Trucstudio - Refactoring clusters

Master Thesis

By: Florian Geldmacher
Supervised by: Michela Pedroni
Prof. Bertrand Meyer

Student Number: 02-911-774
# Index

Abstract ........................................................................................................................................... 3

1 Intro ................................................................................................................................................ 3
   1.1 Problem description ................................................................................................................ 3
   1.2 New implementation .............................................................................................................. 3
   1.3 Difficulties ............................................................................................................................ 4
   1.4 Outline ..................................................................................................................................... 4

2 Background – Differences to the old system .............................................................................. 4
   2.1 Model ....................................................................................................................................... 4
      a Inheritance ............................................................................................................................ 4
      b Model-View-Controller ......................................................................................................... 4
      c Old implementation .............................................................................................................. 5
      d New implementation ............................................................................................................ 6
   2.2 Action ....................................................................................................................................... 8
      a Old implementation .............................................................................................................. 8
      b New implementation ............................................................................................................ 10

3 Implementation ............................................................................................................................ 12

4 Conclusion .................................................................................................................................... 13
   4.1 Conclusion ................................................................................................................................ 13
      a Model ....................................................................................................................................... 13
      b Commands/Actions ................................................................................................................ 13
   4.2 Future work ............................................................................................................................. 13
      a Server ...................................................................................................................................... 13
      b GUI ......................................................................................................................................... 13
      c Graph ...................................................................................................................................... 14
Abstract

TrucStudio is a software framework used to support instructors to divide the content of their lectures into smaller pieces of knowledge. Those pieces are called clusters, trucs and notions.

Earlier work targeting an import facility for OWL (Web Ontology Languages) files\(^1\) into TrucStudio, created the need for a slightly different structuring of the model. The first part of this thesis concerns the redesign of the model. The second part deals with the reimplementation of how actions from the GUI are called.

1 Intro

Today the goal of any university in Europe is to adapt to the Bologna process. The Bologna process was introduced to make the education standard more comparable throughout Europe. This offers many more chances for students to change universities in the context of their studies.

For universities to be more comparable, the courses have to follow a stricter succession of topics to avoid overlapping. A knowledge platform is needed to coordinate this process.

The software project TrucStudio intends to aid in the creation of course curricula and to significantly reduce the management overhead associated with maintaining them. In its current state, it mainly focuses on the definition of course topics (TRUCs – Testable Reusable Units of Cognition) and relations between them to ensure that the course structure is consistent and as complete as possible.

This project rebuilds the model structure of TrucStudio.

1.1 Problem description

Because of an earlier thesis that introduced the possibility to import OWL files into TrucStudio, the underlying model, a strict tree structure, was no longer adequate. The highest levels of knowledge (the clusters) needed to be ordered by a directed acyclic graph (DAG) instead of a tree. Therefore, the model had to be adapted to the new requirement.

In addition to the model adoption, the actions used to transform the model were very difficult to debug and not always correct. Our goal was to simplify the calls by taking out one level of indirection.

1.2 New implementation

The model now conforms to the requirement, i.e. the clusters exhibit a DAG structure. The new system differs from the old one in the way that all entities now inherit from the TS_ENTITY class while the old system had three different super classes (TS_CONTAINED_ENTITY, TS_ENTITY, TS_CONTAINER_ENTITY\(G \rightarrow TS\_ENTITY\)). A direct consequence of this is that the implementation changed to more generic features which are able to process any entity. A lot of code duplication disappeared because of that.

The TS\_ACTION classes are now completely removed from the system. Instead, there are two types of COMMAND classes, namely HISTORY\_COMMAND and SINGLE\_COMMAND. The history is shared between the HISTORY\_COMMAND classes and the TS\_MAIN\_CONTROLLER class.

\(^1\) OWL: http://www.w3.org/TR/owl-features/ | http://en.wikipedia.org/wiki/Web_Ontology_Language
1.3 Difficulties

In the beginning, there was a lot of confusion concerning what the new model should look like. The old model was not intuitive and we had to learn how the application makes use of the different parts. There was a model-view-controller-like implementation. Unfortunately, there were some major differences to the described pattern. The most significant one was that the model knew about the main controller and most of the other classes accessed the main controller via the truc manager (one of the central model classes).

