Operational Applications UI foundation

Collection of C++/Qt common components for Operational Application User Interfaces.

Requirements

- C++17
- CMake 3.14
- Qt 5.12
- libxml2

Installation

```
cmake <source> && make -j4 && ctest
```

Quick start

```
// Create an application model holding all the data of running application.
SessionModel model;

// Populate model with content.
// - The model contains a single item representing a Gaussian distribution.
auto compound = model.InserItem<CompoundItem>;
compound->SetDisplayName("Gaussian");
compound->AddProperty("Mean", 0.0);
compound->AddProperty("StdDev", 1.0);

// Creates ViewModel which can be shown in Qt widgets.
ViewModel view_model(&model);

// Open the view model in standard Qt's tree view.
QTreeView view;
view.SetModel(&view_model);
view.show();

// After editing is complete, save the content in the XML file.
XmlDocument document({&model});
document.save("filename.xml");
```

The context

This model-view-viewmodel (MVVM) framework is intended for large Qt-based scientific applications written in C++. The definition of `large` is quite arbitrary and means something in a line "can be easilly above 50K of LOC".

The Model

The first part of the framework, the model, consists of a group of classes to build a tree-like structure, named `SessionModel`, to handle any data of the GUI session. The data in this model is originated, at least partly, from the persistent storage (database, XML files), and then leaves in the memory of the running application. The model is populated with elements of different complexities: from compound items representing complex domain objects (classes), to elementary items, representing editable properties.

This part of the framework is intentionally made independent of any graphics library. The idea behind is the following:

In large GUIs, the business logic gets quickly spoiled with presentation logic. Graphics library classes (like `QModelIndex` from Qt library) start to appear everywhere, even in places that have nothing to do with graphics. Removing graphics library from dependencies allows focusing more on common needs (i.e. objects construction, property editing, etc) of GUI applications rather than on presentation details. Thus, the intention here is to build an application model to handle the data and logic of GUI while being independent on any particular GUI library.

The ViewModel

The second part defines `ViewModel` and serves as a thin counterpart of `SessionModel` in the Qt world. `ViewModel` doesn't own the data but simply acts as a proxy to different parts of `SessionModel`. It is derived from `QAbstractItemModel` and intended to work together with Qt's trees and tables. The layout of `ViewModel` (i.e. parent/child relationships) doesn't follow the layout of the original `SessionModel`. It is generated on the fly by using strategy who-is-my-next-child provided by the user. In practice, it allows generating Qt tables and trees with arbitrary layouts, based on a common data source, without diving into the nightmare of `QAbstractProxyModel`. Particularly, the aforementioned machinery allows having something in the line of the ancient Qt property browser framework.

The View

The third part, the View, contains few convenience widgets for property editing. In the future this part can be extended with widget library for scientific plotting.

Further reading

- GUI architecture, Martin Fowler

List of features

Model part

- Application model to store arbitrary data of GUI session `(+)`.

- Serialization of application models to XML `(+)`.
- Depends only on C++17 and libxml `(+)`.
- Unique identifiers for all items and memory pool registration `(+)`.
- Multiple models with the possibility for persistent inter-model links `(+)`.
- Undo/redo based on command pattern `(+/-)`.

ViewModel part (depends on Qt5)

- View model to show parts of application model in Qt widgets `(+/-)`.
- Property editors `(+/-)`.
- Automatic generation of widgets from model content `(+/-)`.
- Scientific plotting `(+/-)`.
- Flexible layout of Qt's trees and tables `(+/-)`.

> `(+)` - feature is implemented, `(+/-)` - porting is required from the original `qt-mvvm`.

Size of the framework

- 4000 LOC of libraries
- 4000 LOC of tests (1500 `EXPECT` statements)

Main components

- SessionItem
- CompoundItem
- SessionModel
- Serialization
- to be continued ...

SessionItem

1. Introduction

`SessionItem` class is a base element to build a hierarchical structure representing all the data of the application running. `SessionItem` can contain an arbitrary amount of basic data types, and can be a parent for other `SessionItems`.

The tree of `SessionItem` objects can be built programmatically via `SessionItem` API, or be reconstructed from persistent content (XML and JSON files, for example).

While being an end leaf in some ramified hierarchy the `SessionItem` often plays a role of a single editable/displayable entity. For example, a `SessionItem` can be seen as an aggregate of information necessary to display/edit a single integer number `42` in the context of some view. Then, it will carry:

- An integer number with the value set to `42`.
- A collection of appearance flags, stating if the value is visible, is read-only, should be shown as disabled (grayed out), and so on.
- Other auxiliary information (tooltips to be shown, allowed limits to change, and similar).

