress display.

The non-Flash progress bar is generated using the ProgressBuilder class, which is based on the LearningDesignProcessor. During the parsing of the design, it builds a set of URLs to show completed activities. The data is then passed to a JSP for rendering. This is done every time, so we will have to watch that it doesn't become a performance issue - the code was done quickly rather than being coded to be quick. Having the work done on the server does increase the server load - normally the calculation of what should appear green, blue, etc is done in Flash and hence is done in the client. But it isn't done on every progress (as it in Flash) so hopefully it won't be a big issue.

Authoring Preview

When an author runs a preview, it actually runs a complete lesson, with a single learner in the class. There is no monitoring client. Apart from the differences mentioned for Gates and Grouping below, the following differences apply to Preview.

- The LearningDesign is set up with COPY_TYPE_PREVIEW.
- The LearningDesigns are hidden in the Workspace tree.
- Any activities that are set to Define Later are run with their default content (after warning the user).
- When a Branching Activity is reached, the author is able to select any branch rather than being forced to do the branch that would normally be selected.
- The user is able to "jump ahead" to an unstarted activity using the Flash progress bar. This triggers a force complete of the activities in between the current activity and the activity selected.

System Activities

Parallel Activities

The URLs for handling Parallel Activities are hardcoded. As it is a "structural" screen, there isn't a lot of point making this a pluggable activity, so no point moving the URLs to the SystemTool table. A Parallel Activity is defined to be completed when both of its children are completed.

Parallel activities are one of the hardest activities to get right in the web layer of the progress engine, as their behaviour varies when they are completed and appear in a popup, compared to in the normal window. So when you make changes to them, make sure you test the parallel activities on normal completion, in the popup window and as part of a complex sequence, such as an optional activity.

Optional Activities and Optional Sequences

Optional Activities and Optional Sequences are the same thing on the server. They have different Activity types for the purposes of authoring (to make life easier for Flash) but they are handled in the identical manner. An Optional Activity contains a set of Activity objects, which are displayed on the screen in orderId order and the learner selects which activity to choose. Optional Sequences just happen to be a special case where all contained Activity objects are SequenceActivity objects. The progress engine could cope with a single type of Optional Activity that mixed single normal activities with sequences, but it would be too confusing in the authoring client.

The URLs for handling Optional Activities are hardcoded. As it is a "structural" screen, there isn't a lot of point making this a pluggable activity, so no point moving the URLs to the SystemTool table. An Optional Activity is defined to be completed when the user selects "Finish" on the activity when the minimum number of child activities have been completed, or when the maximum number of activities have been completed.

Grouping

Grouping current supports two sorts of grouping - Random and Teacher Chosen. Random Grouping supports both "Group Learners by Number of Groups" and "Group Learners By Number of Learners Per Group" are marked as extension rather than inclusion because only one of these two conditions can be

applied when the teacher tries to create a random grouping activity. Applying both conditions will implicitly set the limit of the number of total learners in a class, which might make it difficult for a teacher to assign learners into the class when the lesson gets started but the authoring user interface only allows one condition to be set.

Teacher Chosen grouping allows the teacher to create groups and assign users to groups on the fly.

Grouping is a system tool and the logic for grouping is spread throughout the core. As such, we can't "plug in" someone else's grouping algorithm. However it is a SimpleActivity that implements the ISystemTool interface and the URLs for Grouping screens are defined in the lams_system_tool table, so there is the potential for calling a grouping algorithm in another web-app if authoring could be made to support it and we added calls to allow the external web-app to create Group objects.

Object and Database Models

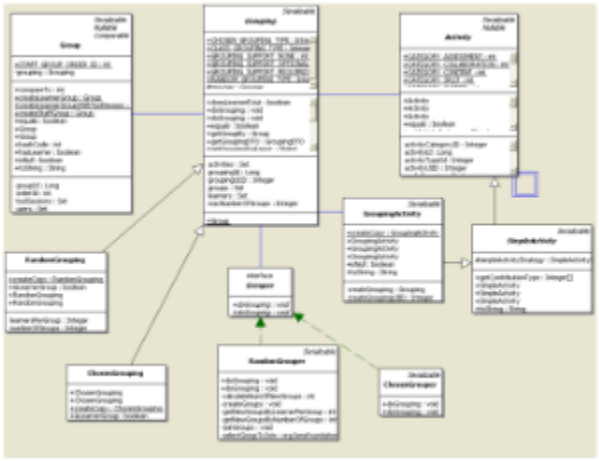

Figure 10 Grouping Domain Object Model

The above diagram shows the object model fragment for grouping implementation. Instead of having two different grouping activities, we delegate the grouping specific operations to the class "Grouping", which is extended by two sub-classes - RandomGrouping and ChosenGrouping. These two classes act as the data holder for two types of groupings. Furthermore, the strategy interface "Grouper" is created purely for calculation purpose. doGrouping method within grouper performs the grouping algorithm in a polymorphic way. The end result of this algorithm is to create a number of groups inside the grouping object.