Another difficulty was the redesign of the actions. As it was not easy to reconstruct what happened when an action was called, we had to find out how to use the different functionality. There were a lot of action sequences that behaved seemingly randomly.

1.4 Outline

The background chapter describes basic differences between the old and the new system. It concentrates on the main differences, in particular the new model and the different implementation of the history. The chapter about the implementation consists of a print-out of the developer guide. The last chapter is a conclusion of this thesis in which the problem description is compared with the outcome. The second part of this chapter gives an outlook in where TrucStudio can be enhanced.

2 Background – Differences to the old system

2.1 Model

There are some major differences between the old and the new model.

a Inheritance

First of all, in the old model, the TS_ENTITY class inherited from the TS_CONTAINED_ENTITY class. Model classes like TS_CLUSTER or TS_TRUC inherited from the TS_CONTAINER_ENTITY[G \rightarrow TS_ENTITY]. The consequence of this was wrongly applied assignment attempts and that parents knew about their children but not vice versa. In the new model, all entities inherit from TS_ENTITY.

Instead of being a container class now each entity has lists about its parents and its children. Furthermore the new implementation allows clusters to have nested clusters and trucs as children. See documentation in chapter 3 Implementation.

b Model-View-Controller

The old model offered the possibility to add and delete entities in the model by calling a function like add_truc_without_action. This was needed to prevent the history to know about the addition of trucs. The lists about contained entities in the truc manager had to be maintained manually by writing directly to the lists of the truc manager. The functions were implemented in the main controller. Usually, the truc manager was called to get the main controller. This means that the model had to know about its controllers.

Now the TS_TRUC_MANAGER class does not know about its controllers. This change is an adaptation to a clean model view controller. The model itself can now be controlled by any controller without knowing about the TS_TRUC_MANAGER. Furthermore, even if all entities can be accessed through the truc-manager, the manipulation of the model is done through the entities themselves.
The advantage of the current implementation is that an entity can be deleted and all of its children will also disappear. Undoing a deletion will also add all children because they are not recursively deleted from system but only the deleted entity is removed from the system, making the children unreachable. Therefore, the entities in the truc manager do not have to be maintained manually any more.

The difference is shown by the following example. First a truc is added to the model in the old way and afterwards the same is done the new way. Supposing the user provided all needed data in a dialog and pressed ok, then on_do_action gets called (in both scenarios).

The red coloured text is the main calling order which is interesting for us.

c Old implementation

**TS_D_TRUC_CREATE**

on_do_action is

```lisp
local c:TS_CLUSTER
do
  c := box.selected_item.data
  if c /= void then
    truc_manager.main_controller.create_truc(txt_name.text, c)
    -- here we see that the main controller is called via the
    -- truc manager
  end
end
```

**TS_MAIN_CONTROLLER**

create_truc(a_name:STRING; c:TS_CLUSTER) is

```lisp
require (create {TS_DBC_REQ}).is_not_void([a_name, c])
local t: TS_TRUC
do
  t := truc_manager.cache.create_truc (a_name)
  -- the truc is now created and has to be added to the model
  add_truc(t, c)
  if preferences_manager.create_new_notion_auto then
    create_notion ("New notion", t)
  end
end
```

add_truc(t:TS_TRUC; c:TS_CLUSTER) is

```lisp
require (create {TS_DBC_REQ}).is_not_void([t, c])
local has_cluster: truc_manager.clusters.has (c)
local action:TS_ACTION
do
  -- the action gets created through the factory, how this differs to the new
  -- model can be seen in the next section where we describe how the
  -- new history is implemented
```
action := history_manager.factory.truc_add_action (t,c)  
action.observer_list_do_actions.extend (agent truc_add_event (t))  
action.observer_list_undo_actions.extend(agent truc_rem_event (t))  
action.observer_list_undo_actions.extend(agent cluster_edit_event_without_select