2. The data of `SessionItem`

The data carried by `SessionItem` is always associated with the role - a unique integer number defining the context in which the data has to be used. They both came in pairs, and the item can have multiple `data/roles` defined.

```
// currently supported elementary data types
using variant_t = std::variant<std::monostate, bool, int, double, std::string,
std::vector<double>>;

// convenience type
using datarole_t = std::pair<variant_t, int>;

// collection of predefined roles
namespace DataRole
{
const int kIdentifier = 0;  //!< item unique identifier
const int kData = 1;        //!< main data role
const int kDisplay = 2;     //!< display name
```

```
  const int kAppearance = 3;  //!< appearance flag
}
```

In the snipped below, the data is set and then accessed for two roles, the display role holding a label and the data role, holding the value.

```
SessionItem item;
item.SetData(42, kData);
item.SetData("Width [nm]", kDisplay)

auto number = item.Data<int>(kData);
auto label = item.Data<std::string>(kDisplay);
```

2.1 Related files

- `variant.h` contains definitions of `variant_t` and `datarole_t` data types. Check it for all suported elementary data types.
- `mvvm_types.h` defines constants and enums. Check it to see current roles, or appearance flags.
- `sessionitemdata.h` contains the definition of `SessionItemData` class. It is a member of `SessionItem` and carries all the logic related to item's data. Most of methods of `SessionItemData` are replicated by `SessionItem`.
- `sessionitemdata.test.cpp` contains unit tests of `SessionItemData` and can be used as an API usage example.

3. Inheriting from `SessionItem`

The `SessionItem` class type name is stored in a string variable and can be accessed via the `GetType()` method:

```
SessionItem item;
std::cout << item.GetType() << std::endl;
>>> "SessionItem"
```

This name is used during item serialization/deserialization and during undo/redo operations to create objects of the correct type in item factories (explained in `sessionmodel.md`).

To inherit from `SessionItem` the new unique name has to be provided in the constructor of the derived class. It is convenient to make this name identical to the class name itself:

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

3.1 Related files

- `itemfactory.h` contains `ItemFactory` class definition. It is used in the context of `SessionModel` for user class registration.

4. Children of SessionItem

`SessionItem` can have an arbitrary amount of children stored in named containers. In pseudo code, it can be expressed

```
class SessionItem
{
  using named_container_t = std::pair<std::string, std::vector<SessionItem*>>;
  std::vector<named_container_t> m_tagged_items;
}
```

Named containers are a convenient way to have items tied to a certain context. The name of the container, `tag`, and the position in it, `index`, can be used to access and manipulate items through their parent `SessionItem`. Before adding any child to `SessionItem`, the container has to be created and its properties defined.

4.1 The `TagInfo` class

The `TagInfo` specifies information about children that can be added to a `SessionItem`. A `TagInfo` has a name, min, max allowed number of children, and vector of all types that children can have.

In the snippet below we register a tag with the name `ITEMS` intended for storing unlimited amount of other `SessionItems`.

```
SessionItem item;
item.RegisterTag(TagInfo("ITEMS", 0, -1));
```

An equivalent way of doing the same is to use convenience factory methods of the `TagInfo` class:

```
SessionItem item;
item.RegisterTag(TagInfo::CreateUniversalTag("ITEMS"));
```

Internally, it leads to the creation of a corresponding named container ready for items to be inserted. In another example, we define a tag with the name `Position` intended for storing the only item of type `VectorItem`.

```
item.RegisterTag(TagInfo("Position", 1, 1, {"VectorItem"});

// or
// item.RegisterTag(TagInfo::CreatePropertyTag("Position", "VectorItem"));
```

4.2 The `TagIndex` class

The `TagIndex` class is a simple aggregate carrying a string with container name, and an index indicating the position in the container.

```
struct TagIndex
{
  std::string tag = {};
  int index = -1;
}
```

The `TagIndex` class uniquely defines the position of a child and it is used in the `SessionItem` interface to access and manipulate items in containers.

4.3 Adding children

There are multiple ways to add children to a parent. In snipped below we register a tag with the name "`ITEMS`" intended for storing an unlimited amount of items of any type. In the next step, we insert a child into the corresponding container and modify its display name. Later, we access the child using the known `TagIndex` to print the child's display name.

```
const std::string tag("ITEMS");
SessionItem item;
item.RegisterTag(TagInfo::CreateUniversalTag(tag));

auto child0 = item.InsertItem({tag, 0});
child0->SetDisplayName("Child");

std::cout << item.GetItem(tag)->GetDisplayName() << "\n";
>>> "Child"
```

There are other alternative ways to add children:

```
// appends new SessionItem
auto child0 = item.InsertItem({tag, -1});

//! appends new PropertyItem
auto child1 = item.InsertItem<PropertyItem>({tag, -1});

// inserts child between child0 and child1 using move semantic
```

```
auto another = std::make_unique<VectorItem>
auto child2 = item.InserItem(std::move(another), {tag, 1});
```

4.5 Related files

- `taginfo.h` defines `TagInfo` class. It holds an information about single tag of `SessionItem`.
- `sessionitemcontainer.h` defines `SessionItemContainer` class. It holds the collection of `SessionItem` objects and `TagInfo` describing container properties.
- `taggeditems.h` defines `TaggedItems` class. It is a member of `SessionItem` and it holds a collection of `SessionItemContainers`.

