Лабораторная работа №2.
Наследование

На этом семинаре мы не ограничимся наследованием, а рассмотрим более широкий круг вопросов:

- наследование классов и полиморфизм;
- отношения между классами и диаграммы классов на языке UML;
- введение в использование паттернов проектирования; технология проектирования программ с учетом будущих изменений.

Наследование классов

Классы в объектно-ориентированных программах используются для моделирования концепций реального и программного мира. Концепции или сущности предметной области находятся в различных взаимоотношениях. Одно из таких взаимоотношений — *отношение наследования* (именуемое также отношением *родитель–потомок* или отношением *обобщение–специализация*).

Например, если в условии задачи 1.2 рассматривать не только треугольники, но и четырехугольники, класс `Tetragon` имел бы общие черты с классом `Triangle`, поскольку и треугольники, и четырехугольники являются частным (специальным, конкретным) случаем более общего понятия «многоугольник». Поэтому было бы логичным создать класс `Polygon`, содержащий элементы, общие для классов `Triangle` и `Tetragon`, а последние два класса объявить «наследниками» базового (родительского) класса `Polygon`. Язык C++ позволяет легко это сделать:

```
class Polygon {
// . . .
};
class Triangle : public Polygon {
public: Show();
};
class Tetragon : public Polygon {
public: Show();
};
```

В этом примере производные классы `Triangle` и `Tetragon` наследуют все элементы базового класса `Polygon`, но каждый из них имеет свой собственный метод `Show`. Иногда отношение наследования называют отношением «*is a*», что можно перевести как «представляет собой». В данном примере `Triangle` *is a* `Polygon`, в такой же мере и

`Tetragon` *is a* `Polygon`.

Общий синтаксис создания производного класса при *простом наследовании*:

```
class имя : ключ_доступа имя_базового_класса {
    // тело класса
};
```

В случае *множественного наследования* после двоеточия перечисляются через запятую все базовые классы со своими модификаторами доступа.

Производный класс, в свою очередь, сам может служить базовым классом. Такой набор связанных классов называется *иерархией классов*. Иерархия чаще всего является деревом, но может иметь и более общую структуру графа.

1. В примерах предыдущего семинара доступ к элементам класса регулировался с помощью двух модификаторов: `private` — закрытая часть класса, `public` — открытая часть класса. Для базовых классов возможно использование еще одного модификатора — `protected`, который определяет *защищенную* часть класса. Ее элементы являются доступными для любого производного класса, но в то же время они недоступны вне классов данной иерархии.

Кроме этого, доступность в производном классе регулируется *ключом доступа к базовому классу*, указываемому в объявлении производного класса. Этот ключ определяет *вид наследования*: открытое (`public`), защищенное (`protected`) или закрытое (`private`).

Открытое наследование сохраняет статус доступа всех элементов базового класса, *защищенное* _ понижает статус доступа `public` элементов базового класса до protected, и наконец, *закрытое* — понижает статусы доступа public и protected элементов базового класса до `private`. Заметим, что в C++ отношение между классами «*is a*» имеет место только при открытом наследовании.

Замещение функций базового класса

Иногда в производном классе требуется несколько иная реализация метода, унаследованного из базового класса. Язык позволяет это сделать. *Замещение* (*переопределение*) метода производится путем объявления в производном классе метода с таким же именем. Если понадобится все-таки вызвать из потомка метод предка, используется операция доступа к области видимости ::, например:

```
#include <iostream>
using namespace std;
class Base {
public:
    void Display() { cout << "Hello, world!  "; }
};
class Derived : public Base {
public:
    void Display() {
        Base::Display();              // Вызов метода базового класса
```

```
        cout << "How are you?  ";
    }
};
class SubDerived : public Derived {
public:
    void Display() {
        Derived::Display();          // Вызов метода базового класса
        cout << "Bye!" << endl;
    }
};
int main() {
    SubDerived sd;
    sd.Display();    // Результат: Hello, world!  How are you?  Bye! }
```

Конструкторы и деструкторы в производном классе

Конструкторы и деструкторы из базового класса не наследуются, поэтому при создании производного класса встает вопрос, нужны ли эти методы и как они должны быть реализованы. Решая эти вопросы, учитывайте следующее.

- Если в базовом классе вообще нет конструктора или есть конструктор по умолчанию, производному классу конструктор нужен, *только* если требуется инициализировать поля, введенные в этом классе.
- Если вы не определили ни одного конструктора, компилятор самостоятельно создаст конструктор по умолчанию, из которого будет вызван конструктор по умолчанию базового класса.
- Если в базовом классе есть конструктор с аргументами, то производный класс должен содержать конструктор со списком аргументов, включающим значения для передачи конструктору базового класса; конструктор базового класса вызывается в списке инициализации.

Необходимость **в деструкторе** для производного класса определяется тем, нужно ли освобождать какие-либо ресурсы, выделенные в конструкторе. Если такой необходимости нет, можно доверить компилятору создать деструктор по умолчанию. В нем обеспечивается вызов деструктора базового класса.

На этапе выполнения программы при создании объекта производного класса сначала вызываются конструкторы базовых классов, начиная с самого верхнего уровня, затем конструкторы объектов-элементов класса и в последнюю очередь — конструктор класса. При уничтожении объекта (например, когда покидается область его видимости) деструкторы вызываются в порядке, обратном вызову конструкторов.

Устранение неоднозначности при множественном наследовании

Предположим, что от базового класса `A`, имеющего некоторый элемент `x`, наследуются два класса, `B` и `C`, а класс `D` является производным от этих двух классов (множественное наследование). Если попытаться обратиться к элементу `x` из методов класса D, компилятор воспримет выражение `D::x` как неоднозначное[1] и прекратит работу. Для решения этой проблемы в C++ предусмотрен механизм, благодаря которому в класс D будет включена только одна копия класса `A`. Достигается это добавлением спецификатора `virtual` перед модификаторами доступа к `A` в объявлениях классов `B` и `C`:

```
class A {
public:    A(int _x = 0) { x = _x; }
protected: int x;
};
class B : virtual public A {            // Виртуальное наследование
public:    void AddB(int y) { x += y; }
};
class C : virtual public A {            // Виртуальное наследование
public:    void AddC(int y) { x += y; }
};
class D : public B, public C {   // Здесь будет только одна копия
полей класса A
public:    void ShowX() { cout << "x = " << x << endl; }
};
int main() {
    D d;
    d.ShowX();                  // Вывод: x = 0
    d.AddB(10);  d.ShowX();     // Вывод: x = 10
    d.AddC(5);   d.ShowX();     // Вывод: x = 15
}
```

Доступ к объектам иерархии

Эффективнее всего работать с объектами одной иерархии через указатель на базовый класс. При открытом (`public`) наследовании такому указателю можно присваивать адрес объекта как базового класса, так и любого производного класса. Такая возможность представляется вполне логичной: в качестве представителя более общего класса используется объект специализированного класса.

Однако если не принять специальных мер, то при работе с таким указателем независимо от того, какой адрес ему присвоен, оказываются доступными *только элементы базового класса*. Например, рассмотрим следующий код:

```
class Base {
public: void Modify();
```

[1] Поскольку неясно, какой элемент класса A имеется в виду: унаследованный через класс B или унаследованный через класс C.

```
};
class Derived : public Base {
public: void Modify();
};
int main() {
    int mode;
    Base* pB;
    cin >> mode;
    if (mode == 1)  pB = new Base;
    else            pB = new Derived;
    pB->Modify();                          // на что указывает pB?
}
```

Так как компилятор не может предсказать, какой выбор будет сделан на этапе выполнения, он выбирает метод по типу указателя `pB`, то есть `Base::Modify`. Такая стратегия называется *ранним (статическим) связыванием*. Поэтому, если в производном классе замещен некоторый метод базового класса (например, `Derived::Modify`), он оказывается недоступным (листинг 2.1).

Листинг 2.1. Раннее связывание

```
#include <iostream>
using namespace std;
class Base {
public:
    Base(int _x = 10) { x = _x; }
    void ShowX() { cout << "x = " << x << "; "; }
    void ModifyX() { x *= 2; }
protected:
    int x;
};
class Derived : public Base {
public:
    void ModifyX() { x /= 2; }
};
int main() {
    Base b;    Derived d;
    b.ShowX();  d.ShowX();                 // Вывод: x = 10; x = 10;
    Base* pB;
    pB = &b;    pB->ModifyX();    pB->ShowX();     // Вывод: x = 20;
    pB = &d;    pB->ModifyX();    pB->ShowX();     // Вывод: x = 20; }
```

Как видите, второй вызов метода `ModifyX` происходит также из базового класса.

Виртуальные методы

Проблема доступа к методам, переопределенным в производных классах, через указатель на базовый класс решается в C++ посредством использования *виртуальных методов*. Чтобы сделать некоторый метод виртуальным, надо в базовом классе предварить его заголовок спецификатором

`virtual`. После этого он будет восприниматься как виртуальный во всех производных классах иерархии.

По отношению к виртуальным методам компилятор применяет стратегию *позднего (динамического) связывания*. Это означает, что на этапе компиляции он не определяет, какой из методов должен быть вызван, а передает ответственность программе, принимающей решение на этапе выполнения, когда уже точно известно, каков тип объекта, адресуемого через указатель. Все сказанное относится также к вызову методов *по ссылке* на базовый класс.

Для реализации динамического связывания компилятор создает *таблицу виртуальных методов* (*vtbl*), а к каждому объекту добавляет скрытое поле ссылки (*vptr*) на таблицу *vtbl*. Это несколько снижает эффективность программы, поэтому рекомендуется делать виртуальными только те методы, которые переопределяются в производных классах.

Чтобы увидеть динамическое связывание в действии, добавьте в листинг 2.1 спецификатор `virtual` перед заголовком метода `ModifyX` в базовом классе:

```
virtual void ModifyX() { x *= 2; }
```

Теперь второй вызов метода `ModifyX` происходит из производного класса, и программа должна вывести

```
x = 10; x = 10; x = 20; x = 5;
```

Виртуальный механизм работает только при использовании указателей и ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется *полиморфным*.

Если базовый класс содержит хотя бы один виртуальный метод, рекомендуется всегда снабжать этот класс *виртуальным деструктором*, даже если он ничего не делает. Наличие виртуального деструктора предотвратит некорректное удаление объектов производного класса, адресуемых через указатель на базовый класс, так как иначе деструктор производного класса вызван не будет[2].

Абстрактные классы. Чисто виртуальные методы

Иногда, разрабатывая иерархию классов, можно получить базовый класс, для которого создание объекта как бы теряет смысл. Например, иерархия классов геометрических фигур может произрастать из базового класса `Shape`, а в качестве производных будут выступать классы `Polygon`, `Ellipse` и т.д. Одним из методов базового класса будет, видимо, функция `Show` — показать объект. Но как можно показывать объект, не имеющий конкретной формы?