(c))
- - instead of showing the next feature call we will directly jump into where
- - the action gets called
history_manager.add_and_execute_action(action)

ensure  
added: truc_manager.trucs.has (t)
has_cluster: t.cluster = c  
end

TS_A_TRUC_ADD

execute_impl is

require
- add truc
else
not_in_data_structure: not truc_manager.trucs.has (truc)
do
- - all those features are inherited from TS_A_TRUC or the
- - corresponding factory class
truc_add (cluster)
end

truc_add(c:TS_CLUSTER) is

-- Add `truc`
require
( create (TS_DBC_REQ)).is_not_void([truc,c])
not_in_data_structure: not truc_manager.trucs.has (truc)
do
- - the cluster gets modified and the truc added
  c.set_modified (True)
c.add_truc (truc)
- - in addition we add the truc to the truc manager by directly (!)
- - manipulating a list which is owned by the truc manager
  truc_manager.trucs.put_last (truc)
ensure  
added: truc_manager.trucs.has(truc)
end

Now the truc is in the system.

d New implementation

TS_D_TRUC_CREATE

on_do_action is

local
  c:TS_CLUSTER  
a_truc: TS_TRUC  
do
  c ?= box.selected_item.data
if c /= void then
- - this one was in the main controller before
- - create a new instance of a truc
  a_truc := truc_manager.create_truc (txt_name.text)
- - all different entities are added with the same feature of the
- - main controller. This is because now all entities inherit from
- - TS_ENTITY
main_controller.add_entity (c, a_truc)
end

TS_MAIN_CONTROLLER

add_entity(a_parent: TS_ENTITY; new_entity: TS_ENTITY) is
  -- add an entity to the model with a command
  require
  new_entity_not_void: new_entity /= Void
  local
  a_cluster: TS_CLUSTER
  a_course: TS_COURSE
  a_redo_procedure, an_undo_procedure: PROCEDURE[ANY, TUPLE]
do
  - - if the parent is not void we add the entity directly to the parent
  - - this has to be the case if the entity is a truc. Only clusters and courses
  - - can be added to the truc manager which is the root
  if a_parent /= Void then
    a_redo_procedure := agent new_entity.add_parent_element (a_parent)
    an_undo_procedure := agent new_entity.del_parent_element (a_parent)
  else
    if new_entity.entity_type.is_equal (C_CLUSTER) then
      a_cluster := new_entity
      a_redo_procedure := agent truc_manager.add_cluster (a_cluster)
      an_undo_procedure := agent truc_manager.del_cluster (a_cluster)
    elseif new_entity.entity_type.is_equal (C_COURSE) then
      a_course := new_entity
      a_redo_procedure := agent truc_manager.add_course (a_course)
      an_undo_procedure := agent truc_manager.del_course (a_course)
  end
end
main_window_controller.graph_view_controller.graph_controller.graph.
set_has been_loaded (False)
create a_command.make_with_undo (agent entity_event_helper
(a_redo_procedure, new_entity, entity_add_event), agent entity_event_helper (an_undo_procedure,
new_entity, entity_delete_event), false)
a_command.execute
end

TS_TRUC

- - the function which gets called by the command in this case

add_parent_element (parent: TS_ENTITY) is
  -- adds the given parent container element to the parents list
  -- if it is not already in there
  require
  (create {TS_DBC_REQ}).is_not_void([parent])
  no_cycle: not has_child(parent)
do
  if not parents.has (parent) then
    parents.put_last (parent)
    parent.add_child_element (current)
    relatives_changed_actions.call ([])
  update_modified
end
ensure
parent_impl_child: current.read_parents.has (parent) implies 
  parent.read_children.has (current)
child_impl_parent: parent.read_children.has (current) implies 
  current.read_parents.has (parent)
end

Now the truc is in the system. The truc manager will get its list of currently used trucs through the 
cluster it knows. The history command will call the add function just as the add_and_execute 
function did in the old model.

The changes are obvious, as the new system only changes the affected entities and updates itself. In 
contrast, the old model had to maintain a list of the entities in the system manually.

2.2 Action

The new implementation takes out one level of indirection. The old implementation used so-called 
ACTION classes. Those classes implemented an execute and an undo feature for every action.

