Системное и прикладное программное обеспечение

Отчёт по второй лабораторной работе. Генератор кода.

## 1. Постановка задачи.

Реализована некоторая система, которая способна генерировать код на языке С++, причём, программы только следующего вида:

class MyClass {

public:

void testFunc1() {

}

virtual void testFunc3() const {

}

protected:

static void testFunc4() {

printf("Hello, world!\n");

}

private:

static void testFunc2() {

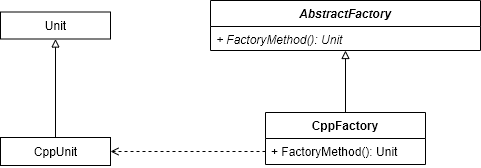
}

};

Требуется реализовать подобную генерацию программ на С# и Java. Таким образом, необходимо расширить возможности предложенной реализации, путём применения фабричных подходов (абстрактная фабрика).

## 2. Предлагаемое решение.

Ввиду того что предлагается использовать фабричный метод, дадим определение этого понятия. Фабричный метод это паттерн, определяющий общий интерфейс для создания объектов, позволяя подклассам изменять тип этих объектов. Для определённости стоит обозначить к чему именно мы собираемся применить данный метод. Так как стоит задача генерации кода программы на разных языках я пришёл к выводу что нужно реализовать функцию, настраивающую генерацию некоторого кода без привязки к конкретному языку, то есть в функцию GenerateProgram будет передаваться фабрика. Я воспользовалась разновидностью данного метода под названием «классическая реализация». Общая диаграмма фабрики выглядит следующим образом:



Использование фабрики накладывает на программу некоторые ограничения, в частности от сюда вытекает необходимость перечислять все поддерживаемые модификаторы доступа в общем классе, в противном случае не получится генерировать одну и ту же программу на нескольких языках. В качестве компромисса в каждый язык была добавлена дополнительная проверка на доступность запрошенного модификатора доступа в данном языке. При попытке использовать недоступный модификатор доступа, используется модификатор по умолчанию.

Для подведения итога по фабрике стоит сказать что в качестве базового интерфейса используется LanguageFactory, а конкретные реализации это CppFactory, CsFactory и JavaFactory. В фабрике реализовано три фабричных метода, отвечающих за создание экземпляров языковой конструкции класса, метода и операции вывода.

Теперь можно подробнее описать языковые конструкции. В данной реализации существует базовая языковая конструкция, и производные от неё: конструкция класса, метода и оператора вывода, и конкретные реализации этих конструкций для каждого языка.

В базовом абстрактном классе языковой конструкции объявлена функция Add, отвечающая за добавление других языковых конструкций в тело текущей. Так как не все языковые конструкции поддерживают данную операцию, она определена по умолчанию и в таком случае выбрасывает исключение о том, что она не поддерживается. Функция Compile отвечает за генерацию самого кода программы, здесь она чисто виртуальная. Также присутствует вспомогательная функция для генерации отступов, на всякий случай она объявлена виртуальной, но данная возможность в дальнейшем никак не используется.

В языковой конструкции класса перечислены все поддерживаемые модификаторы доступа, язык определяет сам, какие из них он поддерживает. В данной реализации за модификаторы доступа отвечает класс, поэтому его тело делится на наборы полей. Количество наборов соответствует количеству доступных модификаторов конкретного языка. За хранение имени создаваемого класса отвечает базовый класс.

Языковая конструкция метода содержит все поддерживаемые модификаторы методов, отличительной чертой от модификаторов доступа здесь является то, что они хранятся побитово. То есть у одного метода может быть сразу несколько модификаторов, например STATIC | CONST. В отличии от конструкции класса, в здесь метод Add реализован на данном этапе, а не для конкретных языков. Это сделано по причине отсутствия флагов у конструкций, добавляемых в метод. Также данный класс отвечает за хранение имени, типа возвращаемого значения, модификаторов и тела метода.

Языковая конструкция оператора вывода не поддерживает добавление дополнительных конструкций, поэтому реализация метода Add остаётся по умолчанию. Соответственно, в данном классе хранится текст, который должен вывести оператор.

Перейдём к конкретным реализациям данных языковых конструкций. Начнём с конструкции класса на языке C++. Сначала задаётся список поддерживаемых модификаторов доступа: public, protected и private. Затем реализуется метод добавления языковых конструкций в тело класса. В нём присутствует проверка входящего модификатора доступа. Если данный модификатор не поддерживается, он заменяется на PRIVATE. В функции генерации кода осуществляется перебор всех полей модификаторов доступа. Если в модификаторе нет полей, он не выводится. При наличии в модификаторе полей, сначала выводится модификатор, а потом для каждой языковой конструкции вызывается метод генерации.

Проверка поддержки заданных флагов производится в функции генерации. Первый из поддерживаемых флагов это STATIC, при его наличии добавляется соответствующую отметка в объявление языковой конструкции метода. Далее идёт флаг VIRTUAL, но так как статическая функция не может быть виртуальной, данный флаг проверяется только если флаг STATIC не был установлен. После этих модификаторов идёт тип возвращаемого значения и имя функции. Между именем функции и телом функции также может быть установлен флаг CONST. Вывод тела метода проще чем класса, ввиду того что у него нет разделения тела на разные области. Таким образом просто перебираются все помещённые в тело метода конструкции и выводятся с учётом увеличения уровня вложенности.

