参考教材6.3，结合自己项目的实际开发，分析采用了哪些面向对象的设计？

1. 面向对象的特征：

* 对象是唯一可标识的且的运行时实体，它们可以设计为消息或请求的目标。
* 对象是可组合的，因为它的数据变量本身可能也是对象，因而封装了对象的内部变量的实现。
* 对象的实现可以通过继承的方式被复用和扩展，用来定义其他对象的实现。
* 面向对象的代码可以是多态的：可以对多个不同但类型相关的对象都起作用的通用代码。相关类型的对象会对一些相同的消息或请求做出响应，但不同类型的对象会有不同的响应。

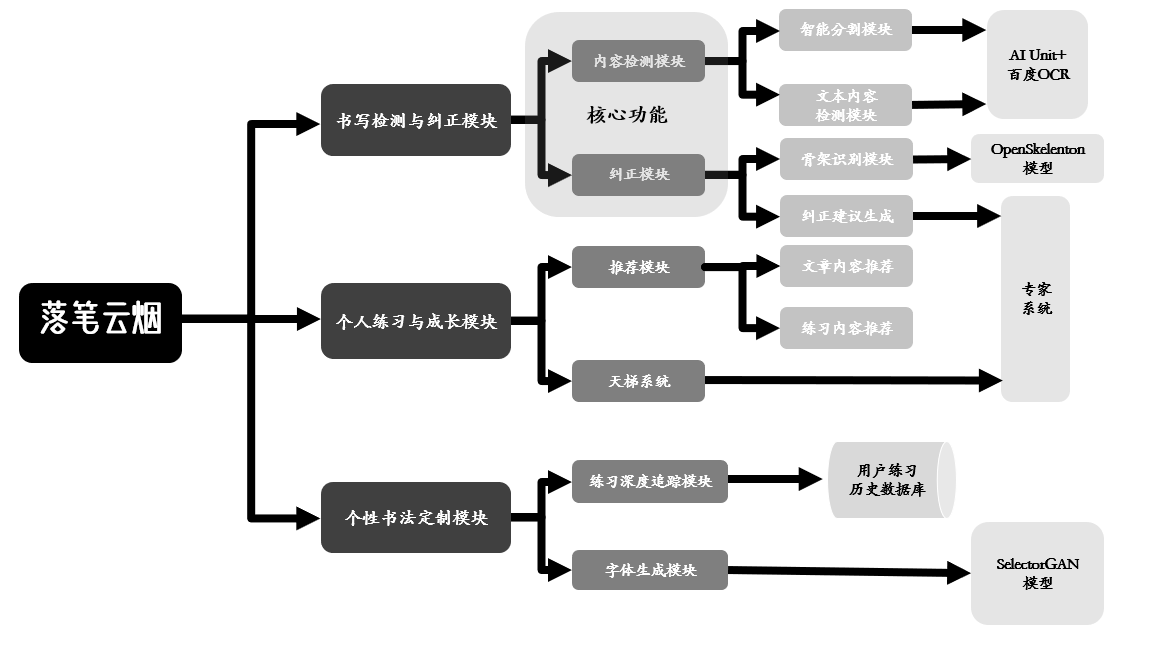
1. 继承与对象组合：

设计中一个关键的决策就是如何最好地组织和关联复杂的对象。在面向对象的系统中，构造大型对象的技术主要有两种：继承和组合。也就是说，可以通过扩展和重载现有类的行为来创建新的类，或者通过组合简单的类来形成一个新类。