The new implementation uses so-called COMMANDS. The main difference is that the built-in 
agents are used now. Agents encapsulate procedures as objects. The main advantages of commands 
are that they only encapsulate the specific procedures. In the old system an editing of an entity 
deleted the entity and added it again with the new properties. The following example demonstrates 
the differences of usage. For simplicity the part cut out of the example from above is taken and a 
look into the history part is given.

a Old implementation

TS_MAIN_CONTROLLER

add_truc(t:TS_TRUC; c:TS_CLUSTER) is
  -- add truc action
  require
    (create (TS_DBC_REQ)).is_not_void([t, c])
  has_cluster: truc_manager.clusters.has (c)
  local
    action:TS_ACTION
  do
    - - here the action gets created, we will go through the creation and 
    - - afterwards just follow the last call in this feature
    action := history_manager.factory.truc_add_action (t, c)
    action.observer_list_do_actions.extend (agent truc_add_event (t))
    action.observer_list_undo_actions.extend(agent truc_rem_event (t))
    action.observer_list_undo_actions.extend(agent cluster_edit_event_without_select

(c))

    - - This time we will go into how the action is called
    history_manager.add_and_execute_action(action)
  ensure
    added: truc_manager.trucs.has (t)
    has_cluster: t.cluster = c
  end

TS_F_ACTION_FACTORY

  - - the instance of the action is created through the factory
  - - The TS_A_TRUC_ADD class implements the features which will be called later
  - - Every procedure we want to call has to implement the TS_ACTION interface

truc_add_action(t:TS_TRUC; c:TS_CLUSTER):TS_A_TRUC_ADD is
require
  (create {TS_DBC_REQ}).is_not_void([t, c])
do
  create Result.make (t, c, trunc_manager)
ensure
  Result_exists: Result /= Void
dend

TS_C_ACTION

add_and_execute_action (a: TS_ACTION) is
  -- execute new action
require
  (create {TS_DBC_REQ}).is_not_void([a])
do
  -- remove redo actions if any
  if not cursor.is_last and not history.is_empty then
    history.prune_right_cursor (history.count - cursor.index, cursor)
  end
  - - registering this action in the history
  history.put_last (a)
cursor.go_after
cursor.back
  a.execute
view.update
from
  history.start
until
  history.off
loop
  --io.put_string (history.item_for_iteration.name + "%N")
  history.forth
end

TS_ACTION

frozen execute is
  -- perform do/redos
  do
    if not actions_initialized then
      init_actions_before_do
      init_actions_after_do
      actions_initialized := true
    end
    - - the list with all actions which should be executed before this action get
    - - called. These are usually cleaning functions of the structure. We add a
    - - trunc so this will probably be empty
    execute_actions_before_do
    execute_impl
    execute_actions_after_do
    notify_do_observer
end

TS_A_TRUC_ADD
execute_impl is
  -- add truc
  require else
    not_in_data_structure: not truc_manager.trucs.has (truc)
  do
    truc_add (cluster)
  end

b New implementation
The following example tries to start at the same point as the old implementation. Only the part of the add_entity feature, which is important for the example, is shown. The rest can be looked up in the section on the new implementation.

TS_MAIN_CONTROLLER

add_entity(a_parent: TS_ENTITY; new_entity: TS_ENTITY) is
  -- add an entity to the model with a command
  require
    new_entity_not_void: new_entity /= Void
  local
    a_cluster: TS_CLUSTER
    a_course: TS_COURSE
    a_redo_procedure, an_undo_procedure: PROCEDURE[ANY, TUPLE]
  do
    if a_parent /= Void then
      -- this are the functions which gets called by execute and undo
      a_redo_procedure := agent new_entity.add_parent_element (a_parent)
      an_undo_procedure := agent new_entity.del_parent_element (a_parent)
    else
      ...
    end
  -- We first create the command. This is like getting the action from the
  -- action factory, the entity helper is used to notify all subscribers
  -- when an entity is added
    create a_command.make_with_undo (agent entity_event_helper
      (a_redo_procedure, new_entity, entity_add_event), agent entity_event_helper (an_undo_procedure,
      new_entity, entity_delete_event), false)
    a_command.execute
  end

