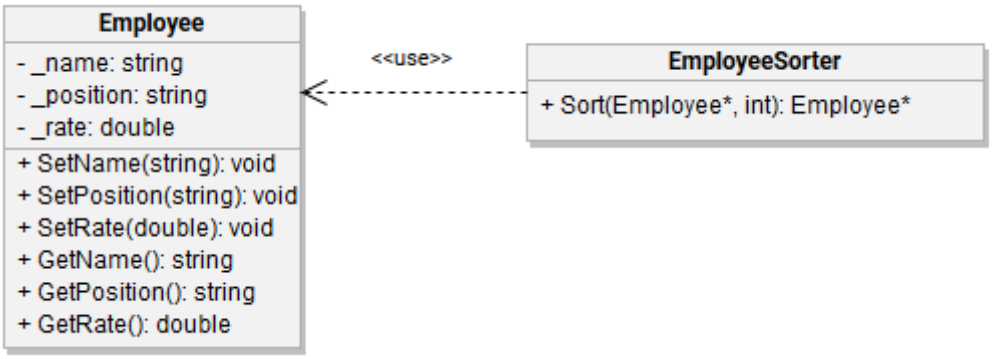
Лабораторная работа №5
Наследование и полиморфизм

Наследование классов

Выполняя предыдущие задания, вы могли обратить внимание, что при написании классов приходится писать много однотипного кода. Более того, код может повторяться из класса в класс. Особенно это касается классов, выполняющих схожие задачи. Например, классы прямоугольника, круга, квадрата могут дублировать целые участки кода с указанием координат центра фигуры, включая проверки входных значений. Помимо самих полей, геттеров и сеттеров, будет дублироваться и код по работе с объектами этих классов. Далее будут рассмотрены концепции объектно-ориентированного программирования, которые позволяют быстрее создавать новые классы на основе уже существующих, устранять дублирование кода в классах со схожей функциональностью и назначением, а также – главное – делать программы более гибкими для добавления новой функциональности.

Рассмотрим пример. В программе по учету сотрудников, использующейся в отделе кадров некоторой компании, существует класс сотрудника Employee:

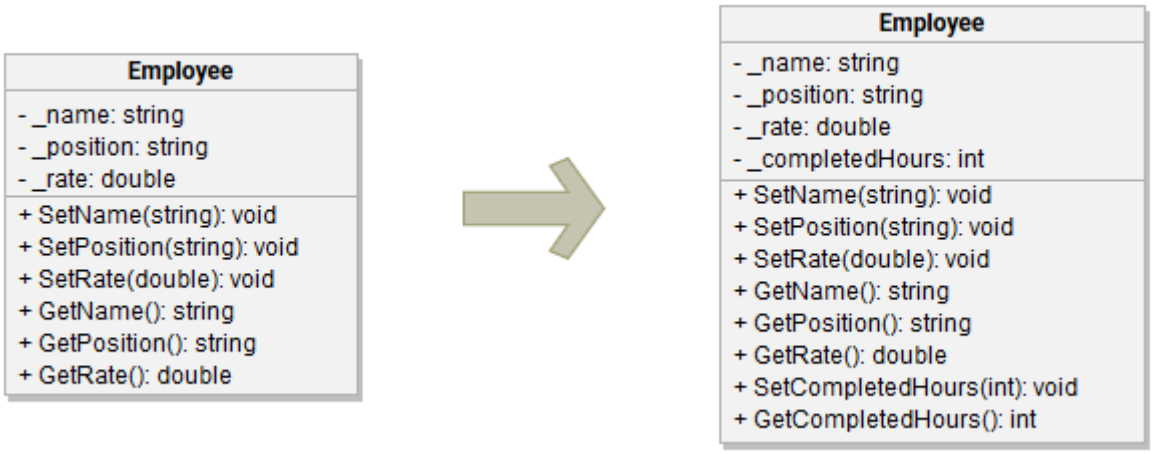


Класс хранит фамилию и имя сотрудника, его должность и оклад. Для работы с этим классом написана функция сортировки сотрудников по фамилии (например, как часть сервисного класса):

```
Employee* EmployeeSorter::Sort(Employee* employees, int employeesCount)
{
    for (int i = 0; i < employeesCount; i++)
    {
        for (int j = 0; j < employeesCount; j++)
        {
            if (employees[i].GetName() < employees[j].GetName())
            {
                auto temporaryEmployee = employees[i];
                employees[i] = employees[j];
                employees[j] = temporaryEmployee;
            }
        }
    }

    return employees;
}
```

Заказчик провел изменения в организации своей компании, и теперь в его штате работают несколько человек с почасовой оплатой. Для работы Сотрудники с почасовой оплатой хранят все те же данные, что и обычные сотрудники, только для расчета их зарплаты также должна храниться информация о количестве часов, отработанных в месяц.

Первый (неправильный) вариант модификации программы, который приходит в голову – это добавить в класс Employee новое поле _completedHours для хранения количества отработанных часов, а также соответствующие геттеры и сеттеры:

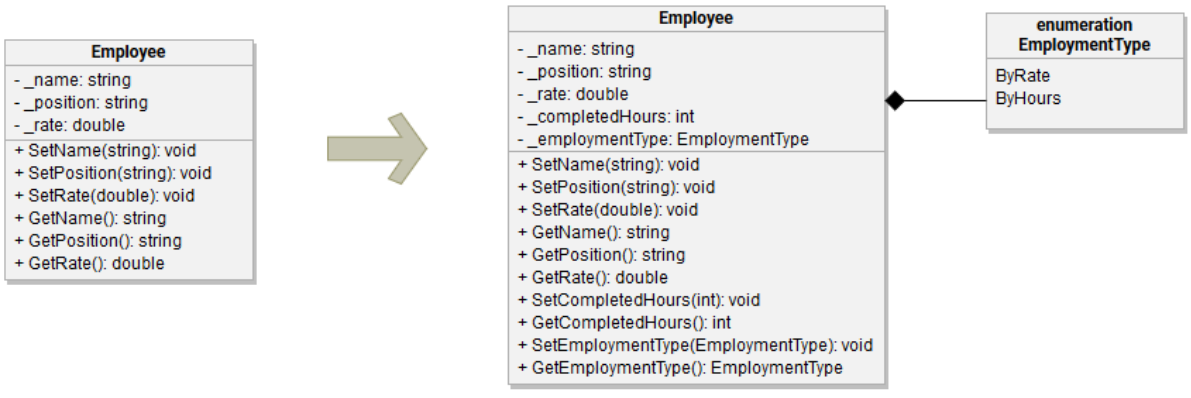


Однако данное решение плохо по двум причинам:

1) У сотрудников, работающих по окладу, а не по часам, теперь тоже есть лишнее поле, которое хранит данные о количестве отработанных часов. Учитывая, что таких сотрудников подавляющее большинство, создание вечно неинициализированного поля внутри класса нерационально.
2) После создания объекта сотрудника нельзя определить, по какой схеме устроен сотрудник – по окладу или по часовой оплате.

Действительно, если в поле часов сотрудника хранится значение 0, мы не знаем – или это сотрудник, работающий по окладу, или это сотрудник с почасовой оплатой, но не отработавший в этом месяце ни одного часа. Можно утвердить договоренность между разработчиками и даже самим заказчиком, что если в поле часов сотрудника хранится значение -1, то это сотрудник по окладу, а если положительное число – то это сотрудник с почасовой оплатой. Или заменить -1 на 168 – ровно столько рабочих часов должно быть у сотрудника по окладу в месяц при среднем количестве рабочих дней в месяце, равным 21. Но любая подобная договоренность плоха тем, что по недосмотру её очень легко нарушить, а сама договоренность приводит к большому количеству лишнего кода, например, на проверку количества часов на -1.

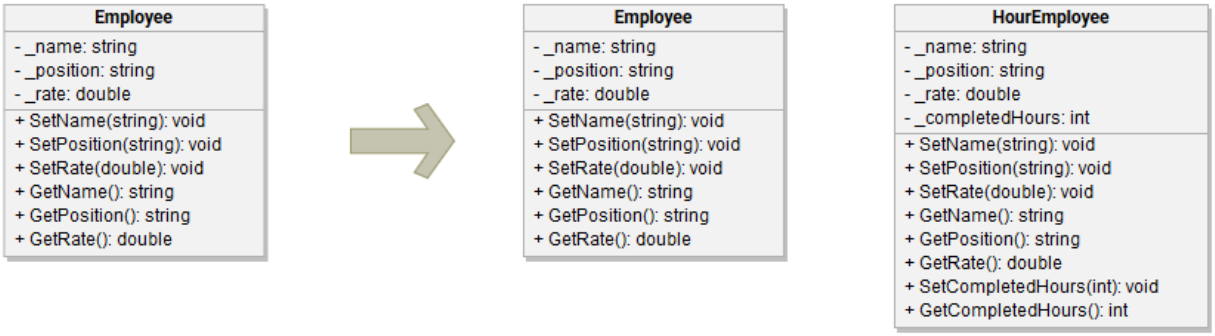
Второй (неправильный) вариант заключается в добавлении в класс поля перечисления EmploymentType, которое будет указывать способ оплаты (почасовой или по окладу):



Благодаря перечислению и дополнительному полю теперь всегда можно определить способ оплаты сотрудника. Однако недостаток с существованием поля количества часов, которое для большинства сотрудников будет не заполнено, остается. Данная проблема может показаться не значительной, но представьте ситуацию, когда сотруднику с почасовой оплатой необходимо, например, хранить пять дополнительных полей – объекты класса Employee фактически будут не заполнены.

Третий (неправильный) вариант заключается в создании нового класса, хранящего данные о сотруднике с почасовой оплатой. В этом варианте уже прослеживается объектный подход: сотрудник с почасовой оплатой

– это новая сущность предметной области, которой ранее не было, а для описания новых сущностей или обязанностей необходимо создать новый класс. Новые классы будут выглядеть следующим образом:



Проблема данного решения становится очевидной уже по диаграмме выше – реализация нового класса на 90% будет скопирована с уже существующего класса, а это дублирование кода. Однако эта проблема гораздо глубже.

Дело в том, что Си++, как и многие языки программирования, является ***статически типизированным*** языком программирования. Это означает, что типы всех переменных и передаваемых аргументов должны быть определены на этапе компиляции, а не этапе выполнения программы. Другими словами, если в статически типизированном языке программирования вы объявили переменную, то тип этой переменной не может быть изменен. Вы можете преобразовать значение переменной в переменную другого типа, но поменять тип исходной переменной вы не можете.

> **Типизация в языках программирования может быть *статической* или *динамической*, *сильной* или *слабой*, *явной* или *не явной*. Типизация – фундаментальное понятие в программировании и во многом определяет структуру программы, принципы составления алгоритмов и подходы к разработке. Си++ обладает слабой явной статической типизацией. Данные понятия не рассматриваются в рамках данного курса, и могут быть изучены читателем самостоятельно.**

Проверку типов при статической типизации выполняет компилятор – при попытке присвоения в переменную одного типа данных значения другого типа данных, компилятор сообщит об ошибке несовпадения типов данных. Например, если в переменную указателя int* попытаться поместить значение double. Вы наверняка сталкивались с этой ошибкой. Аналогично компилятор проверяет, чтобы при вызове функций типы передаваемых переменных совпадали с типами принимаемых функцией аргументов. Например, если функция принимает значение int, а разработчик пытается передать значение double.

Статическая типизация является разумным ограничением в языках программирования, позволяющая избежать многих ошибок еще на этапе компиляции. Однако, она же может потребовать от разработчика написания дополнительного кода для обработки разных типов данных.

Возвращаясь к нашему примеру с классами сотрудников, статическая типизация обязывает нас писать новые сервисные классы, функции и методы для нового класса сотрудника с почасовой оплатой. Например, сервисный класс, предоставляет метод сортировки сотрудников по фамилии, который принимает на вход указатель на тип данных Employee. Метод сортировки не будет работать с сотрудниками с почасовой оплатой, так как для компилятора классы Employee и HourEmployee – это два разных класса, то есть два разных типа данных. Также, мы не сможем хранить объекты Employee и HourEmployee в одном массиве, так как это разные типы данных.

Фактически, создание нового класса для нового типа сотрудника приводит нас к необходимости дублирования ***всего исходного кода***, который работал с классом Employee. При этом в большинстве случаев в дублированном коде потребуется поменять только тип данных Employee на HourEmployee, не внося каких-либо других изменений.

Сравнивая недостатки перечисленных трёх вариантов решения исходной задачи, можно прийти к выводу, что лучше хранение одного лишнего поля в классе, чем дублирование всей программы целиком. Но, как было сказано ранее, добавление не использующихся полей в классы в итоге приведет к созданию классов с десятками пустых полей, работа с которыми будет неудобна ни разработчику, ни пользователю программы.

Если проанализировать недостаток подхода к созданию нового класса HourEmployee, то он заключается в том, что компилятор не знает о связи классов Employee и HourEmployee в предметной области. В предметной области сотрудник с почасовой оплатой является лишь частным случаем обычного сотрудника, и все операции или действия, которые доступны для обычного сотрудника, также справедливы и для сотрудника с почасовой оплатой. Следовательно, необходим механизм, позволяющий сообщить компилятору о наличии такой связи между типами данными. Таким механизмом является ***наследование***.

> **Наследование – механизм языка программирования, позволяющий создавать новые классы с приобретением реализации другого класса. В отличие от агрегирования, где один объект является полем другого объекта, при наследовании один класс (дочерний, производный) является частным, расширенным вариантом другого класса (родительского, базового). При этом все поля и методы базового класса становятся полями и методами дочернего класса.**

Рассмотрим применение наследования на примере задачи с сотрудниками.

Четвертый (правильный) вариант. Данный вариант также предполагает создание нового класса, однако, в отличие от третьего варианта, новый класс HourEmployee будет создан с использованием наследования:

```
class Employee
{
    string _name;
    string _position;
    double _rate;

public:
    void SetName(string name);
    void SetPosition(string position);
    void SetRate(double rate);

    string GetName();
    string GetPosition();
    double GetRate();
};

// Новый (дочерний) класс на основе наследования
class HourEmployee : public Employee
{
    int _completedHours;

public:
    void SetCompletedHours(int completedHours);
    int GetCompletedHours();
};
```

Код с реализацией самих методов опущен, так как их содержание очевидно.

Обратите внимание на строку объявления класса HourEmployee:

```
class HourEmployee : public Employee
```

Данный синтаксис обозначает наследование, т.е. что класс HourEmployee является дочерним к классу Employee. Именно данное указание родительского класса через двоеточие в объявлении нового класса сообщает компилятору о связи данных классов между собой.

Также обратите внимание, что в классе HourEmployee теперь не объявляются поля для хранения фамилии, имени, оклада, а также геттеры и сеттеры для работы с этими полями. Внутри класса HourEmployee объявлены только те поля и методы, которые характерны для HourEmployee, но которых не должно быть в классе обычного сотрудника Employee. Создание класса HourEmployee стало лаконичным, и не содержит никакого дублирования кода.

Однако это не означает, что в классе HourEmployee осталось только одно поле с количеством часов. Объявление класса как наследника класса Employee сообщит компилятору, что все поля и методы класса Employee будут также доступны внутри класса HourEmployee. В этом легко убедиться, создав в функции main две переменных каждого типа и попытавшись обратиться к полям и методам каждого класса:

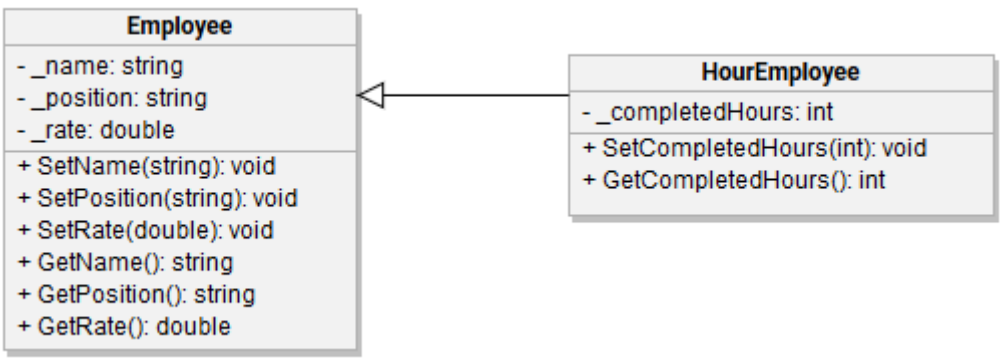
```
void main()
{
    // пример работы с объектом базового класса
    Employee* employee = new Employee();
    employee->SetName("Абаков С.И.");
    employee->SetPosition("Middle Developer");
    employee->SetRate(65000.0);

    HourEmployee* hourEmployee = new HourEmployee();

    // вызов унаследованных методов
    hourEmployee->SetName("Изодин А.Р.");
    hourEmployee->SetPosition("Junior Developer");
    hourEmployee->SetRate(30000.0);

    // вызов новых методов класса
    hourEmployee->SetCompletedHours(170);
}
```

Обращаясь к полям и методам класса HourEmployee, мы можем вызвать любые открытые (public) методы. Однако, это не означает, что наследование работает только для открытых полей и методов. Наследование предполагает приобретение ***всех полей и методов*** базового класса, однако извне класса будут доступны только открытые члены класса, точно также, как они доступны извне для базового.

