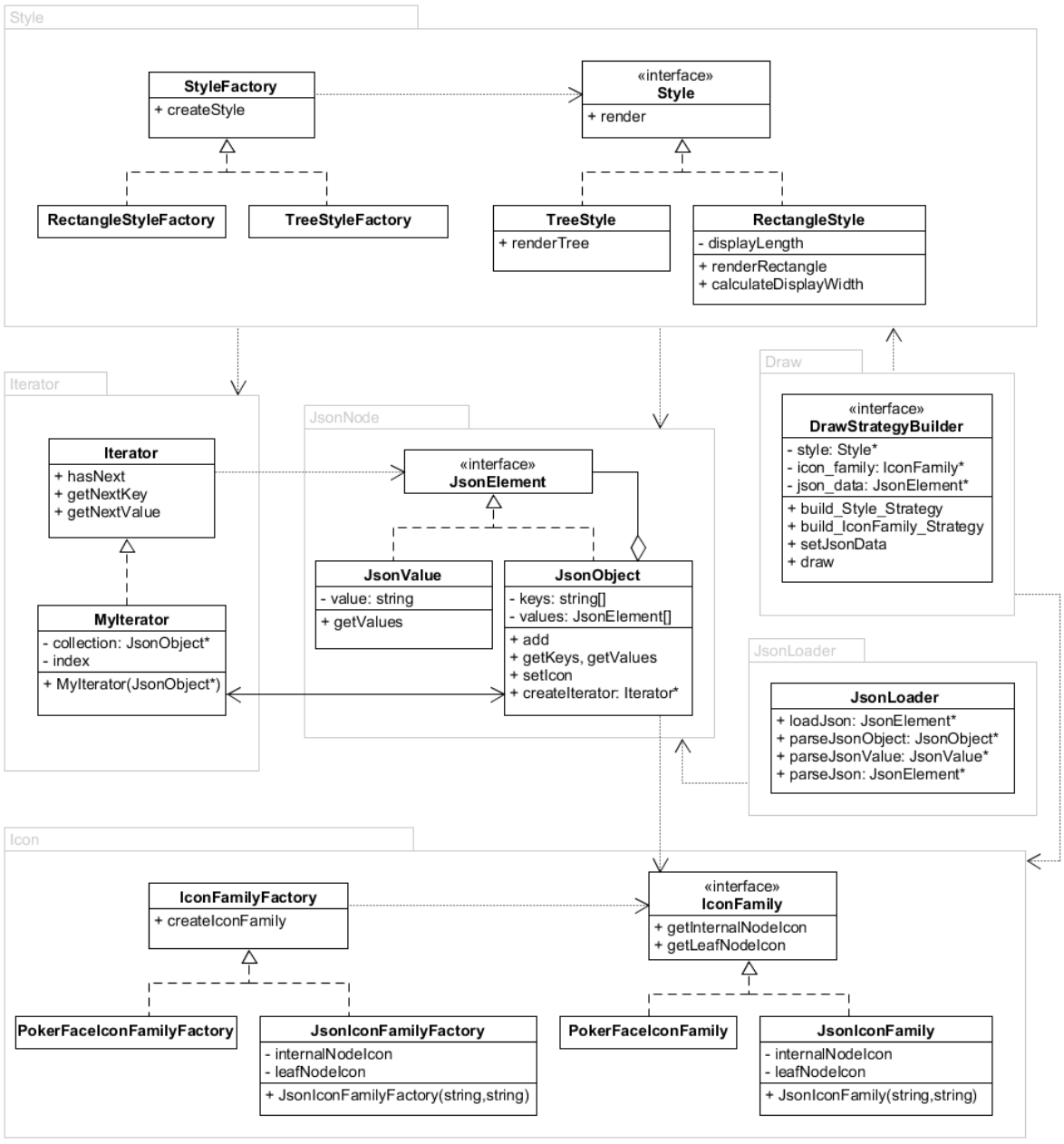
FJE-2.0 设计文档

21307140 李明俊

一. 项目类图及说明



Iterator：使用迭代器模式，包括一个抽象类`Iterator`和一个具体类`MyIterator`，`MyIterator`负责遍历一个`JsonObject`类型节点的子节点。

