Принцип единственности и ответственности

Формулировка: не должно быть больше одной причины для изменения класса

Что является причиной изменения логики работы класса?
Видимо, изменение отношений между классами, введение новых требований или отмена старых.

Если у объекта много ответственности, то и меняться он будет очень часто.

Таким образом, если класс имеет ***больше одной ответственности***, то это ведет к хрупкости дизайна и ошибкам в неожиданных местах при изменениях кода.

Задача: Валидация данных

Проблема

Валидация данных в проекте.

Например, проверка введенного адреса эл. почты, длины имени пользователя, сложности пароля и т.п.

```
class Product
{
 public:
     int price;

    bool IsValid()
    {
        return price > 0;
    }
};
```

Например, объект **Product** начал использовать **CustomerService**, который считает валидным продукт с ценой больше 100 тыс. рублей.

Что делать?

Понятно, что придется изменять наш объект продукта, например:

```
class Product
{
 public:
     int price;

     bool IsValid(bool isCustomerService)
    {
        if (isCustomerService == true)
            return price > 100000;

        return price > 0;
    }
};
```

Решение

Очевидно, что при дальнейшем использовании объекта **Product** логика валидации его данных будет изменяться и усложняться.

Видимо пора отдать ответственность за валидацию данных продукта другому объекту.

Причем надо сделать так, чтобы сам объект продукта не зависел от конкретной реализации его валидатора.

Получаем код:????

Предложить советующий код на C++.

Проблема

Нарушения принципа единственности ответственности – **God object.**
Этот объект знает и умеет делать все, что только можно.
Рассмотрим на примере класс `ImageHelper`.

```
class ImageHelper {
public:
static void Save(Image image){
// сохранение изображение в файловой системе }
static int DeleteDuplicates(){
// удалить из файловой системы все дублирующиеся изображения и вернуть количество удаленных}
static Image SetImageAsAccountPicture(Image image, Account account) {
// запрос к базе данных для сохранения ссылки на это изображение для пользователя}
static Image Resize(Image image, int height, int width){
                // изменение размеров изображения}
static Image InvertColors(Image image){
                // изменить цвета на изображении}
static byte* Download(Url imageUrl){
// загрузка битового массива с изображением с помощью HTTP запроса
        }
        // и т.п.
};
```

Каждая ответственность этого класса ведет к его потенциальному изменению.

Получается, что этот класс будет очень часто менять свое поведение, что затруднит его тестирование и тестирование компонентов, которые его используют.

Такой подход снизит работоспособность системы и повысит стоимость ее сопровождения.

Решение

Решением является разделить этот класс по принципу единственности ответственности: **один класс на одну ответственность.**

Предложить советующий код на C++.

Принцип открытости и закрытости

Формулировка: программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для изменения.

Проблема

Пример нарушения принципа открытости/закрытости – использование конкретных объектов без абстракций.

Предположим, что у нас есть объект **SmtpMailer**.

Для логирования своих действий он использует **Logger**, который записывает информацию в текстовые файлы.

Рассмотрим соответствующие классы.

```
class Logger
{
public:
void Log(string logText)
{// сохранить лог в файле}
};

class SmtpMailer
{
private:
Logger* logger;
public:
SmtpMailer()
{
 logger = new Logger();
}
void SendMessage(string message)
{
 // отправка сообщения
}
};
```

Такая конструкция вполне жизнеспособна до тех, пока мы не решим записывать лог **SmptMailer**'а в базу данных.
Для этого нужно создать класс, который будет записывать все «логи» не в текстовый файл, а в базу данных:
Например.

```
class DatabaseLogger
{
 public:
 void Log(string logText)
 { // сохранить лог в базе данных }
};
```

Теперь нужно изменить класс SmptMailer из-за изменившегося требования:

```
class SmtpMailer
{ private:
 DatabaseLogger *logger;
   public:
  SmtpMailer(){
 logger = new DatabaseLogger();
}
void SendMessage(string message){// отправка сообщения
}
};
```

Теперь нужно изменить класс **SmptMailer** из-за изменившегося требования:

```
 class SmtpMailer
{
private:
DatabaseLogger *logger;
public:
SmtpMailer()
{
 logger = new DatabaseLogger();
}
void SendMessage(string message)
{// отправка сообщения
}
};
```

Но, по принципу единственности ответственности не SmptMailer отвечает за логирование, почему изменения дошли и до него?