Стоит отметить, что к закрытым полям и методам обратиться извне базового класса невозможно. То есть, к ним нельзя обратиться не только, например, в функции main(), но нельзя и обратиться внутри дочернего класса HourEmployee. Другими словами, несмотря на приобретение реализации другого класса, инкапсуляция базового класса при наследовании не может быть нарушена. Однако как управлять доступом к унаследованным полям и методам базового класса из производного без нарушения инкапсуляции будет рассмотрено в следующих подразделах.

Отдельно стоит сказать, что упрощается модификация классов. Так, если в будущем заказчику потребуется добавить новые данные о сотрудниках, или изменить проверку имени/фамилии сотрудника, то при изменении полей и методов внутри класса Employee, они автоматически будут изменены и в дочернем классе HourEmployee.

На диаграммах классов наследование обозначается в виде сплошной линии с закрытой стрелкой от дочернего класса к базовому:



При составлении диаграммы классов, в дочерних классах можно не указывать те члены классы, которые унаследованы и уже указаны в базовом. То есть достаточно указать только те члены класса, которые были добавлены в дочернем классе.

Мы рассмотрели создание класса HourEmployee с применением наследования, однако данный пример не закончен. В последующих разделах также будут рассмотрены преимущества работы с объектами дочерних классов.

Указатель на базовый класс

Ранее было рассмотрено создание дочерних классов с использованием механизма наследования. Теперь рассмотрим работу с объектами базовых и дочерних классов.

Как было указано ранее, язык Си++ относится к статически типизированным языкам программирования. Это, в частности, значит, что при выполнении операций компилятор строго контролирует типы данных, над которыми эти операции осуществляются:

```
void main()
{
    int a = 17;
    double b = 34.5;
    char c = 'A';

    int* p1;
    p1 = &a;
    p1 = &b; // Ошибка компилятора - несовпадение типов данных
    p1 = &c; // Ошибка компилятора - несовпадение типов данных
}
```

В Си++ допустимы явные и неявные преобразования типов, поэтому некоторые операции, такие как присвоение целочисленного значения в вещественную переменную, допустимы. Для пользовательских типов данных явное и неявное преобразование допустимы только тогда, когда разработчик сам определил данные операции для типов данных. Во всех остальных случаях, которых большинство, компилятор сообщит об ошибке. Особенно строго типизация соблюдается в отношении указателей. Указатель на double может хранить адрес только переменной типа double, указатель на char может хранить только тип char.

Однако это поведение несколько отличается при работе с указателями для классов с наследованием.

Создадим два динамических объекта класса Employee и HourEmployee:

```
Employee* employee = new Employee();
employee->SetName("Абаков С.И.");
employee->SetPosition("Middle Developer");
employee->SetRate(65000.0);

HourEmployee* hourEmployee = new HourEmployee();
hourEmployee->SetName("Изодин А.Р.");
hourEmployee->SetPosition("Junior Developer");
hourEmployee->SetRate(30000.0);
hourEmployee->SetCompletedHours(170);
```

Работа с объектами осуществляется через указатели, которые позволяют обратиться к любому открытому члену класса.

Указатель на класс Employee можно назвать ***указателем на базовый класс***. Указатель на базовый класс – это указатель на объекты класса, у которого есть наследники. Указатель на базовый класс обладает уникальным свойством – помимо адресов объектов базового класса, он также может хранить адреса объектов дочерних классов:

```
employee = hourEmployee;
cout << "Адрес объекта в указателе на базовый класс:  " << employee << endl;
cout << "Адрес объекта в указателе на дочерний класс: " << hourEmployee << endl;
```

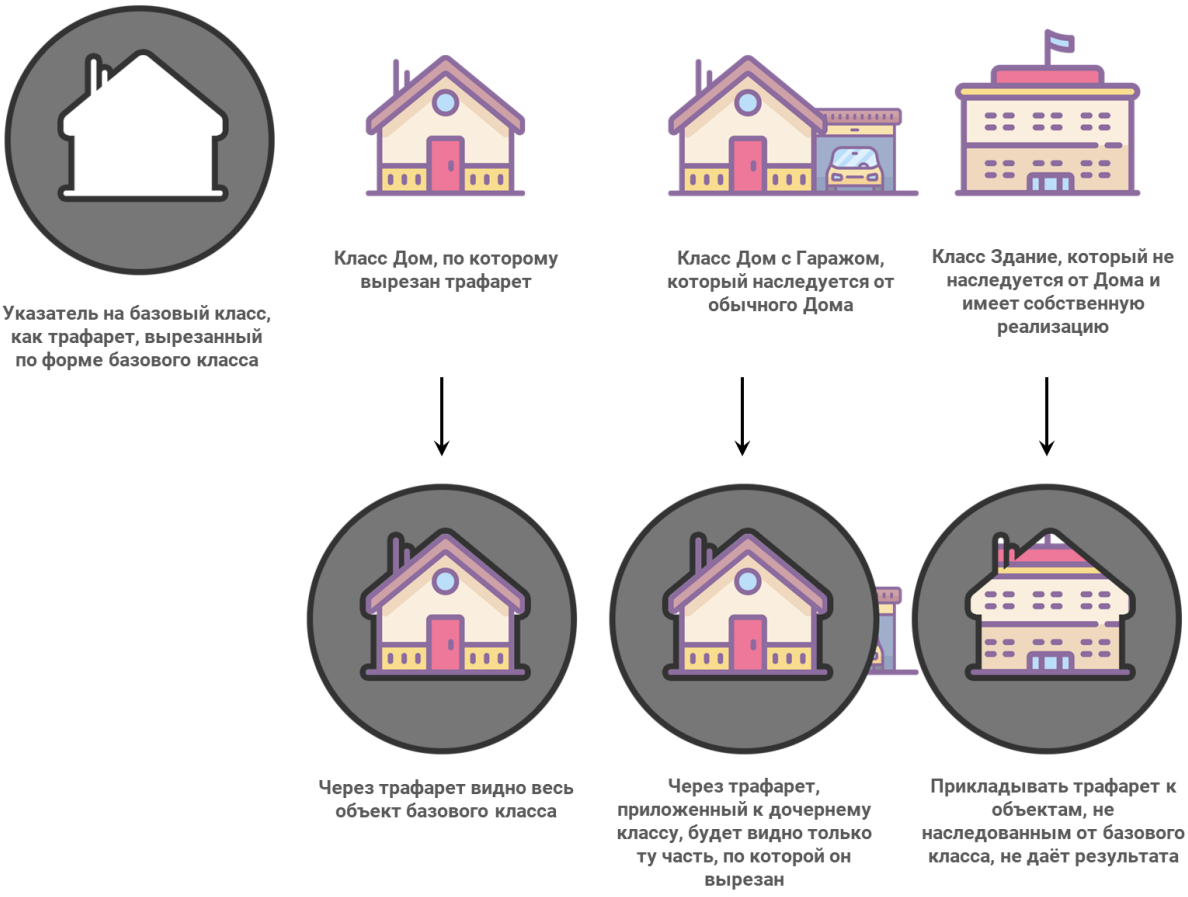
Пример работы программы:

```
Адрес объекта в указателе на базовый класс:  00A069E8
Адрес объекта в указателе на дочерний класс: 00A069E8
```

Можно даже вывести адрес объекта на дочерний класс до присвоения в указатель и после присвоения, чтобы убедиться, что указатель на базовый класс хранит адрес объекта HourEmployee. Но главным преимуществом является то, что указатель на базовый класс позволяет вызывать для объекта дочернего класса все члены класса, которые объявлены в базовом классе. То есть, через указатель на класс Employee мы можем вызвать, например, сеттеры для фамилии, имени и оклада, но нельзя обратиться к сеттеру для количества часов, так как этот сеттер объявлен в дочернем классе HourEmployee, а не базовом Employee:

```
// Обращение к методам дочернего класса через указатель на базовый приведет к ошибке
employee->SetCompletedHours(150);
cout << employee->GetCompletedHours();
```

Чтобы интерпретировать для себя работу с объектами через указатель на базовый класс, представьте себе трафарет, сделанный по форме некоторого объекта. Если вы будете прикладывать этот трафарет к разным объектам, то вы будете видеть только ту часть, которая видна через трафарет, но остальная часть предмета вам будет недоступна. Указатель на базовый класс является таким трафаретом:



Также важно помнить, что трафарет можно прикладывать не ко всем объектам, а только к объектам определенного типа. Иначе то, что вы увидите через трафарет, может вам не понравится.

В завершение примера стоит добавить, что при динамическом создании объекта дочернего класса его объект можно сразу помещать в указатель на базовый:

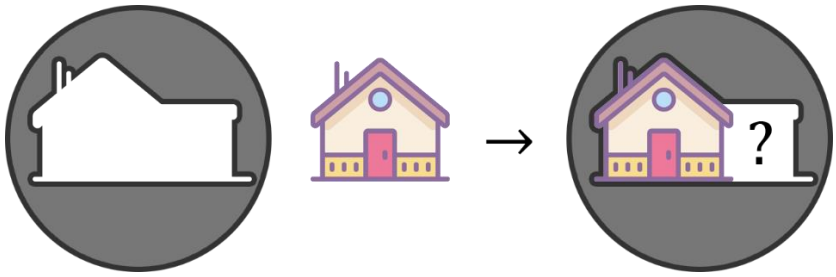
```
Employee* employee = new HourEmployee();
```

Данный код будет абсолютно корректным, но следует помнить, что после создания объекта дочернего класса, узнать программно тип данных объекта, хранящегося в указателе, будет затруднительно – это может помешать вам обратиться к тем полям и методам, которые объявлены в дочернем классе и недоступны через указатель на базовый класс.

Также стоит уточнить, что хранить дочерние объекты можно только в указателе на базовый класс. Если же попытаться поместить объект дочернего класса напрямую в переменную базового класса, то компилятор сообщит об ошибке несоответствия типов данных.

```
// Ошибка компиляции
hourEmployee = new Employee();
```

Последний аспект, который следует помнить о работе с объектами дочерних классов, что хранить объекты базового класса в указателях на дочерний неправильно:



И действительно, интерпретируя данную ситуацию с трафаретом, то вкладывая в трафарет объект базового типа, часть трафарета останется пустой. Попытка обращения к пустой части трафарета вызовет ошибку.

Применение указателя на базовый класс

Почему же указатели на базовый класс имеют такое значение? Дело в том, что именно указатели на базовый класс позволяют избавиться от дублирования кода при работе с дочерними классами.

Например, в сервисном классе для класса Employee существует метод, который формирует строку с информацией о рабочем для последующего вывода на экран:

```
// Сервисный класс, хранящий вспомогательные методы для Employee
class EmployeeTools
{
public:
    string GetDescription(Employee* employee);
};

string EmployeeTools::GetDescription(Employee* employee)
{
    string description;
    description += "Имя: " + employee->GetName();
    description += " Должность: " + employee->GetPosition();
    description += " Оклад: " + to_string(employee->GetRate());

    return description;
}
```

Метод выполняет форматирование информации под заданный формат и может применяться, например, для вывода информации о сотрудниках в виде таблицы (при вызове метода в цикле).

Так как Си++ имеет статическую типизацию, то без механизма наследования пришлось бы создавать сервисный класс с аналогичным методом:

```
// Сервисный класс для HourEmployee, практически полностью совпадающий с EmployeeTools
```

```
class HourEmployeeTools
{
public:
    string GetDescription(HourEmployee* employee);
};

string HourEmployeeTools::GetDescription(HourEmployee* employee)
{
    string description;
    description += "Имя: " + employee->GetName();
    description += " Должность: " + employee->GetPosition();
    description += " Оклад: " + to_string(employee->GetRate());

    return description;
}
```

Если в программе необходимо вывести всех сотрудников с любым способом оплаты в одну таблицу, тогда реализация такого сервисного класса будет полностью дублировать реализацию сервисного класса для базового Employee. Обратите внимание, что реализации отличаются только типом указателя, передаваемого на вход метода. Весь остальной код полностью повторяет исходный класс. Фактически, такое дублирование пришлось выполнять для всех классов, работающих с Employee, чтобы они могли работать и с дочерним HourEmployee.

Однако, используя механизм наследования, мы указываем компилятору, что класс HourEmployee связан с классом Employee. Мы помним, что указатель на базовый класс может хранить объекты дочерних классов и предоставлять доступ ко всем членам класса, доступным для базового класса. Это правило выполняется и для указателей, которые передаются в функцию или возвращаются из функции. Другими словами, если мы написали функцию или метод, который принимает указатель на базовый класс, то мы можем вызывать этот метод не только с объектами базового класса, но и с объектами дочернего класса:

```
void main()
{
    Employee* employee = new Employee();
    employee->SetName("Абаков С.И.");
    employee->SetPosition("Middle Developer");
    employee->SetRate(65000.0);

    HourEmployee* hourEmployee = new HourEmployee();
    hourEmployee->SetName("Изодин А.Р.");
    hourEmployee->SetPosition("Junior Developer");
    hourEmployee->SetRate(30000.0);
    hourEmployee->SetCompletedHours(170);

    // Вызов формирования описания для базового и дочернего объектов
    EmployeeTools* tools = new EmployeeTools();
    cout << tools->GetDescription(employee) << endl;
    cout << tools->GetDescription(hourEmployee) << endl;
}
```

Обратите внимание, что сервисный класс, написанный когда-то для класса Employee, автоматически, благодаря наследованию, способен работать с новым классом HourEmployee. При этом не требуется никаких модификаций в коде.

Распространяя данный пример на разработку в целом, это означает, что код, работающий с базовым классом, будет корректно работать и с дочерними классами. Это даёт огромные возможности для модификации программ и повторному использованию существующего кода. Так, библиотеки, написанные разработчиками десятки лет назад, могут использоваться для работы с новыми классами, написанными разработчиками в наше время.

Представление наследованных объектов в памяти и хранение дочерних объектов в массивах

Для правильного понимания работы наследования необходимо знать представление объектов в оперативной памяти. В первую очередь, это важно для понимания того, как правильно создавать массивы для классов-наследников.

Рассмотрим представление объектов в памяти на примере двух простых фиктивных классах. Классы и их поля не представляют никаких реальных объектов, и для упрощения примера, из класса устранены все методы. В этом примере созданы два класса с полями простых типов данных, при этом второй класс B наследуется от класса A и добавляет пару новых полей. В функции main происходит инициализация полей объектов каждого класса и вывод на экран размеров этих объектов в оперативной памяти в байтах:

```
class A
{
public:
    int X;
    int Y;
    double H;
};

class B : public A
{
public:
    int Z;
    double W;
};
```

```
int main()
{
    A a;
    a.X = 5;
    a.Y = 10;
    a.H = 15.35;

    B b;
    b.X = 3;
    b.Y = 6;
    b.H = 18.5;
    b.Z = 9;
    b.W = 11.2;

    cout << sizeof(a) << endl;
    cout << sizeof(b) << endl;
}
```

После запуска программа выведет на экран следующий результат (результат может отличаться, если для компилятора включена настройка выравнивания размеров полей класса):

```
16
28
```

То есть, размер объекта класса равен сумме размеров всех его полей. Если размер int равен 4 байта, а double – 8 байт, тогда размер объекта класса A равен 16 байтам, а размер объекта класса B будет равен 28.

Если представить оперативную память в виде ячеек по 1 байту, тогда переменные a и b можно показать следующим образом:

Обратите внимание, что:

1) все поля объекта класса располагаются в оперативной памяти последовательно;
2) поля, добавленные в дочернем классе, располагаются сразу же после полей базового класса;
3) все поля располагаются в том же порядке, что указаны в классе – компилятор не сортирует поля по имени или по размеру.

Объект в оперативной памяти представляет только значения полей. Если бы в классах были методы, то на размер объектов методы влияния не имеют. Так, мы знаем, что для представленных классов A и B при компиляции автоматически создаются конструкторы по умолчанию, однако они никак не повлияли на размер объектов при запуске программы – размер объектов равен сумме их полей и только полей. Внося ясность, статические поля в эту сумму не входят и хранятся в оперативной памяти отдельно от объектов.

Почему важно запомнить представление объектов в оперативной памяти? Понимание хранения объектов в памяти компьютера помогает понять, как создать массив, способный хранить объекты базового и дочерних классов одновременно.

Создавая массив какого-либо типа данных, мы указываем количество объектов в массиве:

```
double* array = new double[10];
array[0] = 15.5;
array[5] = -22.1;
```

Как мы знаем, элементы массива располагаются в оперативной памяти последовательно. При выполнении программа определяет размер одного объекта в памяти, умножает размер на количество элементов массива и таким образом вычисляет размер оперативной памяти, который нужно выделить под массив. Это особенно важно при работе с динамической памятью, так как нужно знать точное количество памяти для выделения и последующего освобождения.

Однако если мы создадим массив на базовый класс из десяти элементов, то поместить объекты дочернего класса в этот массив мы не сможем:

```
B b;
A* array = new A[10];

array[0] = b; // Ошибка компиляции или потеря данных
```

Объяснение для этого предельно простое. Каждый элемент массива будет занимать ровно столько памяти, сколько требуется для хранения объекта класса A – 16 байт. Если поместить объект дочернего класса B в любой из элементов массива, то объект из 28 байт просто не поместится в ячейку массива – часть данных фактически залезет на данные другого элемента, тем самым его повредив. В случае Си++ компилятор не позволит провести такую операцию.

Хранение объектов базового класса в массиве дочернего класса также неправильно, как хранение объекта базового класса в указателе на дочерний класс.