Compound item

The `CompoundItem` is a convenience class derived from `SessionItem` that offers several additional methods to simplify the creation of item properties.

1. Adding properties

The property of the `CompoundItem` is another child item, inserted into the named container and carrying the data. There can be only one property item associated with the property name, and it can not be removed from the parent.

In the snippet below the `GaussianItem` carries two properties: one for the mean of the distribution, another for standard deviation.

```
class GaussianItem : public CompoundItem
{
public:
  GaussianItem() : CompoundItem("GaussianItem")
  {
    AddProperty("mean", 0.0);
    AddProperty("std_dev", 1.0);
  }
};
```

A similar effect could be achieved with the following code:

```
SessionItem parent;
parent.SetDisplayName("GaussianItem");

parent.RegisterTag(TagInfo::CreatePropertyTag("mean"));
auto mean_item = parent.InsertItem<PropertyItem>({"mean", 0});
mean_item->SetDisplayName("mean");
mean_item->SetData(0.0, DataRole::kData);

parent.RegisterTag(TagInfo::CreatePropertyTag("std_dev"));
auto std_dev_item = parent.InsertItem<PropertyItem>({"std_dev", 0});
std_dev_item->SetDisplayName("std_dev");
std_dev_item->SetData(1.0, DataRole::kData);
```

2. Accessing properties

The methods `CompoundItem::Property` and `CompoundItem::SetProperty` can be used to access and modify the underlying data of the property item.

```
GaussianItem item;

item.SetProperty("mean", 42.0);

std::cout << item.Property<double>("mean") << "\n";
>>> 42.0
```

The same can be done via `SessionItem` API:

```
GaussianItem item;

item.GetItem("mean", 0)->SetData(42.0, DataRole::kData);

std::cout << item.GetItem("mean", 0)->Data<double>(DataRole::kData) << "\n";
>>> 42.0
```

3. Other compound items as properties

Other `CompoundItems` can be registered as property items too. In the snippet below we define a `VectorItem` with three properties for (X, Y, Z) coordinates.

```
class VectorItem
{
  VectorItem() : CompoundItem("VectorItem")
  {
    AddProperty("X", 0.0);
    AddProperty("Y", 0.0);
    AddProperty("Z", 0.0);
  }
};
```

After that, we define `SphereItem` with the item `VectorItem` registered as a `Position` property.

```
class SphereItem
{
  SphereItem() : CompoundItem("SphereItem")
  {
    AddProperty<VectorItem>("Position");
  }
};
```

4. Remark on the back-compatibility

It is important to stress that the string `"mean"` used during property creation `AddProperty("mean", 0.0)` plays two roles: it is used as a tag name for container, and as a display name for property item.

Given below is an excerpt of an XML obtained after the serialization of `GaussianItem`.

```
<ItemContainer>
  <TagInfo max="1" min="1" name="mean">PropertyItem</TagInfo>
  <Item type="PropertyItem">
    <ItemData>
      <Variant role="0" type="double">42.0</Variant>
      <Variant role="2" type="string">mean</Variant>
    </ItemData>
  </Item>
</ItemContainer>
```

The string `"mean"` appears in `TagInfo` serialization, and the display name of the `PropertyItem`. This might become a problem if the user decide to change the display name of the property item from `"mean"` to `"Mean"`, for example:

```
AddProperty("Mean", 0.0); // mean -> Mean
```

It will then affect the name of the container and will lead to failure if one decides to update a new item from old XML files ("no such container exists"). To reduce the risk it is recommended to use unique names for item containers. One possible way of doing this is shown below:

```
class GaussianItem : public CompoundItem
{
public:
  static const std::string P_MEAN = "P_MEAN";
  GaussianItem() : CompoundItem("GaussianItem")
  {
    AddProperty(P_MEAN, 0.0)->SetDisplayName("Mean");
  }
};
```

Here the container was registered using the name which unlikely to be changed, and the display name is set separately. It also allows to access properties using string constants, instead of literals.

```
std::cout << item.Property<double>(GaussianItem::P_MEAN) << "\n";
>>> 42.0
```

5. Remark on conventional class API

The `CompoundItem` allows conveniently registering class properties. These properties are based on the same `SessionItem` machinery and are the subject to all benefits that the `SessionItem` hierarchy offer:

- Serialization.
- Editing in Qt widget.
- Undo/redo.