Классы, для которых нет смысла создавать объекты, объявляют как *абстрактные*. Абстрактный класс — это класс, содержащий хотя бы один *чисто виртуальный метод* (метод, объявленный в классе, но не имеющий

[2] В небольшой учебной программе это может пройти без последствий. В нетривиальных программах возможно появление труднодиагностируемых ошибок.

конкретной реализации). Синтаксис объявления такого метода дополняется конструкцией `= 0`, например:

```
virtual void Show() = 0;
```

Компилятор не допустит создания объекта для абстрактного класса, если вы по забывчивости попытаетесь это сделать. Если в производном классе хотя бы одна чисто виртуальная функция базового класса останется без конкретной реализации, то производный класс также будет абстрактным классом, для которого запрещено создавать объекты.

Отношения между классами. Диаграммы классов на языке UML

Полиморфизм и наследование делают язык C++ чрезвычайно мощным инструментом для реализации объектно-ориентированной технологии проектирования.

Для эффективного восприятия новых технологических идей полезно знать хотя бы основы ставшего уже стандартом *унифицированного языка моделирования UML* (Unified Modeling Language).

Язык UML является визуальным средством представления *моделей программ*, то есть их графического представления в виде различных диаграмм, отражающих связи между объектами в программном коде.

Одной из основных диаграмм языка UML является *диаграмма классов*. Она описывает классы и отношения, существующие между ними. Такая диаграмма чрезвычайно удобна для сопоставления различных вариантов проектных решений. Класс изображается на диаграмме классов UML в виде прямоугольника, состоящего из трех частей. Имя класса указывается в верхней части. В средней части приводится список *атрибутов* с указанием типов (атрибут класса в UML соответствует термину *поле класса* в C++).

В нижней части записывается список *операций* (методов класса), возможно, с указанием списка типов аргументов и типа возвращаемого значения. Имя *абстрактного класса*, так же как и имена *абстрактных операций* (чисто виртуальных методов), выделяются курсивом. Перед именем поля или метода может указываться спецификатор доступа с помощью одного из трех символов: + для `public`, – для `private`, # для `protected`. Для статических элементов класса после спецификатора доступа записывается символ $.

Во второй и третьей частях могут указываться не все элементы класса, а только те, которые представляют интерес на данном уровне абстракции. Если обе эти части пусты, они могут быть опущены. Например, варианты изображения класса `Triangle` на различных этапах анализа решения задачи 1.2 показаны на рис. 2.1.

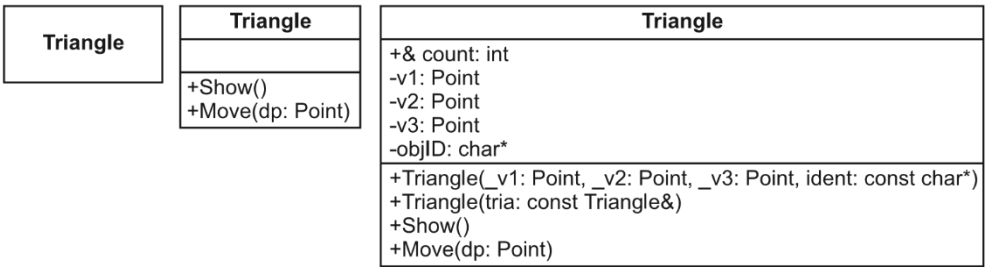

Рис.2.1. Варианты изображения класса Triangle на диаграмме классов UML

Большинство объектно-ориентированных языков поддерживают следующие виды отношений между классами: ассоциация, наследование, агрегация и зависимость. Рассмотрим, что они означают и как изображаются на диаграмме классов.

Ассоциация

Если два класса концептуально взаимодействуют друг с другом, такое взаимодействие называется *ассоциацией*. Например, желая смоделировать торговую точку, мы вводим две абстракции: товары (класс `Product`) и продажи (класс `Sale`). Объект класса `Sale` — это некоторая сделка, в которой продано от 1 до `n` объектов класса `Product`. На рис. 2.2 ассоциация между этими двумя классами показана соединяющей их линией. Над линией рядом с обозначением класса может быть указана так называемая *кратность* (*multiplicity*), указывающая, сколько объектов данного класса может быть связано с одним объектом другого класса. Символ звездочки * обозначает «произвольное количество».

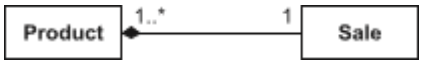

Рис. 2.2. Отношения ассоциации

Ассоциация представляет наиболее абстрактную семантическую связь между двумя классами, выявляемую на ранней стадии анализа. В дальнейшем она, как правило, конкретизируется, переходя в одно из рассматриваемых далее отношений.

Наследование

Отношение *обобщения* (наследования, «*is a*») между классами показывает, что *подкласс* (производный класс) разделяет атрибуты и операции, определенные в одном или нескольких *суперклассах* (базовых классах). На диаграмме классов отношение наследования показывают линией со стрелкой в виде незакрашенного треугольника, которая указывает на базовый класс. Допускается объединять несколько стрелок в одну (см. рис. 2.3), с тем чтобы разгрузить диаграмму.

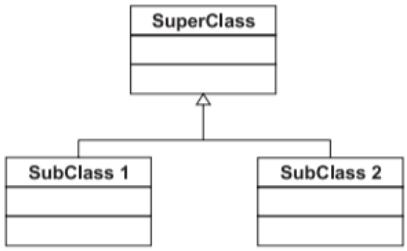

Рис. 2.3 Отношения наследования

Отношение *агрегации* показывает, что один класс содержит в качестве составной части объекты другого класса. Иными словами, это отношение `целое_часть`, или отношение «*has a*», между двумя классами. На диаграмме такая связь обозначается линией со стрелкой в виде незакрашенного ромба, которая указывает на `целое`. На рис. 2.4 показан пример так называемой *нестрогой* агрегации (или просто агрегации). Действительно, конкретный объект класса `СпортЗал` может содержать не все компоненты (спортивные снаряды), присутствующие на схеме.

Как распознать агрегацию в программном коде? Для реализации нестрогой агрегации `часть` включается в `целое` *по ссылке*: на языке C++ это обычно указатель на соответствующий класс. Если этот указатель равен нулю, то компонент отсутствует. В зависимости от решаемой задачи такой компонент может появляться и исчезать динамически в течение жизни объекта `целое`.

Рис. 2.4. Отношения агрегации

Строгая агрегация имеет специальное название — *композиция*. Она означает, что компонент не может исчезнуть, пока объект `целое` существует. Пример такого отношения мы уже встречали при решении задачи 1.2, в которой класс `Triangle` содержал в себе три объекта класса `Point`. На диаграмме отношение композиции обозначается линией со стрелкой в виде закрашенного ромба (рис. 2.5).

Проще всего композицию реализовать включением объектов-компонентов *по значению*. В то же время возможна реализация и включением по ссылке, но тогда времена жизни компонентов и объекта `целое` должны совпадать.

Зависимость

Отношение *зависимости* (*использования*) показывает, что один класс (`Client`) пользуется услугами другого класса (`Supplier`), например:

- метод класса `Client` использует значения некоторых полей класса `Supplier`;
- метод класса `Client` вызывает некоторый метод класса `Supplier`;
- метод класса `Client` имеет сигнатуру, в которой упоминается класс `Supplier`. На диаграмме такое отношение изображается пунктирной линией со стрелкой, указывающей на класс `Supplier`. Например, отношения зависимости существует между классами `Triangle` и Point из задачи 1.2 (рис. 2.6).

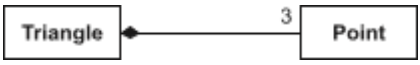

Рис. 2.5. Отношение композиции

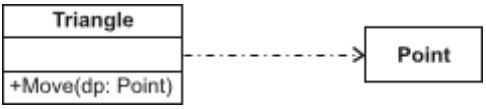

Рис. 2.6. Отношение зависимости (использования)

Паттерны проектирования

Одним из важнейших достижений в области ООП является методология паттернов проектирования, иногда называемых шаблонами проектирования[3]. Впервые она была представлена в книге [6].

Паттерн — это описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте. Паттерны выявляются по мере накопления опыта разработок, когда программист использует одну и ту же схему организации и взаимодействия объектов в разных контекстах.

Поначалу паттерны можно было рассматривать как особенно изящный и хорошо продуманный способ решения определенного класса задач. Однако практика показала, что применение паттернов наряду с объектной технологией позволяет вывести процесс проектирования из области интуиции и представить систему на более высоком уровне абстракции — на уровне схем взаимодействия объектов. Любая абстракция исключает частные,

[3] Слово «шаблон» в отношении термина «pattern» не вполне удачно, поскольку вызывает путаницу с термином «template». Поэтому во многих книгах издательства «Питер» используется термин «паттерн». — *Примеч. ред.*

второстепенные детали, и обычно при этом удается выделить в проблеме ее переменные и постоянные составляющие.

Основные трудности при разработке элегантной и удобной в сопровождении архитектуры вызывает идентификация так называемого «вектора изменений». Выявление важнейшего фактора перемен в проектируемой системе дает опорную точку для построения дальнейшей архитектуры.

Итак, паттерны предназначены прежде всего для *инкапсуляции изменений*. Если рассматривать их с этой точки зрения, то некоторые паттерны нам уже встречались. Например, наследование тоже может рассматриваться как паттерн, пусть даже реализованный на уровне компилятора. Оно выражает различия в поведении (переменная составляющая) объектов, обладающих одинаковым интерфейсом (постоянная составляющая). Впрочем, обычно возможности, напрямую поддерживаемые языком программирования, не принято относить к паттернам.

Один из фундаментальных принципов, провозглашенный в книге [6], гласит: «Отдавайте предпочтение композиции объектов перед наследованием классов». И действительно, иногда композиция радикально упрощает архитектуру, в которой кажется уместным наследование, а полученная архитектура становится более

гибкой.

Краткие сведения о паттернах в данном разделе приведены по книге [6]. Разумеется, для более глубокого изучения этой темы нужно обратиться к специальной литературе, например, [5], [6] и [34].

Все паттерны в соответствии с их назначением разделены на три группы: порождающие, структурные и паттерны поведения. *Порождающие паттерны* описывают способы создания объектов, *структурные* — схемы организации классов и объектов, а *паттерны поведения* определяют типичные схемы взаимодействия классов и объектов. В каждой группе можно, в свою очередь, выделить два уровня, определяющих, применяется паттерн к классам или объектам. В табл. 2.1 приведена классификация паттернов.

Описание каждого паттерна в [6] выполнено по одной и той же схеме, включающей его назначение, область применения, структурную схему (в форме диаграммы классов UML), описание входящих в него объектов и их взаимодействий, а также пример реализации. Такой уровень документирования делает возможным использование паттерна в различных конкретных случаях, возникающих при проектировании программных систем.