Мы знаем, что указатель на базовый класс может хранить объекты дочернего класса. Также мы знаем, что любой указатель занимает ровно 8 байт. Объединив эти две идеи, мы можем создать ***массив указателей на базовый класс***. В отличие от массива объектов базового класса, размер ячеек которого будет зависеть от размера объекта, ячейки массива указателей всегда будут занимать 8 байт, вне зависимости от того, на какие именно объекты указывают эти указатели:

```
A** pointersArray = new A* [10];

pointersArray[0] = new A();
pointersArray[1] = new B();
```

Этот приём позволяет хранить объекты базового и дочернего классов в одном массиве. Для чего это нужно?

Возвращаясь к примеру с сотрудниками компании, в отдельных ситуациях удобнее хранить всех сотрудников в одном массиве, чем хранить сотрудников в разных массивах в зависимости от типа оплаты труда. Напри-

мер, пользователю будет удобнее просматривать всех сотрудников компании в едином списке, чем проверять по двум разным спискам. А для того, чтобы реализовать вывод отсортированного списка всех сотрудников компании на экран, удобнее это делать, как все сотрудники хранятся в одном массиве. В противном случае, пришлось бы выполнять отдельно сортировку сотрудников разного типа и выводить их на экран по отдельности.

Зная, как хранить объекты наследуемых классов в одном массиве, мы также можем передавать такие массивы в функции. Например, реализовать сортировку сотрудников по фамилии в методе сервисного класса:

```
Employee** Sort(Employee** employees, int employeesCount)
{
    for (int i = 0; i < employeesCount; i++)
    {
           for (int j = 0; j < employeesCount; j++)
           {
                  if (employees[i]->GetName() < employees[j]->GetName())
                  {
                         auto temporaryEmployee = employees[i];
                         employees[i] = employees[j];
                         employees[j] = temporaryEmployee;
                  }
           }
    }

    return employees;
}
```

Обратите внимание, что подобный метод может сортировать как массив сотрудников класса Employee, как массив сотрудников класса HourEmployee, так и массивы с разными сотрудниками. В том числе, если в будущем в программе понадобится добавить новый тип сотрудников, унаследованный от класса Employee, то модифицировать метод сортировки не придется – он сможет выполнять сортировку сотрудников нового класса без каких-либо дополнительных модификаций. Так как помимо сортировки сотрудников в программе наверняка будут методы поиска, категоризации, фильтрации, вывода и т.п., хранение объектов в массивах указателей на базовый класс значительно упрощает поддержку программы и добавление новой функциональности.

Вызов конструкторов при наследовании

Ранее в примере в классах специально были опущены конструкторы, чтобы разобрать данный вопрос в отдельном подразделе.

Создавая класс, мы можем определить способ создания его объектов с помощью конструкторов. Если разработчик не создаст конструктор самостоятельно, то компилятор создаст конструктор по умолчанию автоматически. Это будет конструктор без параметров и не выполняющий никаких операций. Но чаще разработчик создаёт конструктор с параметрами для инициализации полей.

При создании объектов дочернего класса происходит вызов конструкторов как дочернего класса, так и базового. При этом конструктор базового класса отвечает за выделение памяти и инициализацию полей базовой части класса, а конструктор дочернего класса отвечает за выделение памяти и инициализацию только тех полей, которые были добавлены в дочернем классе. Аналогично работают и деструкторы класса.

Разберем процесс создания и уничтожения объектов на следующем примере:

```
class Base
{
    int _x;

public:
    Base();
    ~Base();
```

```
class Child : public Base
{
    int _y;

public:
    Child();
    ~Child();
```

```
};

Base::Base()
{
    cout << "Вызван конструктор Base"
         << endl;
}

Base::~Base()
{
    cout << "Вызван деструктор Base"
         << endl;
}
```

```
};

Child::Child()
{
    cout << "Вызван конструктор Child"
         << endl;
}

Child::~Child()
{
    cout << "Вызван деструктор Child"
         << endl;
}
```

Два класса, базовый и дочерний, в которых объявлены конструкторы и деструкторы. В методы мы специально поместили вывод на экран, чтобы наглядно увидеть, в какой последовательности вызываются конструкторы и деструкторы. Далее в функции main() мы сначала создадим и уничтожим объект класса Base, чтобы посмотреть вызов методов для обычного класса. А затем создадим и уничтожим объект класса Child, чтобы посмотреть вызовы методов для дочернего класса:

```
int main()
{
    cout << "Проверка создания и уничтожения объектов класса Base" << endl;
    Base* base = new Base();
    delete base;

    cout << endl;
    cout << "Проверка создания и уничтожения объектов класса Child" << endl;
    Child* child = new Child();
    delete child;
}
```

В результате запуска программы мы увидим следующее:

```
Проверка создания и уничтожения объектов класса Base
Вызван конструктор Base
Вызван деструктор Base

Проверка создания и уничтожения объектов класса Child
Вызван конструктор Base
Вызван конструктор Child
Вызван деструктор Child
Вызван деструктор Base
```

Для объекта класса Base нет ничего необычного – результат соответствует вызову конструктора и вызову деструктора в функции main(). Но для класса Child получен более интересный результат. Как видно из результата запуска, при вызове конструктора класса Child сначала автоматически будет вызван конструктор базового класса Base, и только затем будет вызван сам конструктор Child (ни в классе Child, ни в функции main() мы не вызываем конструктор класса Base самостоятельно). Для деструктора мы видим обратный порядок: сначала вызывается деструктор класса Child и по его завершению вызывается деструктор класса Base.

Данный пример наглядно иллюстрирует механизм наследования, где дочерний класс является своеобразной надстройкой для базового. Поэтому при создании объектов дочернего класса сначала надо создать основание – часть из базового класса -, а затем саму надстройку. При уничтожении объектов класса, наоборот, сначала разрушить надстройку, а затем основание.

В примере выше разобран случай конструкторов без параметров. Однако, как правило, конструкторы класса имеют параметры. Более того, в базовом классе может быть объявлен не один конструктор, а несколько,

каждый со своим набором параметров. Логично, что в языке программирования нужен некоторый механизм, который позволит дочернему классу определить:

1) какой именно конструктор базового класса вызывать перед конструктором дочернего класса;
2) какие значения параметров должны быть переданы в конструктор базового класса.

Реализуется это предельно просто:

```
class Base
{
    int _x;

public:
    Base(int x);
    ~Base();
};

Base::Base(int x)
{
    _x = x;
}

Base::~Base()
{
}
```

```
class Child : public Base
{
    int _y;

public:
    Child(int x, int y);
    ~Child();
};

Child::Child(int x, int y) : Base(x)
{
    _y = y;
}

Child::~Child()
{
}
```

Во-первых, создадим в базовом классе конструктор с параметром, который будет инициализировать закрытое поле значением.

Во-вторых, создадим в дочернем классе конструктор с параметрами. Обратите внимание, что конструктор должен принимать параметры как для полей дочернего класса, так и для полей базового класса. Так как клиентский код в функции main() не вызывает напрямую конструктор для Base, следовательно, все необходимые для инициализации данные должны быть переданы в конструктор дочернего класса.

В-третьих, при реализации конструктора дочернего класса, после объявления аргументов метода необходимо написать вызов необходимого конструктора базового класса с передачей в него параметров. В нашем случае, это вызов конструктора Base() с передачей в него локальной переменной x, которая является входным аргументом для конструктора Child(). Сам конструктор Child() выполняет инициализацию собственных полей (поля _y), а за инициализацию полей базового класса отвечает конструктор базового класса.

Разумеется, конструктор дочернего класса может выполнить инициализацию всех полей, как унаследованных, так и собственных – такой приём также можно встретить в практике. Однако в подавляющем большинстве случаев, рекомендуется выносить логику инициализации полей в конструктор того класса, где эти поля объявлены. Также, если в базовом классе есть поля доступные только на чтение, то они могут быть проинициализированы только в базовом классе – следовательно, конструктор дочернего класса обязан вызывать конструктор базового.

> **В обозначенном примере инициализация полей класса выполняется конструктором напрямую, без сеттеров. Это может привести к нарушению целостности данных. В реальных классах конструкторы должны присваивать значения в поля исключительно через сеттеры.**

Например, для классов Employee и HourEmployee реализация конструкторов выглядела бы следующим образом:

```
Employee::Employee(string name, string position, double rate)
{
    SetName(name);
    SetPosition(position);
    SetRate(rate);
}
```

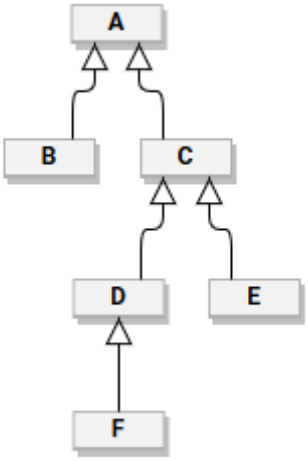
```
HourEmployee::HourEmployee(string name, string position, double rate, int completedHours):
                                                        Employee(name, position, rate)
{
    SetCompletedHours(completedHours);
}
```

Если в родительском классе содержится несколько конструкторов, при реализации конструктора в дочернем классе вы можете указать любой из них. Вызов нужного конструктора будет определяться набором аргументов, переданных в конструктор. Если в родительском классе есть только один конструктор без параметров, указывать его в реализации конструктора дочернего класса не надо.

Говоря о реализации деструкторов, указывать базовый деструктор при реализации деструктора дочернего класса также не надо. Правила языка Си++ определяют, что в любом классе может быть только один деструктор, и этот деструктор всегда без параметров – по этой причине компилятор самостоятельно определит вызов деструктора базового класса в деструкторе дочернего.

Иерархия наследования

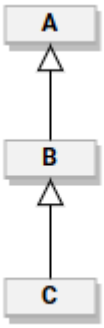
Наследование не ограничивается двумя классами. Оно может быть продолжено как вертикально – у дочерних классов могут собственные наследники с новыми полями и методами – и горизонтально – у любого класса может быть не один класс-наследник, а несколько. Таким образом, аналогично агрегированию, с помощью наследования создается иерархия классов.



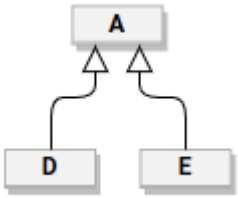
Конечно, наследование не применяется в программах отдельно от других связей – базовые и дочерние классы могут быть как агрегированы и использованы другими классами, так и сами агрегировать и использовать другие классы. Все виды связей между классами программы и составляют архитектуру программы.

Вне зависимости от роста иерархии наследования, правила работы указателей на базовый класс, также как и другие правила, сохраняются.

Например, если класс A имеет дочерний класс B, а класс B имеет дочерний класс C, то объекты класса C могут храниться как в указателе на базовый класс B, так и в указателе на базовый класс A. Сам класс B является одновременно и базовым (то отношению к классу C), так и дочерним (по отношению к классу A).



Если у класса A есть два прямых класса-наследника D и E, тогда объекты классов D и E могут храниться в указателе на класс A. Но при этом объекты класса D не могут храниться в указателе на класс E, а объекты класса E не могут храниться в указателе на класс D.



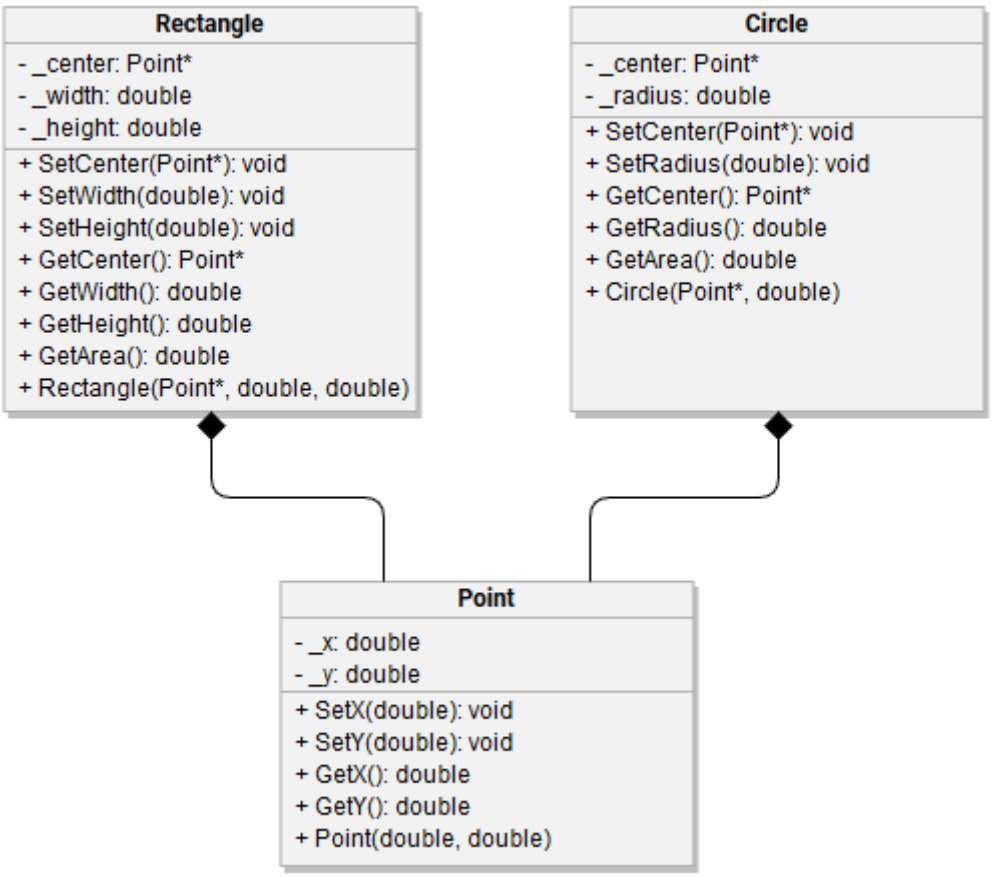
Рефакторинг с созданием базового класса

Далеко не всегда при разработке программ сначала происходит создание базового класса, а затем, при модификации программы, появляются новые дочерние классы. Часто создание базового класса идет в обратном порядке – разработчики создают несколько классов для решения задач пользователя, затем замечают, что в программе есть классы со схожим назначением и дублирующимся кодом, и только после этого создают базовый класс. В базовый класс выносится общая функциональность, а сервисные классы или более высокоуровневые классы переписываются на использование указателя на базовый класс вместо использования конкретных классов в тех местах, где это допустимо.

Такая модификация программы относится к понятию рефакторинга. **Рефàкторинг** – модификация программы с целью улучшения структуры кода или его читаемости без изменения его функциональности. Термин "рефакторинг" можно было бы перевести на русский как "реорганизация кода", однако слово "рефакторинг" крепко вошло в речь разработчиков. Важно подчеркнуть, что рефакторинг не добавляет новой функциональности самой программе, т.е. со стороны конечного пользователя программа выглядит и работает как и раньше. Рефакторинг проводится для того, чтобы в дальнейшем самим разработчикам было удобнее работать с исходным кодом и проще добавлять новую функциональность. Фактически, рефакторинг – это наведение порядка на рабочем месте.

К рефакторингу относится не только выделение базовых классов, но и любые работы по реорганизации кода, исключению дублирования и повышении его читаемости. Данной теме посвящена отдельная книга Мартина Фаулера "Рефакторинг. Улучшение существующего кода", в данном же разделе разберем только один пример, связанный с наследованием.

Определить дублирование кода в программе можно по диаграммам классов или по самому исходному коду:



На диаграмме видно два класса, в которых повторяется часть функциональности. Сами классы прямоугольника Rectangle и круга Circle имеют схожее назначение – они описывают геометрические фигуры. К повторяющейся функциональности можно отнести поле, геттер и сеттер, описывающие центр геометрической фигуры, а также можно увидеть, что в обоих классах есть метод с одинаковым названием GetArea(), выполняющий расчет площади фигуры.

Диаграмма классов позволяет нам увидеть ***потенциально*** дублирующийся код, однако в этом надо убедиться, сравнив реализации методов для обоих классов. Так, если сеттеры в обоих классах совпадают (в них используются одинаковые проверки), то эти методы в будущем можно будет вынести в базовый класс. Если же реализации методов будут отличаться, то вынести их в базовый класс нельзя. Еще раз подчеркнем, что в базовый класс выносится только общая, дублирующаяся функциональность. Если методы имеют одинаковое название, но принципиально разную реализацию, эти методы дублирующимися не считаются.

```
class Point
{
    double _x;
    double _y;

public:
    void SetX(double x);
    void SetY(double y);

    double GetX();
    double GetY();

    Point(double x, double y);
};

void Point::SetX(double x)
{
```

```
    _x = x;
}

void Point::SetY(double y)
{
    _y = y;
}

double Point::GetX()
{
    return _x;
}

double Point::GetY()
{
    return _y;
}

Point::Point(double x, double y)
{
    SetX(x);
    SetY(y);
}

////////////////////////////////////////////////////////////
class Rectangle
{
    Point* _center;
    double _width;
    double _height;

public:
    void SetCenter(Point* center);
    void SetWidth(double width);
    void SetHeight(double height);

    Point* GetCenter();
    double GetWidth();
    double GetHeight();

    double GetArea();

    Rectangle(Point* center, double width, double height);
};

void Rectangle::SetCenter(Point* center)
{
    _center = center;
}

void Rectangle::SetWidth(double width)
{
    _width = width;
}

void Rectangle::SetHeight(double height)
{
    _height = height;
}
```

```
Point* Rectangle::GetCenter()
{
    return _center;
}

double Rectangle::GetWidth()
{
    return _width;
}

double Rectangle::GetHeight()
{
    return _height;
}

double Rectangle::GetArea()
{
    return GetWidth() * GetHeight();
}

Rectangle::Rectangle(Point* center, double width, double height)
{
    SetCenter(center);
    SetWidth(width);
    SetHeight(height);
}

////////////////////////////////////////////////////////////
class Circle
{
    Point* _center;
    double _radius;

public:
    void SetCenter(Point* center);
    void SetRadius(double radius);

    Point* GetCenter();
    double GetRadius();

    double GetArea();

    Circle(Point* center, double radius);
};

void Circle::SetCenter(Point* center)
{
    _center = center;
}

void Circle::SetRadius(double radius)
{
    _radius = radius;
}

Point* Circle::GetCenter()
{
    return _center;
}
```

```
double Circle::GetRadius()
{
    return _radius;
}

double Circle::GetArea()
{
    return 3.14159 * GetRadius() * GetRadius();
}


Circle::Circle(Point* center, double radius)
{
    SetCenter(center);
    SetRadius(radius);
}
```

В ходе просмотра исходного кода можно определить, что дублируются:

- поле _center;
- геттер GetCenter();
- сеттер SetCenter();
- часть конструкторов, инициализирующих центр.

