# **Funny JSON Explorer**

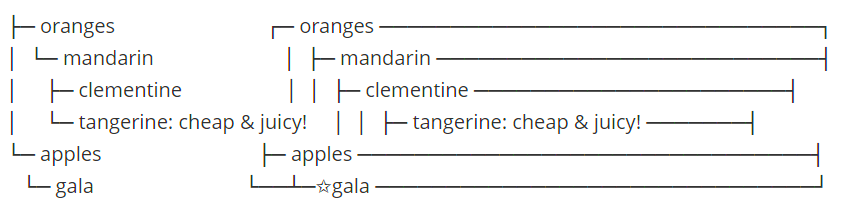
万聂林21310247

## **一、实验要求**

Funny JSON Explorer（**FJE**），是一个JSON文件可视化的命令行界面小工具

fje -f <json file> -s <style> -i <icon family>  
​

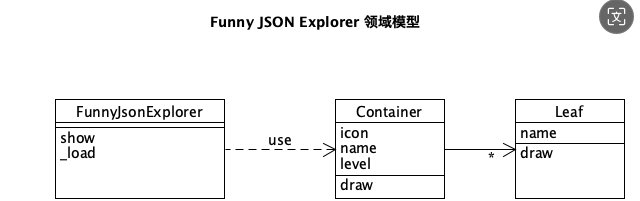
{  
   oranges: {  
       'mandarin': {                           ├─ oranges  
           clementine: null,                   │ └─ mandarin  
           tangerine: 'cheap & juicy!' -=>     │     ├─ clementine  
       }                                       │     └─ tangerine: cheap & juicy!  
   },                                           └─ apples  
   apples: {                                       ├─ gala  
       'gala': null,                               └─ pink lady  
       'pink lady': null  
   }  
}  
````  
​  
FJE可以快速切换\*\*风格\*\*（style），包括：树形（tree）、矩形（rectangle）；  
​

  
   树形（tree）                                   矩形（rectangle）

也可以指定**图标族**（icon family），为中间节点或叶节点指定一套icon

├─♢oranges                                   
│ └─♢mandarin                               
│     ├─♤clementine                          
│     └─♤tangerine: cheap & juicy!      
└─♢apples                                    
   └─♤gala                                   
​  
poker-face-icon-family: 中间节点icon：♢ 叶节点icon：♤

### **领域模型**



### **作业要求**

基于上述需求描述和领域模型，按照设计模式要求，进行软件设计，并编码实现（任何语言均可）。

### **设计模式**

使用**工厂方法**（Factory）、**抽象工厂**（Abstract Factory）、**建造者**（Builder）模式、**组合模式**（Composition），完成功能的同时，使得程序易于扩展和维护。

（必做）：不改变现有代码，只需添加新的抽象工厂，即可添加新的风格

（选做）：通过配置文件，可添加新的图标族

### **作业提交**

设计文档：类图与说明，说明使用的设计模式及作用

运行截图：两种风格，两种图标族，共计4次运行fje的屏幕截图

源代码库：公开可访问的Github repo URL

### **参考资料**

unicode 制表符与图标： <https://unicode.yunser.com/>

## 预备知识

### 1. 工厂方法（Factory Method）

\*\*定义：\*\*

工厂方法模式是一种创建型设计模式，它通过创建工厂类来代替直接实例化对象的方式，使得子类可以决定实例化哪个类。

\*\*应用场景：\*\*

当一个类无法预见需要创建哪个类的实例时，或者一个类希望由其子类来指定它所创建的对象时使用。

\*\*示例：\*\*

```cpp

// 产品接口

class Product {

public:

virtual void use() = 0;

};

// 具体产品A

class ProductA : public Product {

public:

void use() override {

std::cout << "Using ProductA" << std::endl;

}

};

// 具体产品B

class ProductB : public Product {

public:

void use() override {

std::cout << "Using ProductB" << std::endl;

}

};

// 工厂方法接口

class Factory {

public:

virtual Product\* createProduct() = 0;

};

// 具体工厂A

class FactoryA : public Factory {

public:

Product\* createProduct() override {

return new ProductA();

}

};

// 具体工厂B

class FactoryB : public Factory {

public:

Product\* createProduct() override {

return new ProductB();