Таблица 2.1. Классификация паттернов проектирования

Назначение/ Уровень	Порождающие паттерны	Структурные паттерны	Паттерны поведения
Класс	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

В качестве примера мы приведем здесь описание только паттерна `Стратегия`.

Паттерн Стратегия (`Strategy`)

Этот паттерн инкапсулирует семейство алгоритмов, делая их взаимозаменяемыми. Применять его целесообразно в следующих случаях:

- имеются родственные классы, различающиеся только поведением. Стратегия позволяет гибко конфигурировать класс, задавая одно из возможных поведений;
- требуется иметь несколько разных вариантов алгоритма. Варианты алгоритмов могут быть реализованы в виде иерархии классов, что позволяет вычленить общую для всех классов функциональность. Инкапсулированные алгоритмы можно затем применять в разных контекстах;
- в классе определено много вариантов поведения, что представлено разветвленными условными операторами. В этом случае проще перенести код из ветвей в отдельные классы стратегий.

Диаграмма классов паттерна приведена на рис. 2.7.

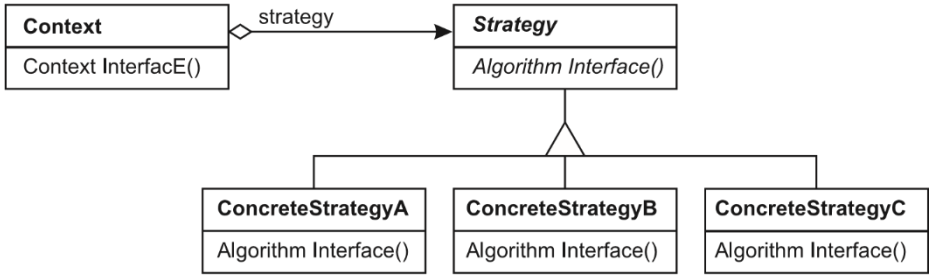

Рис. 2.7. Структура паттерна Стратегия

Паттерн состоит из следующих классов.

- **Strategy** (стратегия) — объявляет общий для всех поддерживаемых алгоритмов интерфейс. Класс `Context` пользуется этим интерфейсом для вызова конкретного алгоритма, определенного в классе `ConcreteStrategy`.

- **ConcreteStrategy** (конкретная стратегия) — наследник класса `Strategy`. Реализует алгоритм, использующий интерфейс, объявленный в классе `Strategy`.

- **Context** (контекст) — конфигурируется объектом класса `ConcreteStrategy`. Хранит ссылку на объект класса `Strategy` и может определять интерфейс, который позволяет объекту `Strategy` получить доступ к данным контекста.

Пример реализации. Термин «стратегия» означает лишь то, что у проблемы имеется несколько решений. Допустим, что вы забыли, как зовут встреченного вами знакомого. Из неловкого положения можно выйти несколькими способами, как показано в листинге 2.2 (идея примера позаимствована из [38]).

Листинг 2.2. Использование паттерна Strategy

```
#include <iostream>
using namespace std;
class NameStrategy {
public: virtual void greet() = 0;
};
class SayHi : public NameStrategy {
public: void greet() { cout << "Привет! Как дела?" << endl; }
};
class Ignore : public NameStrategy {
public: void greet() { cout << "(Сделать вид, что не заметил
человека)" << endl; }
};
class Admission : public NameStrategy {
public: void greet() { cout << "Простите, я забыл Ваше имя." << endl;
}
};
class Context { // ------------Контекст управляет выбором стратегии:
NameStrategy& strategy;
public:
    Context(NameStrategy& strat) : strategy(strat) {}
    void greet() { strategy.greet(); }
};
int main() {
    SayHi     sayhi;
    Ignore    ignore;
    Admission admission;
```

```
    Context c1(sayhi), c2(ignore), c3(admission);
    c1.greet(); c2.greet(); c3.greet();
}
```

Завершая краткое введение в паттерны проектирования, заметим, что хороший критерий профессионализма — выделение паттернов самостоятельно, на основании собственного опыта. При решении следующих задач мы покажем, как это делается.

Проектирование программы с учетом будущих изменений

Одним из показателей качества программной системы является ее способность к модификациям в связи с появлением новых требований заказчика. На предыдущем семинаре уже отмечалось, что большое значение имеет правильная декомпозиция системы на компоненты (модули, классы, функции).

Однако тактические решения локальных проблем также влияют на то, сколь сложным окажется добавление новой функциональности к разработанному ранее проекту. Сейчас мы рассмотрим некоторые концепции поиска таких решений.

Одной из причин негибкости процедурно-ориентированных систем является частое использование в них управляющих конструкций на базе оператора `switch`. Рассмотрим типичную ситуацию: некоторая функция `SomeFunc` в зависимости от значения параметра `mode` вызывает одну из обрабатывающих процедур:

```
typedef enum { mode1, mode2, mode3 } ModeType;
void SomeFunc (ModeType mode) {
     switch (mode) {
         case mode1:  DoSomething1();   break;
         case mode2:  DoSomething2();   break;
         case mode3:  DoSomething3();   break;
    }
}
```

Допустим, что перечень значений перечисляемого типа `ModeType` согласован с заказчиком и утвержден техническим заданием. Вся система, таким образом, разрабатывается с использованием утвержденного перечня. Однако через некоторое время заказчик обнаруживает, что необходимо предусмотреть еще одно значение параметра `mode`. Хорошо, если он еще платежеспособен и сможет заказать модификацию системы. Но для разработчиков наступят черные дни, когда придется искать по всем модулям переключатели `switch` и вносить нужные добавления. При этом потребуется полное повторное тестирование всех модулей, использующих эту функцию, так как могут появиться неожиданные побочные эффекты.

Попытаемся найти более удачное решение проблемы типа «переключатель». Одна из рекомендаций, приведенных в [6], звучит так: «Найдите то, что должно или может *измениться* в вашем дизайне, и

инкапсулируйте сущности, подверженные изменениям!» В нашем случае переменной сущностью являются вызываемые процедуры. Можем ли мы их инкапсулировать? Да — если реализуем эти процедуры как виртуальные методы полиморфных объектов!

Идея заключается в следующем: объект типа «переключатель» (класс `Switch`) должен иметь дело с некоторой абстрактной «процедурой вообще», а это можно реализовать, создав абстрактный класс `AbstractEntity` с чисто виртуальным методом `DoSomething`. Конкретные процедуры в виде одноименных замещенных методов будут содержаться в производных от `AbstractEntity` классах `Entity1`, `Entity2`, и т. д. Чтобы делать «осознанный» выбор, класс `Switch` должен быть осведомлен об адресах объектов `Entity1`, `Entity2`, и т. д. Это можно обеспечить, поместив список адресов в одно из его полей типа `std::vector<AbstractEntity*>`[4]. Описываемая идея поясняется на диаграмме классов (рис. 2.8).

Обратите внимание, что между классами `Switch` и `AbstractEntity` имеется отношение `композиции` (стрелка с закрашенным ромбиком), так как класс `Switch` содержит поле типа `std::vector<AbstractEntity*>`. В то же время класс `Switch` *использует* класс `AbstractEntity` (пунктирная стрелка), поскольку один из его методов возвращает значение типа `AbstractEntity*`.

Как же все это работает? Клиент обращается с запросом `SelectEntity` к объекту `Switch` (то есть вызывает одноименный метод). Неважно, как реализован метод `SelectEntity`, существенно то, что он возвращает значение типа `AbstractEntity*`, содержащее адрес объекта одного из подклассов класса `AbstractEntity`. После этого через полученный указатель клиент вызывает конкретную процедуру `DoSomething` одного из объектов `Entity1`, `Entity2`, ...

Чем примечательно предлагаемое решение? Необычайной легкостью модификации. Если возникла необходимость дополнить список переключаемых сущностей еще одним `k-м` экземпляром, то достаточно добавить производный класс `EntityK`, объявить объект этого класса и добавить адрес нового объекта в список
`vector<AbstractEntity*>`.

Очевидно, что предложенное на рис. 2.8 решение есть не что иное, как еще один *паттерн проектирования*, который мы назвали `Switch`. Пример его реализации приводится в решении задачи 2.1. Понятно, что скорее всего аналогичное решение уже неоднократно применялось программистами на практике, но ведь решение становится паттерном только после его описания на требуемом уровне абстракции!

---

[4] Класс `std::vector` содержится в стандартной библиотеке шаблонов STL, которая изучается на семинаре 15. Тем не менее мы будем пользоваться этим удобным инструментом работы с динамическими массивами уже сейчас.

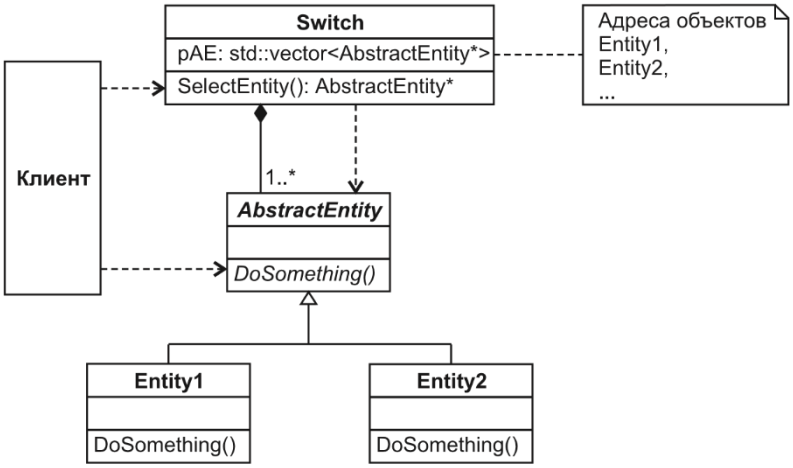

Рис. 2.8. Паттерн Switch (переключатель)

Задача 2.1. Функциональный калькулятор

Разработать программу, имитирующую работу функционального калькулятора, который позволяет выбрать с помощью меню какую либо из известных ему функций, затем предлагает ввести значение аргумента и, возможно, коэффициентов и после ввода выдает соответствующее значение функции.

В первой версии калькулятора «база знаний» содержит две функции:

- экспоненту $y = e^x$;
- линейную функцию $y = a\,x + b$.

Решение задачи начнем с выявления понятий (классов) и их взаимосвязей.

Интерфейс пользователя нашего калькулятора, как следует из его описания, должен обеспечить меню для выбора вида функции. У нас уже есть опыт создания меню для консольных приложений, в которых отсутствует оконная графика, — например, в задаче 10.2. Функция `menu` в той программе выводит список выбираемых пунктов с номерами и после ввода пользователем номера пункта возвращает это значение главной функции. Далее следует традиционный переключатель:

```
switch (Menu()) {
    case 1: Show(...);  break;
    case 2: Move(...);  break;
    ...
    case 5: // Конец работы
}
```

Но вряд ли стоит повторять такое решение. Ведь мы уже знаем, что подобная реализация может резко усложнить модификацию программы. Мы

также вооружены новым паттерном проектирования `Switch`, который дает красивое, легко модифицируемое решение. Попробуем применить этот паттерн на практике. Несложно увидеть, что роль класса `Switch` здесь будет играть класс `Menu`.