Весь остальной код является уникальным для своих классов. Отдельно стоит указать, что метод GetArea() для обоих классов имеет одинаковую сигнатуру, но реализация расчета площади для круга и квадрата принципиально отличается. По этой причине, метод GetArea() не относится к дублирующемуся коду.

Теперь, определив дублирующиеся участки кода, создадим базовый класс. Выбирая название для базового класса, необходимо найти существительное, которое описывало бы оба дочерних класса. Так как прямоугольник и круг описывают разные геометрические фигуры, то базовый класс следует назвать GeometricFigure или просто Figure. Иногда, начинающие разработчики не думают над названием и придумывают что-то на подобие RectangleOrCircle. Такие названия сложные, неправильные с точки зрения наследования от него новых геометрических фигур, и, главное, вводящие больше путаницы для читающих код разработчиков. Название базового класса должно быть простым, но при этом обобщать смысл дочерних сущностей.

Код базового класса будет следующим:

```
class GeometricFigure
{
    Point* _center;

public:
    void SetCenter(Point* center);
    Point* GetCenter();

    GeometricFigure(Point* center);
};

void GeometricFigure::SetCenter(Point* center)
{
    _center = center;
}

Point* GeometricFigure::GetCenter()
{
    return _center;
}

GeometricFigure::GeometricFigure(Point* center)
```

```
{
    SetCenter(center);
}
```

Теперь дублирующийся код можно удалить из классов Rectangle и Circle. Важно не забыть добавить в объявление дочерних классов наследование от базового:

```
class Rectangle : public GeometricFigure
```

Следующим шагом рефакторинга является просмотр кода, который использует объекты прямоугольников и кругов. Так, если сторонние классы работают, например, с прямоугольником, используя только методы из базового класса, то такие сторонние классы могут быть переписаны на использование базового указателя.

Например, в программе может существовать сервисный класс, который выполняет выравнивание геометрических фигур по центру – два его метода принимают либо массив кругов Circle, либо массив прямоугольников Rectangle, и присваивают им одинаковые координаты центра. Так как для работы этих методов необходим вызов только сеттера SetCenter(), вынесенного в общий класс, то можно избавиться от дублирования кода в сервисном классе, заменив два метода на один, работающий с указателем на базовый класс. В будущем, если в программу будут добавлены другие геометрические фигуры, то сервисный класс автоматически сможет выполнять и их выравнивание.

В данном примере в рамках базового класса устранилось дублирование 25 строк кода. Это может показаться незначительным улучшением исходного кода. Однако на практике дублируемого кода даже в рамках трёх классов может быть существенно больше, включая дублирование в клиентском коде. Реальные программы могут состоять из сотен и тысяч классов. Вторым преимуществом выделения базового класса является возможность внесения изменений в базовый класс (например, дополнительных проверок в сеттерах), что автоматически добавит эти проверки во все классы наследников. Есть также третье преимущество, связанное с полиморфизмом, но оно будет раскрыто в последующих разделах.

Модификатор доступа protected

Механизм инкапсуляции создает уникальную область видимости внутри класса, доступ к которой разграничивается с помощью модификаторов private и public. Сделано это для того, чтобы клиентский код, то есть код, который будет использовать некий класс не мог получить доступ к тем членам класса, которые нужны исключительно для внутренней работы класса. Например, чтобы клиентский не мог напрямую обратиться к полям класса и изменить их значение. Или не дать доступ к закрытым методам класса – сеттерам для полей доступных только на чтение или методов валидации, используемых в тех же сеттерах.

Однако наследование – это более тесная связь, чем просто использование объектов класса. При наследовании один объект фактически приобретает реализацию другого объекта, даже если она закрыта модификатором доступа private. В отдельных ситуациях дочерним классам бывает необходим доступ к полям базового класса, к его закрытым сеттерам или другим скрытым методам. Но делать члены базового класса открытыми (public) ради задач дочернего класса будет плохой идеей, ведь поля и методы станут доступными не только для дочерних классов, но и для любого другого клиентского кода.

С этой целью, совместно с механизмом наследования добавлен модификатор доступа protected ("защищенный"). ***Модификатор доступа protected*** – модификатор доступа, делающий члены класса доступными внутри базового и его дочерних классов, но не доступными для остального кода.

Рассмотрим пример:

```
class Person
{
    string _surname;
    string _name;

    void AssertCorrectName(string name);

public:
```

```
    void SetSurname(string surname);
    void SetName(string name);

    string GetSurname();
    string GetName();

    Person(string surname, string name);
};

void Person::SetSurname(string surname)
{
    AssertCorrectName(surname);
    _surname = surname;
}

void Person::SetName(string name)
{
    AssertCorrectName(name);
    _name = name;
}

void Person::AssertCorrectName(string name)
{
    for (int i = 0; i < name.length(); i++)
    {
            if (isalpha(name[i]))
            {
                    throw exception("Недопустимые символы в строке");
            }
    }
}

class Teacher : public Person
{
    string _position;
    void AssertPosition(string position);

public:
    void SetPosition(string position);
    string GetPosition();

    Teacher(string surname, string name, string position);
};

void Teacher::SetPosition(string position)
{
    AssertPosition(position);
    _position = position;
}

void Teacher::AssertPosition(string position)
{
    for (int i = 0; i < position.length(); i++)
    {
            if (isalpha(position[i]))
            {
                    throw exception("Недопустимые символы в строке");
            }
    }
```

```
}
```

Внутри базового класса Person содержится закрытый метод валидации имени и фамилии – имя и фамилия не должны содержать запретных символов, знаков препинания и т.д.

Такая же проверка выполняется в дочернем классе преподавателя Teacher для поля должности. Наблюдается явное и нежелательное дублирование кода. Так как метод валидации в классе Person находится под модификатором private, вызвать его внутри класса Teacher не представляется возможным. Перемещение метода валидации под модификатор public будет неправильным, так как это нарушает инкапсуляцию класса Person.

В данном случае необходимо поместить метод валидации базового класса под модификатор protected:

```
class Person
{
    string _surname;
    string _name;

protected:
    void AssertString(string string);

public:
    void SetSurname(string surname);
    void SetName(string name);

    string GetSurname();
    string GetName();

    Person(string surname, string name);
};

void Person::SetSurname(string surname)
{
    AssertString(surname);
    _surname = surname;
}

void Person::SetName(string name)
{
    AssertString(name);
    _name = name;
}

void Person::AssertString(string string)
{
    for (int i = 0; i < string.length(); i++)
    {
            if (isalpha(string[i]))
            {
                    throw exception("Недопустимые символы в строке");
            }
    }
}
```

В таком случае, защищенный метод будет доступен в дочернем классе, и его можно будет вызвать в сеттере должности преподавателя.

```
class Teacher : public Person
{
    string _position;
```

```
public:
    void SetPosition(string position);
    string GetPosition();

    Teacher(string surname, string name, string position);
};

void Teacher::SetPosition(string position)
{
    AssertString(position);
    _position = position;
}
```

При этом, если со стороны клиентского кода мы попытаемся обратиться к защищенному методу через базовый или дочерний классы, то компилятор сообщит об ошибке:

```
void main()
{
    Person* person = new Person("Богородов", "Дмитрий");
    person->AssertString("Строка"); // Ошибка компиляции

    Teacher* teacher = new Teacher("Богородов", "Дмитрий", "Профессор");
    teacher->AssertString("Строка"); // Ошибка компиляции
}
```

Приём с модификатором protected часто применяется с конструкторами, которые тоже могут быть перенесены под protected. Таким образом, вы можете создать конструктор, который смогут вызывать дочерние классы (чтобы избавиться от дублирования при инициализации объектов), но при этом данный конструктор нельзя будет вызвать в клиентском коде.

Таким образом, в объектно-ориентированном программировании существует три модификатора доступа, создающих уникальные области видимости для членов классов:

- private – доступ к членам класса только внутри класса;
- protected – доступ к членам класса только внутри класса и его наследниках;
- public – доступ к членам класса в любой точке клиентского кода;

Модификатор protected даёт некоторую свободу дочерним классам в работе с приобретенной в результате наследования реализацией. Но не следует делать переносить все закрытые поля и методы под модификатор protected в предположении, что это может понадобится в будущем. Модификатор protected следует применять только в тех случаях, когда вы уверены, что: а) у данного класса могут появиться наследники; б) для реализации наследников понадобятся закрытые методы базового класса. Если вы не уверены хотя бы в одном из двух пунктов, оставляйте члены класса под модификатором private. В любом случае, открыть доступ к неиспользуемым вовне членам класса в будущем будет проще, чем закрывать доступ к неправильно используемым членам класса, нарушающим логику работы программы.

Модификаторы наследования

При объявлении дочернего класса вы могли обратить внимание на то, что перед названием базового класса указывается модификатор доступа public:

```
class HourEmployee: public Employee
```

Данный модификатор доступа влияет на модификаторы доступа унаследованных членов класса. Например, если при наследовании указать модификатор доступа public, то все члены базового класса будут унаследованы без изменения своих модификаторов доступа: открытые члены класса останутся открытыми, закрытые – закрытыми, защищенные – защищенными.

Однако, доступность унаследованных членов класса можно изменить. Так, если при наследовании указать модификатор доступа protected:

```
class HourEmployee: protected Employee
```

то все открытые члены базового класса станут защищенными внутри дочернего. То есть, создав в клиентском коде объекты Employee и HourEmployee, то для объекта Employee можно будет вызвать сеттер SetName(), а для объекта HourEmployee вызвать унаследованный сеттер SetName() будет невозможно, так как в классе HourEmployee его модификатор доступа поменяется на protected. При этом закрытые члены класса останутся закрытыми – то есть смена модификатора наследования не может повысить режим доступа к членам базового класса, а только понизить. Тем самым гарантируется обход инкапсуляции базового класса в том варианте, в которой её задумал автор класса.

Если указать модификатор наследования private:

```
class HourEmployee: private Employee
```

то все унаследованные члены класса станут под модификатором доступа private в дочернем классе. Здесь стоит отметить, что те члены класса, которые в базовом классе были под модификатором доступа private – будут унаследованы, но недоступны для вызова в дочернем классе. Те же члены класса, которые в базовом классе были public и protected, станут private внутри дочернего класса и могут быть вызваны внутри дочернего класса.

Модификаторы доступа наследования используются в сложных иерархиях наследования классов, чтобы уменьшить внешние интерфейсы дочерних классов. В подавляющем большинстве случаев используется модификатор наследования public.

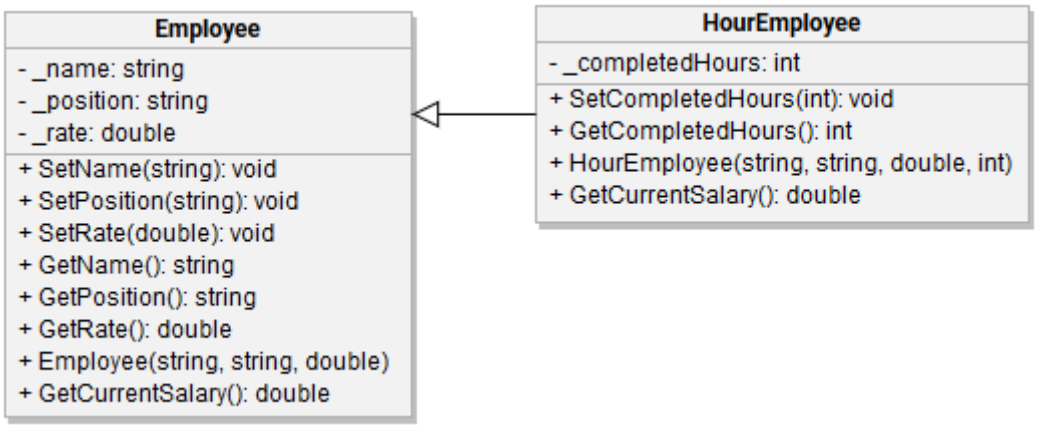
Виртуальные и чисто виртуальные функции

Наследование редко применяется как самостоятельный механизм. На практике наследование часто связано с применением виртуальных и чисто виртуальных функций. Эта комбинация в дальнейшем формирует понятие полиморфизма – одного из ключевых понятий объектно-ориентированного программирования.

Рассмотрим пример. В ранее рассмотренной программе по учету сотрудников существует два класса, описывающих сотрудников – это Employee и HourEmployee. Заказчик попросил добавить функциональность по расчету зарплаты за текущий месяц:

- Обычные сотрудники должны получать зарплату в размере оклада.
- Сотрудники с почасовой оплатой должны получать зарплату на основе размера оклада и отработанных часов. Считается, что сотрудник в месяц должен отрабатывать 168 часов. Следовательно, зарплата сотрудника с почасовой оплатой будет равна количеству отработанных часов, умноженных на стоимость часа работы, где стоимость часа равна окладу, разделенному на 168 часов.

Так как все данные, необходимые для расчета зарплаты сотрудника хранятся в самом сотруднике, и никаких иных сторонних данных для этого не требуется, логично реализовать методы расчета зарплаты внутри классов сотрудников:




```
double Employee::GetCurrentSalary()
{
    return _rate;
}

double HourEmployee::GetCurrentSalary()
{
    // 168 правильнее перенести в константы класса и пояснить смысл значения
    return GetCompletedHours() * GetRate() / 168;
}
```

Но при данной реализации возникает проблема – компилятор сообщает, что в классе HourEmployee уже определен метод GetCurrentSalary().

Объясняется это тем, что после добавления метода GetCurrentSalary() в базовый класс Employee, он автоматически наследуется в дочерний класс HourEmployee. Объявить второй метод с таким же названием внутри класса HourEmployee уже не получиться.

Казалось бы, проблему можно решить простым переименованием метода GetCurrentSalary() во что-то другое, например, GetCurrentSalary(). Но данное решение только маскирует проблему, а не решает её. На самом же деле при переименовании мы получим следующую функциональность:

- В класс HourEmployee все равно наследуется метод GetCurrentSalary(), которого по смыслу класса там быть не должно – работник с почасовой оплатой не должен иметь метода с расчетом зарплаты по окладу.
- Если мы храним всех сотрудников в массиве по указателю на базовый класс Employee, то, обращаясь к объекту HourEmployee через указатель на базовый класс, можно вызвать метод GetCurrentSalary() и рассчитать для сотрудника с почасовой оплатой зарплату на основе оклада.

Ошибку, описанную во втором пункте, допустить очень легко – уже через месяц активной разработки программы вы забудете, что в классе HourEmployee есть два метода для расчета зарплаты, один из которых неправильный. Вероятнее всего, вы будете работать с объектами сотрудников через указатель на базовый класс и, очевидно, будете вызывать тот метод, который увидите через указатель – то есть неправильно считать зарплату части сотрудников. Это может быть трагично для тех сотрудников, что стараются отработать в месяц сверхурочные часы для большей зарплаты.

С точки зрения предметной области, эту проблему можно было бы сформулировать следующим образом: дочерний объект умеет решать ту же задачу, что и родительский объект, но другим способом.

Трактуя проблему с позиции предметной области, необходим некий механизм, который бы позволял дочернему классу переопределять реализацию унаследованных методов, сохраняя сигнатуру метода базового класса. Данный механизм называется ***виртуальными функциями***.

Виртуальные функции – это методы класса, реализация которых может быть переопределена в дочерних классах. В языке Си++ виртуальные функции обозначаются с помощью модификатора virtual:

```
class Employee
{
    string _name;
    string _position;
    double _rate;

public:
    void SetName(string name);
    void SetPosition(string position);
    void SetRate(double rate);

    string GetName();
    string GetPosition();
    double GetRate();

    double virtual GetCurrentSalary();

    Employee(string name, string position, double rate);
};
```

После того, как мы обозначим функцию GetCurrentSalary() в классе Employee виртуальной, её реализацию можно будет поменять в дочернем классе. Дочерний класс не обязан переопределять унаследованную виртуальную функцию – он может использовать готовую реализацию из базового класса. Однако, если необходимо переопределить реализацию, то необходимо:

- объявить в дочернем классе функцию с аналогичной сигнатурой;
- обозначить функцию ключевым словом override (перегрузка, переопределение);
- сделать новую реализацию функции в рамках дочернего класса.