HISTORY_COMMAND

make_with_undo (an_action: like action;
  an_undo_action: like undo_action; a_value: like is_once_command) is
  -- Set `action' to `an_action'.
  -- Set `undo_action' to `an_undo_action'.
  -- Set `is_once_command' to `a_value'.
  require
    an_action_not_void: an_action /= Void
    an_undo_action_not_void: an_undo_action /= Void
  do
    action := an_action
    undo_action := an_undo_action
    is_once_command := a_value
  ensure
    action_set: action = an_action
    undo_action_set: undo_action = an_undo_action
    is_once_command_set: is_once_command = a_value
end

execute is
    -- Call `action`
    -- with an empty tuple as arguments.
    do
        if action.valid_operands ([[]]) then
            if is_once_command and then history.has (Current) then
                history.extend (clone (Current), [])
            else
                -- registering the command in the history, the history is
                -- shared between the main controller and the commands
                -- therefore, the main controller can now undo this
                -- command without having to care about the registration
                history.extend (Current, [])
            end
        end
        -- call the command, which is in this example
        -- new_entity.add_parent_element (a_parent)
        -- where `new_entity` is the truc and `a_parent` is the cluster it
        -- belongs to
        action.call ([[]])
    end

This implementation differs a lot from the old one. The main difference is that no new classes need
to be implemented for additional actions which should be performed. The command only needs the
procedure and executes it. The granularity of modification is a lot finer than before. If several
commands should be encapsulated, single commands implement commands that are not registered
in the history. The composite commands encapsulate the single commands to one big command.
3 Implementation

In the following section the developer guide and the user guide concerning this thesis is given.
Developer guide model

Abstract

This is the developer guide concerning the class hierarchy of Trucstudio's basic entities (clusters, trucs aso.). The implementation of an OWL file import required to change the structure from a tree to a DAG (Directed Acyclic Graph). This page describes both, the data model and the controller. The implementation is based on the MVC (Model View Controller) pattern.

Design

Class hierarchy

The following image shows the new class design of the entity classes. Every entity inherits from the TS_ENTITY class, where the basic functionality is implemented. For example adding/deleting children is implemented in TS_ENTITY which helps to keep the parent-child-relations consistent.

Data Model

The data model is based on a DAG. The DAG consists of entities. These entities are clusters, trucs, notions, courses and lectures. The top level container, where all entities can be accessed, is the truc manager. This is the root node. The definition of the entities can be found in Course management with TrucStudio - M. Pedroni, M. Oriol, B. Meyer, E. Albonico, L. Angerer
The entities themselves are doubly-linked; each knows its parents as well as its children. The true manager does not have an explicit list of all entities as they can be found by walking through the structure. The following picture illustrates the parent relations.

![Diagram of entity parent relations]

The image below shows how the structure is implemented in TrucStudio. The courses and lectures are also included as they are also entities. Notions do not know about the lectures they belong to.
Controller

The implementation follows the well-known MVC pattern. The main controller is responsible for adding/removing entities and the history.

The controller uses the command pattern to implement a history allowing undo and redo of actions called by the GUI. More information can be found at: Commands and history

Implementation

Data Model

Truc Manager

The truc manager is the entry point of the model. It contains the clusters and it manages the ts_cl_root, which contains the courses. Adding clusters to the model is done directly in the truc manager. All other operations can be applied to the model itself (Adding or removing entities should only be done in the parent and children entities, respectively.).

The truc manager also offers features to get new entities. Those entities are created by the cache the truc manager is bound to. Lists to access all the entities in the system exist. Those lists are updated whenever an entity is added/removed.

TS_CL_ROOT

Entities

All elements in the DAG inherit from TS_ENTITY. TS_ENTITY offers features to add and delete parent and children entities. These features ensure the consistency of the DAG (no cycles).

It defines the basic features like setting a name or updating the is_modified status as well as a mechanism to subscribe to if an element has changed or if a relation has changed. The view will therefore directly subscribe to the entity it belongs to rather than to the main controller.