Что является изменяемой сущностью в нашей задаче? — Вид функции, для которой нужно выполнить вычисления (в более общем случае — вид некоторого объекта, для которого нужно выполнить некоторую операцию). Следовательно, требуется инкапсулировать эту изменяемую сущность так, чтобы класс `Menu` имел дело с некоторой абстрактной «функцией вообще» (с некоторым «объектом вообще»). Отсюда вывод: необходим абстрактный класс `Function`, обеспечивающий единый унифицированный интерфейс для всех его потомков, в данном случае — для классов `Exp` и `Line`. В результате мы приходим к диаграмме классов на рис. 2.9.

На диаграмме, естественно, показаны не все элементы классов, а только наиболее существенные для данного этапа анализа. Кроме метода `Calculate`, выполняющего роль метода `DoSomething` в паттерне `Switch`, появился метод `GetName`, извлекающий наименование функции. Этим методом будет пользоваться объект `Menu` для вывода перечня выбираемых функций.

Прежде чем привести код программы, скажем несколько слов о работе с классами `std::string` и `std::vector`. Более подробно они изучаются на семинаре 12.

Класс `string` позволяет создать строку $_s$ типа `string`, с которой очень удобно работать: ее можно инициализировать значением С-строки: `std::string s("С-строка");` или присвоить ей значение другой строки посредством операции присваивания =. Память, выделенная для объекта $_s$, автоматически освобождается, как только программа покидает область его видимости. Для использования класса необходимо подключить заголовочный файл `<string>`.

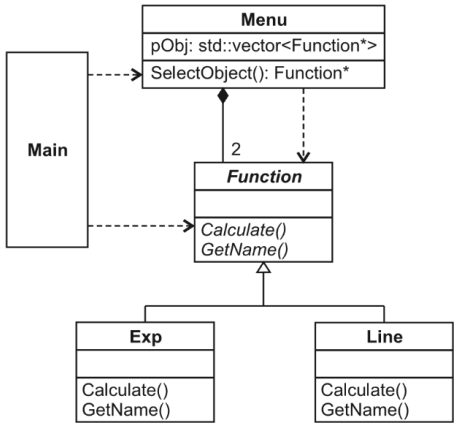

Рис.2.9. Диаграмма классов программы «Функциональный калькулятор»

Контейнер `vector` является шаблонным классом и позволяет создавать динамические массивы (термины «*вектор*» и «*динамический массив*» мы будем использовать как синонимы). Для использования класса необходимо подключить заголовочный файл `<vector>`. В классе есть конструктор по умолчанию, создающий вектор нулевой длины, а также конструктор с инициализацией создаваемого вектора обычным одномерным С-массивом. Можно добавить в конец имеющегося вектора новый элемент с помощью метода `push_back`. Доступ к любому элементу вектора можно выполнять по индексу, как и в обычном массиве. Текущее количество элементов в векторе можно определить методом `size`.

Поскольку `vector` — шаблонный класс, он позволяет создавать конкретные экземпляры классов для любого типа элементов, задаваемого в качестве аргумента в угловых скобках после имени класса. Поясним использование класса на примере, в котором показано использование векторов `v1, v2, v3`, предназначенных для хранения элементов типа `char`, `int` и `double` соответственно:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    int a[5] = { 5, 4, 3, 2, 1 };
    double b[] = { 1.1, 2.2, 3.3, 4.4 };
    vector<char> v1;
    v1.push_back('A');  v1.push_back('B');  v1.push_back('C');
    for (int i = 0; i < v1.size(); ++i) cout << v1[i] << ' '; cout <<
endl;
    vector<int> v2(a, a + 5);
    for (int i = 0; i < v2.size(); ++i) cout << v2[i] << ' '; cout <<
endl;
    vector<double> v3(b, b + sizeof(b) / sizeof(double));
    for (int i = 0; i < v3.size(); ++i) cout << v3[i] << ' '; cout <<
endl; }
```

Эта программа должна вывести на экран:

```
A B C
5 4 3 2 1
2.2 3.3 4.4
```

Обратите внимание на два способа инициализации вектора одномерным массивом. Первый (для вектора `v2`) используется, когда длина массива задана константой. Второй (для вектора `v3`) оказывается единственно возможным, когда длина массива определяется на стадии компиляции. В листинге 2.3 приведена реализация программы «Функциональный калькулятор».

Листинг 2.3. Функциональный калькулятор

```
//////////////////////////////////////////////////////Function.h
#ifndef FUNCTION_H
#define FUNCTION_H
#include <string>
class Function {
public:

    virtual ~Function() {}
    virtual const std::string& GetName() const = 0;             // 1
    virtual void Calculate() = 0;
protected:
    double x;                       // аргумент
};
#endif /* FUNCTION_H */
// ////////////////////////////////////////////////////// Exp.h
#include "Function.h"
class Exp : public Function { // -------- Класс для представления
функции y = e ^ x
public:
    Exp() : name("e ^ x") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;          // математическое обозначение функции
};
extern Exp f_exp;
// ////////////////////////////////////////////////////// Exp.cpp
#include <iostream>
#include <math.h>
#include "Exp.h"
using namespace std;
void Exp::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter x = ";
    cin >> x; cin.get();
    cout << "y = " << exp(x) << endl; cin.get();
}
Exp f_exp;    // Глобальный объект
// ////////////////////////////////////////////////////// Line.h
#include "Function.h"
class Line : public Function { // --- Класс для представления функции
y = a * x + b
    public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;   // математическое обозначение функции
    double a, b;
};
```

```
extern Line f_line;
// ////////////////////////////////////////////////////// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;
void Line::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = ";   cin >> a;
    cout << "Enter b = ";   cin >> b;
    cout << "Enter x = ";   cin >> x;
    cin.get();
    cout << "y = " << (a * x + b) << endl;
    cin.get();
}
Line f_line;    // Глобальный объект
// ////////////////////////////////////////////////////// Menu.h
#include <vector>
#include "Function.h"
class Menu {
public:
    Menu(std::vector<Function*>);
    Function* SelectObject() const;
private:

    int SelectItem(int) const;
    std::vector<Function*> pObj;
};
// ////////////////////////////////////////////////////// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Function*> _pObj) : pObj(_pObj) {
    pObj.push_back(0);                  // для выбора пункта 'Exit'
}
Function* Menu::SelectObject() const {
    int nItem = pObj.size();
    cout << "=====================================\n";
    cout << "Select one of the following function:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i+1 << ". ";  // номер пункта меню на 1 больше индекса
массива pObj
        if (pObj[i]) cout << pObj[i]->GetName() << endl;
        else  cout  << "Exit" << endl;
    }
    int item = SelectItem(nItem);
    return pObj[item - 1];
}
int Menu::SelectItem(int nItem) const {
    cout << "-------------------------------------\n";
    int item;
```

```
    while ( true ) {
        cin >> item;
        if ((item > 0) && (item <= nItem) && (cin.peek() == '\n')) {
            cin.get();   break;
        }

        else {
            cout << "Error (must be number from 1 to " << nItem <<
"):" << endl;
            cin.clear();
            while ( cin.get() != '\n' ) {};
        }
    }
    return item;
}
////////////////////////////////////////////////////// Main.cpp
#include <iostream>
#include "Exp.cpp"
#include "Line.cpp"
#include "Menu.cpp"
using namespace std;
Function* pObjs[] = { &f_exp, &f_line };
vector<Function*> funcList(pObjs, pObjs + sizeof(pObjs) /
sizeof(Function*));
int main() {
    Menu menu(funcList);
    while ( Function* pObj = menu.SelectObject() )
        pObj->Calculate();
    cout << "Bye!\n";
}
```

Несколько пояснений по тексту программы.

В абстрактном классе `Function` конструктор отсутствует. Класс содержит *виртуальный* деструктор (см. рекомендацию, приведенную в разделе «Виртуальные методы»). Существует рекомендация размещать в абстрактном классе *только методы*, а все поля объявлять в производных классах (то есть делать класс чисто интерфейсным). Этим гарантируется, что любые изменения и дополнения не затронут абстрактный класс. В данном случае мы все-таки описали поле `x` для значения аргумента, поскольку рассматриваем только функции с одним аргументом и при этом полагаем, что аргумент всегда имеет тип `double`. А вот значения коэффициентов содержатся в полях производных классов.

Обратите внимание на сигнатуру метода `GetName` в классе `Function` (*оператор 1*). Метод объявлен константным, так как он не должен изменять состояние полей класса. Кроме того, чтобы исключить вызов конструктора копирования, мы хотим вернуть значение по ссылке: `std::string&`. Однако

сигнатуру `virtual std::string& GetName` const не пропустит компилятор, так как возврат по ссылке `string&` противоречит модификатору `const`. Выход из этого противоречия — возврат по константной ссылке: `const std::string&`.

Конкретные методы `Calculate` в классах `Exp` и `Line` обеспечивают ввод исходных данных (аргумент и, по необходимости, значения коэффициентов) и вычисляют значение соответствующей функции. Отметим, что в модуле `Exp.cpp` объявлен глобальный объект `Exp f_exp`, а в модуле `Line.cpp` — глобальный объект `Line f_line`. Конструктор класса `Menu` обеспечивает инициализацию поля `pObj` вектором объектов типа `Function*`, передаваемым через его аргумент. После копирования аргумента в поле `pObj` (в инициализаторе конструктора) к вектору с помощью метода `pObj.push_back(0)` добавляется нулевой указатель, который используется для реализации пункта меню `Exit` (см. метод `Menu::SelectObject`).

В методе `Menu::SelectObject` количество пунктов меню определяется вызовом `pObj.size`. После вывода оператором `for` списка пунктов меню вызывается метод `SelectItem`, обеспечивающий защищенный от ошибок ввод пользователем номера выбираемого пункта. Заметим, что метод `SelectItem` объявлен в секции `private` класса `Menu`, так как он используется только внутри класса.

В главном модуле main.cpp объявлен глобальный массив `pObjs[]`, содержащий список адресов объектов `&f_exp, &f_line`. Этим массивом инициализируется вектор `funcList`.

Функция `main` лаконична: в ней объявлен объект `menu` класса `Menu`, которому передается вектор `funcList`, а затем идет цикл `while`, работа которого очевидна. Метод `menu.SelectObject` возвращает адрес конкретного объекта (`f_exp` или `f_line`), через который вызывается метод `Calculate` соответствующего класса. При возврате нулевого адреса цикл завершается, и программа завершается.

Необходимо также пояснить, по какому принципу размещаются *директивы* `using` в тексте модулей. Напомним, что благодаря директиве `using namespace std` становятся видимыми (известными компилятору) все имена из пространства имен `std` стандартной библиотеки C++. В нашей программе это, например, имена `cin, cout, string, vector`.

Есть и другие способы сделать видимым некоторое конкретное имя: с помощью *объявления* `using` (например, `using std::string` – действует до конца блока, в котором оно сделано) или с помощью квалификатора `std::` (например, `std::string`). Проще всего использовать директиву `using`, но при этом возникают две опасности: возможность конфликта имен из пространства `std` с вашими именами и, что более неприятно, возможность неоднозначной трактовки кода компилятором при беспорядочном употреблении директив

using. В этом вопросе мы солидарны с Гербом Саттером [25] и приводим ниже его рекомендации.

СОВЕТ

Правило №1. *Не используйте* директивы using, равно как и объявления using, *в заголовочных файлах*. Именно в этом случае возможно появление указанных выше проблем. Поэтому в заголовочных файлах используйте только квалификатор std:: перед конкретным именем.

Правило №2. *В файлах реализации* директивы и объявления using не должны располагаться *перед* директивами include. Иначе использование using может изменить семантику заголовочного файла из-за введения неожиданных имен.

Откомпилируйте и проверьте работу программы. Еще раз сравните код функций main в этой программе и в задаче 10.2, чтобы увидеть элегантность нового решения. А теперь представим, что от заказчика поступило предложение сделать наш калькулятор более «мощным», добавив в него новую функцию — параболу. Доработка сведется к добавлению в проект класса Parabola:

```
// ////////////////////////////////////////////////// Parabola.h
#include "Function.h"
// Класс для представления функции y = a * x^2 + b * x + c
class Parabola : public Function {
public:
    Parabola() : name("a * x^2 + b * x + c") {}
    const std::string& GetName() const { return name; }
    void Calculate();
protected:
    std::string name;  // мат. обозначение функции
    double a, b, c;
};
extern Parabola f_parabola;

// ////////////////////////////////////////////////// Parabola.cpp
#include <iostream>
#include "Parabola.h"
using namespace std;
void Parabola::Calculate() {
    cout << "Calculation for function y = " << name << endl;
    cout << "Enter a = ";   cin >> a;
    cout << "Enter b = ";   cin >> b;
    cout << "Enter c = ";   cin >> c;
    cout << "Enter x = ";   cin >> x;
    cin.get();
    cout << "y = " << (a * x * x + b * x + c) << endl;
    cin.get();
```

```
}
Parabola f_parabola;     // Глобальный объект
```

Кроме этого, в основном модуле `main.cpp` необходимо добавить директиву `#include "Parabola.h"`;, а в списке инициализации массива `pObjs` — адрес нового объекта `&f_parabola`. Все! Весь прежний код проекта остается без изменений. Новая версия калькулятора готова.

Задача 2.2. Продвинутый функциональный калькулятор

Наш заказчик никак не угомонится. Теперь ему пришла идея расширить набор операций, которые способен выполнять функциональный калькулятор: пусть он либо вычисляет значение заданной функции для некоторого аргумента, либо выполняет табуляцию функции в заданном интервале с заданным шагом.

Идея неплохая. Это позволит резко повысить спрос на рынке на наш калькулятор, поскольку калькуляторы других фирм не умеют это делать. За работу!

Нам нужно решить две проблемы. Во-первых, видимо, придется добавить в базовый класс Function новый чисто виртуальный метод Tabulation, а также заместить его во всех подклассах. То есть придется вносить изменения в код всех классов иерархии `Function`, а это плохо. Во-вторых, в продвинутом функциональном калькуляторе необходимо реализовать двухуровневое меню: на первом уровне выбирается вид функции, на втором уровне — вид операции. В предыдущей задаче мы справились с проблемой *легко модифицируемого кода* одноуровневого меню с помощью паттерна проектирования `Switch`. Неужели на втором уровне придется писать по старинке гирлянду `case'ов` в блоке оператора `switch`? Как мы знаем, это плохо для дальнейшей модификации программы.

Попробуем найти более удачное решение. А что, если попытаться развить идею одноуровневого переключателя (паттерн `Switch`) для случая двух уровней? Например, переключатель такого типа мог бы выбирать на первом уровне некоторую конкретную сущность EntityA_I из иерархии абстрактного базового класса `AbstractEntityA`, а затем, на втором уровне, — некоторую конкретную сущность `EntityB_J` из иерархии абстрактного базового класса `AbstractEntityB`, после чего клиенту предоставляется возможность реализовать требуемую ассоциацию между выбранными сущностями[5].

Паттерн проектирования `DoubleSwitch`, реализующий эту идею, имеет диаграмму классов, показанную на рис. 2.10.

Обратите внимание, что метод `SelectEntityA` класса `Switch` возвращает указатель `AbstractEntityA*`, который передается далее в качестве аргумента методу `SelectEntityB`, осуществляющему выбор на втором уровне уже *для*

[5] Красиво сказано, не правда ли?

конкретной сущности первого уровня. Абстрактный класс второго уровня `AbstractEntityB` позволяет клиенту абстрагироваться от вида выполняемой операции: он имеет дело с «операцией вообще» Operate.

После выбора на втором уровне благодаря полиморфизму клиент может вызвать метод Operate, относящийся к конкретной сущности второго уровня. Передача ссылки `AbstractEntityA*` методу `Operate` разрешает последнему иметь доступ к атомарным операциям `AtomOperate1`, `AtomOperate2` и т. п., определенным в конкретном объекте иерархии `AbstractEntityA`. Совокупность этих примитивов должна быть достаточной для реализации любого метода Operate в иерархии `AbstractEntityB`. Как правило, это операции, обеспечивающие доступ ко всем защищенным или закрытым полям объектов иерархии `AbstractEntityA` или выполняющие обработку информации, специфичную для данного класса.

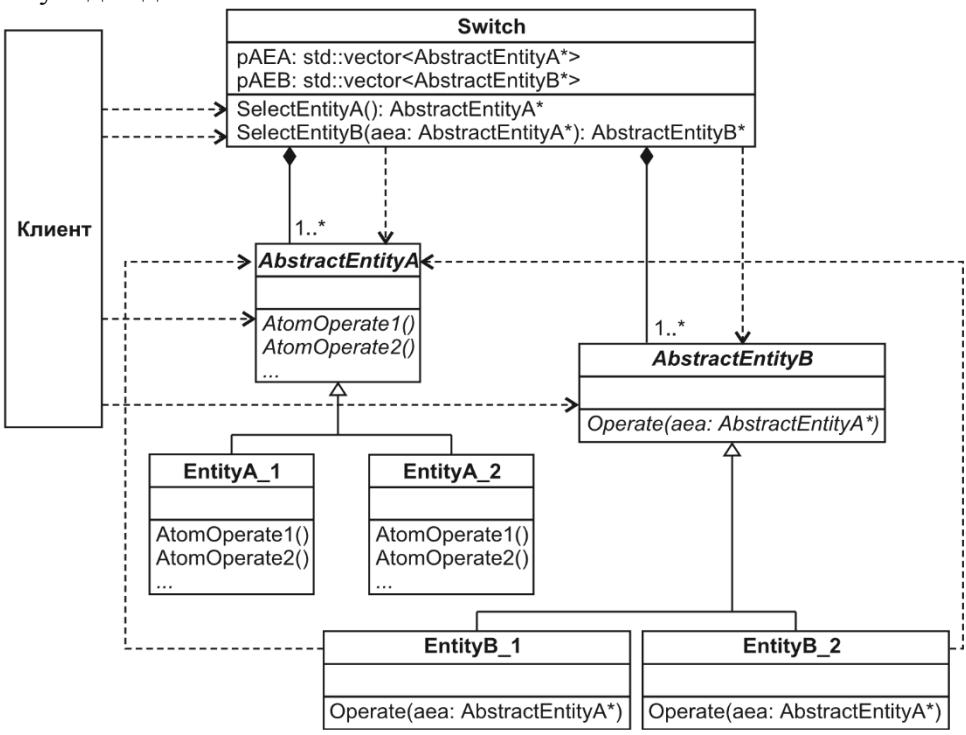

Рис. 2.10. Паттерн DoubleSwitch

Теперь попытаемся применить этот паттерн к нашей задаче. Ясно, что в качестве класса `Switch` будет выступать класс `Menu`, как и в предыдущей задаче, а в качестве класса `AbstractEntityA` — класс `Function`. Роль класса `AbstractEntityB` мы поручим новому абстрактному классу `Action`, обеспечивающему унифицированный интерфейс для конкретных классов

Calculation, Tabulation, AnyAction. Последний класспустышку мы добавим просто для иллюстрации того, что список выполняемых операций может легко расширяться.

Роль метода AbstractEntityB::Operate(AbstractEntityA*) будет возложена на метод Action::Operate(Function*). Ну и, наконец, метод Calculate перекочует из иерархии базового класса Function в конкретный класс Calculation, принадлежащий иерархии базового класса Action. Взамен мы должны пополнить иерархию Function такими атомарными операциями, как SetArg — установить значение аргумента, SetCoeff — установить значения коэффициентов, GetVal — получить значение функции.

Мы не будем приводить здесь диаграмму классов предлагаемого решения, оставляя это в качестве упражнения читателю. Текст программы приведен в листинге 2.4.

Листинг 2.4. Продвинутый функциональный калькулятор

```
// ////////////////////////////////////////////////// Function.h
#ifndef FUNCTION_H
#define FUNCTION_H
#include <string>
class Function {
public:
    virtual ~Function() {}
    void SetArg(double arg) { x = arg; }
    virtual void SetCoeff() = 0;
    virtual double GetVal() const = 0;
    virtual const std::string& GetName() const = 0;
protected:
    double x;      // аргумент
};
#endif /* FUNCTION_H */ //
////////////////////////////////////////////////////////// Exp.h
#include <math.h>
#include "Function.h"
class Exp : public Function { // --- Класс для представления функции y
= e ^ x
public:
    Exp() : name("e ^ x") {}
    const std::string& GetName() const { return name; }
    void SetCoeff() {}
    double GetVal() const { return exp(x); }
private:
    std::string name;   // мат. обозначение функции
};
extern Exp f_exp;
// ////////////////////////////////////////////////////// Exp.cpp
#include "Exp.h"
Exp f_exp;   // Глобальный объект
// ////////////////////////////////////////////////////// Line.h
```