JsonNode：使用组合模式，该文件中的类负责以树的形式存储 Json 文件中的信息，其中的`setIcon`可以将指定图标族插入到树中的每一个节点key的最前面。同时，该文件也属于迭代器模式的一部分，`JsonObject`类中的`createIterator`函数可以创建一个`MyIterator`实例对自己的子节点进行遍历，创建时会把自己的指针传入以供`MyIterator`的初始化。

JsonLoader：负责处理输入的 Json 文件，将 Json 文件中的信息以树的形式保存在`JsonElement`类型的对象中，方便后续对其处理和渲染

Style：使用工厂方法，设定了不同的 FJE 风格，当前有矩形风格和树形风格，每一种风格中的`render`函数可以将`JsonElement`类型的数据转换成对应的风格。同时，在该文件实现的不同style类也是策略模式的一部分。

Icon：使用工厂方法，原理和Style相似，当前有扑克风格和自定义风格，可以使用`getInternalNodeIcon`和`getLeafNodeIcon`函数获取到当前Icon中间节点和叶子节点的图标。扑克风格的实现是直接返回两种图标，自定义风格是接收了来自`input/icon.json`中的图标然后返回，因此可以在该文件中自定义想使用的图标族。同时，在该文件实现的不同Icon族也是策略模式的一部分。

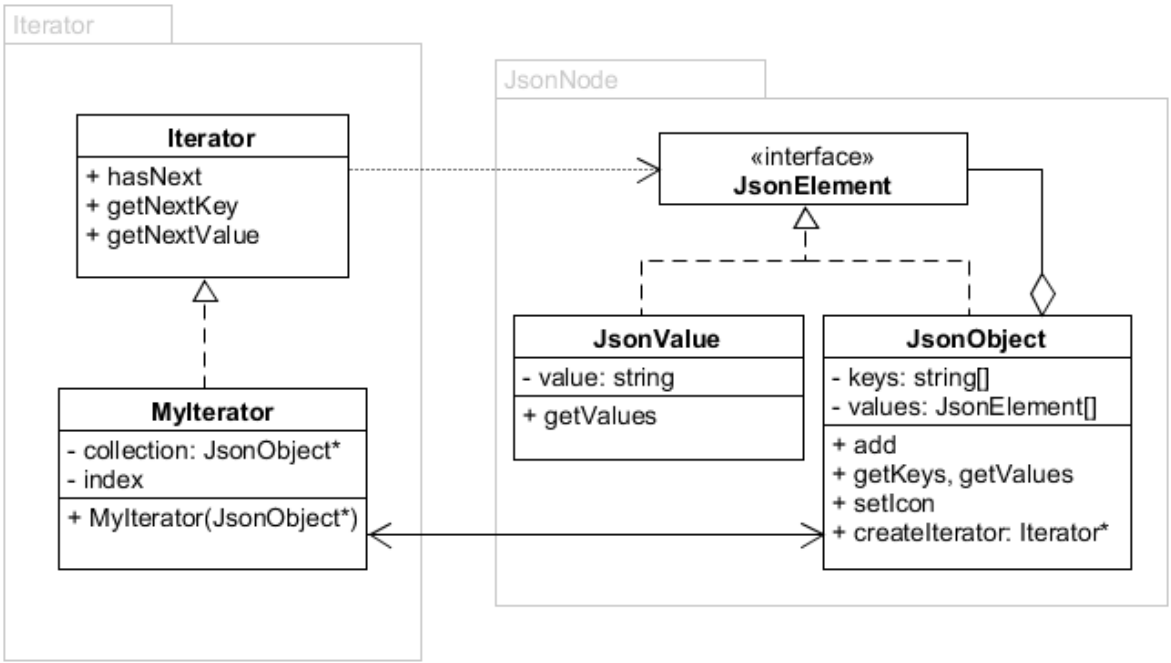
Draw：使用建造者模式和策略模式，根据选择的策略构建出所需要的FJE，`draw`函数可以返回当前 FJE 渲染后的结果

