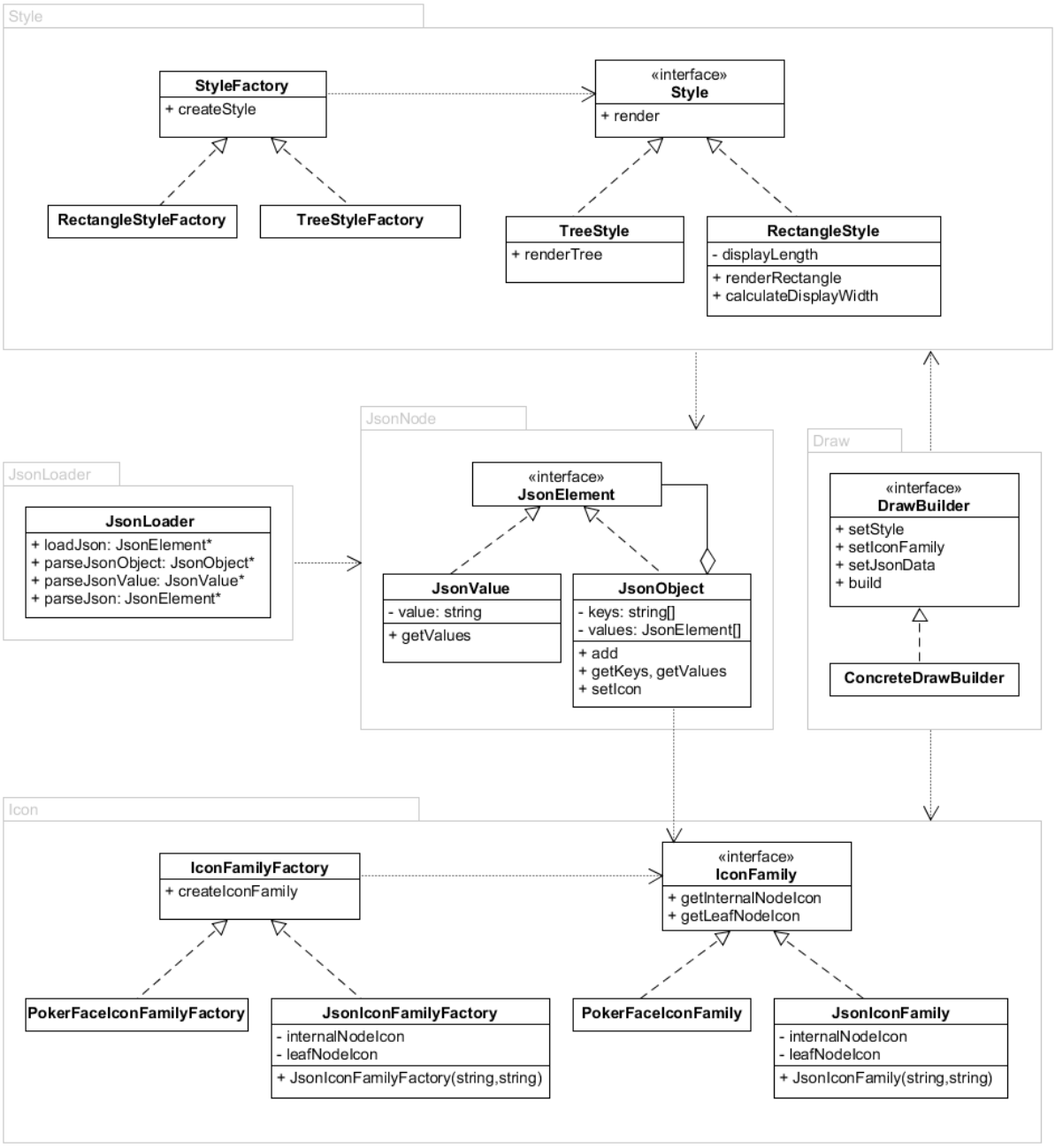
FJE 设计文档

21307140 李明俊

一. 项目类图及说明



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

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

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

Draw：使用建造者模式，构建出所需要的FJE，`build`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

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如下：

```
// 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;
};
```

工厂方法模式将对象创建的代码集中在一个地方，避免了在多个地方重复编写相同的实例化代码，从而提高了代码的可读性和可维护性。同时，使用工厂方法也提高了项目的灵活性和可扩展性，当项目需要添加新的风格或者坐标族时，只需要添加新的工厂方法即可对其创建进行统一管理。

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：

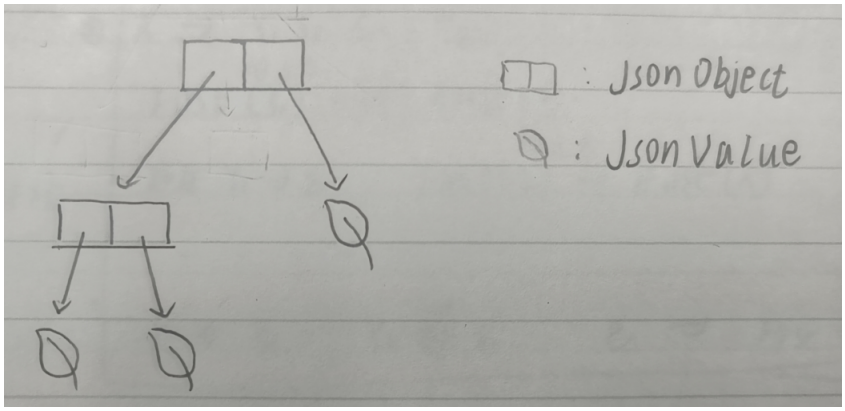
```
// 抽象组件
class JsonElement {
public:
    virtual ~JsonElement() = default;
};

// 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



建造者模式：

项目中的DrawBuilder使用到了建造者模式，部分代码如下：

```
class DrawBuilder {
public:
    virtual void buildStyle() = 0;
    virtual void buildIconFamily() = 0;
    virtual void SetJsonData(std::shared_ptr<JsonElement> json_data) = 0;
    virtual std::string build() const = 0;
    virtual ~DrawBuilder() = default;
};

class ConcreteDrawBuilder1 : public DrawBuilder {
public:
    void buildStyle() override;
    void buildonFamily() override;
    void SetJsonData(std::shared_ptr<JsonElement> json_data) override;
    std::string build() const override;
private:
    std::shared_ptr<Style> style;
    std::shared_ptr<IconFamily> icon_family;
    std::shared_ptr<JsonElement> json_data;
};
//......
```

通过这种方式，就可以利用建造者模式构建出我们所需的不同类型的FJE。例如，可以用使用`ConcreteDrawBuilder1`构建出矩形风格+扑克图标的FJE，`ConcreteDrawBuilder2`构建出树形风格+自定义图标的FJE …… 然后build函数可以计算出每一种FJE中的渲染结果，方便不同类型的FJE的统一构建与管理。通过一个 Builder 接口按顺序创建部件，以便将对象的构建与表示分离，从而提高了代码的可读性和可维护性。（在最终代码中，为了使用及测试方便，将build改为了set直接设置）