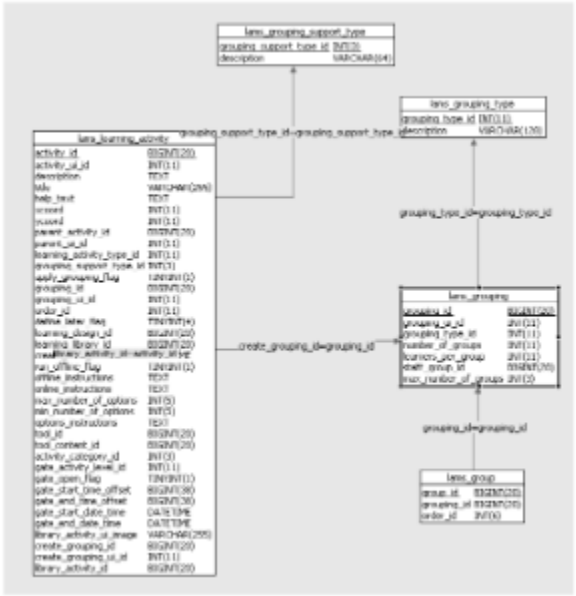

Figure 11 Database Model

Compared to the domain object model, the database design is much simpler. Most of the inheritance relationships you have seen are mapped using "Table per class hierarchy" to improve efficiency.

Ultimately, LAMS should be able to cope with tool having a non-grouped mode and grouped mode, hence the **GroupingSupportType**. Although LAMS 2.0 tools are not able to freely switch between grouped and non-grouped mode this database design will allow more flexibility in this area in the future. There was a debate over what levels of grouping would mean to a tool and as all our tools only care about getting "some group of users", we implemented a very simple version of the grouping support until we have an actual case where we want to do something more complex.

At present the three modes are used to control the users associated with the tool session, so the tools don't have to do anything with respect to the grouping type. The three basic supported types are:

1. NONE - It means that the activity that doesn't support grouping. Any tools sessions created will contain only a single user.
2. OPTIONAL - It means that the grouping capability is available but not compulsory. When the author applies a grouping to an activity, it becomes a grouped activity. Otherwise it is a non-grouped activity. If there was a grouping applied, then a tool session is created for each group, otherwise there is one tool session for the whole LessonClass.
3. REQUIRED - It means that the grouping capability is compulsory for a particular activity. If the author doesn't apply grouping on an activity, LAMS will apply class level grouping on this tool to avoid system failure. So in practise, OPTIONAL and REQUIRED have the same outcome on the LAMS server.

The Authoring Flash client also uses this value to determine whether or not to allow (OPTIONAL) or force (REQUIRED) the user to select a grouping for the activity.

All LAMS tools currently use OPTIONAL.

Preview

When running a lesson in preview, there is no monitoring screen so a teacher chosen grouping cannot be done. In preview, when a chosen grouping is reached, the author sees a screen indicating that a chosen grouping would normally be done here, but that a random grouping will be done so that the preview will proceed. LAMS then applies the default random grouping algorithm and preview continues.

Gates

A gate is used to stop the class at some point of the lesson. This is useful for teacher to control the pace of the lesson and review what students have done, or to allow all the students to move on to an activity together (ie stop a few learners running ahead of the class). Once a gate is opened, it cannot be shut.

There are three types of Gates:

1. **Permission Gate** This gate won't be open until the teacher permits the students to go through. The teacher can open the gate any time he wants to at the monitoring UI.
2. **Schedule Gate** The teacher sets up a time offset against the lesson start time in authoring. Once the lesson started, the system scheduler will run gate scheduler to open and close the gate automatically according to the predefined condition.
3. **Sync Gate** This is very similar to the LAMS 1.0 sync point. It stops the learners in front of the gate until all the learners have reached the gate. However the definition of "all the learners" is "all the learners who have actually started the lesson". So if you have 30 users assigned as learners but only 3 learners have actually started the lesson, then the Sync Gate will open when the three learners have reached the gate. This has avoided the problem in LAMS 1.0 where a learner missing that day would hold up the Sync Gate, but it means that Sync Gates are not very useful at the start of a lesson.

At present it is not possible to make a gate work over a subset of the class. This means that a Sync Gate won't work in Branching or Optional Sequences, as it is likely that not all the learners in the class will be in the same branch or sequence - otherwise there wouldn't be much point branching! This is a shortcoming of the current model. See Separate Instances of System Tools (below).

Gates are a system tool and the logic for grouping is spread throughout the core. As such, we can't "plug in" someone else's gate algorithm. However it is a SimpleActivity that implements the ISystemTool interface and the URLs for Gate screens are defined in the lams_system_tool table, so there is the potential for calling a grouping algorithm in another web-app if authoring could be made to support it.