二. 项目设计模式及作用

（一）迭代器模式：

迭代器部分的类图如下所示，迭代器文件包括包括一个抽象类`Iterator`和一个具体类`MyIterator`，`MyIterator`负责遍历一个`JsonObject`类型节点的子节点（因此初始化时要传入`JsonObject*`），可以利用`getNextKey`和`getNextValue`函数迭代获得下一个子节点的信息，`index`记录当前遍历到的节点下标。

迭代器的另一部分是在`JsonObject`类中实现的，`JsonObject`类中的`createIterator`函数可以创建一个`MyIterator`实例对自己的子节点进行遍历，创建时要把自己的指针传入以供`MyIterator`的初始化。



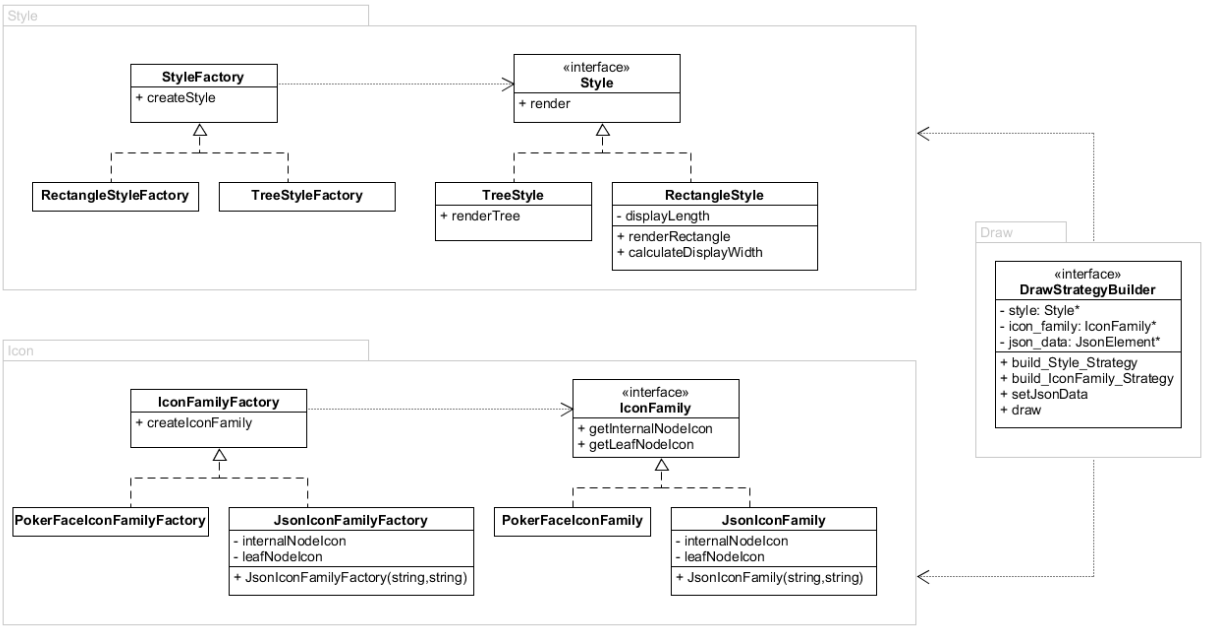
实现上述迭代器后，就可以通过下面的方式遍历`JsonObject`类型节点的子节点，这可以使遍历一个复杂对象的内部结构变得更加简洁和清晰，用户不需要了解对象的内部结构，也不需要编写复杂的循环和条件判断来遍历元素，只需使用迭代器提供的接口方法即可。同时，使用迭代器模式还可以统一包含不同内部结构的对象的遍历接口，支持多种遍历方式，将对象的遍历逻辑与对象本身的实现分离，从而提高代码的可读性和可维护性。

```
auto* obj = dynamic_cast<JsonObject*>(data.get());
if (obj) {
    auto iterator = obj->createIterator();
    while(iterator->hasNext()){
        const auto& key = iterator->getNextKey();
        const auto& value = iterator->getNextValue();
        //......do something
    }
```