Для класса HourEmployee это будет выглядеть следующим образом:

```
class HourEmployee : public Employee
{
    int _completedHours;

public:
    void SetCompletedHours(int completedHours);
    int GetCompletedHours();

    double GetCurrentSalary() override;

    HourEmployee(string name, string position, double rate, int completedHours);
};
```

Обратите внимание, что переопределенный в дочернем классе метод также называется GetCurrentSalary(), но компилятор не будет сообщать об ошибке – так как методы обозначены ключевыми словами virtual и override, компилятор понимает, что это не два разных метода, а это две реализации одного и того же метода.

Главное преимущество виртуальных функций открывается при их использовании. Теперь, если в клиентском коде создать указатель на базовый класс и поместить в него объект класса HourEmployee, то при вызове метода GetCurrentSalary() через указатель на базовый класс, будет вызываться реализация дочернего класса:

```
void main()
{
    Employee* employee =
            new Employee("Яковлев Д.Б.", "QA Engineer", 55000.0);
    cout << employee->GetCurrentSalary() << endl;
    delete employee;
```

```
    employee =
           new HourEmployee("Морозов Н.А.", "Senior Developer", 110000.0, 202);
    cout << employee->GetCurrentSalary() << endl;
    delete employee;
}
```

В результате работы программы на экране будут показаны рассчитанные значения зарплат:

```
55000
132262
```

Несложно убедиться, что для сотрудника Employee была вызвана реализация класса Employee, а для сотрудника HourEmployee через указатель на базовый класс была вызвана реализация дочернего класса.

То есть, при вызове метода вызываемая реализация будет определяться не на этапе компилирования, а на этапе выполнения программы. Это даёт потрясающую гибкость при разработке приложений.

Так, без виртуальных функций пришлось бы хранить отдельные (независимые) методы для расчета зарплат для разных сотрудников. Если бы понадобилось рассчитать зарплаты для всех сотрудников, то пришлось бы отдельно вызывать расчет зарплат для обычных сотрудников и отдельно для сотрудников с почасовой оплатой – только по тому, что в классах формально разные методы расчета зарплаты. Благодаря виртуальным функциям, можно выполнить расчет зарплат всех сотрудников, хранящихся в едином массиве:

```
double GetAllSalaries(Employee** employees, int employeesCount)
{
    double sum = 0.0;

    for (int i = 0; i < employeesCount; i++)
    {
           sum += employees[i]->GetCurrentSalary();
    }

    return sum;
}
```

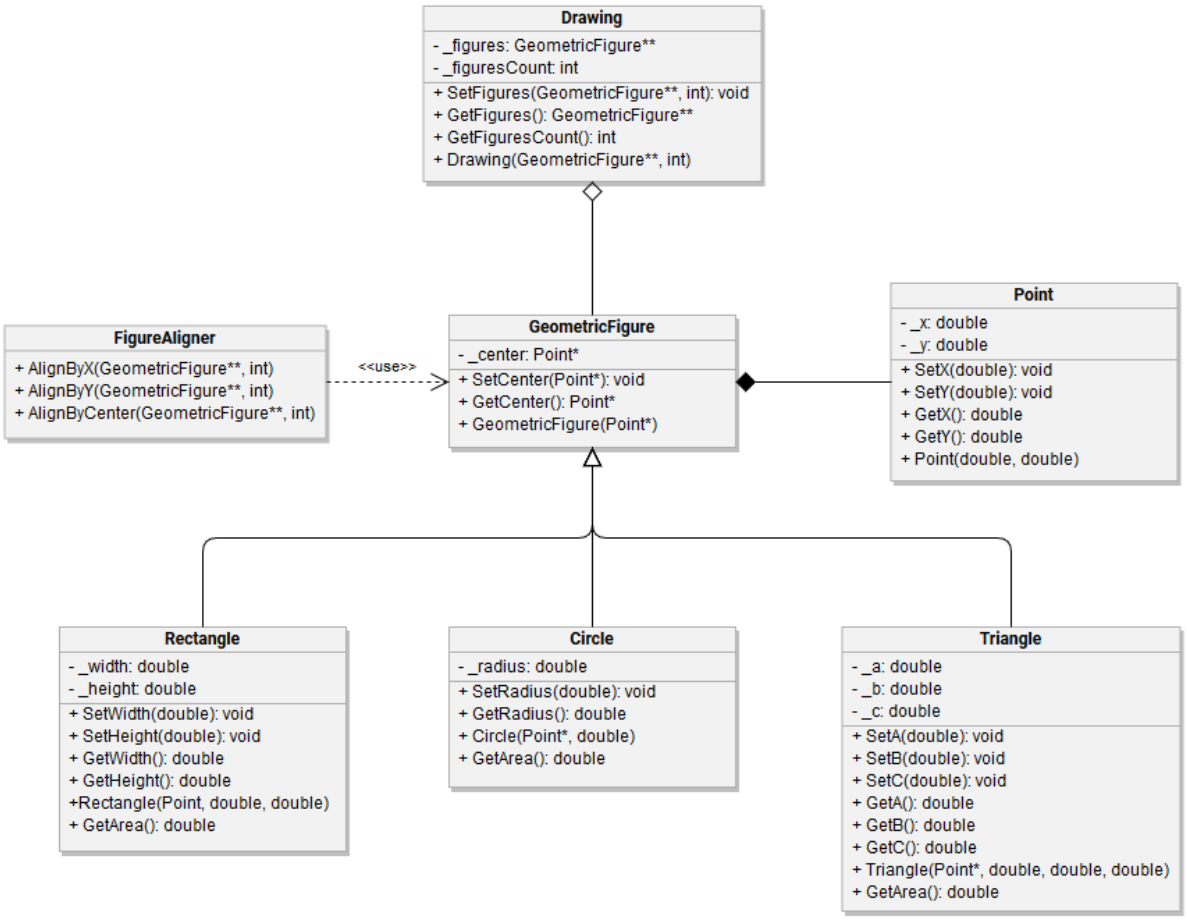
Клиентскому коду теперь без разницы какой именно объект скрывается за указателем на базовый класс – клиентский код будет работать со всеми дочерними классами, даже не подозревая об их существовании. И действительно, в примере выше функция явно знает только о базовом классе и не содержит знания о каком-либо дочернем классе.

Говоря о наследовании виртуальных функций правильнее говорить о ***наследовании интерфейса***, а не реализации. То есть в данном случае наследование выполняется не для того, чтобы приобрести реализацию метода, а для того, чтобы приобрести интерфейс базового класса и спрятать за этим интерфейсом новую реализацию.

Классы с переопределенными виртуальными функциями называются ***полиморфными классами***. Возможность работы с объектами с общим интерфейсом, но без знания о том, какой объект скрывается за интерфейсом, называется ***полиморфизмом***.

Метафорой для полиморфизма может послужить пример с правами на вождение. Автомобили от разных производителей имеют разную реализацию внутри, но общий интерфейс в виде руля, педалей и коробки передач. Благодаря общему интерфейсу, получив водительские права, вы, как "клиент" автомобиля, можете управлять любым автомобилем, не зная о его конкретной реализации. Это и называется полиморфизмом. Без полиморфизма вам бы пришлось получать отдельные права для каждого автомобиля.

Чисто виртуальные функции. Рассмотрим другой пример. В программе для построения чертежей имеются классы геометрических фигур. В основе чертежей лежат примитивы, например, прямоугольники, круги и треугольники. Для всех трёх примитивом существует базовый класс геометрической фигуры GeometricFigure, определяющий общую функциональность. Также у каждого примитива есть свой собственный метод расчета площади фигуры GetArea().



Благодаря общему интерфейсу можно создать сервисный класс FigureAligner (align – "выровнять"), выполняющий выравнивание любых геометрических фигур по X, по Y или по центру – приём, часто требующийся в различных САПР, 2D- и 3D-редакторах, или при создании диаграмм.

В программе также присутствует класс Drawing (чертеж), который хранит массив всех фигур, из которых состоит чертеж.

Заказчик попросил добавить в программу функцию, которая выводила бы на экран общую площадь всей детали, нарисованной на чертеже. Для этой задачи можно добавить метод в класс Drawing (так как он хранит все примитивы чертежа).

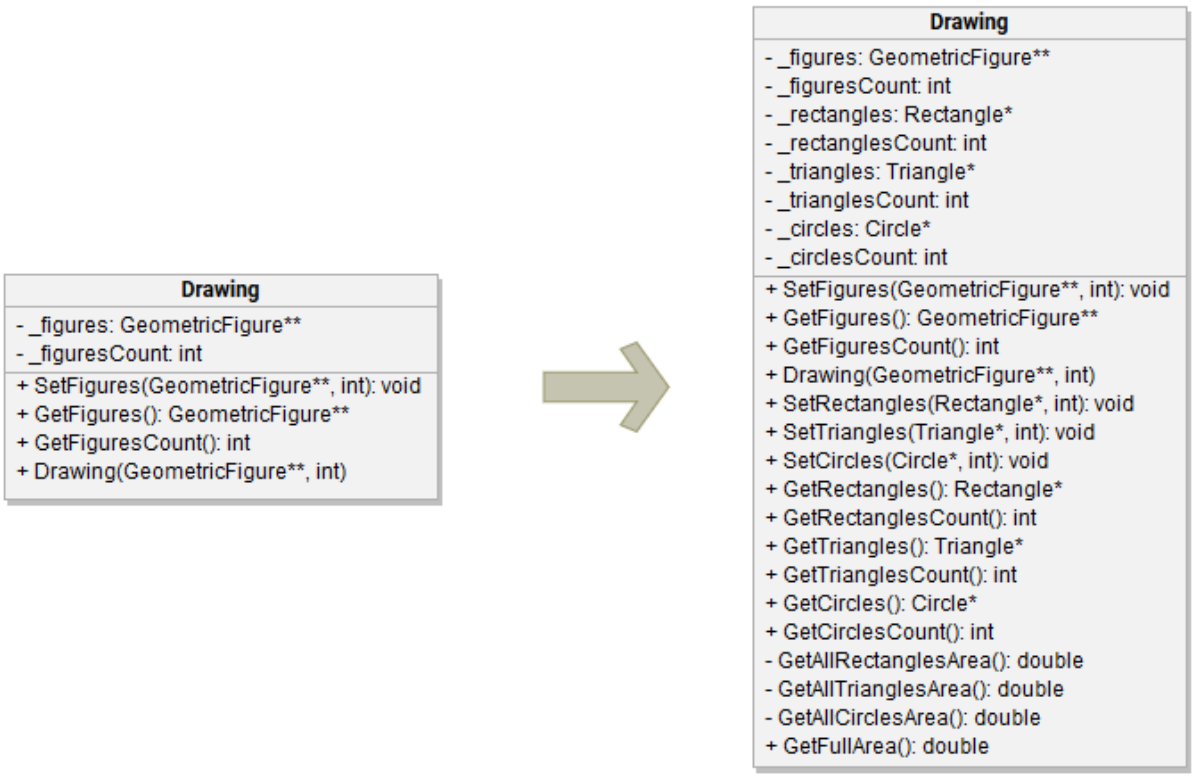
Если не применять механизм виртуальной функции, то каждый примитив имеет собственную реализацию расчета площади, ***не являющуюся частью общего интерфейса***. Это значит, что когда в системе необходимо сделать функцию, затрагивающую разные классы в одной иерархии наследования, то придется работать с каждым типом объекта по отдельности. В нашем случае, это означает, что нужно будет считать отдельно площади для прямоугольников, кругов и треугольников, а затем складывать получившиеся суммы вместе. Работа с отдельными примитивами раздельно также означает и их раздельное хранение – ведь сейчас чертеж хранит все фигуры в одном массиве через указатель на базовый класс. А так как в базовом классе нет метода расчета площади, то и вызвать расчет площади ни для одной фигуры невозможно.

Следовательно, (без механизма виртуальных функций) в программе необходимо сделать следующие изменения:

- добавить в класс Drawing три новых массива под каждый тип геометрических фигур;
- при создании новых фигур в программе, примитивы нужно будет поместить как в общий массив всех примитивов, так и в массив для конкретного типа (если не поместить в общий массив, то перестанет работать выравнивание; а если не поместить в конкретный, то не получиться реализовать расчет площади);
- при удалении фигур с чертежа, нужно будет удалять их из двух массивов;

- в класс Drawing добавить закрытые методы расчета площади для каждой отдельной геометрической фигуры;
- в класс Drawing добавить открытый метод, выполняющий расчет общей площади чертежа с вызовом закрытых методов для отдельных фигур.

После модификации класс Drawing будет выглядеть следующим образом:



Особенно стоит обратить внимание на реализацию методов по расчету площадей:

```
double Drawing::GetAllRectanglesArea()
{
    double sum = 0.0;

    for (int i = 0; i < _rectanglesCount; i++)
    {
            sum += _rectangles[i].GetArea();
    }
}

double Drawing::GetAllTrianglesArea()
{
    double sum = 0.0;

    for (int i = 0; i < _trianglesCount; i++)
    {
            sum += _triangles[i].GetArea();
    }
}

double Drawing::GetAllCirclesArea()
{
    double sum = 0.0;
```

```
    for (int i = 0; i < _circlesCount; i++)
    {
            sum += _circles[i].GetArea();
    }
}

double Drawing::GetFullArea()
{
    double sum = GetAllRectanglesArea();
    sum += GetAllTrianglesArea();
    sum += GetAllCirclesArea();
    return sum;
}
```

Очевидно, что без механизма виртуальных функций приходится создавать большое количество побочного дублирующегося кода. Кроме этого, если в будущем заказчику понадобится добавить в программу новый примитив, например, овал, то программа потребует дополнительного дублирования кода в самых разных частях программы.

Чтобы избавиться от дублирования кода при хранении и обработке геометрических фигур, необходимо добавить виртуальный метод в базовый класс GeometricFigure. И здесь мы обнаружим проблему: в базовом классе нет данных о форме фигуры, а, следовательно, сделать реализацию метода GetArea() в базовом классе невозможно. Невозможно рассчитать площадь, зная только центр геометрической фигуры и не знать её размеров и формы. Другими словами, сущность Геометрическая фигура ***слишком абстрактна***. Мы знаем, что у любой закрытой геометрической фигуры есть площадь (то есть сам метод может быть в базовом классе), но нет универсального решения расчета площади для любых фигур (то есть, у метода не может быть реализации в обобщенном классе).

Можно применить искусственный приём, и сделать ложную реализацию метода GetArea() в классе GeometricFigure(), которая будет возвращать 0 или отрицательное значение:

```
double virtual GeometricFigure::GetArea()
{
   return 0;
}
```

Но такое решение неправильное по ряду причин. Ключевая из них заключается в том, что если разработчик новой геометрической фигуры, например, овала, не переопределит в дочернем классе метод GetArea(), то все овалы в программе будут рассчитываться с нулевой площадью.

Другими словами, необходим механизм, позволяющий определить ***в базовом классе только интерфейс*** метода, но не создавать его реализацию, и при этом ***обязать дочерний класс*** добавить собственную реализацию метода. Таким механизмом являются чисто виртуальные функции.

Чисто виртуальные функции – это виртуальные функции, которые не имеют реализации в базовом классе.

Синтаксис чисто виртуальных функций выглядит следующим образом:

```
double virtual GetArea() = 0;
```

Объявление чисто виртуальной функции выполняется внутри объявления самого класса, т.е. в заголовочном файле. Фактически, разработчику вместо добавления реализации в фигурных скобках необходимо присвоить функции значение 0. В *.cpp файле класса (или в любой другой части проекта) дополнительно добавлять реализацию метода не нужно.

Чисто виртуальные функции вводят дополнительные ограничения на использование объектов классов. Например, до создания чистой виртуальной функции, в программе можно было создавать экземпляры класса GeometricFigure. Разумеется, с точки зрения пользователя, создавать внутри чертежа некоторую абстрактную фигуры без формы нет никакой необходимости – реальный чертеж должен состоять строго из

определенных фигур заданного размера и формы. Однако код программы позволял создавать объекты базового класса. Но что должно произойти, если создать объект базового класса и попытаться вызвать для него чисто виртуальную функцию без реализации?

Логично, что программа не может самостоятельно принимать решение о том, что нужно делать в ситуации, когда разработчик не указал реализацию метода. Для программы эта ситуация схожа с ситуацией обращения к переменной, значение которой равно nullptr. Для программы эта ситуация некорректна, и должна завершится ошибкой.

По этой причине, если в классе есть чисто виртуальные функции без реализации, то компилятор ***запрещает создавать экземпляры*** данного класса. Классы, имеющие хотя бы одну чисто виртуальную функцию без реализации, называются ***абстрактными классами***. Абстрактные классы предназначены для определения общего интерфейса для дочерних классов, но создание объектов базового класса становится недоступным.

Стоит отметить, что ***использование указателей на абстрактный класс*** остается возможным. В этом указателе могут храниться объекты любого дочернего класса, в котором определена реализация для всех чисто виртуальных функций.

Если дочерние классы не определяют реализацию унаследованных чисто виртуальных функций базового класса, то они автоматически становятся абстрактными. То есть, создание объектов данного класса также становится невозможным. Это становится хорошей защитой от ситуаций, когда разработчик забыл переопределить функции базового класса.