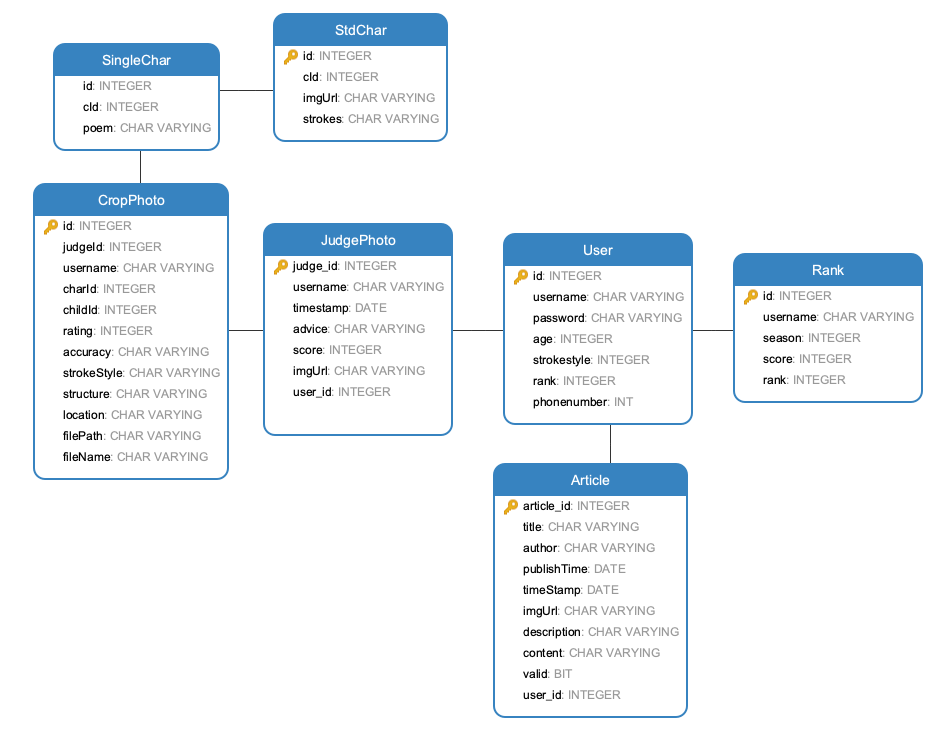
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 技术问题 | 对象 | 描述 | 难度 | 依赖关系 |
| 图像降噪 | ①图像降噪 | 输入一张图片，去除图片中的噪声 | 中等 |  |
| 手写字检测 | ②手写字检测 | 输入降噪后的手写字的图片，检测出图片中所有手写的字，返回Bounding Box和对应标签 | 简单 | 依赖① |
| 整体书写评价 | ③单字评价 | 在多字图片中提取单字重心及Bounding Box | 中等 | 依赖② |
|  | ④整齐度计算 | 计算整体重心偏移量方差以及距离方差，并进行合理评估打分 | 简单 | 依赖③ |
| 书写报告生成 | ⑤参数预处理 | 根据整体书写评价得到的各项参数做相关预处理 | 简单 | 依赖③④ |
|  | ⑥自然语言生成 | 根据得到的各项评价参数生成用户可以理解的书写报告 | 中等 | 依赖⑤ |
| 汉字骨架识别 | ⑦关键点热力图/笔画PAF预测 | 根据输入的图片，使用深度学习模型预测关键点的热力图，笔画方向PAF图 | 高 | 依赖② |
|  | ⑧OKS算法 | 通过NMS算法的变形OKS算法，过滤掉距离较近的同样类型的关键点 | 中等 | 依赖⑦ |
|  | ⑨PAF路径计算 | 通过计算不同连接路径的向量与PAF的积分，得到这一连接的分数 | 高 | 依赖⑦⑧ |
|  | ⑩最优子集搜索 | 根据汉字包含的笔画信息，构建多个可能的集合，计算不同集合的PAF分数，得到最优的笔画集合 | 高 | 依赖⑦⑧ |
| 汉字图像对齐 | ⑪图像缩放 | 通过检测出的关键点对图像进行缩放 | 低 | 依赖⑦⑧⑨⑩ |
|  | ⑫图像旋转 | 通过深度学习模型预测端正一个汉字所需的旋转角度 | 中等 | 依赖② |
| 汉字纠正 | ⑬模版图像位置匹配 | 根据检测出的汉字骨架，找出模版汉字与输入汉字的最佳对应位置 | 低 | 依赖⑦⑧⑨⑩ |
|  | ⑭差异计算 | 根据检测出的汉字骨架，判断是否缺少笔画、笔画间的长度、倾斜度的差异，并将结果反馈给用户 | 低 | 依赖⑦⑧⑨⑩ |
|  | ⑮专家系统匹配 | 根据检测出的汉字骨架与识别出的汉字信息，匹配专家系统中已经录入好的指导信息，判断当前书写的缺陷所在，返回对应的指导建议 | 高 | 依赖②⑦⑧⑨⑩ |
| 单字评分 | ⑦单字评价模型 | 根据识别出的汉字骨架与汉字纠正模块的差异计算结果，给出当前汉字的评分 | 中等 | 依赖②⑦⑧⑨⑩⑭ |