（二）策略模式：

策略模式部分的类图及代码如下所示，之前实现的不同种style和icon实际就是两个方向的策略，这里将策略模式和建造者模式进行了一个简单的合并，将两个方向策略的选择，构建，绘制集中在同一个类`DrawStrategyBuilder`中：

1. `build_Style_Strategy`函数接收要使用的`StyleFactory`，创建特定种类的`Style`实例
2. `build_IconFamily_Strategy`函数接收要使用的`IconFamilyFactory`，创建特定种类的`IconFamily`实例
3. `setJsonData`函数负责接收所要处理的数据
4. `draw`函数根据生成的策略处理输入数据，返回最终结果



```
class DrawStrategyBuilder {
public:
    void build_Style_Strategy(std::unique_ptr<StyleFactory> styleFactory);
    void build_IconFamily_Strategy(std::unique_ptr<IconFamilyFactory>
iconFamilyFactory);
    void setJsonData(std::shared_ptr<JsonElement> json_data);
    std::string draw();

private:
    std::shared_ptr<Style> style;
    std::shared_ptr<IconFamily> icon_family;
    std::shared_ptr<JsonElement> json_data;
};
```

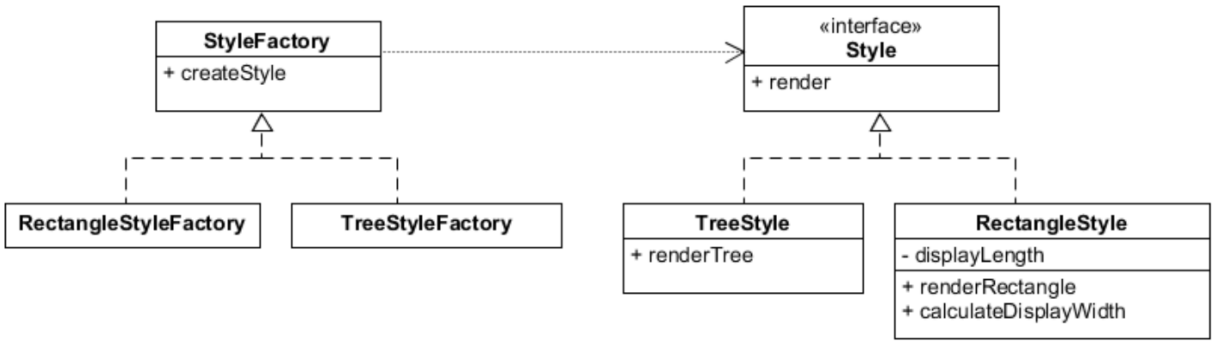
实现策略模式后，就可以使用下面的代码简单地实现策略的选择构建与执行。策略模式通过将具体算法的实现从上下文类中抽离出来，使得算法可以独立于使用它的客户端类变化，不仅简化了客户端类的实现，还允许在不修改客户端代码的情况下扩展新的算法。同时，使用策略模式可以替换掉大量的条件语句，客户端类不再需要通过条件判断来选择执行的算法，而是通过策略对象的多态性来决定具体执行哪个算法，从而提高代码的可读性和可维护性。

```
    // 使用 建造者模式 + 策略模式
    auto builder = std::make_shared<DrawStrategyBuilder>();
    builder->build_Style_Strategy(std::move(style_factory));
    builder->build_IconFamily_Strategy(std::move(icon_family_factory));
    builder->setJsonData(json_data);
    std::string result = builder->draw();
```

（三）工厂方法:

项目中的 style 和 icon 使用到了工厂方法，其中style用到工厂方法的部分代码及类图如下，不同风格类 `TreeStyle` 和 `RectangleStyle` 均继承 `Style`，从而实现了多态，使得 `Style*` 可以自动转换为子类指针，不同风格类中包含的 `render` 函数可以按照自己的风格类型处理数据。