There are five kinds of entities: clusters, trucs, notions (different granularity of how to divide topics to teach), courses and lectures (time and topic container).

Children inside a cluster are either clusters (DAG) or trucs (tree structure).
All entities have a so called original_entity. This entity is a copy of the main parts of the current entity. It is used to update the is_modified flag. If update_modified is called the diff function will compare the original entity with the current entity. If they differ, the flag is set. With help of the original_entity, we know that we have changed the entity after downloading it from the server. The original_entity is set each time we download an entity.

**Cluster**

Clusters are different to the other entities as they are represented in a DAG. A cluster can either be a top cluster, a cluster nested in another or both. For the model, especially when storing or loading it, it has to know if the cluster is a top cluster or not. Therefore, a cluster has a feature is_top_level. When committed to the repository, this feature is not submitted as it does not have to be the case that we want a top cluster in the committing project to be a top cluster in another project. Nested clusters are new to the model. They are implemented as children of other clusters. The nested clusters build the newly introduced DAG structure.

**Controller**

The controller is divided into several classes, see developer guide controller. The main controller in the system is responsible for the model changes. It implements all possible event handlers for entities. There are different command implementations to change entities. You can, for example, add an entity by calling

```python
add_entity(a_parent_entity; a_child_entity)
```

or move an entity by calling

```python
move_entity(the_item_to_move; the_target)
```
Developer guide model

Abstract

We implement the Model-View-Controller pattern. Most of the controllers are located at $Strucestudio/src/tstudio/controller$.

Design

As there is a lot to control, there is a basic controller ancestor class, TS_CONTROLLER. Furthermore, there is a main controller, TS_MAIN_CONTROLLER. The main controller is directly associated with the truc manager. This means that most of the changes in the model are initiated by the main controller. For every GUI (e.g. property view, server view) there is another controller to simplify the structure for the developer.

Implementation

Main controller

The main controller implements the features to modify the model. It is directly connected to the truc manager. It also manages other controllers like the main window controller, the property controller, the server controller and the graph controller.

Model

The main controller offers a simple interface to manipulate the data in the model. To add an entity, just call add_entity(a_parent: TS_ENTITY; a_new_entity: TS_ENTITY). This feature will add the new entity to the model. If the parent is void, the new entity has to be either a cluster or a course and is then added at the top level to the truc manager. All other entities are added if the parent entity is correct. In addition, all procedures subscribed for an add event will be notified. Deleting entities works the same way but it also deletes relations between the entities. The feature delete_entity_from_system is useful to delete every occurrence of a cluster out of every instance that refers to it and the truc manager if it is a top level cluster. As the model is a DAG, there could be multiple parent-child relations for clusters. The feature edit_entity(do_action: PROCEDURE[ANY, TUPLE]; undo_action: PROCEDURE[ANY, TUPLE]) allows to edit an entity. Editing entities happens with the finest granularity. This means, instead of deleting and adding an entity to the model when only having edited a text field (e.g. name), the new implementation needs only the specific procedure. Therefore, finding an undo function is very easy. Every procedure is saved in the history. The main controller inherits from SHARED_HISTORY, which enables the access to the feature history.

The main controller is usually used to manipulate the data. Nevertheless, it is also possible to do this without using the main controller. By using HISTORY_COMMANDS, the main controller will not be notified. Nevertheless, the procedures are saved in the history and all subscribed procedures
inside the entities are notified. Therefore, by undoing a change through the main controller, the commands used outside the main controller will be undone as well.

Controllers

The main controller manages also the different controllers in the system. The interface of the main controller gives access to the different controllers which are:

- the `truc_notion_tree_controller` (manages the clusters, trucs and notions)
- the `cln_tree_controller` (manages the courses and lectures)
- the `current_p_controller` (refers to the current properties controller)
- the `rpc_provider` (supports the repository)
- the `matimport_manager` (allows reading in material and creating lectures out of it)
- the `main_window_controller` (manages the main window and controlling the graph and the server view)
Developer guide I/O

Abstract

This page describes the general input and output handling. There are some fundamentals that are described in Design. Furthermore, it describes how the input is implemented in Truestudio and where the source is located.