However, the extensive usage of `CompoundItem` API to manipulate item's properties has its disadvantages:

- Code is becoming cluttered with constructs like `item.Property<double>(GaussianItem::P_MEAN)`.
- Further refactoring might become problematic because of the lack of compile-time checks.

These problems can be addressed by using a conventional class API along with property machinery in the background:

`GaussianItem.h:`

```
class GaussianItem : public CompoundItem
{
public:
  GaussianItem();

  double GetMean() const;

  void SetMean(double value);

  double GetStdDev() const;

  void SetStdDev(double value);
};
```

`GaussianItem.cpp:`

```
static const std::string kMean = "kMean";
static const std::string kStdDev = "kStdDev";

GaussianItem::GaussianItem() : CompoundItem("GaussianItem")
{
    AddProperty(kMean, 0.0)->SetDisplayName("Mean");
    AddProperty(kStdDev, 1.0)->SetDisplayName("StdDev");
}

double GaussianItem::GetMean() const
{
  return Property<double>(kMean);
}

void GaussianItem::SetMean(double value)
```

```
{
  SetProperty(kMean, value);
}

double GaussianItem::GetStdDev() const
{
  return Property<double>(kStdDev);
}

void GaussianItem::SetStdDev(double value)
{
  SetProperty(kStdDev, value);
}
```

With this approach class API explicitly communicates its responsibilities, and implementation details of property machinery remain hidden from the users of the class.

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.

Serialization

Introduction

The `SessionModel` can be serialised to XML file, and then restored from it, using followind code:

```
SessionModel model;
XmlDocument document({&model});
document.save("filename.xml");

model.clear(); // clear the model or modify it in any way

document.load("filename.xml");

// at this point, the model will be exactly as at the moment of saving
}
```

Multiple models can be saved in single XML file, if needed:

```
SessionModel model;
PulseScheduleModel pulse_schedule_model;
ComponentModel component_model;

XmlDocument document({&model, &pulse_schedule_module, &component_model});
document.save("filename.xml");
}
```

Examples of serialization

Empty model

`C++`

```
TestModel model;
```

`XML`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Document>
  <Model type="TestModel"/>
</Document>
```

Model with single item

C++

```cpp
TestModel model;
model.InserItem<PropertyItem>();
```

XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Document>
  <Model type="TestModel">
    <Item type="PropertyItem">
      <ItemData>
        <Variant role="0" type="string">{5c383869-750e-4b7d-af18-fbca95494254}
</Variant>
        <Variant role="2" type="string">PropertyItem</Variant>
      </ItemData>
      <TaggedItems defaultTag=""/>
    </Item>
  </Model>
</Document>
```

`PropertyItem` with data

C++

```cpp
PropertyItem item;
item.setData(42, DataRole::kData);
item.setData("width", DataRole::kDisplay);
```

XML

```xml
<Item type="Property">
  <ItemData>
    <Variant role="0" type="string">{8f923bfc-94b3-456e-b222-0c81f19b8f5f}
</Variant>
    <Variant role="1" type="int">42</Variant>
```

```
    <Variant role="2" type="string">width</Variant>
  </ItemData>
  <TaggedItems defaultTag=""/>
</Item>
```

Parent with single child

C++

```
SessionItem parent;
parent.setDisplayName("parent_name");
parent.registerTag(TagInfo::CreateUniversalTag("defaultTag"), /*set_as_default*/
true);

auto child = parent.insertItem(std::make_unique<PropertyItem>(),
TagIndex::append());
child->setDisplayName("child_name");
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<Item type="SessionItem">
  <ItemData>
    <Variant role="0" type="string">{ca0fc80b-1246-4a69-8896-c6df46e3aa99}
</Variant>
    <Variant role="2" type="string">parent_name</Variant>
  </ItemData>
  <TaggedItems defaultTag="defaultTag">
    <ItemContainer>
      <TagInfo max="-1" min="0" name="defaultTag"/>
      <Item type="Property">
        <ItemData>
          <Variant role="0" type="string">{9459511d-1096-4fa1-a3e8-eb1d7386694d}
</Variant>
          <Variant role="2" type="string">child_name</Variant>
        </ItemData>
        <TaggedItems defaultTag=""/>
      </Item>
    </ItemContainer>
  </TaggedItems>
</Item>
```