}

};

```

### 2. 抽象工厂（Abstract Factory）

\*\*定义：\*\*

抽象工厂模式提供一个接口，用于创建一系列相关或互相依赖的对象，而无需指定它们的具体类。

\*\*应用场景：\*\*

当系统要独立于产品的创建和组织时，或者当系统需要一个产品家族而不是某个具体产品时使用。

\*\*示例：\*\*

```cpp

// 产品接口

class ProductA {

public:

virtual void use() = 0;

};

class ProductB {

public:

virtual void eat() = 0;

};

// 具体产品A1

class ProductA1 : public ProductA {

public:

void use() override {

std::cout << "Using ProductA1" << std::endl;

}

};

// 具体产品B1

class ProductB1 : public ProductB {

public:

void eat() override {

std::cout << "Eating ProductB1" << std::endl;

}

};

// 抽象工厂接口

class AbstractFactory {

public:

virtual ProductA\* createProductA() = 0;

virtual ProductB\* createProductB() = 0;

};

// 具体工厂1

class Factory1 : public AbstractFactory {

public:

ProductA\* createProductA() override {

return new ProductA1();

}

ProductB\* createProductB() override {

return new ProductB1();

}

};

```

### 3. 建造者（Builder）模式

\*\*定义：\*\*

建造者模式是一种创建型设计模式，它允许使用逐步构建复杂对象的方式，而不需要关心每个部分是如何组装的。

\*\*应用场景：\*\*

当需要创建一个复杂的对象，其构建过程应独立于对象的组成部分以及它们的装配方式时使用。

\*\*示例：\*\*

```cpp

// 产品类

class Product {

public:

void setPartA(const std::string& part) { partA = part; }

void setPartB(const std::string& part) { partB = part; }

void show() {

std::cout << "Product with " << partA << " and " << partB << std::endl;

}

private:

std::string partA;

std::string partB;

};

// 抽象建造者接口

class Builder {

public:

virtual void buildPartA() = 0;

virtual void buildPartB() = 0;

virtual Product\* getResult() = 0;

};

// 具体建造者

class ConcreteBuilder : public Builder {

public:

ConcreteBuilder() { product = new Product(); }

void buildPartA() override { product->setPartA("PartA1"); }

void buildPartB() override { product->setPartB("PartB1"); }

Product\* getResult() override { return product; }

private:

Product\* product;

};

// 指挥者

class Director {

public:

void construct(Builder& builder) {

builder.buildPartA();

builder.buildPartB();

}

};

```

### 4. 组合模式（Composite Pattern）

\*\*定义：\*\*

组合模式是一种结构型设计模式，它将对象组合成树形结构以表示部分-整体的层次结构，使得客户端可以统一地处理单个对象和组合对象。

\*\*应用场景：\*\*

当需要表示对象的部分-整体层次结构，或希望客户端可以统一处理单个对象和组合对象时使用。

\*\*示例：\*\*

```cpp

// 组件接口

class Component {

public:

virtual void add(Component\* component) {}

virtual void remove(Component\* component) {}

virtual void display(int depth) = 0;

};

// 叶子节点

class Leaf : public Component {

public:

Leaf(const std::string& name) : name(name) {}

void display(int depth) override {

std::cout << std::string(depth, '-') << name << std::endl;

}

private:

std::string name;

};

// 组合节点

class Composite : public Component {

public:

void add(Component\* component) override {

children.push\_back(component);

}

void remove(Component\* component) override {

children.erase(std::remove(children.begin(), children.end(), component), children.end());

}

void display(int depth) override {

std::cout << std::string(depth, '-') << "Composite" << std::endl;

for (auto child : children) {

child->display(depth + 2);

}

}

private:

std::vector<Component\*> children;