Design

Truestudio uses XML to save the output. This is helpful as XML has a lot of benefits: First of all, XML can be read by humans; this is useful to debug the files manually. Secondly, the more important benefit is that XML can easily be transformed. The gobo library offers built-in parsers for XML files and transformers. This offers the possibility to output different formats. The hypergraph, for example, has another XML format to describe its structure. The XML-file used to describe the *.txm (truestudio projects) is very flat. All entities are written on the same level and interlinked with references. This is the case because there could be a lot of relations between the entities. The file could explode in size if we saved all the entities in a tree structure.

Implementation

Storing

Truestudio has project files that are stored in *.txm format. $truestudio/src/tstudio/root/cluster/data/TSData.dtd describes the structure of the project files.

The source output generation is loaded in the subcluster $truestudio/src/tstudio/io. To save files, we call the save_project feature in TS_C_MAIN_WINDOW, which is the controller of the main window. This feature uses the library class XML_DOCUMENT to create the file in the filesystem. It saves xml-nodes which are then appended to the document.

Those nodes have to be provided by us. We implement TS_XML_* classes for all elements we want to save. All those classes inherit from the TS_XML_ELEMENT class which is the main class to save data into xml_elements. The top-level node is the project node. All entities are subnodes of this one.

Each entity specific class is saving its attributes as described in the TSData.dtd to an xml-element.

Loading

The mechanism to load files is more or less the reverse of saving them. The feature open_txm_file in the TS_C_MAIN_WINDOW loads a file into a XML_DOCUMENT. Afterwards, the TS_XML_PROJECT_ELEMENT is used to load all the nodes.

In this class the nodes generated by loading the XML_DOCUMENT will be processed. At the moment this is not optimized. See future work.
Output generation

To generate any output, such as webpages, Trucstudio saves the current project to a temporary file, which is found at $Trucstudio/src/tstudio/root_cluster/xsl/in.txm. This file can then be transformed with the help of the xsl files, located at $Trucstudio/src/tstudio/root_cluster/xsl/templates.

GUI

The output generation is located at '$Trucstudio/src/tstudio/output_generation'. The main class for the dialog which is shown in the GUI is the TS_OUT_WIZARD_CONTROLLER. In this class the GUI elements are initialized.

Webpages

The xsl files to transform the in.txm file to webpages can be found in the directory $Trucstudio/src/tstudio/root_cluster/xsl/templates.

Hypergraph

The xsl files to transform the in.txm file into the format for the hypertree can be found in the directory $Trucstudio/src/tstudio/root_cluster/xsl/templates/hypertree.
Developer guide command

The pattern

The command pattern allows to undo/redo commands. It is based on the well known book of the gang of four Design Patterns: Elements of Reusable Object-Oriented Software. The exact explanation can be found in the book or on the wikipedia page command pattern.

Implementation

The pattern is described in a generic way for object-oriented programming languages. In Eiffel, the built-in agent mechanism is available. An agent encapsulates a procedure into an object. This mechanism simplifies the originally described pattern implementation. The two most important classes are the command and the history class. The first one represents the actual command. The arguments to be provided at creation time are the do/redo and the undo command (agent). Because we use a special implementation of the pattern, the history is shared. By calling the command to execute, it will subscribe itself in the history. The history class maintains the history. There are the main features undo/redo and the status access can_redo/can_undo. The latter two indicate if the last command can be re- or undone. This is necessary as there could be once calls filled in.

Our implementation introduces a feature to clear the history. This is useful when a new document is loaded.

The implementation is taken from the library implemented by Karine Arnout in her PhD Thesis: Karine Arnout: From Patterns to Components; PhD Thesis, 2004

Commands

The commands are implemented according to the pattern. The creation feature of a command takes three arguments. The first is a procedure which is called to execute the call and which is also called when any call should be redone. The second argument is the procedure to undo any change and the third is a boolean which describes whether it is a once call or not.