В предложенную реализацию конструкции оператора вывода языка C++ я внёс некоторое изменение. Дело в том что в остальных языках будут использованы операции вывода, добавляющие после себя перенос строки, а предложенный printf() этого не делает. По этому я приняла решение заменить его на более привычный для C++ потоковый вывод через std::cout с добавлением std::endl.

Следующий реализованный язык это C#. Так как основные аспекты реализации каждой языковой конструкции мы уже рассмотрели на примере языка C++, в дальнейшем стоит лишь упомянуть чем реализации конкретных языков отличаются от уже рассмотренных. В случае языка C# и его конструкции класса стоит отметить, что он поддерживает три дополнительных модификатора доступа: internal, proteted internal и private protected. Кроме того, тело класса уже не разбивается на наборы полей явно, вместо этого модификатор доступа указываются в начале каждого метода. Что касается конструкции метода, то из неё удаляется модификатор CONST, ввиду его отсутствия в данном языке.

Языковые конструкции Java больше похожи на конструкции C#, чем на C++. В конструкции класса так же отсутствует разделение на поля. В конструкции метода теряем ещё один модификатор – VIRTUAL, но за то появляется новый модификатор FINAL.

Теперь, когда все использованные в решении конструкции рассмотрены, перейдём к их использованию. Я немного расширил данный пример, добавив к нему подкласс ClassB. Сделала я это для того чтобы наглядно показать от куда в языке C# появляются модификаторы доступа у классов. Вносить какие-либо ещё изменения в предложенный пример необходимости нету, так как если какой-то язык не поддерживает какой-то модификатор, он его просто игнорирует. Настройка и генерация кода программы происходит в функции GenerateProgram которая в качестве аргумента принимает фабрику языка. Это сделано для того, чтобы эту функцию можно было использовать для любого из языков. Соответственно вызовы таких функций как CppClassUnit() были заменены на работу с фабрикой factory->CreateClass(). Вызов данной функции и создание фабрик осуществляется в главной функции. В ней также присутствует отлов исключений, сделан он для того, чтобы при возникновении исключения в процессе генерации программы одного из языков, у программы оставался шанс быть построенной на других языках.

В некоторых местах используются умные указатели, нужны они для того чтобы они сами следили за тем когда они должны будут удалиться из памяти.

## 3. Коды программ.

unit.h – базовая языковая конструкция

class\_unit.h – базовая языковая конструкция класса

method\_unit.h – базовая языковая конструкция метода

print\_operator\_unit.h – базовая языковая конструкция оператора вывода

cpp/unit.h – языковые конструкции языка C++

cpp/class\_unit.h

cpp/method\_unit.h

cpp/print\_operator\_unit.h

cs/unit.h – языковые конструкции языка C#

cs/class\_unit.h

cs/method\_unit.h

cs/print\_operator\_unit.h

java/unit.h – языковые конструкции языка Java

java/class\_unit.h

java/method\_unit.h

java/print\_operator\_unit.h

language\_factory.h – фабрика языков

main.cpp

Все методы классов определены в заголовочных файлах с целью избегания через чур большого количество файлов в проекте. Все реализации фабрик находятся в одном файле по той же причине.

## 4. Инструкция пользователя.

Взаимодействие с языковыми классами предлагается осуществлять по средством предоставленных фабрик: CppFactory, CsFactory и JavaFactory.

Для добавления в код экземпляра класса необходимо воспользоваться методом LanugageFactory::CreateClass, принимающим на вход название желаемого класса.

Конструкция класса имеет поддержку добавления в тело подклассов и методов, которым при добавлении можно выбрать модификатор доступа: ClassUnit::Add.

Доступные модификаторы доступа: PUBLIC, PROTECTED, PRIVATE, INTERNAL, PROTECTED\_INTERNAL, PRIVATE\_PROTECTED.

Конструкция метода также имеет поддержку добавления в тело, но уже только операторов. Также при создании метода можно указать его собственные модификаторы, такие как: STATIC, CONST, VIRTUAL, FINAL.

Из операторов доступен только оператор вывода: LanguageFactory::CreatePrintOperator. Он принимает на вход текст, который необходимо вывести.

## 5. Тестирование.

Первым делом стоит проверить компилируется ли сгенерированные программы. Если запустить программу, предложенную по умолчанию в файле main.cpp, и проверить её на всех трёх языках, то можно убедиться, что всё работает как задумано.

Также можно проверить что будет если попытаться вызвать метод Add для оператора вывода, как мы помним, у него данная операция не поддерживается.

oper->Add(method, 0);

В результате, как и предполагалось, для всех языков выводится исключение о том, что он не поддерживается.

This type of unit doesn't support the Add method.

Также у нас есть ещё один вариант для исключительной ситуации. На случай если что-то пошло не по плану и по какой-то причине была предпринята попытка добавить в тело класса или функции пустой указатель.

method->Add(nullptr);

Получаем соответствующее уведомление

The unit is nullptr.