不同风格类工厂 `TreeStyleFactory` 和 `RectangleStyleFactory` 均继承 `StyleFactory`，也实现了多态，使得调用 `StyleFactory*` 中的 `createStyle` 函数时可以自动创建所需类型的风格实例。




```
// 风格接口工厂
class StyleFactory {
    virtual std::shared_ptr<Style> createStyle() const = 0;
    virtual ~StyleFactory() = default;
```

```
};

// 树形风格工厂
class TreeStyleFactory : public StyleFactory {
    std::shared_ptr<Style> createStyle() const override;
};

// 矩形风格工厂
class RectangleStyleFactory : public StyleFactory {
    std::shared_ptr<Style> createStyle() const override;
};
```

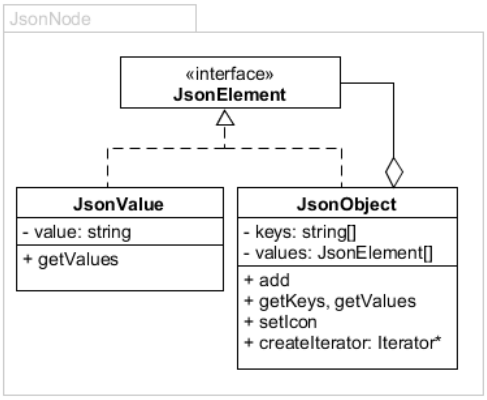
实现上述工厂方法后，就可以通过下面代码创建所需类型的风格，因此工厂方法有以下作用：

1. **解耦对象创建和使用**：将对象的创建与使用分离，使得客户端代码不需要知道具体的类名和创建过程，只需要依赖抽象的工厂接口，提高了代码的灵活性和可维护性
2. **提高项目的可扩展性**：当项目需要添加新的风格或者icon族时，只需添加一个新的工厂类和对应的产品类，不需要修改现有的客户端代码，从而符合开闭原则（对扩展开放，对修改关闭）
3. **简化对象的创建过程**：可以封装复杂的对象创建过程，客户端只需调用工厂方法即可得到对象，无需关心对象创建的具体细节。
4. **便于管理对象生命周期**：可以在工厂类中集中管理对象的创建和销毁逻辑，便于控制对象的生命周期，防止内存泄漏等问题。
5. **实现代码复用**：将创建对象的逻辑集中在工厂类中，避免了在多个地方重复创建对象的代码，提高了代码的复用性。

```
std::unique_ptr<StyleFactory> style_factory;
std::shared_ptr<Style> mystyle = styleFactory->createStyle();
```

（四）组合模式：

项目中的 JsonNode 使用到了组合模式，使用到组合模式的部分代码及类图如下




```
// 抽象组件
class JsonElement {
public:
    virtual ~JsonElement() = default;
```

```
};

// 组合组件：json对象，非叶子节点
class JsonObject : public JsonElement {
public:
    void add(const std::string& key, std::shared_ptr<JsonElement> value);
    //......
private:
    //......
    std::vector<std::shared_ptr<JsonElement>> values;
};

// 叶子组件：json值
class JsonValue : public JsonElement {
    //......
    std::string value;
};
```

`JsonObject` 和 `JsonValue` 均继承自 `JsonElement`，而 `JsonObject` 又包含了一个 `JsonElement` 类型的数组，这种方式实际上就可以形成一种树形结构（如下图示例所示），`JsonObject` 是树中的非叶子节点，`JsonValue` 是树中的叶子节点，这样就可以在处理 json 文件时将其中的信息以树的形式保存下来。

除了提供灵活的层级结构（如树）外，组合模式还定义了一个统一的接口（`JsonElement`），使得客户端代码可以一致地处理简单和复杂对象。这样，无论是处理单个 `JsonValue` 还是包含多个子节点的 `JsonObject`，客户端代码都可以使用相同的方式进行操作，简化了代码逻辑。