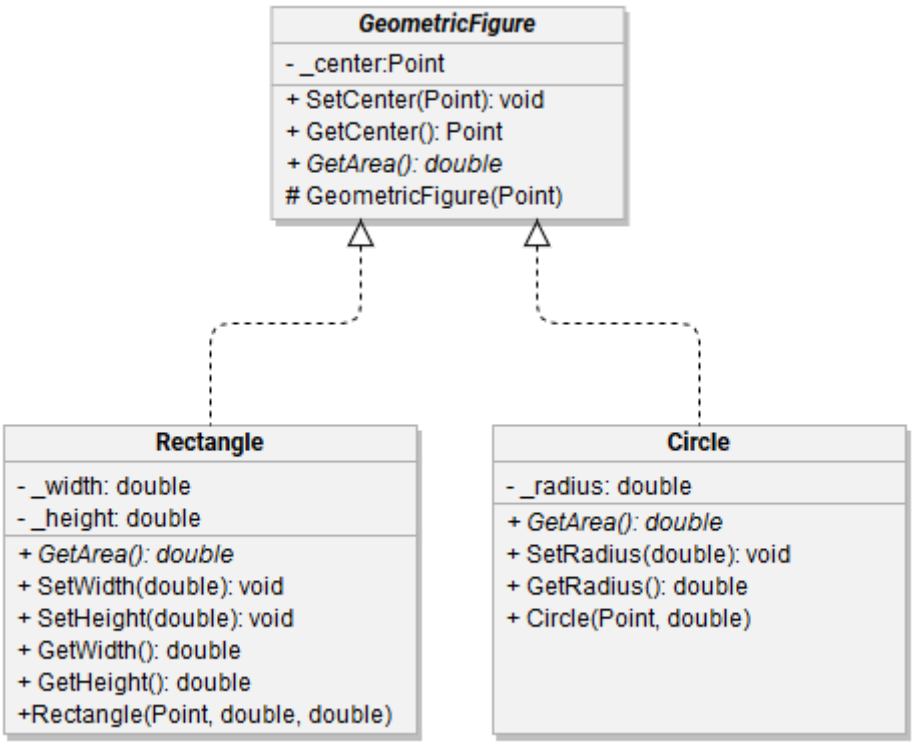
Последнее важное определение данного раздела, это интерфейс. ***Интерфейс*** – это класс, содержащий только чисто виртуальные функции. То есть, интерфейс как базовый класс не задает никакой реализации для методов, и предназначен для использования в качестве указателя на базовый класс и предоставления доступа к реализациям дочерних классов. Другими словами, интерфейс предназначен для реализации полиморфизма в программе. В отличие от интерфейсов, просто абстрактные классы могут помимо виртуальных методов хранить поля и обычные методы. Как правило, это поля и методы, являющиеся общей реализацией для всех дочерних классов. Таким образом, абстрактные классы позволяют избавиться от дублирования кода, необходимого ***для всех*** наследников. Если реализация хотя бы одного наследника полностью отличается от реализации остальных дочерних классов, то необходимо применять интерфейс.

Полиморфизм на диаграммах классов

В понятиях диаграмм классов нет связи полиморфизма. Однако диаграммы классов различают наследование с целью приобретения готовой реализации и наследование с целью переопределения (реализации) виртуальных функций, или, точнее, реализации приобретенного интерфейса.

Как было показано ранее, наследование с целью приобретения готовой реализации обозначается сплошной линией с закрытой стрелкой, направленной от дочернего к базовому классу.

Наследование с целью реализации приобретенного интерфейса, или проще говоря, связь реализации обозначается пунктирной линий с закрытой стрелкой, направленной от дочернего к базовому классу.



Сразу стоит сказать, что виртуальные и чисто виртуальные функции обозначаются курсивом. Если класс содержит хотя бы одну чисто виртуальную функцию, он называется абстрактным классом. Названия абстрактных классов на диаграммах также обозначаются курсивом.

У читателя может возникнуть вопрос – зачем вводить разные термины "полиморфизм" и "реализация" для обозначения одной и той же механики? Но на самом деле, понятия полиморфизма и реализации отличаются друг от друга.

Полиморфизм – это не само наследование от абстрактного класса с последующей реализацией интерфейса, а это ***использование*** объектов с одинаковым интерфейсом без знания о том, какой именно объект скрывается за интерфейсом. Связь реализации в свою очередь показывает само ***наследование*** интерфейса с последующим его переопределением. Другими словами, понятие полиморфизма на диаграмме классов будет скрываться не за отдельной связью "реализации", а будет скрываться за связью использования/агрегирования между клиентским классом и классом-интерфейсом. Понимание разницы между этими двумя терминами поможет в понимании самого полиморфизма как инструмента, применяемого для решения практических задач.

Интерпретация наследования

Аналогично агрегированию, наследование выражает определенную связь между объектами реального мира. Так как вся объектно-ориентированная концепция старается создать структуру программы, соответствующую реальной предметной области, важно понимать, какие связи реальных объектов можно выражать в программе в виде наследования классов, а какие связи нельзя. Как и любой инструмент, наследование предназначено для решения определенных задач. Применение наследования для задач ему не свойственных только навредит программе и усложнит её разработку.

> **Почему это важно? Наследование – это самая сильная связь между классами, которая только может быть. Поведение дочерних классов при наследовании зависит от базового сильнее, чем поведение класса-контейнера от класса-части при композиции/агрегации. Изменение в поведении базового класса автоматически приводит к изменению поведения дочерних классов, а, следовательно, разработчику необходимо проверить правильность работы как базового класса, так и дочерних. Как результат, даже небольшие изменения в классе могут привести к большим изменениям в самых разных частях программы.**

В литературе часто можно найти интерпретацию наследования как ***частного случая***, т.е. дочерний класс является частным случаем базового класса. Однако такая интерпретация вредна, так как может истолкована не верно. Правильной интерпретацией наследования стоило бы называть расширением базового класса, т.е. дочерний класс ***расширяет функциональность*** базового класса.

Например, дом с гаражом расширяет функциональность обычного дома. То есть, он выполняет все функции обычного дома, но при этом ***добавляет новые функции*** в виде гаража.

Другой пример, это работник с почасовой оплатой. Он также выполняет все функции обычного сотрудника, но расширяет данные о нём с помощью ***добавления новых данных*** о себе – количестве отработанных в месяц часов.

Однако примером неправильной интерпретации может служить наследование квадрата от прямоугольника. Из курса школьной геометрии известно определение квадрата как ***частного случая*** прямоугольника с равными сторонами. Изучив наследование с неправильной интерпретацией, начинающие разработчики часто применяют наследование в аналогичных случаях, проводя параллель с понятием "частный случай" из повседневного использования. Однако если рассмотреть пример внимательнее, то мы обратим внимание, что для описания прямоугольника необходимо знание его ширины и высоты, а для описания квадрата необходимо знание только его стороны.

Другими словами, наследуя квадрат от прямоугольника, мы ***уменьшаем количество знаний*** об объекте. Однако в исходном коде это будет означать, что квадрат для своей работы использует не все унаследованные поля и методы.

> **Уменьшение знаний об объекте или его функций противоречит назначению наследования.**

Начинающему разработчику может показаться не критичным ситуация, при которой дочерний класс не нуждается в части наследованной функциональности. Однако использование подобного класса создает сложности из-за неоднозначности интерпретации его работы.

Например, разработчик квадрата может при наследовании может поступить двумя способами:

1) работать только с полем ширины, игнорируя в коде поле высоты;
2) искусственно менять значение ширины при изменении высоты и наоборот, т.е. при изменении любого из полей автоматически изменять значение второго поля.

Однако работая с геометрическими фигурами через указатель на базовый класс, разработчику клиентской части будет неочевидно поведение программы в такой ситуации:

```
Rectangle* rectangle;
...
// в указателе может храниться как прямоугольник, так и квадрат
rectangle->SetWidth(5);
rectangle->SetHeight(8);
```

Клиентский код вызывает методы установки длины и ширины для объекта в указателе. По завершению этого кода разработчик будет ожидать, что в полях хранятся разные значения. Какое из этих значений считать истинным для размера стороны квадрата? Если спросить разработчика класса квадрата, то, разумеется, он объяснит реализацию класса. Или можно посмотреть в исходный класса. Но проблема в том, что разработчик клиентской части считает вышеописанный код корректным, и он не подумает о том, что надо уточнить реализацию дочерних классов у других разработчиков! Зачем спрашивать другого разработчика, если ты не знаешь о существовании потенциальной проблемы?

> **По этой причине, допустима только одна интерпретация наследования – это расширение обязанностей класса. Если же частный случай приводит к уменьшению обязанностей, применение наследования с высокой вероятностью приведет к проблемам в работе программы.**

Другое важное правило: наследование должно применяться только для тех объектов, которые в реальном мире описываются аналогичной связью. Наследование не должно применяться для тех объектов, между которыми нет аналогичной связи в реальном мире.

Например, базовый класс человека может содержать поля имени и даты рождения. Вполне допустимо, чтобы наследниками такого класса были студент и преподаватель – у каждого из них есть имя и собственная дата рождения. Неправильным применением будет наследование домашнего животного от человека.

Некоторые начинающие разработчики применяют наследования исходя не из реального смысла объектов, а исходя из подобия полей и методов внутри классов. Так, у домашнего животного, как и у человека, есть имя и дата рождения. Видя одинаковые данные в двух классах, начинающие разработчики не стесняясь могут унаследовать домашнее животное от человека. В будущем, при модификации базового класса может случиться, что у домашних животных появится работа, имущество, паспортные данные и т.п. При этом разработчик, добавляющий новые данные в базовый класс, может даже не подозревать о том, что у класса человека может быть наследник в виде домашнего питомца. Проблема заключается как раз в интерпретации классов – если в реальном мире два объекта не связаны наследованием, то разработчик не ожидает такой связи и в исходном коде программы. Поиск ошибок в такой программе или её модификация могут стать очень долгим процессом.

Другой пример неправильного применения наследования – наследование от базового класса, основываясь только на типах данных полей. Данный пример встречается у студентов, по этой причине он приведен здесь. Например, разработчику необходимо создать класс данных о пользователе сайта, в которых будет указаны фамилия, имя и дата рождения (два строковых поля и поле типа даты соответственно). Однако разработчик знает, что в системе уже есть класс аккаунта, в котором также есть два строковых поля и поле типа даты. Только в классе аккаунта эти поля соответствуют логину, паролю и дате регистрации на сайте. Видя, что типы данных совпадают, а, возможно, совпадают даже проверки внутри сеттеров, разработчик наследует класс пользователя от существующего класса аккаунта. Помимо того, что унаследованные поля не совпадают по имени с теми именами, которые требуется (поле фамилии теперь называется _login, а дата рождения – registrationDate), проблема заключается в том, что при наследовании кардинально меняется смысл унаследованных полей. Однако эта проблема не волнует студентов, руководствующихся принципом "оно же работает", и продолжающих писать плохой код.

> **При наследовании в дочернем классе не должен меняться смысл полей и методов базового класса**

Другими примерами изменения смысла полей в дочерних классах может служить изменение единиц измерений. Например, в базовом классе товара стоимость указана в долларах, а в классе-наследнике стоимость уже подразумевается в рублях. Этот пример особенно коварен, так как тип данных и имя поля соответствуют его содержанию – вещественное поле стоимости Cost. Но поиск ошибки в коде может потребовать значительного времени. Особенно это критично, если такая ошибка попадет в готовую программу, и у конечного пользователя спишется неправильная сумма денег. Это приведет к потере репутации разработчиков, денежным компенсациям и, с высокой вероятностью, закрытию компании.

> **Если поле класса описывает величину в единицах измерения – единицы измерения не должны меняться в дочерних классах**

Последнее правило правильного применения наследования звучит так:

> **Базовый класс не должен знать о существовании дочерних классах или о данных и методах, относящихся исключительно к дочерним классам.**

У базового класса может быть не один дочерний класс, а несколько. Более того, как будет показано на примере полиморфизма в последующих разделах, чаще всего базовые классы предназначены для многократного наследования.

Однако вовремя рефакторинга при попытке выделить базовый класс из существующих разработчик может совершить ошибку, и перенести в базовый класс те поля, которые должны оставаться в дочерних классах.

Например, в примере базового класса человека для преподавателя и студента, разработчик может перенести в базовый класс поле номера студенческого билета и соответствующие геттер и сеттер. Как результат, при наследовании от базового класса, у преподавателя также появляется номер студенческого билета, что, разумеется, ошибка реализации. Если базовый класс описывает некоторого абстрактного человека, у такой сущности не может быть студенческого билета, так как это данные, которыми обладает не каждый объект данного класса.

Часто данная проблема комбинируется со сменой смысла поля. Например, вместо того, чтобы исправить базовый класс и убрать поле студенческого билета, разработчик пытается выкрутиться и, либо делает поле у преподавателя всегда равным нулю, либо меняет его смысл на номер пропуска. Фактически, разработчик маскирует одну ошибку другой, что категорически неправильно.

По этой же причине неправильно, если в методах базового класса используются объекты дочернего класса. Базовый класс не должен ничего знать о дочерних классах. В случае языка Си++ использование объектов дочернего класса можно отследить по подключению через директиву include файлов с дочерними классами.

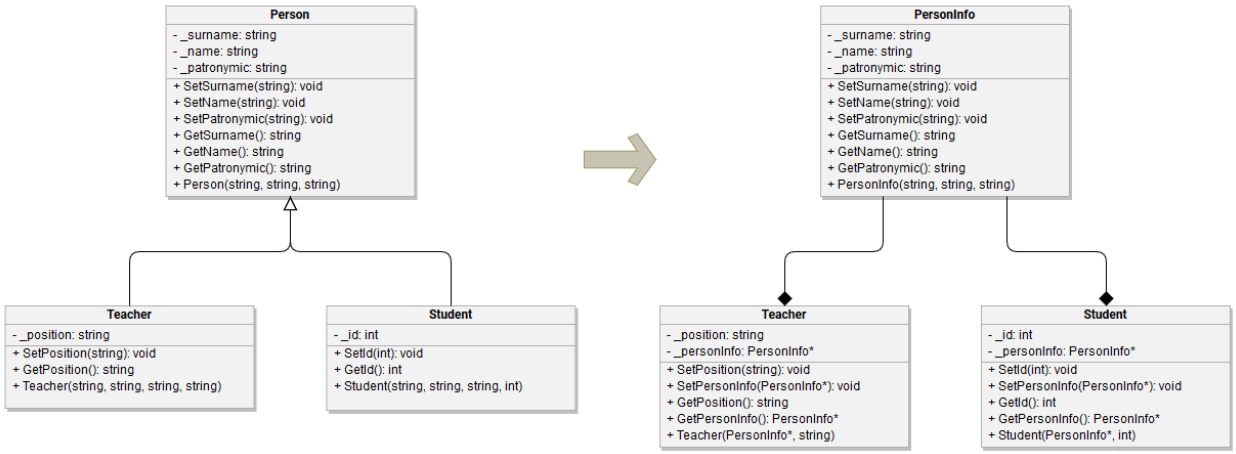
Все примеры данного раздела подразумевают применение наследования как самостоятельного механизма языка программирования. На практике наследование применяется для создания полиморфизма, а точнее наследования интерфейса базового класса, а не его реализации.

> **В случае создания полиморфных объектов, наследование применяется не для расширения функциональности, а для приобретения интерфейса, описанного в базовом классе.**

Используя наследование, необходимо четко понимать, для какой цели вы применяете этот механизм – приобретение готовой реализации или приобретение интерфейса. Понимание цели позволит избежать ошибок при разработке программы.

Сравнение наследования и агрегирования

Как было сказано ранее, наследование – это самая сильная связь между классами. По этой причине её использование должно быть осторожным. На практике, если это возможно, связь наследования стараются заменить на агрегирование. Например, наследование преподавателя и студента от общего класса человека можно заменить по следующей схеме:



Теперь данные о человеке хранятся в классе PersonInfo, который композируется обоими классами. Данная структура позволяет хранить те же самые данные о студентах и преподавателях, но при этом уменьшает связность между классами – модификация класса PersonInfo с меньшей вероятностью приведет к изменению классов Teacher и Student, в отличие от реализации с наследованием.

Обратите внимание, что PersonInfo содержит все те же поля и методы, что и класс Person. Зачем же понадобилось переименовывать класс Person в PersonInfo?

Дело заключается именно в правильной интерпретации связей между классами с их реальными объектами в предметной области. Так, композиция подразумевает, что "один объект является частью другого объекта" или "один объект хранит в себе другой объект". Если применить данную трактовку к нашему примеру, то имеем два разных пояснения предметной области. Просто сравните эти две формулировки:

1) Студент всегда хранит в себе человека.
2) Студент всегда хранит в себе информацию о человеке.

Вторая формулировка понятна для любого слушателя, будь то разработчик, заказчик или пользователь, которому объясняют как связаны данные внутри программы. Первая же формулировка вызывает много вопросов, в первую очередь, физиологических. Другими словами, переименование класса обусловлено правильным пониманием предметной области для всех участников процесса разработки.

По существу, есть несколько причин, по которым не следует заменять наследование агрегированием:

1) для сохранения инкапсуляции класса необходимо использование модификатора protected: при агрегировании невозможно предоставить уникальный доступ к отдельным методам агрегируемого класса, в то время как модификатор доступа protected позволяет создать уникальную область видимости членов класса, ограниченную иерархией наследования.
2) использование наследования для реализации полиморфизма: заменить полиморфизм на любую другую связь, при этом не потеряв значительно в лаконичности исходного кода, невозможно.