Object Model

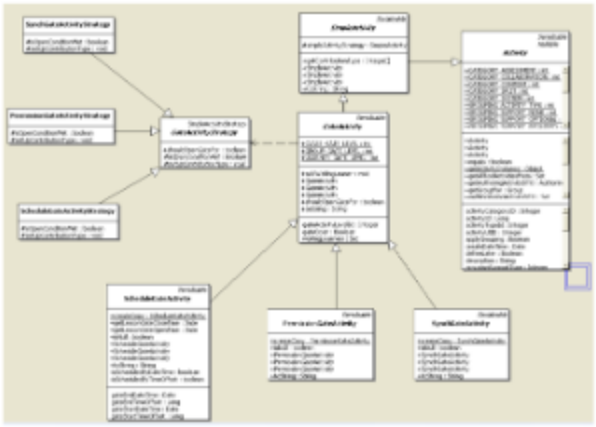

Figure 12 Gate Domain Object Diagram

Gate activity is part of entire activity hierarchy. It defines the data holder for different types of gates in LAMS 2.0 Strategy pattern has been applied here to check up the condition of opening gate in a polymorphic manner.

Gate Algorithm

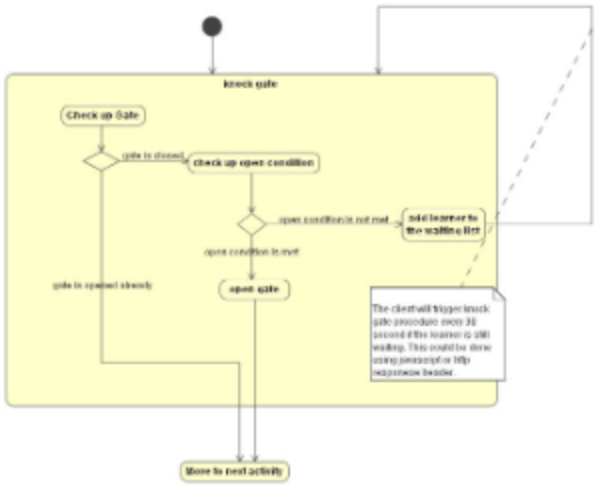

Figure 13 Open Synch Gate Activity Diagram

The knock gate procedure will firstly triggered by the completion of the activity before the gate activity in a learning design. As we have shown in the note, waiting learner will knock gate every 30 seconds. This is achieved via a META HTTP-EQUIV="Refresh" statement, and knocking on the gate will open it when enough learners are waiting.

Similarly, the Permission Gate and Schedule Gate will do a refresh to check if the Gate is open. Once the gate is open, the learner is taken onto the next activity automatically, saving the user clicking refresh constantly.

Preview

When running a lesson in preview, there is permission gates and schedule gates would be a problem. In preview, when these gates are reached, the author sees a screen indicating that a gate would normally apply here, but that the gate will be opened immediately so that the preview will proceed. LAMS then opens the gate and preview continues.

Branching

See Branching

Future Improvements

Lesson and Progress Data Caching

At present, the lesson and progress data are not cached, although the core User objects are cached. These are obvious candidates for caching, as they are complex structures and are required on every calculation of the next activity. Just putting the objects straight into the user's normal HttpSession or even the SharedSession doesn't work well. The HttpSession varies from web-app to web-app and some of the code to execute the "next" functionality is run within the tool's web-app, although the code itself lives in the lams-learning.jar. Putting it in the SharedSession would avoid this, but then we have to have two different entries for the lesson and progress data for the preview screen and the learner screen as both can be running simultaneously. Also, putting the lesson in the SharedSession is not a good idea as the Lesson is the same for all the users in the Lesson and potentially we could be caching a Learning Design (attached to the Lesson) which is now out of date due to a Live Edit. So the Lesson and the LearningDesign will probably need to be cached using the JBoss cache, which interacts with Hibernate so that the object in the cache is up to date.

In general, there is a lot of "put activity in the request", "get learner progress somehow", "put activity id/learning design id in the request parameters", etc in the progress engine. Originally it was carefully tuned so that each system screen got only what it needed, but then we'd change the screen to do more and we'd need more parameters, etc so it grew higglepiggldy. When we look at caching the progress data then we need to clean up where we put the activity in the request.

Making System Tools More Pluggable

At present, the system tools cannot be easily replaced with other versions of the tool. For most of the system tools this is not an issue, but we already have people wanting to use different grouping algorithms (say to access the groups already configured in their LMS) and potentially people may wish to open gates based on external systems. The lams_system_tool table has been set up to try to make a way to do this in the future and to make it easier to set up new sorts of branching (as you can tweak URLS in a table rather than hardcoding them). But before we can have pluggable groupings we need to address how this is done in authoring, as the grouping choices are hardcoded into the Flash interface and into the Activity class hierarchy.

Separate Instances for System Tools

Every tool activity will be associated with a set of tool sessions to hold its learning time data for a different group. However, system tools (e.g. gates, grouping, branching and sequence activities) do not have the equivalent of a tool session. Therefore it is not possible to easily assign a grouping to a gate, or restrict a gate to the users currently doing a branch. To do this, we would need to make the gate logic know about the pseudo-groupings used for branching. It also means that we have no way of creating subgroups (groups of groups), which I think we could in LAMS 1.0.