Лабораторная работа № 2
Разработка программ с использованием принципа подстановки Лисков (LSP), принципа разделения интерфейсов (ISP) и принципа инверсии (DIP)

Цель работы: получить навыки разработки программ с использованием принципа подстановки Лисков (LSP), принципа разделения интерфейсов (ISP) и принципа инверсии (DIP).

1. Теоретические сведения

Принцип подстановки Лисков

Принцип подстановки Лисков - принцип организации подтипов в объектно-ориентированном программировании. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 г., выглядело следующим образом. Если для каждого объекта *O1* типа *S* существует объект *O2* типа *T*, такой, что для любой программы *P*, определенной в терминах *T*, поведение *P* не изменяется при замене *O2* на *O1*, то *S* является подтипом *T*.

То есть, иными словами, класс S может считаться подклассом *T*, если замена объектов *T* на объекты *S* не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так: функции, использующие указатели или ссылки на базовые классы, должны иметь возможность использовать объекты классов-наследников, не зная об этом.

Если код, оперирующий ссылками на базовые классы, должен знать обо всех его наследниках и изменяться с появлением каждого нового наследника, то этот код не отвечает принципу подстановки Лисков, а значит, не отвечает и принципу открытости-закрытости.

Принцип подстановки Лисков реализуется за счет абстрагирования и выделения общего функционала в базовый класс.

Принцип разделения интерфейсов

Принцип разделения интерфейсов относится к тем случаям, когда классы имеют избыточный, т. е. слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы.

Принцип разделения интерфейсов можно сформулировать так: клиенты не должны вынужденно зависеть от методов, которыми не пользуются.

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге мы приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

В этом случае интерфейс класса разделяется на отдельные части, которые

составляют раздельные интерфейсы. Затем эти интерфейсы независимо друг от друга могут применяться и изменяться. В итоге применение принципа разделения интерфейсов делает систему слабосвязанной, и тем самым ее легче модифицировать и обновлять.

Принцип инверсии зависимостей

Данный принцип гласит, что, во-первых, классы высокого уровня не должны зависеть от низкоуровневых классов. При этом оба должны зависеть от абстракций. Во-вторых, абстракции не должны зависеть от деталей, но детали должны зависеть от абстракций. Классы высокого уровня реализуют бизнес- правила или логику в системе (приложении). Низкоуровневые классы занимаются более подробными операциями, другими словами, они могут заниматься записью информации в базу данных или передачей сообщений в операционную систему или службы и т. п.

Говорят, что высокоуровневый класс, который имеет зависимость от классов низкого уровня или какого-либо другого класса и много знает о других классах, с которыми он взаимодействует, тесно связан. Когда класс явно знает о дизайне и реализации другого класса, возникает риск того, что изменения в одном классе нарушат другой класс.

Поэтому необходимо держать эти высокоуровневые и низкоуровневые классы слабо связанными, насколько это возможно. Чтобы обеспечить это, необходимо сделать их зависимыми от абстракций, а не друг от друга.

2. Задания к лабораторной работе

Для заданного варианта задания (*предметной области*) разработать UML-диаграмму классов и диаграмму последовательности.

Разработать программу решения задания в виде консольного приложения (C++, C#) с использованием принципа подстановки Лисков (LSP), принципа разделения интерфейсов (ISP) и принципа инверсии (DIP). Допускается вводить дополнительные понятия предметной области. Наделите классы атрибутами и функциональностью по вашему усмотрению. В программе предусмотрите тестирование функциональности созданных объектов классов.

Отчет по лабораторной работе – файл формата pdf. Формат имени файла отчета: <НомерГруппы >_<ФамилияСтудента>.pdf. В отчет включить построенные диаграммы и исходный код программы (при необходимости и заголовочные файлы). Формат отчета см. в Приложении. Отчет загрузить в LMS.

При защите лабораторной работы: уметь объяснить логику и детали работы программы; реализацию принципов из раздела 1 на примере разработанной программы.

Варианты заданий

1. Студент, преподаватель, заведующий кафедрой, персона.
2. Организация, страховая компания, нефтегазовая компания, завод.
3. Журнал, книга, печатное издание, учебник.
4. Тест, экзамен, выпускной экзамен, испытание.
5. Строительное сооружение, театр, производственный корпус, гостиница.
6. Игрушка, телевизор, товар, молоко.
7. Квитанция, накладная, документ, счёт.
8. Автомобиль, поезд, самолёт, транспортное средство.
9. Директор, инженер, работник, менеджер.
10. Корабль, пароход, парусник, корвет.
11. Двигатель, бензиновый двигатель, дизельный двигатель, реактивный двигатель.
12. Деталь, электронный прибор, механизм, изделие.
13. Выпускник вуза, бакалавр, магистр, инженер.

Распределение вариантов заданий

№ по списку группы	№ варианта	№ по списку группы	№ варианта	№ по списку группы	№ варианта
1	1	11	11	21	8
2	2	12	12	22	9
3	3	13	13	23	10
4	4	14	1	24	11
5	5	15	2	25	12
6	6	16	3	26	13
7	7	17	4	27	1
8	8	18	5	28	2
9	9	19	6	29	3
10	10	20	7	30	4

Технологии конструирования программного обеспечения

Отчет по лабораторной работе № 00

Группа: 000-000 **Студент**: Иванов Иван Иванович

Задание на лабораторную работу

<Формулировка задания согласно варианту
……
……..>

Диаграмма классов

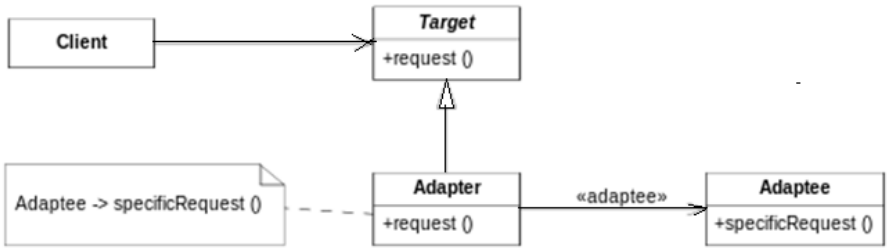
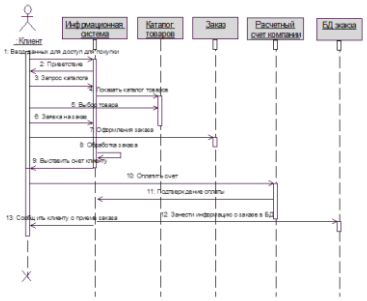


Диаграмма последовательности



Исходный код программы

```
#include "pch.h"
#include "CppUnitTest.h"
#include "../Add/Add.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace UnitTest1
{
	TEST_CLASS(UnitTest1)
	{
	public:

		TEST_METHOD(TestMethod1)
		{
			Assert::AreEqual(2, add(1, 1));
		}
```

……