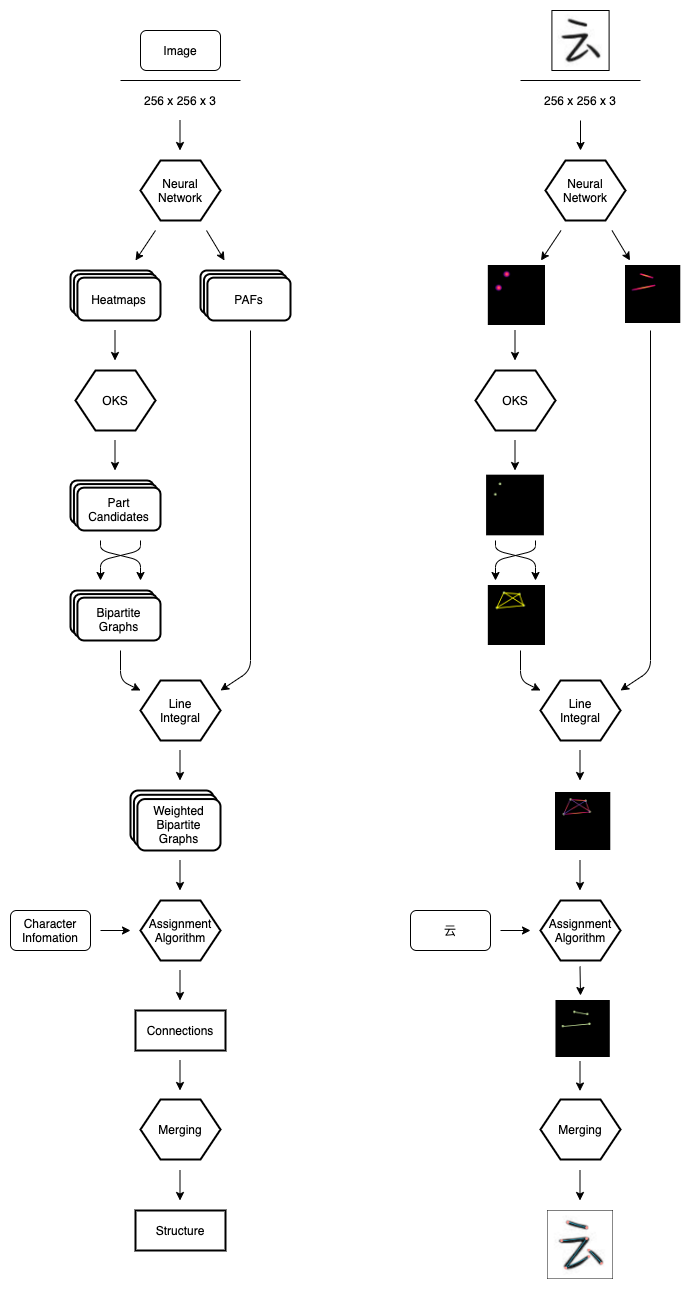
三大模块对象：书写检测与纠正、个人练习与成长、个性书法定制：

1. 书写检测与纠正模块作为核心功能模块，主要围绕汉字骨架识别的OpenSkeleton算法对用户的书写内容进行分析，并依靠专家系统为其提出合理建议，改善用户的书写习惯，让书写更加协调美观；系统有机结合了AIUnit提供的端侧算力支持，算法可以更迅速地得到用户书写内容，极大提升用户使用体验。
2. 书写检测与纠正模块作为核心功能模块，主要围绕汉字骨架识别的OpenSkeleton算法对用户的书写内容进行分析，并依靠专家系统为其提出合理建议，改善用户的书写习惯，让书写更加协调美观；系统有机结合了AIUnit提供的端侧算力支持，算法可以更迅速地得到用户书写内容，极大提升用户使用体验。
3. 书写检测与纠正模块作为核心功能模块，主要围绕汉字骨架识别的OpenSkeleton算法对用户的书写内容进行分析，并依靠专家系统为其提出合理建议，改善用户的书写习惯，让书写更加协调美观；系统有机结合了AIUnit提供的端侧算力支持，算法可以更迅速地得到用户书写内容，极大提升用户使用体验。