```
#include "Function.h"
class Line : public Function { // --- Класс для представления функции
y = a * x + b
public:
    Line() : name("a * x + b") {}
    const std::string& GetName() const { return name; }
    void SetCoeff();
    double GetVal() const { return (a * x + b); }
private:
    std::string name;   // мат. обозначение функции
    double a, b;
};
extern Line f_line;
// ////////////////////////////////////////////////////// Line.cpp
#include <iostream>
#include "Line.h"
using namespace std;
void Line::SetCoeff() {
    cout << "Enter a = ";   cin >> a;
    cout << "Enter b = ";   cin >> b;
}
Line f_line;   // Глобальный объект
// ////////////////////////////////////////////////////// Action.h
#ifndef ACTION_H
#define ACTION_H
#include "Function.h"
class Action {
public:
    virtual ~Action() {}
    virtual void Operate(Function*) = 0;
    virtual const std::string& GetName() const = 0;
};
#endif /* ACTION_H */
// ////////////////////////////////////////////////// Calculation.h
#include "Action.h"
class Calculation : public Action {
public:
    Calculation() : name("Calculation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;   // обозначение операции
};
extern Calculation calculation;
// ////////////////////////////////////////////////// Calculation.cpp
#include <iostream>
#include "Calculation.h"
using namespace std;
void Calculation::Operate(Function* pFunc) {
```

```
    cout << "Calculation for function y = " << pFunc->GetName() <<
endl;
    pFunc->SetCoeff();
    double x;
    cout << "Enter x = ";   cin >> x;
    cin.get();
    pFunc->SetArg(x);
    cout << "y = " << pFunc->GetVal() << endl;
    cin.get();
}
Calculation calculation;   // Глобальный объект
// ////////////////////////////////////////////////// Tabulation.h
#include "Action.h"
class Tabulation : public Action {
public:
    Tabulation() : name("Tabulation") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;   // обозначение операции
};
extern Tabulation tabulation;
// ////////////////////////////////////////////////// Tabulation.cpp
#include <iostream>
#include <iomanip>
#include "Tabulation.h"
using namespace std;
void Tabulation::Operate(Function* pFunc) {
    cout << "Tabulation for function y = ";
    cout << pFunc->GetName() << endl;
    pFunc->SetCoeff();
    double x_beg, x_end, x_step;
    cout << "Enter x_beg = ";    cin >> x_beg;
    cout << "Enter x_end = ";    cin >> x_end;
    cout << "Enter x_step = ";   cin >> x_step;
    cin.get();
    cout << "-----------------------" << endl;
    cout << "    x          y" << endl;
    cout << "-----------------------" << endl;
    for (double x = x_beg; x <= x_end; x += x_step) {
        pFunc->SetArg(x);
        cout << setw(6) << x << setw(14) << pFunc->GetVal() << endl;
    }
    cin.get();
}
Tabulation tabulation;   // Глобальный объект
// ////////////////////////////////////////////////// AnyAction.h
#include "Action.h"
class AnyAction : public Action {
public:
```

```
    AnyAction() : name("Any action") {}
    const std::string& GetName() const { return name; }
    void Operate(Function*);
private:
    std::string name;   // обозначение операции
};
extern AnyAction any_action;
// ////////////////////////////////////////////////// AnyAction.cpp
#include <iostream>
#include "AnyAction.h"
using namespace std;
void AnyAction::Operate(Function*) {
    cout << "Здесь могла бы быть Ваша реклама!" << endl;
    cin.get();
}
AnyAction any_action;   // Глобальный объект
// ////////////////////////////////////////////////////// Menu.h
#include <vector>
#include "Function.h"
#include "Action.h"
class Menu {
public:
    Menu(std::vector<Function*>, std::vector<Action*>);
    Function* SelectObject() const;
    Action* SelectAction(Function*) const;
private:
    int SelectItem(int) const;
    std::vector<Function*> pObj;
    std::vector<Action*> pAct;
};
// ////////////////////////////////////////////////////// Menu.cpp
#include <iostream>
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Function*> _pObj, vector<Action*> _pAct)
        : pObj(_pObj), pAct(_pAct) {
    pObj.push_back(0);                  // для выбора пункта 'Exit'
}
Function* Menu::SelectObject() const {
    // Возьмите код аналогичной функции из листинга 2.3
}
Action* Menu::SelectAction(Function* pObj) const {
    int nItem = pAct.size();
    cout << "======================================\n";
    cout << "Select one of the following Action:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << pAct[i]->GetName() << endl;
      }
    int item = SelectItem(nItem);
    return pAct[item - 1];
```

```
}
int Menu::SelectItem(int nItem) const {
    // Возьмите код аналогичной функции из листинга 2.3
}
// ////////////////////////////////////////////////////// Main.cpp
#include <iostream>
#include "Exp.cpp"
#include "Line.cpp"
#include "Calculation.cpp"
#include "Tabulation.cpp"
#include "AnyAction.cpp"
#include "Menu.cpp"
using namespace std;
Function* pObjs[] = { &f_exp, &f_line };
vector<Function*> funcList(pObjs, pObjs +
sizeof(pObjs)/sizeof(Function*));
Action* pActs[] = { &calculation, &tabulation, &any_action };
vector<Action*> operList(pActs, pActs + sizeof(pActs) /
sizeof(Action*));
int main() {
    Menu menu(funcList, operList);
    while ( Function* pObj = menu.SelectObject() ) {
        Action* pAct = menu.SelectAction(pObj);
        pAct->Operate(pObj);
    }
    cout << "Bye!\n";
}
```

Обратите внимание на следующее:

- метод `Exp::SetCoeff` имеет пустое тело, то есть ничего не делает, поскольку для вычисления экспоненты коэффициенты не требуются;
- в связи с тем, что все методы класса `Exp` — встроенные, файл реализации `Exp.cpp` содержит только объявление глобального объекта `f_exp`.

Отметим, что предложенное решение позволяет легко модифицировать наш калькулятор, решая проблемы добавления как новых функций, так и новых операций.

Задача 2.3. Работа с объектами символьных и шестнадцатеричных строк

Написать программу, демонстрирующую работу с объектами двух типов: символьная строка (SymbString) и шестнадцатеричная строка (HexString), для чего создать систему соответствующих классов.

Каждый объект должен иметь как минимум два атрибута: идентификатор и значение, представленные в виде строк символов. В поле значения объекты SymbString могут хранить произвольный набор символов, в

то время как объекты HexString — только изображение шестнадцатеричного числа.

Клиенту (функции main) должны быть доступны следующие операции: создать объект, удалить объект, показать значение объекта (строку символов), показать изображение эквивалентного десятичного числа (только для объектов HexString), показать изображение эквивалентного двоичного числа (только для объектов HexString). Предусмотреть меню, позволяющее демонстрировать эти операции.

При решении этой задачи мы воспользуемся еще одним из видов диаграмм UML —*диаграммой видов деятельности* (*Activity Diagram*), чтобы расширить наш кругозор в сфере современных методов проектирования.

Диаграмма видов деятельности UML очень похожа на старые блок-схемы алгоритмов. В ней точками принятия решений и переходами описывается последовательность шагов, именуемых в UML *видами деятельности* (*Activity*). Каждый вид деятельности, ассоциируемый обычно с некоторой процедурой, изображается прямоугольником с закругленными углами, а переходы между ними — стрелками. Точки принятия решений можно изображать одним из двух способов: либо просто показываются все возможные переходы после завершения некоторого вида деятельности, либо изображается переход к маленькому ромбику, похожему на блок ветвления, а затем все возможные пути выходят из этого ромбика.

Анализируя условие задачи, мы пытаемся, во-первых, выявить объекты/классы предметной области и, во-вторых, сконструировать основной алгоритм, работающий с этими объектами. В данном случае представляется естественным алгоритм, изображенный в виде диаграммы видов деятельности на рис. 2.11.

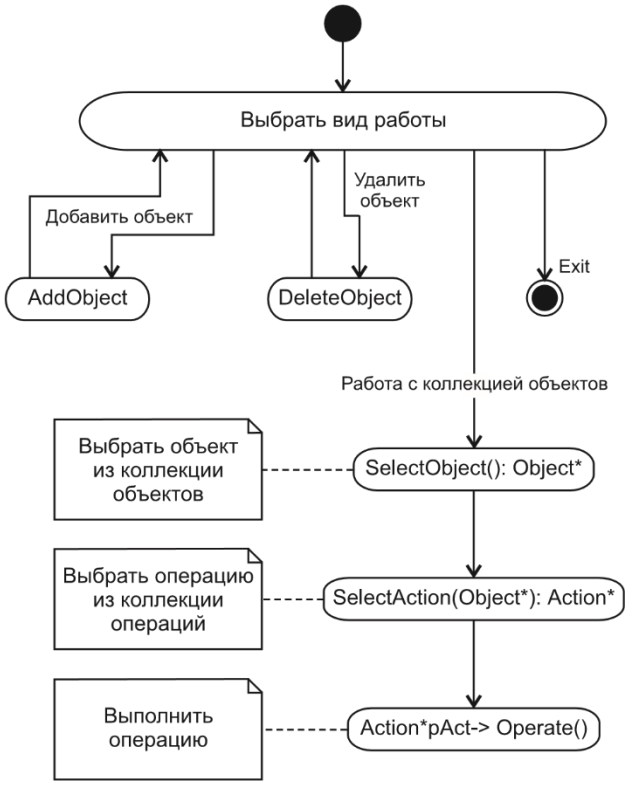

Рис. 2.11. Диаграмма видов деятельности для задачи 2.3

Начальная точка алгоритма обозначается в виде закрашенного кружка, конечная — в виде «глазка». При необходимости на диаграмме можно размещать комментарии в виде прямоугольника с текстом, напоминающего листок бумаги с отогнутым углом.

Графическое представление алгоритма позволяет более четко выявить решаемые проблемы. Так, меню в нашей программе должно обеспечить выбор на трех уровнях: 1) выбор вида работы (добавить объект к коллекции, удалить объект из коллекции, работать с коллекцией объектов, выйти из программы); 2) выбор объекта из коллекции объектов; 3) выбор операции из коллекции операций.

Опираясь на опыт реализации двухуровневого меню в предыдущей задаче, подумаем о способе реализации трехуровневого меню. Первой приходит мысль: а не создать ли нам паттерн проектирования `ThreeLevelSwitch`, развивающий идеи паттерна `DoubleSwitch`? Однако с каждым повышением размерности переключателя код будет становиться все сложней для понимания. Поэтому важно вовремя остановиться, помня, что любую идею можно довести до абсурда.

Принимаем следующий постулат: перечень действий (работ), выбираемых на первом уровне, тщательно продуман заказчиком, и вероятность его изменения крайне мала. В связи с этим первый уровень меню мы реализуем без использования паттернов — в блоке оператора `switch`, а вот для второго и третьего уровней применим уже испытанный паттерн `DoubleSwitch`, который позволит в будущем легко добавлять как типы обрабатываемых объектов, так и виды применяемых операций.

Теперь можно поговорить о классах. Два из них: `SymbString` и `HexString` — продиктованы условием задачи. Именно объекты этих классов будут пополнять коллекцию объектов, с которыми в дальнейшем можно выполнять те или иные операции (в нашем случае — показать значение объекта, показать изображение эквивалентного десятичного числа и т. д.). Для реализации паттерна проектирования `DoubleSwitch` нам потребуется абстрактный базовый класс `AString`, наследниками которого будут классы `SymbString` и `HexString`, а также абстрактный базовый класс `Action`, имеющий в качестве дочерних классы `ShowStr, ShowDec, ShowBin`.

Для реализации меню создадим, как и в предыдущей задаче, класс `Menu`, содержащий аналогичные методы `SelectObject` и `SelectAction`, но к ним в компанию добавим метод `SelectJob`. Поскольку список операций, применяемых к объектам, заранее известен (эти операции инкапсулированы в классах `ShowStr`, `ShowDec` и `ShowBin`), этот список, как и раньше, разместим в поле `pAct`. А вот со списком объектов ситуация более сложная, чем в задаче 2.2, так как эти объекты должны создаваться и уничтожаться динамически — в процессе работы программы. Пока отложим этот вопрос, мы вскоре к нему вернемся.

Какие сущности в решаемой задаче не охвачены перечисленными классами? В условии задачи упоминаются операции: *создать объект* и *удалить объект*. Конечно, можно поручить эти операции функции `main`, разместив заодно в ней и коллекцию обрабатываемых объектов, но это было бы в высшей степени нехорошо! Почему? — Потому, что такое решение понизило бы сцепление внутри главного клиента main.

СОВЕТ

Прилагайте максимум усилий к тому, чтобы каждая часть кода — каждый модуль, класс, функция, — отвечала за выполнение одной четко определенной задачи.

Последуем этому совету и создадим класс $_{\text{Factory}}$, отвечающий за создание и удаление объектов из иерархии `AString`. Класс будет содержать поле `std::vector<AString*> pObj` для хранения коллекции объектов, и два метода: `AddObject` и `DeleteObject`.

Пора переходить к кодированию. Предлагаемое решение приведено в листинге 2.5.

Листинг

2.5. Работа с объектами символьных и шестнадцатеричных строк

