SessionModel

Introduction

The `SessionModel` is the main class to hold the hierarchy of `SessionItem` objects. It contains a single root `SessionItem` as an entry point to other top-level items through the model API. In pseudo-code it it can be expressed as:

```
class SessionModel
{
public:
  SessionModel() : m_root_item(std::make_unique<SessionItem>()) {}

  SessionItem* GetRootItem() { return m_root_item.get();}

  tempate<typename T>
  void InsertItem(SessionItem* parent, const TagIndex& tag_index);

private:
  std::unique_ptr<SessionItem> m_root_item;
};
```

Inserting items

To insert an item in a model, one has to use the `InsertItem` method and provide a pointer to a parent item and `TagIndex` specifying the position in the parent's containers. When no arguments are provided, appending to an invisible root item is assumed.

```
SessionModel model;

// appends compound to the root item
auto parent = model.InsertItem<CompoundItem>();

// inserting child into parent's tag
auto child0 = model.InsertItem<PropertyItem>(parent, {"tag", 0});
```

In the example above it is assumed that the `parent` item is properly configured, and can accept items of a given type under the given tag.

Accessing items

Use `GetRootItem` method to access root item:

```
auto root = model.GetRootItem();
```

It is possible to access top-level items of a certain type using the `GetTopItems` method:

```
auto top_items = model.GetTopItems<CompoundItem>();
```

With the help of the `iterate` function from the `Utils::` namespace it is possible to visit all items in a model. In the example below the model is visited iteratively and all existing items are collected:

```
std::vector<const SessionItem*> visited_items;
auto visitor = [&](const SessionItem* item) { visited_items.push_back(item); };
Utils::iterate(model.GetRootItem(), visitor);
```

Registering custom items to use with the model

By default, `SessionModel` knows about the existence of only a few items:

- `SessionItem` is the basic construction element of the model.
- `PropertyItem` is the one that carries the data only, and doesn't have children
- `CompoundItem` provides an additional API for convenient handling of item's properties
- `VectorItem` and item to carry (x, y, z) data.

The list can be extended by registering additional types to use with the model. This will enable item insertion, serialization, undo/redo, and the possibility for item copying.

In the snippet below we define two custom items: `SegmentItem` and `PulseScheduleItem`. The `PulseScheduleItem` is additionally configured to accept an unlimited amount of `SegmentItems` as children under the tag `Segments`.

```
class SegmentItem
{
  static const std::string Type = "SegmentItem";
  SegmentItem : public SessionItem(Type) {}
};

class PulseScheduleItem
{
```

```
  static const std::string Type = "PulseScheduleItem";
  PulseScheduleItem : public SessionItem(Type)
  {
    // Define tag "Segments" for unlimited amount of SegmentItem children .
    // The tag is declared as default to omit tag name in operations with
children.
    RegisterTag(TagInfo::CreateUniversalTag("Segments", {SegmentItem::Type}),
/*as_default*/true);
  }
};
```

Then, after the registration of two new items in a model using `RegisterItem` method, items become available for further manipulation.

```
SessionModel model;
model.RegisterItem<SegmentItem>();
model.RegisterItem<PulseScheduleItem>();

// create top level PulseScheduleItem
auto pulse_schedule = model.InsertItem<PulseScheduleItem>();

// add segments to it
auto segment0 = model.InsertItem<SegmentItem>(pulse_schedule);
auto segment1 = model.InsertItem<SegmentItem>(pulse_schedule);
```

It also became possible to use string type to create an object of the `SegmentItem` type:

```
auto segment = model.InsertNewItem(SegmentItem::Type, pulse_schedule);
assert(dynamic_cast<SegmentItem*>(segment));
```

Memory pool

Every `SessionItem` carries a unique identifier that is assigned to it at the moment of construction. This identifier can be accessed using the `GetIdentifier` method:

```
SessionItem item;
std::cout << item.GetIdentifier() << std::endl;
>>> "4c281780-1bf6-4d98-8de2-b9775085c755"
```

When `SessionItem` is inserted in the model, this identifier gets registered in the model's `ItemPool` together with the item's address. When the item is removed from the model, the record is removed too. During serialization of the model, and following reconstruction of the model from serialized content, items' identifiers are preserved.

Knowing the identifier, one can find the `SessionItem` address. It allows using identifiers for cross-linking between model parts, also for the case of different models.

```
SessionModel model;

auto item = model.InsertItem<SessionItem>();
auto identifier = item->GetIdentifier();

auto found_item = model.FindItem(identifier);
assert(item == found_item);
```

Multiple models in the application

Often it is convenient to have more than one model in an application. Each model should have a unique name and might have its own set of constituent items. In the snippet below we define a PulseScheduleModel intended for storage of a pulse schedule with some convenience API.

```
class PusleScheduleModel : public SessionModel
{
public:
  PulseScheduleModel() : SessionModel("PulseScheduleModel")
  {
    RegisterItem<PulseScheduleItem>();
    RegisterItem<SegmentItem>();
    RegisterItem<TransitionItem>();
  }

  PulseSceduleItem* CreatePulseSchedule()
  {
    return InsertItem<PulseSceduleItem>();
  }
};
```

To provide the possibility for cross-linking between different models, all involved models should be initialized with the same memory pool. This will allow items from one model to access items in another model if identifiers are known.

In the snippet below we prepare two models to use shared `ItemPool` object.

```
class PulseScheduleModel : public SessionModel
{
public:
  PulseScheduleModel(std::shared_ptr<ItemPool> pool) :
SessionModel("PulseScheduleModel", pool) {}
};

class RTFConfigurationComponentModel : public SessionModel
```

```
{
public:
  RTFConfigurationComponentModel(std::shared_ptr<ItemPool> pool) :
SessionModel("RTFConfigurationComponentModel", pool) {}
};
```

Then, we create one common pool and two different model instances.

```
auto pool = make_shared<ItemPool>();

PulseScheduleModel pulse_schedule_model(pool);
auto pulse_schedule = pulse_schedule_model.InsertItem<PulseSchedule>;
auto identifier = pulse_schedule->GetIdentifier();

RTFConfigurationComponentModel component_model(pool);
assert(pulse_schedule == component_model.Find(identifier));
```

With that, every model can find items in another model, if identifiers are known.