Запрет наследования от класса

Есть ряд причин, по которым стоит запрещать возможность наследования от некоторых классов:

1) Не дать другому разработчику принять неправильное архитектурное решение. Ранее было сказано, что наследование как самостоятельный от полиморфизма механизм стоит применять осторожно. В некоторых случаях в архитектуре напрашиваются "легкие" решения с помощью наследования, однако в конечном счете это может привести к проблемам разработки. К сожалению, привести конкретные примеры таких ситуаций в рамках данного учебного курса затруднительно.
2) Запретить наследование от классов, находящихся в поставляемых библиотеках. Помимо разработки конечных продуктов, существуют компании, которые занимаются разработкой библиотек и инструментов для других разработчиков ПО. Поставляя библиотеки по лицензии, вы даете возможность другим разработчикам использовать ваши классы. Однако получая доступ к вашим классам, разработчики других команд могут "подсмотреть" реализацию классов с помощью простого наследования. Так, например, можно узнать защищенные члены ваших классов. Фактически, возможность наследования создает лазейку для изучения чужой интеллектуальной собственности. Впрочем, программисты придумали много других, более эффективных способов к изучению работы чужих библиотек.
3) Помешать другому разработчику нарушить правильность работы вашей программы. Опять же, поставляя библиотеки сторонним разработчикам, необходимо гарантировать, что они не вмешаются в работу ваших классов. Например, один из приёмов, применяемых в разработке, это создание полиморфных классов с ***защищенными виртуальными*** методами. Это интересный приём, его можно часто встретить в библиотеках с элементами пользовательского интерфейса. Однако, если сторонний разработчик выполнит наследование от вашего класса, переопределит защищенные методы и передаст их в использование вашим же классам (например, по указателю на базовый класс), то это может нарушить работу всей программы.

Запрет наследования от определенных классов решает вышеописанные проблемы.

Для запрета наследования от класса необходимо написать ключевое слово final при объявлении класса:

```
class Employee final
{
...
};
```

В результате при попытке наследования от класса Employee компилятор сообщит об ошибке. Модификатор final может применяться не только для базового класса, но и для наследников. Например, указание модификатора final у класса HourEmployee вместо класса Employee позволит другим разработчикам создавать новых наследников от Employee, но запретит создавать дочерние классы от HourEmployee.

Однако, не стоит запрещать наследование во всех классах. Как и само наследование, запрет наследования должен применяться обдуманно.

Наследование, полиморфизм и статические классы

В предыдущих разделах, где рассматривалось создание статических полей, методов и классов упоминалось, что статические классы используются в ограниченном количестве случаев, и их использование нежелательно.

Еще одной важной причиной для ***неиспользования*** статических методов и классов заключается в том, что статические методы и классы не могут быть полиморфными. Работа полиморфизма заключается в работе с некоторым объектом, для которого неизвестен его истинный класс, но известен его интерфейс для работы с этим объектом. Применение статического метода предполагает вызов метода через имя конкретного класса, а не переменной с объектом. Статический метод всегда вызывается от лица определенного класса.

Таким образом, если часто использовать статические методы и классы, вы не даете себе использовать наследование и полиморфизм – два мощнейших инструмента объектно-ориентированного программирования, позволяющих в разы уменьшить количество исходного кода и сделать систему более гибкой.

Общий принцип составления и оформления диаграмм классов

В завершение курса объектно-ориентированного программирования хотелось бы представить принцип составления диаграмм классов. Как показал многолетний опыт преподавания, у студентов возникают затруднения с тем, как правильно составить диаграмму классов по готовому исходному коду, например, для пояснительной записки в курсовой работе или выпускной квалификационной работе. В реальной работе разработчика данный навык важен для изучения чужого кода и сторонних библиотек – чтобы понять, как устроен чужой код, лучше всего составить его диаграмму классов. Видение архитектуры чужого кода или библиотеки в виде одной или нескольких небольших диаграмм позволяет лучше понять работу кода, чем изучение отдельных классов в виде исходного кода.

Речь идет о составлении диаграммы именно по готовой программе, так как составление диаграммы для проектирования системы – это совершенно другая задача, требующая от разработчика навыков проектирования объектно-ориентированных систем, а также так называемых паттернов проектирования. В данном же разделе рассматривается составление диаграмм по исходному коду.

На самом деле, процесс составления диаграмм предельно прост и состоит из следующих шагов:

1) составить карточки классов с полями и методами;
2) определить существующие связи между классами;
3) если между двумя классами несколько связей, то оставить наиболее сильную связь или наиболее значимую с точки зрения архитектуры;
4) расположить классы на диаграмме, желательно соблюдая иерархию классов;
5) выровнять размеры и расположение классов по сетке;
6) расставить на диаграмме поясняющие комментарии, если нужно.

Теперь подробно о каждом шаге.

Составить карточки классов с полями и методами. Необходимо открыть все файлы исходного кода и для каждого пользовательского типа данных создать карточку. Для классов карточка делится на три части, где в верхней части указано название, в центральной – поля, и в нижней – методы. Названия полей и методов на карточке должны соответствовать названиям из исходного кода, также необходимо правильно указать модификатор доступа и тип данных. Статические поля и методы подчеркиваются, виртуальные методы пишутся курсивом. Работа по составлению карточек классов чисто механическая, вам нужно просто пролистывать исходный код и записывать в карточку каждый новый член класса, который вы видите в коде. В случае Си++ это делать удобно, так как все члены класса уже удобно выписаны в заголовочных файлах (*.h) – конечно, если вы правильно оформляете код.

Определить существующие связи между классами. Именно определение правильных связей для классов создает наибольшую трудность у студентов. Опытный разработчик выполняет шаги 2 и 3 машинально – он просто сразу рисует правильную связь между классами. Но если вы затрудняетесь сразу определить ключевую связь между двумя классами, и не можете понять, а есть ли связь между классами вообще, то следуйте следующему алгоритму.

Для того, чтобы определить связи класса с другими классами, необходимо просматривать код класса сверху вниз. Ни в коем случае нельзя определять связь на основе указанных вверху директив include – они могут быть ложными, подключать файлы, которые не используются классом. Просматривайте код самого класса, а не директивы в начале файла.

При просмотре кода класса, обращайте внимание на следующие элементы:

- Если при объявлении класса в строке через двоеточие указано имя другого класса, значит поставьте между этими классами связь наследования;
- Если имя другого класса указано как тип данных какого-либо поля, значит это связь агрегации или композиции:
    - если объект для поля создается в конструкторе этого же класса, а уничтожается в деструкторе – это композиция;
    - если объект для поля создается непосредственно перед конструктором этого класса (для этого надо будет найти все участки кода, где используется данный класс) – то это также композиция;
    - в остальных случаях можно считать, что это агрегация;
- Если определили связь агрегации или композиции, необходимо также определить кратность связи:
    - если в классе только одно поле другого класса, ставьте кратность "1 к 1";
    - если в классе несколько полей другого класса, ставьте кратность "1 к n", где n – количество полей;
    - если поле является массивом или указателем на массив, ставьте кратность "1 к n" или "1 к *", где n – количество элементов массива (если оно строго задано), а * - это любое количество элементов (если оно неизвестно);
- Если имя другого класса указано в качестве входных аргументов метода, выходного типа данных или встречается внутри самого метода, значит это связь использование.
- Если метод является перегрузкой виртуального метода, значит это связь реализации (полиморфизм).

После просмотра всего кода класса, вы определите все связи с другими классами. Необходимо повторить для всех классов.

Оставить наиболее сильную и наиболее значимую связь между классами. В результате просмотра классов на предыдущем шаге вероятно, что между двумя произвольными классами может быть более одной связи. На диаграммах классов не принято рисовать между классами более одной связи, поэтому надо оставить только одну. Оставляйте наиболее сильную связь. По силе связи располагаются в следующем порядке:

- наследование (самая сильная);
- композиция;
- агрегация;
- реализация;
- использование (самая слабая).

Однако, если в классе есть перегрузка виртуальных функций, и при разработке класса вы применили наследование для создания полиморфизма, то вместо наследования на диаграмме следует показать связь реализации. Так как полиморфизм нельзя реализовать без наследования, то связи наследования и реализации на диаграммах определяются исходя из причин разработки класса.

Расположить классы на диаграмме, соблюдая иерархию. Все последующие шаги, включая этот, студенты выполняют крайне редко, из-за чего их диаграммы сложно воспринимаются. Любая информация воспринимается лучше при её структурировании и оформлении. Информация без структуры и без оформления воспринимается плохо. Более того, плохо оформленная презентация создает ощущение, что разработка самой программы выполнялась "на коленке", и программа не очень надежная. Чтобы не портить впечатление о вашем проекте и о вас как разработчике, потратьте несколько минут для аккуратного оформления диаграммы.

С точки зрения иерархии следует соблюдать простое правило – диаграмма читается сверху вниз от высокоуровневых классов к низкоуровневым. В случае альбомного расположения листа (или слайда презентации) также можно располагать слева направо. Человек привык читать информацию сверху вниз, в том числе диаграммы, графики, таблицы и т.п. Поэтому вверху диаграммы располагаются те классы, которые агрегируют/композируют остальные или те классы, от которых происходит наследование/реализация. Далее дерево агрегирования или наследования спускается вниз. Если класс использует другой, более низкоуровневый

класс (например, точка будет более низкоуровневой по отношению к геометрической фигуре), то более низкоуровневый класс имеет смысл нарисовать ниже первого. Если класс использует аналогичный по уровню класс (например, преподаватель и студент могут считаться на одном уровне в иерархии классов), то их стоит рисовать на одном уровне и на диаграмме.

Как правило, в начале на диаграмме рисуются иерархии агрегирования и наследования, а затем располагаются используемые классы, например, сервисные. Сервисные классы можно рисовать на одном уровне с классами, которые они используют, но на уровень ниже классов, которые используют этот сервисный класс.

Если диаграмма немного не вмещается в размеры страницы, допустимо размещение отдельных классов не согласно иерархии, а в доступном для этого месте. Если же диаграмма значительно не вмещается в размеры страницы (например, на страницу А4 отчета), то разместите и распечатайте диаграмму на листе большего размера (А3 и более). Не надо делать диаграмму мелкой. Помните, что диаграмма в первую очередь делается для комфортного изучения программы, а не просто для отчетности. Если вам самому некомфортно читать диаграмму, то попытайтесь его переделать таким образом, чтобы было комфортно.

Кроме того, если расположение классов на диаграмме создает большое количество длинных линий связей с множеством перегибов, попробуйте расположить классы таким образом, чтобы линии стали проще. В конечном счете, линии связей имеют не меньшее, а, возможно, и большее значение на диаграммах.

Иногда на диаграммах бывают небольшие классы или перечисления, которые связаны линиями практически со всеми остальными классами программы. Например, в САПР или 3D-редакторах многие классы будут связаны с классом точки Point. Простой класс, хранящий три координаты {X, Y, Z} приведет к большому количеству связей на диаграмме и её станет некомфортно читать. Если у вас на диаграмме появляется такой элементарный класс, как точка или время, от которого исходят десятки связей к остальным классам, то такой класс лучше убрать с диаграммы, и пояснить его отдельно после диаграммы. Аналогичная ситуация может быть с перечислениями.

Выровнять размеры и расположение классов. Несколько простых операций сделают вашу диаграмму значительно аккуратнее, а, следовательно, более комфортной для чтения.

1) Если классы-наследники или полиморфные классы на диаграмме имеют примерно одинаковое количество членов класса – сделайте их одинакового размера. Одинаковый размер классов подчеркнет, что эти классы одного уровня в иерархии. Размер определяется наибольшим по размеру на диаграмме классом. Как правило, в редакторах диаграмм есть готовая кнопка, по нажатию на которой все выбранные классы становятся одного размера.
2) Аналогично можно выровнять сервисные классы и перечисления.
3) Выровняйте классы наследники или агрегируемые классы по верхней границе. Для этой (и последующих) операции в редакторах также есть специальная кнопка.
4) Выровняйте расстояния между классами, находящимися на одном уровне. Очень плохо смотрится, когда, например, у класса есть три наследника, но два из них расположены друг к другу впритык, а третий отстоит от них на большом расстоянии.
5) Попробуйте представить поверх диаграммы сетку с клетками, например, в 1 или 0,5 см. Выровняйте расположение классов по линиям сетки. Аналогично выровняйте по линиям сетки и размеры самих классов. Выравнивание по сетке – один из эффективных дизайнерских и типографских приёмов для комфортной передачи информации.
6) Разместите линии связей под прямыми углами, либо под углами в 45 или 30 градусов. Диаграмма выглядит крайне неаккуратно, если все связи расположены под разными углами. Если между классами возможно провести связь под прямым углом, то обязательно сделайте связь под прямым углом. Если связь под прямым углом сделать не получается, то лучше нарисовать связь с перегибом под прямым углом или в 45 градусов, чем одну линию под неправильным углом.

Несмотря на громоздкость описанных шагов, все они выполняются достаточно быстро, так как для каждого шага по выравниванию в редакторах диаграмм уже есть готовые инструменты в одно нажатие мыши.

Расставить на диаграмме комментарии, если нужно. К сожалению, диаграмма далеко не всегда бывает понятной, даже при правильном оформлении. Оставьте поясняющие комментарии к ключевым классам диаграммы, например:

1) Если назначение класса непонятно из названия, либо название слишком сложное – оставьте поясняющий комментарий.

2) Если класс выполняет сложные или неочевидные проверки данных.
3) Если для работы отдельных методов класса необходимо проинициализировать объект определенным образом.
4) Если для работы класса нужно подключение к БД, интернету или внешнему устройству.
5) Если класс выполняет загрузку конфигурационных и иных файлов, то в комментарии можно указать расположение этих файлов или их расширение.
6) Если по какой-то причине вам пришлось принять неочевидное решение в разработке, следует пояснить причину такого решения. Например, для работы с классом из сторонней библиотеки вам пришлось создать дополнительный класс-прослойку, так как работать со сторонней библиотекой напрямую возникали проблемы.

Это лишь некоторые случаи, когда имеет смысл оставить поясняющий комментарий. Оставлять комментарии ко всем классам не надо – лишние комментарии только засоряют диаграмму. Оставляйте комментарии только к ключевым и сложным для понимания классам.

Программы для составления диаграмм классов. Существует множество программ, в которых можно составить диаграммы классов:

- **Десктоп-приложения:**
  - Sparx Architect
  - Microsoft Visio
- **Веб-приложения:**
  - visual-paradigm.com
  - create.ly
  - draw.io
  - yuml.me
  - и др.
- **Встроенные:**
  - средства Visual Studio по составлению диаграмм.

По опыту работы рекомендуем в рамках обучения программированию использовать visual-paradigm.com. После попыток составления диаграмм в самых разных редакторах, именно visual-paradigm.com показал себя как наиболее удобный для создания небольших диаграмм: простота создания карточек классов, инструменты по их выравниванию, инструменты по группировке и перемещению классов относительно друг друга, а также соблюдение нотации UML. К сожалению, в остальных программах некоторые из перечисленных пунктов либо отсутствуют, либо неудобны в использовании.

Стоит также сказать, что существуют программы, способные автоматически составить диаграммы классов по исходному коду. Например, Sparx Architect. Это большое корпоративное приложение для ведения проектной документации, в числе прочих своих функций способно составлять диаграммы классов для большинства современных языков программирования. Оно автоматически составляет карточки классов, заполняя их поля и методы. Однако программа не может определить правильность связей – это можно установить только понимая контекст данных классов и предметной области, чего компьютер не может. То есть, после составления карточек классов, пользователю остается:

1) расставить правильные связи между классами;
2) удалить члены классов, которые могут быть не принципиальными для изучения архитектуры;
3) расположить классы друг относительно друга для комфортного просмотра диаграммы;
4) выровнять размеры и расположение классов;
5) расставить комментарии, если нужно.

Конечно, разработчику диаграммы остается достаточно работы по оформлению диаграммы, но наиболее рутинная часть выполняется автоматически. Также стоит сказать, что ручное составление карточек классов или их редактирование в Sparx Architect сделано крайне неудобно, эта работа может показаться адом. В конечном счете, данная программа не предназначена для обычного рисования диаграмм, а направлена на решение более серьезных задач при разработке сложных корпоративных приложений.

В любом случае, в рамках учебного курса или обучения в вузе начинающему разработчику рекомендуется составлять диаграммы вручную – именно ручное составление диаграмм позволяет до конца усвоить отличия связей между собой, а главное, помогает пониманию объектной концепции.

Архитектура приложения "Модель-Вид"

В рамках данного курса рассматривается создание программ, количество классов в которых не превышает и десятка. В реальных приложениях количество классов может исчисляться сотнями и даже тысячами классов. Чтобы разработчики легко ориентировались в исходном коде таких приложений, применяются специальные шаблоны проектирования, или ***паттерны***.

Существует множество паттернов и принципов проектирования приложений, как высокоуровневых, так и касающихся отдельных небольших задач. Всё это тема курса ***объектно-ориентированного анализа и проектирования***, следующего после объектно-ориентированного программирования.

Однако в данном курсе стоит упомянуть один из принципов, который всегда закладывается в архитектуру приложений. Это принцип разделения программы на Модель и Вид.

Модель – это те классы и типы данных, которые отвечают за представление предметной области и правил взаимодействия реальных объектов. Модель также часто называют бизнес-логикой программы. **Вид** – это те классы и типы данных, которые отвечают за отображение данных пользователю, ввод данных и вывод данных. Например, всё, что связано с работой с консолью (cout и cin) – всё относится к Виду.