As an example for editing the name of an entity in the model we could call:

create a_command.make_with_undo (agent an_entity.set_name("new name"), agent an_entity.set_name (an_entity.name), false)
a_command.execute

We see that the command is only completed by calling a_command.execute. There are two different kinds of commands implemented. The following sections describe the differences. There is a third class to combine several commands.

HISTORY_COMMAND

The history command is used whenever a command should register itself in the history. A history command does not need the main controller to register the command in the history. The history is shared by all history commands and the main controller.
Most of the commands used by the application are history commands. This is a consequence of the requirement that most of the actions taken in the GUI should be undoable.

**SINGLE_COMMAND**

A single command is an implementation of the command that does not subscribe itself in the history. This is the only difference to the history command inside the command class. The main advantage is its usage in the composite command.

**COMPOSITE_COMMAND**

The composite command is a container for several commands. It inherits from the composite class. This container allows to store several commands. They are supposed to have a do and an undo feature. If we call execute in the composite_command, it will execute all commands in the list. To undo all, it will execute all the undo procedures of the commands in the list.

A composite_command is very useful to clean all dependencies and connect them again when undoing the command to mention only one example.

It is used as follows:

```python
create composite_command.
create a_single_command.make_with_undo (agent redo_procedure, agent undo_procedure, false)
create another_single_command.make_with_undo (agent another_redo_procedure, agent another_undo_procedure, false)
composite_command.add (a_single_command)
composite_command.add (another_single_command)
create history_command.make_with_undo (agent composite_command.redo, agent composite_command.undo, false)
history_command.execute
```

This is how we can call several commands and undoing will undo all of the commands rather than only the last one.

**HISTORY**

As mentioned before, the history is shared between history_commands and the application. The history stores a two way list of commands. It has a cursor that points to the current place in the history. When a new command is added, it will cut off the tail of the two way list and append the new command.

The history keeps track of the commands and whether a command can be undone or redone.

**Differences old to new**

The new implementation has major differences to the old one. As seen above, the new implementation uses commands instead of actions. To show the differences, flowcharts for both are given.

The green arrows follow through the calling sequence. The red arrows belong to procedures where the path is continued in the procedure where they startet.

The old implementation:
The new implementation:
4 Conclusion

4.1 Conclusion

a Model
The model now has the required functionality. Clusters can be nested and are ordered in a DAG. In addition, the true manager maintains the lists of the entities. All changes are made directly through the respective entities. There is no need to keep track of any modifications, as the model will now update the status after every change. If an initial change is undone, the model will recognize it and set the is_modified flag to false.

To avoid code duplication, a lot of the features were refactored one level up into the TS_ENTITY class.

b Commands/Actions
The newly entered commands differ drastically from the old implementation. At first sight it seems that the adaptations are a lot more complicated. This is a consequence of the more generic way to call the functions.

The advantages of the new commands are that a developer does not have to implement a class and factory function for every action he/she wants to have. There is no longer any need to register a command in the history as a command does this automatically. All the action sequences to edit actions that used to be in the main controller are now in the entities. If an entity is added or deleted, the main controller notifies all subscribers.

4.2 Future work
There is a lot left to do in TrueStudio. The following section only discusses the parts that concern this project.

a Server
The server needs a new database scheme. The server uses a scheme in which clusters are not nested. When an entity is shared on a server it cannot be shared on another server. As there is no way to delete entities from the server at the moment, an entity cannot be moved from one server to another. There has to be an implementation for the server that enables moving of entities as well as telling the server that an entity is deleted.

Another issue is that loading all entities is slow. It seems that every entity is called individually. This can probably be accelerated by making one look up for all entities.

b GUI
There is a problem with the trees in the GUI. The cursor of a visual tree cannot be accessed. This is because it is not allowed to have two items with the same content.

The implemented drag and drop function to move clusters to a lecture only adds the notions of the
selected cluster. Notions owned by a nested cluster will not be added to the lecture. The exact semantics of such a move has to be determined and implemented.

\textit{c Graph}

The graph represents the model. All operations transforming the model are displayed in the graph. Therefore, the graph is correct. However, it is very slow. Therefore, all transformation call a refresh of the graph. This should be changed in future versions of \textit{TrucStudio}.