```
// ////////////////////////////////////////////////////// AString.h
#pragma once
#ifndef ASTRING_H
#define ASTRING_H
#include <string>
class AString {
public:
    virtual ~AString() {}
    virtual const std::string& GetName() const = 0;
    virtual const std::string& GetVal() const = 0;
    virtual int GetSize() const = 0;
};
#endif //ASTRING_H
// ////////////////////////////////////////////////// SymbString.h
#pragma once
#include <string>
#include "AString.h"
class SymbString : public AString {
public:
    SymbString(std::string _name) : name(_name) {}
    SymbString(std::string _name, std::string _val) : name(_name),
val(_val) {}
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name, val;
};
// ////////////////////////////////////////////////// HexString.h
#pragma once
#include <string>
#include "AString.h"
const std::string alph = "0123456789ABCDEF";
```

```
bool IsHexStrVal(std::string);
class HexString : public AString {
public:
    HexString(std::string _name) : name(_name) {}
    HexString(std::string, std::string);
    const std::string& GetName() const { return name; }
    const std::string& GetVal() const { return val; }
    int GetSize() const { return val.size(); }
private:
    std::string name, val;
};
// ////////////////////////////////////////////////// HexString.cpp
#include <iostream>
#include "HexString.h"
using namespace std;
bool IsHexStrVal(string _str) {                                  // 1
    for (int i = 0; i < _str.size(); ++i)
        if (-1 == alph.find_first_of(_str[i]))
            return false;
    return true;
}
HexString::HexString(string _name, string _val) : name(_name) {
    if (IsHexStrVal(_val))  val = _val;
}
// ////////////////////////////////////////////////////// Action.h
#pragma once
#ifndef ACTION_H
#define ACTION_H
#include "AString.h"
class Action {
public:
    virtual ~Action() {}
    virtual void Operate(AString*) = 0;
    virtual const std::string& GetName() const = 0;
protected:
    long GetDecimal(AString* pObj) const;
};
#endif /* ACTION_H */
// ////////////////////////////////////////////////// Action.cpp
#include <iostream>
#include "Action.h"
#include "HexString.h"
using namespace std;
long Action::GetDecimal(AString* pObj) const {
    if (dynamic_cast<HexString*>(pObj)) {
        long dest;
        string source = pObj->GetVal();
        sscanf(source.c_str(), "%lX", &dest);
        return dest;
    }
```

```
    else { cout << "Action not supported." << endl; return -1; }
}
// ////////////////////////////////////////////////// ShowStr.h
#pragma once
#include "Action.h"
class ShowStr : public Action {
public:
    ShowStr() : name("Show string value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name;   // обозначение операции
};
extern ShowStr show_str;
// ////////////////////////////////////////////////// ShowStr.cpp
#include <iostream>
#include "ShowStr.h"
using namespace std;
void ShowStr::Operate(AString* pObj) {
    cout << pObj->GetName() << ": " << pObj->GetVal() << endl;
    cin.get();
}
ShowStr show_str;   // Глобальный объект//
////////////////////////////////////////////////// ShowDec.h
#pragma once
#include "Action.h"
class ShowDec : public Action {
public:
    ShowDec() : name("Show decimal value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string name;   // обозначение операции
};
extern ShowDec show_dec;
// ////////////////////////////////////////////////// ShowDec.cpp
#include <iostream>
#include "ShowDec.h"
#include "HexString.h"
using namespace std;
void ShowDec::Operate(AString* pObj) {
    cout << pObj->GetName() << ": ";
    long decVal = GetDecimal(pObj);
    if (decVal != -1) cout << GetDecimal(pObj) << endl;
    cin.get();
}
ShowDec show_dec;   // Глобальный объект
// ////////////////////////////////////////////////// ShowBin.h
#pragma once
#include "Action.h"
```

```
class ShowBin : public Action {
public:
    ShowBin() : name("Show binary value") {}
    void Operate(AString*);
    const std::string& GetName() const { return name; }
private:
    std::string GetBinary(AString*) const;
    std::string name;   // обозначение операции
};
extern ShowBin show_bin;
// ////////////////////////////////////////////////// ShowBin.cpp
#include <iostream>
#include "ShowBin.h"
#include "ShowDec.h"
#include "AString.h"
using namespace std;
void ShowBin::Operate(AString* pObj) {
    cout << pObj->GetName() << ": " << GetBinary(pObj) << endl;
    cin.get();
}
string ShowBin::GetBinary(AString* pObj) const {
    int nBinDigit = 4 * pObj->GetSize();
    char* binStr = new char[nBinDigit + 1];
    for (int k = 0; k < nBinDigit; ++k)
        binStr[k] = '0';
    binStr[nBinDigit] = 0;
    long decVal = GetDecimal(pObj);
    if (-1 == decVal) return string("");
    int i = nBinDigit - 1;
    while ( decVal > 0 ) {
        binStr[i--] = 48 + (decVal % 2);
        decVal >>= 1;
    }
    string temp(binStr);
    delete [] binStr;
    return temp;
}
ShowBin show_bin;    // Глобальный объект
// ////////////////////////////////////////////////// Factory.h
#pragma once
#ifndef FACTORY_H
#define FACTORY_H
#include <vector>
#include "AString.h"
class Factory {
    friend class Menu; public:
    Factory() {}
    void AddObject();
    void DeleteObject();
private:
```

```
    std::vector<AString*> pObj;
};
#endif //FACTORY_H
// ////////////////////////////////////////////////// Factory.cpp
#include <iostream>
#include "Factory.h"
#include "Menu.h"
#include "SymbString.h"
#include "HexString.cpp"
using namespace std;
#define MAX_LEN_STR 100
void Factory::AddObject() {
    cout << "----------------------------------------\n";
    cout << "Select object type:\n";
    cout << "1. Symbolic string" << endl;
    cout << "2. Hexadecimal string" << endl;
    int item = Menu::SelectItem(2);
    string name;
    cout << "Enter object name: ";
    cin >> name; cin.get();

    cout << "Enter object value: ";
    char buf[MAX_LEN_STR];
    cin.getline(buf, MAX_LEN_STR);
    string value = buf;
    AString* pNewObj;
    switch (item) {
        case 1: pNewObj = new SymbString(name, value); break;
        case 2: if (!IsHexStrVal(value)) {
                    cout << "Error!" << endl;
                    return;
                }
                pNewObj = new HexString(name, value); break;
    }
    pObj.push_back(pNewObj);     cout << "Object added." << endl;
}
void Factory::DeleteObject() {
    int nItem = pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl; cin.get();
        return;
    }
    cout << ".....................................\n";
    cout << "Delete one of the following Object:\n";
    for (int i = 0; i < nItem; ++i)
        cout << i + 1 << ". " << pObj[i]->GetName() << endl;
    int item = Menu::SelectItem(nItem);
    string objName = pObj[item - 1]->GetName();
    pObj.erase(pObj.begin() + item - 1);
    cout << "Object " << objName << " deleted." << endl;
```

```
    cin.get();
}
// ////////////////////////////////////////////////////// Menu.h
#pragma once
#include <vector>
#include "AString.h"
#include "Action.h"
#include "Factory.h"
typedef enum { AddObj, DelObj, WorkWithObj, Exit } JobMode;
class Menu {
public:
    Menu(std::vector<Action*>);
    JobMode    SelectJob() const;
    AString*   SelectObject(const Factory&) const;
    Action*    SelectAction(const AString*) const;
    static int SelectItem(int);
private:
    std::vector<Action*> pAct;
};
// ////////////////////////////////////////////////////// Menu.cpp
#include <iostream>
#include "AString.h"
#include "SymbString.h"
#include "HexString.h"
#include "Menu.h"
using namespace std;
Menu::Menu(vector<Action*> _pAct) : pAct(_pAct) {}
JobMode Menu::SelectJob() const {
    cout << "=====================================\n";
    cout << "Select one of the following job mode:\n";
    cout << "1. Add object" << endl;
    cout << "2. Delete object" << endl;     cout << "3. Work with
object" << endl;
    cout << "4. Exit" << endl;
    int item = SelectItem(4);
    return (JobMode)(item - 1);
}
AString* Menu::SelectObject(const Factory& fctry) const {
    int nItem = fctry.pObj.size();
    if (!nItem) {
        cout << "There are no objects." << endl; cin.get();
        return 0;
    }
    cout << ".....................................\n";
    cout << "Select one of the following Object:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << fctry.pObj[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return fctry.pObj[item - 1];
```

