Примеры кода SOLID

Принцип единственной ответственности (Single Responsibility Principle)

У класса должен быть только один мотив для изменения

```
class User {
private:
        string m_name;
        string m_lastName;
        string m_password;
        string m_email;

public:
        User(string name, string lastName, string password, string email) :
                m_name(name), m_lastName(lastName), m_password(password),
m_email(email) {}

        // Getters and Setters
};

class RepositoryUser {
public:
        void saveUser(User user);
        void deleteUser(User user);
        void getOne(User user);
};

class LoggerUser {
public:
        void logToExcel(User user);
        void logToTextFile(User user);
};

class PrinterUser {
        void printToConsole(User user);
        void printToPDF(User user);
};
```

Принцип открытости/закрытости (Open/closed Principle)

Расширяйте классы, но не изменяйте их первоначальный код.

```
class Animal {
        virtual void makeSound() = 0;
};

class Perrot : Animal {
        void makeSound() override;
};

class Cow : Animal {
        void makeSound() override;
};

class Cat : Animal {
        void makeSound() override;
};
```

Принцип подстановки Лисков (Liskov Substitution Principle)

Формулировка No1: если для каждого объекта o1 типа S существует объект o2 типа T, который для всех программ P определен в терминах T, то поведение P не изменится, если o2 заменить на o1 при условии, что S является подтипом T.

Формулировка No2: Функции, которые используют ссылки на базовые классы, должны иметь возможность использовать объекты производных классов, не зная об этом.

Другими словами: если нужно добавить какое-то ограничение в переопределенный метод, и этого ограничения не существует в базовой реализации, то, нарушается принцип подстановки Liskov.

```
class Person {
        void eat();
        void sleep();
};

class AndroidDeveloper : Person {
        void think();
        void writeCode();
};

class Teacher : Person {
        void teaching();
```

```
        void makeStudyProgram();
};

class Children : Person {
        void growUp();
        void play();
};
```

Принцип разделения интерфейса (Interface Segregation Principle)

Клиенты не должны зависеть от методов, которые они не используют.

```
class IFlyable {
        virtual void fly() = 0;
};

class IDrivable {
        virtual void drive() = 0;
};

class ISwimable {
        virtual void swim() = 0;
};

class Helicopter : IFlyable {
        void fly() override;
};

class Car : IDrivable {
        void drive() override;
};

class Ship : ISwimable {
        void swim() override;
};
```

Принцип Инверсий зависимостей (Dependency Inversion Principle)

Классы верхних уровней не должны зависеть от классов нижних уровней. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны

зависеть от абстракций.

```
// Абстракция
class INotification {
public:
        virtual void send(const string& message) = 0;
};

// Модуль нижнего уровня
class EmailNotificaton : public INotification {
        void send(const string& message) override;
};

// Модуль нижнего уровня
class SMSNotification : public INotification {
        void send(const string& message) override;
};

// Модуль верхнего уровня
class NotificationService : public INotification {
private:
        INotification* notification;
public:
        void sendNotification(const string& message) {
                notification->send(message);
        }
};
```