Ранее на этом не акцентировалось внимание, и если, например, класс прямоугольника в одном из своих методов выводил данные на экран (метод содержал инструкции для cout), то это считалось допустимым. Однако на практике следует выделять работу с пользовательским интерфейсом в отдельные классы. Так, правильнее сделать класс прямоугольника, в котором не будет никакого использования cout или cin или иных операторов и функций, связанных с пользовательским интерфейсом, а ввод/вывод данных прямоугольника на пользовательский интерфейс вынести в отдельный сервисный класс. Аналогично следует поступать с остальными классами.

При таком разделении очень важно, чтобы ни один класс, относящийся к Модели, не использовал и не обращался к классам из Вида. Вид может использовать объекты Модели, Модели категорически запрещено использование объектов Вида.

Для того, чтобы не перепутать, какие классы относятся к Модели, а какие – к Виду, их могут разместить в отдельных подпапках проекта или, что применяется чаще, в разных проектах. Фактически, Модель компилируется в самостоятельную dll, которую затем подключает Вид и использует. В таком случае гарантируется, что Модель не подключит Вид в качестве своей библиотеки, и, следовательно, не будет её использовать.

Почему такое разделение важно? Разделение "Модель-Вид" дает несколько преимуществ:

- **Если Модель не использует конкретный вид интерфейса, то в будущем можно сделать разные интерфейсы для одной и той же программы**. Например, если вы разрабатываете мобильное приложение, то вы можете вынести бизнес-логику в отдельную библиотеку (не работающую с пользовательским интерфейсом), а для неё сделать два пользовательских интерфейса – для Android и iOs. Если же Модель, например, будет содержать обращение к конкретным командам пользовательского интерфейса Android, то использовать эту библиотеку для iOs уже не получиться.
- **Если Модель не требует работы с пользовательским интерфейсом, бизнес-логика может быть протестирована с помощью автоматических тестов**. Когда класс бизнес-логики при своей работе запрашивает данные, например, из консоли, то тестировать такие классы придется вручную – вам придется самостоятельно вводить данные с клавиатуры. Если же класс бизнес-логики при своей работе использует данные извне, например, в качестве аргументов метода, а результаты работы возвращает через методы или поля класса, тогда разработчик может написать небольшие функции, которые будут передавать в класс тестовые данные и сверять результат работы с заранее заданным значением. Автоматические тесты выполняются в разы быстрее ручного тестирования, а, главное, их можно перезапускать любое количество раз, что упрощает разработку и повышает надежность программ

Примечание: автор немного лукавит, говоря, что классы, работающие с консолью, можно протестировать только вручную. Технически, можно перенаправить поток консоли и также из исходного кода вводить данные и получать результаты из класса. Но: 1) такой подход реализуется гораздо дольше, чем при разделении на Модель-Вид; 2) код может обрасти постоянными вызовами дополнительных функций по преобразованию значений в строки и строк в значения – чтобы проверять текстовый вывод в консоли.

Разделение "Модель-Вид" одно из самых простых. В современных приложениях данный паттерн эволюционировал, и появились более сложные схемы разделения, например "Модель-Вид-Контроллер" (MVC), "Модель-Вид-МодельВида" (MVVM), "Чистая архитектура" и др. Каждый из них предназначен для решения определенных архитектурных задач, и даже могут относится к конкретным платформам (десктоп, мобильные приложения, веб). Однако все эти паттерны объединяет общий принцип разделения архитектуры на Модель и Вид.

Задания

Наследование

5.1.1 Создайте класс Person со строковыми полями имени, фамилии, отчества. Создайте соответствующие сеттеры, геттеры и конструктор класса.

5.1.2 Создайте дочерний от класса Person класс Student с дополнительными полями номера зачетной книжки и года поступления. Создайте соответствующие сеттеры, геттеры и конструктор класса. Конструктор дочернего класса должен быть основан на наследовании от конструктора базового класса.

5.1.3 Создайте дочерний от класса Person класс Teacher с дополнительными полем должности. Создайте соответствующие сеттеры, геттеры и конструктор класса. Конструктор дочернего класса должен быть основан на наследовании от конструктора базового класса.

5.1.4 Создайте функцию ShowName(Person* person), принимающую указатель на базовый класс. Функция принимает на вход объект типа Person (или объект производного класса), и выводит на экран фамилию, имя и отчество. Пример вывода:

```
Безбородов Николай Александрович
```

5.1.5 Создайте функцию main, в которой создайте объект базового класса Person. Вызовите функцию ShowName() с экземпляром базового класса. Создайте объект дочернего класса Student и вызовите функцию ShowName() с экземпляром дочернего класса. Создайте объект дочернего класса Teacher и вызовите функцию ShowName() с экземпляром дочернего класса. Убедитесь, что функция одинаково работает с объектами всех трёх классов и корректно выводит данные на экран.

5.1.6 Убедитесь, что нет утечек памяти.

5.1.7 Нарисуйте UML-диаграмму классов для созданных классов

Рефакторинг с выделением базового класса

Работа разработчика неразрывно связана с умением читать чужой код, находить в нём ошибки проектирования или ошибки работы, и исправлять их.

5.2.1 Избавьтесь от дублирования кода с помощью выделения базового класса в следующем коде:

```
////////////////////// Post - пост в блоге пользователя с платным аккаунтом

class Post
{
    string _title;
    string _text;

public:
    void SetTitle(string title);
    void SetText(string text);

    string GetTitle();
    string GetText();

    Post(string title, string text);
};

void Post::SetTitle(string title)
{
    _title = title;
}
```

```
void Post::SetText(string text)
{
    _text = text;
}

string Post::GetTitle() { return _title; }
string Post::GetText() { return _text; }

Post::Post(string title, string text)
{
    SetTitle(title);
    SetText(text);
}


////////////////////// User - обычный пользователь

class User
{
    int _id;
    string _login;
    string _password;

    void SetId(int id);

public:
    void SetLogin(string login);
    void SetPassword(string password);

    int GetId();
    string GetLogin();
    string GetPassword();

    User(int id, string login, string password);
    bool IsCorrectPassword(string password);
};

void User::SetId(int id)
{
    _id = id;
}

void User::SetLogin(string login)
{
    _login = login;
}

void User::SetPassword(string password)
{
    _password = password;
}

int User::GetId() { return _id; }
string User::GetLogin() { return _login; }
string User::GetPassword() { return _password; }

User::User(int id, string login, string password)
{
    SetId(id);
    SetLogin(login);
```

```
    SetPassword(password);
}

bool User::IsCorrectPassword(string password)
{
    return (password == _password);
}

////////////////////// Paid User - пользователь с платным аккаунтом

class PaidUser
{
    int _id;
    string _login;
    string _password;
    Post* _posts;
    int _postsCount;

    void SetId(int id);

public:
    void SetLogin(string login);
    void SetPassword(string password);
    void SetPosts(Post* posts, int postsCount);

    int GetId();
    string GetLogin();
    string GetPassword();
    Post* GetPosts();
    int GetPostsCount();

    PaidUser(int id, string login, string password, Post* posts, int postsCount);
    PaidUser(int id, string login, string password);
    bool IsCorrectPassword(string password);
};

void PaidUser::SetId(int id)
{
    _id = id;
}

void PaidUser::SetLogin(string login)
{
    _login = login;
}

void PaidUser::SetPassword(string password)
{
    _password = password;
}

void PaidUser::SetPosts(Post* posts, int postsCount)
{
    if (postsCount < 0)
    {
            throw exception("Posts count must be more than 0");
    }
    _posts = posts;
    _postsCount = postsCount;
}
```

```
int PaidUser::GetId() { return _id; }
string PaidUser::GetLogin() { return _login; }
string PaidUser::GetPassword() { return _password; }
Post* PaidUser::GetPosts() { return _posts; }
int PaidUser::GetPostsCount() { return _postsCount; }

PaidUser::PaidUser(int id, string login, string password, Post* posts, int postsCount)
{
    SetId(id);
    SetLogin(login);
    SetPassword(password);
    SetPosts(posts, postsCount);
}

PaidUser::PaidUser(int id, string login, string password):
           PaidUser(id, login, password, nullptr, 0)
{
}

bool PaidUser::IsCorrectPassword(string password)
{
    return (password == _password);
}
```

5.2.2 Избавьтесь от дублирования кода в следующих функциях с помощью создания одной общей функции, работающей с объектами обоих классов через указатель на базовый класс.

```
// Метод принимает массив всех зарегистрированных бесплатных пользователей,
// ищет пользователя с введенным именем и паролем.
// Если такого пользователя нет, возвращается nullptr.
// Если пользователь есть, но пароль не подошел, то генерируется исключение.
User* Login(User** users, int usersCount, string enteredLogin, string enteredPassword)
{
    for(int i = 0; i < usersCount; i++)
    {
           if (users[i]->GetLogin() == enteredLogin)
           {
                  if (users[i]->IsCorrectPassword(enteredPassword))
                  {
                         return users[i];
                  }
                  else
                  {
                         throw exception("Uncorrect password");
                  }
           }
    }
    return nullptr;
}

// Метод принимает массив всех зарегистрированных платных пользователей,
// ищет пользователя с введенным именем и паролем.
// Если такого пользователя нет, возвращается nullptr.
// Если пользователь есть, но пароль не подошел, то генерируется исключение.
PaidUser* Login(PaidUser** paidUsers, int paidUsersCount, string enteredLogin, string
enteredPassword)
{
    for (int i = 0; i < paidUsersCount; i++)
```

```
    {
        if (paidUsers[i]->GetLogin() == enteredLogin)
        {
            if (paidUsers[i]->IsCorrectPassword(enteredPassword))
            {
                return paidUsers[i];
            }
            else
            {
                throw exception("Uncorrect password");
            }
        }
    }
    return nullptr;
}
```

5.2.3 Замените в функции main() использование ранее существовавших функций на вызов новой общей функции Login(). Массив пользователей программы при этом объедините в единый массив всех пользователей через массив указателей на базовый класс:

```
void main()
{
    User** users = new User * []
    {
        new User(100000, "morkovka1995", "1995morkovka"),
        new User(100001, "ilon_mask", "X æ A-12"),
        new User(100002, "megazver", "password"),
        new User(100003, "yogurt", "ksTPQzSu"),
    };

    PaidUser** paidUsers = new PaidUser * []
    {
        new PaidUser(200000, "TheKnyazz", "JHPzPGFG"),
        new PaidUser(200001, "system_exe", "UgfkDGmU"),
        new PaidUser(200002, "RazorQ", "TBgRnbCP"),
        new PaidUser(200003, "schdub", "CetyQVID"),
    };

    string login = "megazver";
    string password = "password";
    User* loginedUser = Login(users, 4, login, password);

    cout << "Signed in as: " << loginedUser->GetLogin() << endl;

    login = "system_exe";
    password = "UgfkDGmU";
    PaidUser* loginedPaidUser = Login(paidUsers, 4, login, password);

    cout << "Signed in as: " << loginedPaidUser->GetLogin() << endl;

    for (int i = 0; i < 4; i++)
    {
        delete users[i];
    }
    delete[] users;

    for (int i = 0; i < 4; i++)
    {
        delete paidUsers[i];
```

```
    }
    delete[] paidUsers;
}
```

5.2.4 Убедитесь, что после выделения базового класса и общей функции, код работает верно. ***Если код работает неправильно, найдите ошибку и исправьте её***. Результатом работы функции main() должен быть следующий текст:

```
Signed in as: megazver
Signed in as: system_exe
```

Также убедитесь в отсутствии утечек памяти.

5.2.5 Подсчитайте, сколько строк кода было до рефакторинга (устранения дублирования) и после. Рассчитайте на сколько процентов уменьшился исходный код программы благодаря наследованию.

5.2.6 Добавьте в базовый класс в метод SetLogin() проверку, чтобы логин не содержал знаки пунктуации `{`, `}`, `<`, `>`, `@`, `#`, `$`, `%`, `^`, `:`, `*`. Убедитесь, что проверка логина работает как для базового класса, так и для дочернего – то есть, механизм наследования работает.

5.2.7 Нарисуйте UML-диаграмму классов для созданных классов.

Полиморфизм

Следующая программа должна реализовать систему скидок в приложении для магазинов. Скидки действуют на товары определенной категории. Есть скидки процентные (скидка предоставляется в заданном размере от стоимости товара определенной категории), и есть скидки сертификатные (скидка предоставляется на заданную в сертификате сумму, но не превышающую стоимость товара).

5.3.1 Создайте класс товара Product с полями названия, категории и стоимости. Сделайте соответствующие геттеры, сеттеры и конструктор класса. Для поля категории создайте перечисление CategoryType. Стоимость не может быть отрицательной, а также превышать 100 000.

5.3.2 Создайте абстрактный базовый класс DiscountBase. В классе создайте:

- закрытое поле _category (категория товара, на которую предоставляется скидка).
- закрытый сеттер SetCategory() для категории товара.
- открытый геттер GetCategory() для категории товара.
- чисто виртуальный открытый метод Calculate(Product* product): double – который в будущем для дочерних классов будет реализовывать расчет стоимости товара после применения скидки.
- защищенный конструктор по категории товара, на которую предоставляется скидка.

5.3.3 Создайте дочерний класс PercentDiscount. Помимо отнаследованных от DiscountBase полей и методов, создайте в классе:

- закрытое поле _percent (размер скидки в процентах).
- открытый сеттер SetPercent() для размера скидки – значение должно быть в диапазоне от 0 до 100.
- открытый геттер GetPercent() для размера скидки.
- открытый конструктор класса по категории товара и размеру скидки – реализация конструктора дочернего класса должна обращаться к защищенному конструктору базового класса.
- переопределение открытого метода Calculate(Product* product): double. Метод должен провести сравнение категории товара с категорией скидки. Если категории совпадают, то производится расчет стоимости товара с учетом скидки в процентах. Метод возвращает новую стоимость товара (то есть, если скидка составляет 15%, то метод должен вернуть 85% стоимости товара). Если категории товара и скидки не совпадают, то метод возвращает полную стоимость товара.

5.3.4 Создайте второй дочерний класс CertificateDiscount. Помимо отнаследованных от DiscountBase полей и методов, создайте в классе:

- закрытое поле _amount (размер сертификата в рублях).

- открытый сеттер SetAmount() для размера сертификата – значение должно быть в диапазоне от 0 до 10 000.
- открытый геттер GetAmount() для размера сертификата.
- открытый конструктор класса по категории товара и размеру сертификата – реализация конструктора дочернего класса должна обращаться к защищенному конструктору базового класса.
- переопределение открытого метода Calculate(Product* product): double. Метод должен провести сравнение категории товара с категорией скидки. Если категории совпадают, то производится расчет стоимости товара с учетом скидки по сертификату. Метод возвращает новую стоимость товара. То есть, если сертификат превышает стоимость товара, то метод вернет 0 – товар полностью оплачен сертификатом, при этом размер сертификата уменьшается на стоимость товара (чтобы сертификат нельзя было применить дважды к разным товарам). Если сертификат меньше стоимости товара, то метод возвращает разницу стоимости товара и размера сертификата, при этом размер сертификата уменьшается до 0. Если категории товара и скидки не совпадают, то метод возвращает полную стоимость товара.

5.3.5 Создайте функцию void ShowCheckWithDiscount(DiscountBase* discount, Product* products, int productsCount), принимающую указатель на базовый класс скидки, а также массив товаров. Функция должна перебрать в цикле все товары массива, и рассчитать для них скидку через экземпляр discount. Для каждого товара на экран выводится его название, старая стоимость и новая стоимость. После цикла программа выводит на экран общую стоимость всех товаров с учетом скидки. Пример вывода на экран (скидка применена для первых двух товаров в размере 25%):

```
TV LG49N000    Old Cost: 40000    New Cost: 30000
Micromax Q1    Old Cost: 2000     New Cost: 1500
Pantum M650    Old Cost: 8000     New Cost: 8000
HP LasetJet    Old Cost: 11000    New Cost: 11000
Full Cost with Discount: 50500
```

5.3.6 Создайте функцию main, в которой создайте объекты обоих дочерних классов. Вызовите функцию ShowCheckWithDiscount() для каждого из созданных объектов дочерних классов. Убедитесь, что для каждого объекта вызывается собственная реализация полиморфного метода соответствующего класса. В частности, убедитесь, что при применении скидки по сертификату после её применения к одному товару, её размер уменьшается на стоимость товара.

Убедитесь, что в программе нет утечек памяти.

5.3.7 Убедитесь, что компилятор не позволяет создать экземпляры абстрактного класса DiscountBase, а только указатели на него.

5.3.8 Нарисуйте UML-диаграмму классов для созданных классов

Список литературы

1. **Наследование** / Metanit.com: сайт о программировании // https://metanit.com/cpp/tutorial/5.10.php
2. **Виртуальные функции и их определение** / Metanit.com: сайт о программировании // https://metanit.com/cpp/tutorial/5.11.php
3. **Абстрактные классы** / Metanit.com: сайт о программировании // https://metanit.com/cpp/tutorial/5.12.php
4. **Наследование в C++: beginner, intermediate, advanced** / Хабр: площадка блогов о программировании // https://habr.com/ru/post/445948/
5. Лафоре Р. **Объектно-ориентированное программирование на Си++.** Глава 9 Наследование
6. Лафоре Р. **Объектно-ориентированное программирование на Си++.** Глава 11 Виртуальные функции