```
}
Action* Menu::SelectAction(const AString* pObj) const {
    if (!pObj) return 0;
    int nItem = pAct.size();
    cout << ". . . . . .\n";
    cout << "Select one of the following Action:\n";
    for (int i = 0; i < nItem; ++i) {
        cout << i + 1 << ". " << pAct[i]->GetName() << endl;
    }
    int item = SelectItem(nItem);
    return pAct[item - 1];
}
int Menu::SelectItem(int nItem) {
    // Возьмите код аналогичной функции из листинга 2.3
}
// ////////////////////////////////////////////////////// Main.cpp
#include <iostream>
#include "Action.cpp"
#include "ShowStr.cpp"
#include "ShowDec.cpp"
#include "ShowBin.cpp"
#include "Factory.cpp"
#include "Menu.cpp"
using namespace std;
Action* pActs[] = { &show_str, &show_dec, &show_bin };
vector<Action*> actionList(pActs, pActs +
sizeof(pActs)/sizeof(Action*));
int main() {
    Factory factory;
    Menu    menu(actionList);
    JobMode jobMode;
    while ( (jobMode = menu.SelectJob()) != Exit ) {
        switch (jobMode) {
            case AddObj: factory.AddObject();     break;
            case DelObj: factory.DeleteObject();  break;
            case WorkWithObj:
                AString* pObj = menu.SelectObject(factory);
                Action* pAct = menu.SelectAction(pObj);
                if (pAct)  pAct->Operate(pObj); break;
        }
        cin.get();
    }
    cout << "Bye!\n";
}
```

Обратим внимание на наиболее интересные моменты реализации.

В модуле `HexString.cpp` размещена глобальная функция `IsHexStrVal`, проверяющая, соответствует ли аргумент `_str` изображению шестнадцатеричного числа. Проверка выполняется с помощью метода `find_first_of` класса `string`, который возвращает индекс вхождения символа,

заданного аргументом, в строку alph. Последняя объявлена в файле HexString.h и содержит набор допустимых символов для изображения шестнадцатеричного числа. Если символ не найден, метод find_first_of возвращает значение –1. Функция IsHexStrVal используется в конструкторе класса HexString, предотвращая присваивание полю val некорректного значения, а также в методе AddObject класса Factory, блокируя ошибочный ввод информации пользователем. В базовом классе Action реализован метод GetDecimal(AString* pObj), возвращающий *значение* десятичного числа для передаваемого через указатель pObj *изображения* шестнадцатеричного числа. Почему в базовом классе? — Потому, что это значение необходимо получать как в методе ShowDec::Operate, так и в методе ShowBin::GetBinary.

В методе Action::GetDecimal применена технология использования *информации о типе на этапе выполнения*, сокращенно *RTTI (RunTime Type Information)*. Дело в том, что заранее (на этапе компиляции) неизвестно, объекты каких типов будут передаваться данному методу как аргумент. Возможно появление объекта любого производного класса иерархии AString. В то же время из условия задачи известно, что получение десятичного и двоичного представлений корректно только для шестнадцатеричных строк. Поэтому мы используем операцию динамического приведения типов dynamic_cast, выполняющую *понижающее преобразование* из типа базового класса к типу производного класса. Операция возвращает адрес объекта производного класса, указанного в угловых скобках: <HexString*>, если преобразование возможно (если pObj действительно является адресом объекта класса HexString), либо 0 в противном случае.

В методе ShowBin::GetBinary сначала вычисляется десятичное значение decVal для шестнадцатеричной строки, а затем в цикле while формируется символьный массив binStr, содержащий символы ‘$_0$’ и ‘$_1$’ для двоичного изображения числа. Очередной символ вычисляется извлечением младшего бита из decVal операцией decVal % 2 и последующим его преобразованием к коду ASCII, для чего к значению бита (0 или 1) прибавляется 48. После извлечения двоичное представление decVal сдвигается на один разряд вправо.

В классе Menu метод SelectItem объявлен как статический. Дело в том, что подзадачу ввода целого числа для выбора пункта меню нужно решать не только в классе Menu, но и в классе Factory. Чтобы не дублировать код, мы сделали метод SelectItem статическим и теперь можем вызывать его из другого класса, разумеется, предваряя операцией доступа к области видимости Menu::.

В методе Factory::AddObject добавление нового объекта к коллекции pObj производится вызовом метода push_back класса vector.

В методе `Factory::DeleteObject` удаление объекта с заданным адресом осуществляется с помощью метода erase класса vector. Так как значение `item` на единицу больше, чем индекс элемента в контейнере `vector`, то адрес удаляемого объекта вычисляется выражением `pObj.begin() + item – 1`, где begin — метод, возвращающий адрес начального элемента в контейнере.

Итоги

1. Наследование и полиморфизм — важнейшие механизмы ООП. Наследование позволяет объединять классы в семейства связанных типов, что дает им возможность совместно использовать общие поля и методы.
2. В языке C++ родительский класс называется *базовым*, а дочерний класс — *производным*. Отношения между родительскими классами и их потомками называются *иерархией наследования*.
3. Полиморфное использование объектов из одной иерархии базируется на двух механизмах: а) возможности использования указателя или ссылки на базовый класс для работы с объектами любого из производных классов, б) технологии *динамического связывания* при выборе виртуального метода через указатель на базовый класс. Фактический выбор операции, которая будет вызвана, происходит во время выполнения программы в зависимости от типа выбранного объекта.
4. Наряду с отношением наследования, классы могут находиться и в других отношениях: *ассоциации*, *агрегации*, *композиции* и *зависимости* (*использования*). Удобным средством отображения взаимоотношений классов является *диаграмма классов* на языке UML.
5. Современное программирование базируется не только на идеях ООП, но и на применении паттернов проектирования, аккумулирующих наиболее удачные решения типичных проблем, неоднократно возникавших при разработке ПО.

Задания

Вариант 1 - 20

1. При выполнении задания необходимо построить UML-диаграмму классов программного модуля.
2. Тестирование и отладка:
   - 2.1. Внимательно прочитать условие задачи.
   - 2.2. Убедиться, что программа корректно работает на примерах.
   - 2.3. Составить план тестирования (проанализировать классы входных данных).
   - 2.4. Тестировать программу.
   - 2.5. Выполнить декомпозицию программы на отдельные блоки и покрыть каждый из них юнит-тестами

Примечание:

Условие задачи сформулировать внутри файла readme.md. С соответствующим форматированием. Файл поместить в папку проекта.

Диаграммы прикрепить в качестве отдельного файла в папку проекта.

Отладку решения задачи с помощью юнит-тестов, проводить в файле проекта (после отладки юнит-тесты закомментировать). При необходимости применять ввод входных данных из текстового файла .

Общая часть заданий для вариантов 1–20

Написать программу, демонстрирующую работу с объектами двух типов, `T1` и `T2`, для чего создать систему соответствующих классов. Каждый объект должен иметь идентификатор (в виде произвольной строки символов) и одно или несколько полей для хранения состояния (текущего значения) объекта.

Клиенту (функции `main`) должны быть доступны следующие основные операции (методы): создать объект, удалить объект, показать значение объекта и прочие дополнительные операции (зависят от варианта). Операции по созданию и удалению объектов инкапсулировать в классе `Factory`. Предусмотреть меню, позволяющее продемонстрировать заданные операции.

При необходимости в разрабатываемые классы добавляются дополнительные методы (например, конструктор копирования, операция присваивания и т.п.) для обеспечения надлежащего функционирования этих классов.

Варианты 1–10

В табл. 2.2 и 2.3 перечислены возможные типы объектов и возможные дополнительные операции над ними. Рассматриваются только целые положительные числа.

Таблица 2.2. Перечень типов объектов

Класс	**Объект**
SymbString	Символьная строка (произвольная строка символов)
BinString	Двоичная строка (изображение двоичного числа)
OctString	Восьмеричная строка (изображение восьмеричного числа)
DecString	Десятичная строка (изображение десятичного числа)
HexString	Шестнадцатеричная строка (изображение шестнадцатеричного числа)

Таблица 2.3. Перечень дополнительных операций (методов)

Операция (метод)	**Описание**
ShowBin()	Показать изображение двоичного значения объекта
ShowOct()	Показать изображение восьмеричного значения объекта
ShowDec()	Показать изображение десятичного значения объекта
ShowHex()	Показать изображение шестнадцатеричного значения объекта

operator +(T& s1, T& s2)[6]	для объектов SymbString — конкатенация строк s1 и s2; для объектов прочих классов — сложение соответствующих численных значений с последующим преобразованием к типу T
operator –(T& s1, T& s2)	для объектов SymbString — если s2 содержится как подстрока в s1, результатом является строка, полученная из s1 удалением подстроки s2; в противном случае возвращается значение s1; для объектов прочих классов — вычитание соответствующих численных значений с последующим преобразованием к типу T

Примечание. Первые четыре операции могут применяться к объектам любых классов, за исключением класса `SymbString`.

Таблица 2.4 содержит спецификации вариантов 1–10.

Таблица 2.4. Спецификации вариантов 1–10

Вариант	T1	T2	Операции (методы)
1	SymbString	BinString	ShowOct(), ShowDec(),ShowHex()
2	SymbString	BinString	operator +(T&, T&)
3	SymbString	BinString	operator –(T&, T&)
4	SymbString	OctString	operator +(T&, T&)
5	SymbString	OctString	operator –(T&, T&)
6	SymbString	DecString	ShowBin(), ShowOct(), ShowHex()
7	SymbString	DecString	operator +(T&, T&)
8	SymbString	DecString	operator –(T&, T&)
9	SymbString	HexString	operator +(T&, T&)
10	SymbString	HexString	operator –(T&, T&)

Варианты 11–20

В табл. 2.5 и 2.6 перечислены возможные типы объектов и дополнительные операции над ними.

Таблица 2.7 содержит спецификации вариантов 11–20.

Таблица 2.5. Перечень типов объектов

Класс	Объект	Класс	Объект
Triangle	Треугольник	Tetragon	Четырехугольник
Quadrate	Квадрат	Pentagon	Пятиугольник
Rectangle	Прямоугольник		

Таблица 2.6. Перечень дополнительных операций (методов)

Операция (метод)	Описание
Move()	Переместить объект на плоскости
Compare(T& ob1, T& ob2)	Сравнить объекты ob1 и ob2 по площади

[6] Здесь T — любой из типов, T1 или T2.

IsIntersect(T& ob1, T& ob2)	Определить факт пересечения объектов ob1 и ob2 (есть пересечение или нет)
IsInclude(T& ob1, T& ob2)	Определить факт включения объекта ob2 в ob1

Таблица 2.7. Спецификации вариантов 11–20

Вариант	**T1**	**T2**	**Операции (методы)**
11	Triangle	Quadrate	Move(), Compare(T&, T&)
12	Quadrate	Pentagon	Move(), IsIntersect(T&, T&)
13	Triangle	Rectangle	Move(), Compare(T&, T&)
14	Triangle	Rectangle	Move(), IsIntersect(T&, T&)
15	Rectangle	Pentagon	Move(), IsInclude(T&, T&)
16	Triangle	Tetragon	Move(), Compare(T&, T&)
17	Triangle	Tetragon	Move(), IsIntersect(T&, T&)
18	Triangle	Tetragon	Move(), IsInclude(T&, T&)
19	Triangle	Pentagon	Move(), Compare(T&, T&)
20	Triangle	Pentagon	Move(), IsIntersect(T&, T&)