};

```

### 总结

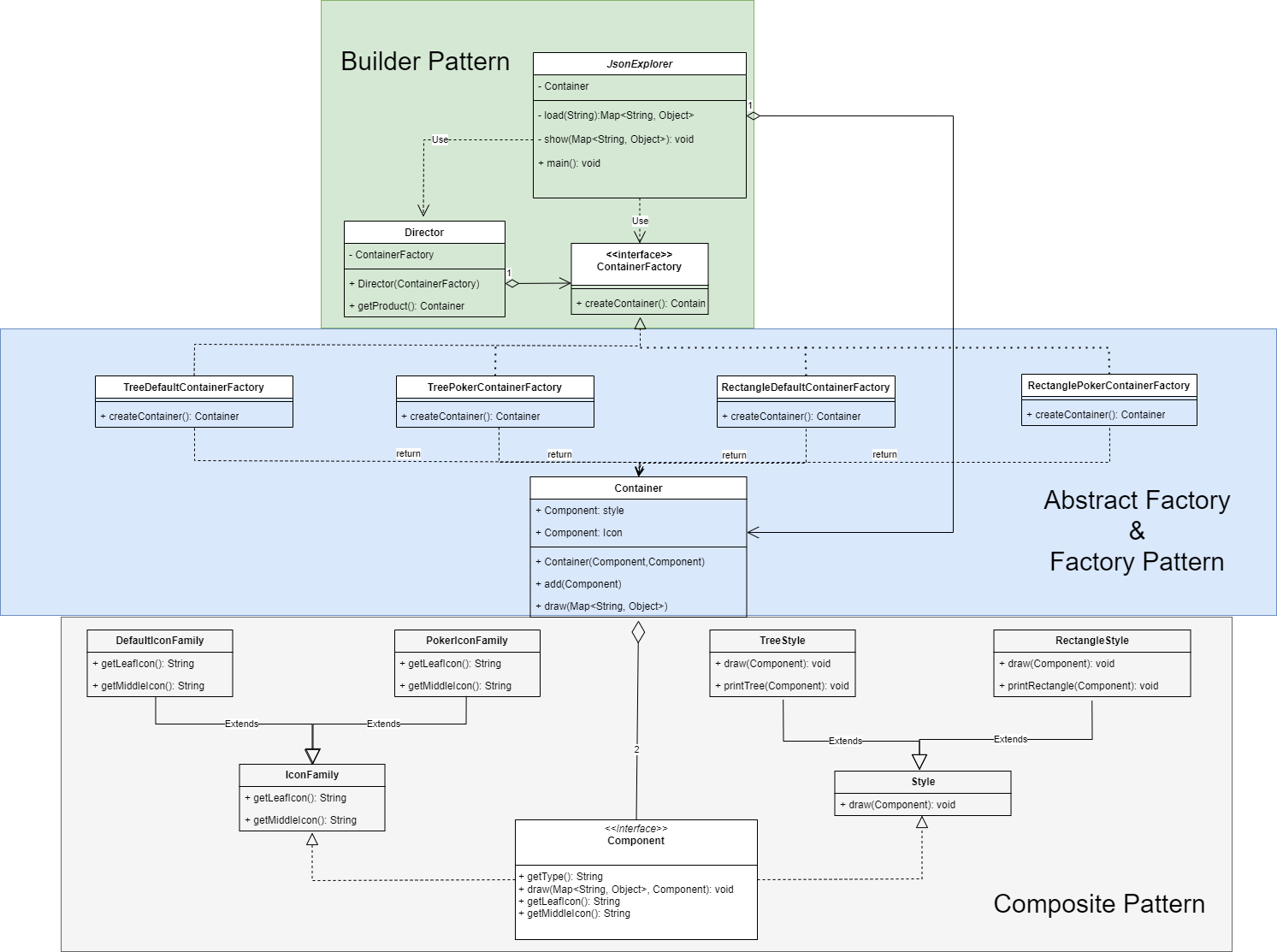
- \*\*工厂方法\*\*：用于创建单一类型对象的工厂类。

- \*\*抽象工厂\*\*：用于创建一系列相关或依赖对象的工厂类。

- \*\*建造者\*\*：用于分步骤创建复杂对象。

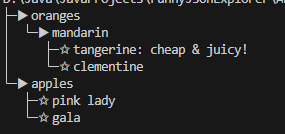
- \*\*组合模式\*\*：用于将对象组合成树形结构，方便处理部分-整体关系。

#### **三、类图**

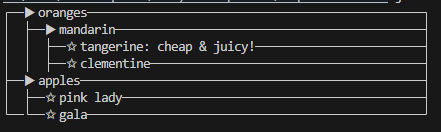


## **结果展示：**

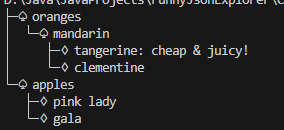
### **Default图标族+树形风格**



### **Default图标族+矩阵风格**



### **Poker图标族+树形风格**



### **Poker图标族+矩阵风格**