对象过程设计：



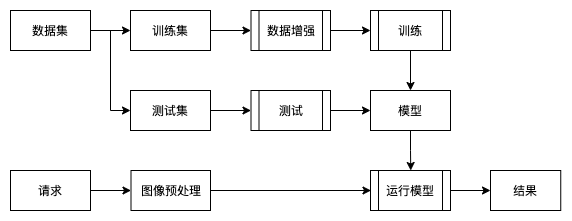


从CMU提出的OpenPose模型中得到启发，使用PAFs（Part Affinity Fields）来编码汉字书写图像中的笔画方向信息，通过预测关键点位置的置信图来得到不同笔画的关键点的位置。得到关键点的置信图和笔画的PAFs后，便可以通过贪心算法来计算不同连接的分数，之后通过识别出的汉字信息，构造可能的汉字笔画的集合，通过计算不同集合的分数得到可能性最大的连接集合，将集合中的关键点与相连即可得到汉字的骨架结构。

1. 可替换性：

子类必须保持其父类的行为，这样，客户端代码才能把它的实例也当成其父类的实例来同等对待。

* 子类支持父类的所有方法，并且它们的签名是兼容的。也就是说，子类的方法的参数和返回的类型对于父类方法的对应的参数和返回的类型，是可替代的，这样，对父类方法的任何调用也会被子类所接受。
* 子类的方法必须满足父类方法的规格说明。这两个类的方法行为不一定要完全相同，但是子类必须不违反父类方法的前置条件以及后置条件。
* 子类必须保留父类中声明了的所有性质。



训练集与测试集作为数据集的子类，可以都当作数据集对象来处理。

问题：

亓龙雨： 接口和抽象类有什么区别？

首先接口和抽象类的设计目的就是不一样的。接口是对动作的抽象，而抽象类是对根源的抽象。

1. 抽象类和接口都不能被直接实例化，如果二者要实例化，就涉及到多态，如果抽象类要实例化，那么抽象类定义的变量必须指向一个子类对象，这个子类继承了这个抽象类并实现了这个抽象类的所有抽象方法。如果接口要实例化，那么这个接口定义的变量要指向一个子类对象，这个子类必须实现了这个接口所有的方法。
2. 抽象类要被子类继承，接口要被子类实现。
3. 接口里面只能对方法进行声明，抽象类既可以对方法进行声明也可以对方法进行实现。
4. 抽象类里面的抽象方法必须全部被子类实现，如果子类不能全部实现，那么子类必须也是抽象类。接口里面的方法也必须全部被子类实现，如果子类不能实现那么子类必须是抽象类。
5. 口里面的方法只能声明，不能有具体的实现。这说明接口是设计的结果，抽象类是重构的结第
6. 抽象类里面可以没有抽象方法。
7. 抽象级别（从高到低）：接口>抽象类>实现类。
8. 抽急类的功能应该要远多于接口，但是定义抽象类的代价较高。因为高级语言一个类只能继承一个父类，即你在设计这个类的时候必须要抽象出所有这个类的子类所具有的共同属性和方法；但是类（接口）却可以继承多个接口，因此每个接口你只需要将特定的动作方法抽象到这个接口即可。也就是说，接口的设计具有更大的可扩展性，而抽象类的设计必须十分谨慎。

韦世强：接口在应用中的模式：

1. 策略模式：策略模式的思想是，针对一组算法，将每一种算法封装到具有共同接口的实现类中，接口的设计者可以在不影响调用者的情况下对算法做出改变。

// 接口：教练

interface Coach {

    // 方法：防守

    void defend();

}

// 何塞·穆里尼奥

class Hesai implements Coach {

    @Override

    public void defend() {

        System.out.println("防守赢得冠军");

    }

}

// 德普·瓜迪奥拉

class Guatu implements Coach {

    @Override

    public void defend() {

        System.out.println("进攻就是最好的防守");

    }

}

public class Demo {

    // 参数为接口

    public static void defend(Coach coach) {

        coach.defend();

    }

    public static void main(String[] args) {

        // 为同一个方法传递不同的对象

        defend(new Hesai());

        defend(new Guatu());

    }

}

1. 适配器模式的思想是，针对调用者的需求对原有的接口进行转接。生活当中最常见的适配器就是HDMI（英语：High Definition Multimedia Interface，中文：高清多媒体接口）线，可以同时发送音频和视频信号。

interface Coach {

    void defend();

    void attack();

