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.