Потому что нарушен наш принцип открытости/закрытости. **SmptMailer** не закрыт для модификации. Пришлось его изменить, чтобы поменять способ хранения его логов.

Решение

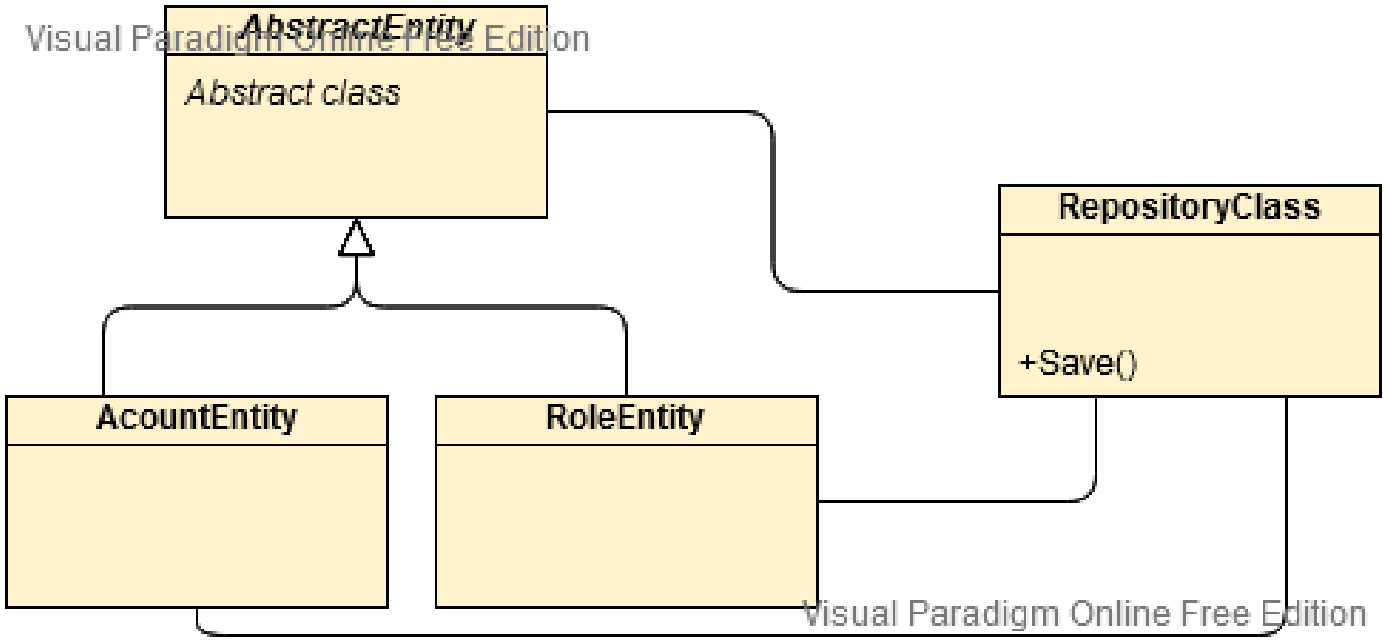
В данном случае защитить SmtpMailer поможет выделение абстракции.

Предложить советующий код на C++.

Проблема

Например, ас есть иерархия объектов с абстрактным родительским классом **AbstractEntity** и класс **Repository**, который использует абстракцию.

При этом вызывая метод **Save** у **Repository** мы строим логику в зависимости от **типа входного** параметра:



```
class AbstractEntity(абстрактный класс){
}
class AccountEntity : public AbstractEntity{
}
 class RoleEntity : public AbstractEntity{
}
class Repository
{
  void Save(AbstractEntity entity)
  { if (entity is AccountEntity){
        // специфические действия для AccountEntity}
      if (entity is RoleEntity){
                        // специфические действия для RoleEntity
              }
        }
}
```

Из предложенного псевдокода видно, что объект **Repository** придется менять каждый раз, когда мы добавляем в иерархию объектов с базовым классом **AbstractEntity** новых наследников или удаляем существующих.

Условные операторы будут множится в методе **Save** и тем самым усложнять его.

Решение

Чтобы решить данную проблему, необходимо логику сохранения конкретных классов из иерархии **AbstractEntity** вынести в конкретные классы **Repository**.

Для этого мы должны выделить интерфейс **IRepository** и создать хранилища **AccountRepository** и **RoleRepository.**
Предложить советующий код на C++.