}

// 抽象类实现接口，并置空方法

abstract class AdapterCoach implements Coach {

    public void defend() {};

    public void attack() {};

}

// 新类继承适配器

class Hesai extends AdapterCoach {

    public void defend() {

        System.out.println("防守赢得冠军");

    }

}

public class Demo {

    public static void main(String[] args) {

        Coach coach = new Hesai();

        coach.defend();

    }

}

1. 工厂模式：简单工厂模式也被称为静态工厂模式;使用简单工厂模式可以将产品的“消费”和生产完全分开，客户端只需要知道自己需要什么产品，如何来使用产品就可以了，具体的产品生产任务由具体的工厂类来实现。工厂类根据传进来的参数生产具体的产品供消费者使用。这种模式使得更加利于扩展，当有新的产品加入时仅仅需要在工厂中加入新产品的构造就可以了

黄一鸣：什么是迪米特法则：

一个软件实体应当尽可能少的与其他实体发生相互作用。每一个软件单位对其他的单位都只有最少的知识，而且局限于那些与本单位密切相关的软件单位。迪米特法则的初衷在于降低类之间的耦合。由于每个类尽量减少对其他类的依赖，因此，很容易使得系统的功能模块功能独立，相互之间不存在（或很少有）依赖关系。迪米特法则不希望类之间建立直接的联系。如果真的有需要建立联系，也希望能通过它的友元类来转达。因此，应用迪米特法则有可能造成的一个后果就是：系统中存在大量的中介类，这些类之所以存在完全是为了传递类之间的相互调用关系——这在一定程度上增加了系统的复杂度。

public class LODRightTest {

    public static void main(String[] args) {

        Phone2 phone2 = new Phone2();

        phone2.readBook();

    }

}

class Phone2 {

    private App2 app2 = new App2();

    public void readBook() {

        app2.read();

    }

}

class App2 {

    private Book2 book2 = new Book2("设计模式");

    public void read() {

        System.out.println(book2.getTitle());

    }

}

class Book2 {

    private String title;

    public Book2(String title) {

        this.title = title;

    }

    public String getTitle() {

        return title;

    }

    public void setTitle(String title) {

        this.title = title;

    }

}

尹永琪：如果X类型要用到B类型的一些函数 , 现在你想让X类型也兼容C类型的同名的函数，那应该怎么做？

B和C的接口假设为func , 那么把BC改成A类型的派生类 , A类型有一个虚函数func(虚函数就是只是说我的派生类会实现 , 但我不会实现 , 根据多态的时候如果是B转A则调用A的func就是调用B的func) , 然后X类型里写的不是B的实例而是A的实例 , 这样传入B/C/....都可以调用func ,不需要再改X的代码了：

#include <iostream>

class Person {

public:

    virtual void Strength() = 0;

};

class Adult : public Person {

public:

    virtual void Strength()

    {

        std::cout << "Adult have big Strength!" << std::endl;

    }

};

class Child : public Person {

public:

    virtual void Strength()

    {

        std::cout << "Child have Small Strength!" << std::endl;

    }

};

class CALL {

public :

    Person \*ptr;

    void testfunc(Person \*input) {

        ptr = input;

        ptr->Strength();

    }

};

int main() {

    CALL call;

    Child X;

    call.testfunc(&X);

    Adult Y;

    call.testfunc(&Y);

    return 0;

}

李博远：什么是依赖倒置原则？

程序要依赖于抽象接口，不要依赖于具体实现。简单的说就是要求对抽象进行编程，不要对实现进行编程，这样就降低了客户与实现模块间的耦合。

核心思想：高层模块不应该依赖底层模块，二者都该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象；

说明：高层模块就是调用端，低层模块就是具体实现类。抽象就是指接口或抽象类。细节就是实现类。

通俗来讲： 依赖倒置原则的本质就是通过抽象（接口或抽象类）使个各类或模块的实现彼此独立，互不影响，实现模块间的松耦合。

问题描述： 类A直接依赖类B，假如要将类A改为依赖类C，则必须通过修改类A的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；假如修改类A，会给程序带来不必要的风险。

解决方案： 将类A修改为依赖接口interface，类B和类C各自实现接口interface，类A通过接口interface间接与类B或者类C发生联系，则会大大降低修改类A的几率。

好处：依赖倒置的好处在小型项目中很难体现出来。但在大中型项目中可以减少需求变化引起的工作量。使并行开发更友好。