DOI 10.37882/2223–2966.2021.10.16

ОСОБЕННОСТИ ИСПОЛЬЗОВАНИЯ ВИРТУАЛЬНЫХ ФУНКЦИЙ В ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММАХ

FEATURES OF USING VIRTUAL FUNCTIONS IN OBJECT-ORIENTED PROGRAMS

P. Novikov

Summary. The paper discusses the features of using virtual functions in object-orient programs that a common for different programming languages. Techniques of reducing the source code based on the use of virtual functions overridden in the class inheritance hierarchy were proposed. The situations with virtual functions were studied, when it is possible both to increase of machine time consumptions and to reduce these consumptions.

Keywords: virtual functions, override of methods, class inheritance hierarchy, polymorphic objects, static methods.

Новиков Павел Владимирович
К.т.н., доцент, Московский авиационный институт (национальный исследовательский университет)
novikov.mai@mail.ru

Аннотация. В статье рассмотрены особенности использования виртуальных функций в объектно-ориентированных программах, общие для разных языков программирования. Предложены приёмы экономии исходного программного кода, основанные на применении виртуальных функций, переопределённых в иерархии наследования классов. Изучены ситуации с виртуальными функциями, приводящие как к росту затрат машинного времени, так и к уменьшению этих затрат.

Ключевые слова: виртуальные функции, переопределение методов, классы в иерархии наследования, полиморфные объекты, статические методы.

В настоящей статье продолжается публикация результатов детального изучения автором некоторых отдельных приёмов объектно-ориентированного программирования, начатая в [1]. Эта работа посвящена исследованию различных особенностей использования виртуальных функций, как их называют в C++, (или виртуальных методов, как их называют в других объектно-ориентированных языках, см. [2–6]) с использованием имеющихся в алгоритмических языках возможностей измерения времени работы программы.

Как известно, виртуальная функция, член класса в C++ (или виртуальный метод в других объектно-ориентированных языках) подключаются к объектно-ориентированной программе не на этапе компиляции, а на этапе выполнения [2–5]. Другими словами, все ссылки на виртуальную функцию (виртуальный метод) разрешаются по факту вызова этой функции (этого метода) объектом [5]. Значит то, **какая** из переопределяемых в иерархии классов виртуальных функций (виртуальных методов) будет выбрана зависит от того, **объект какого класса** будет создан. Такой механизм назван «поздним связыванием» (см. [2, 4, 5]). Для реализации позднего связывания адреса виртуальных функций (виртуальных методов) хранятся в создаваемых для каждого класса таблицах виртуальных методов (Virtual Method Table, VMT) по одной для каждого класса. Наиболее интересно использование виртуальных функций в классах, соединённых в иерархию наследования. При этом важно заметить, что перевод слова *virtual* с английского означает "фактический" (см., например, у Павловской Т.А. в [5]), а не «кажущийся», как порой трактуют. То есть если функция, член класса (или метод), объявлена как virtual, то это означает, что фактически именно она (а не функция с тем же именем, но член другого класса) будет вызвана при посылке сообщения объекту этого же класса. При посылке сообщения к объекту непосредственно этой функцией такая ситуация является очевидной, но более интересна ситуация, когда сообщение к объекту выполняется с помощью другой функции, в теле которой вызывается эта виртуальная функция.

С прикладной точки зрения интересно рассмотреть следующие особенности использования виртуальных функций (виртуальных методов), не слишком строго и подробно изученные ранее. Это, прежде всего, приёмы и способы для экономии исходного кода объектно-ориентированных программ, упомянутые в [3] и весьма полно представленные в [6]. Затем проблема увеличения времени вычислений [5], а также и совсем небезынтересный нюанс, связанный с уменьшением (!) затрат на вычисления при определённых условиях.

Экономия исходного кода объектно-ориентированной программы при использовании виртуальных функций, переопределённых в иерархии наследова-

ния классов, позволяющая правильно использовать те унаследованные методы (функции, члены класса), которые вызывают эти переопределённые виртуальные функции.

Ниже рассмотрены фрагменты программ, иллюстрирующие эту экономию. Пример: Рассмотрим простейшую иерархию классов «Точка»-«Окружность» для работы с графическими объектами на экране. Класс Point описывает объекты-точки с защищёнными полями — экранными координатами X и Y. Доступ к полям данных осуществляется с помощью открытых методов-акцессоров GetX, GetY, PutX(…) и PutY(…). Метод Draw изображает объект-точку на экране, а метод Mask прячет объект-точку, рисуя её цветом фона. Метод Move(dX, dY) класса Point выполняет сдвиг объекта-точки, спрятав его методом Hide, увеличив координаты X и Y на dX и dY, соответственно, а затем изобразив объект-точку на новом месте с помощью метода Draw.

Пример 1. Объявление класса Point

```
class Point
{protected: int X;
int Y;
public: Point(int, int);
int GetX(), GetY();
void PutX(int), PutY(int);
void Draw ();
void Mask ();
void Move(int, int);};
```

Класс Circle, наследник класса Point, с полями — экранными координатами центра окружности X и Y, унаследованными от класса Point, а также с открытым полем R — радиусом окружности. Открытые методы-акцессоры GetX, GetY, PutX(…) и PutY(…) наследуются от класса Point. Метод Draw изображает объект-окружность на экране, а метод Mask прячет объект-окружность, рисуя её цветом фона. Метод Move(dX, dY) осуществляет сдвиг объекта-окружности на величину dX по оси X и на величину dY по оси Y. Методы Draw, Mask и Move класса Circle переопределяют одноимённые методы класса Point. Метод Move(int dX, int dY) класса Circle выполняет сдвиг объекта-окружности, скрыв его методом Mask, увеличив координаты X и Y на dX и dY, соответственно, а затем изобразив объект-окружность на новом месте с помощью метода Draw.

Пример 2. Объявление класса Circle

```
class Circle: public Point //класс Circle — наследник класса Point
{public: int R;
Circle(int, int, int);
void Draw (); // переопределение унаследованного метода Draw
void Mask (); // переопределение унаследованного метода Mask
void Move(int, int);}; // переопределение унаследованного метода Move
```

Очевидно, что код метода Move класса Point и код метода Move класса Circle совпадают. Отличие только в том, какому классу метод (функция, член класса) принадлежит. Определения методов это подтверждают:

Пример 3. Определение методов Point:: Move(…) и Circle:: Move(…)

```
void Point:: Move(int dX, int dY) {Mask ();
PutX(GetX() + dX);
PutY(GetY() + dY);
Draw ();}
void Circle:: Move(int dX, int dY) {Mask ();
PutX(GetX() + dX);
PutY(GetY() + dY);
Draw ();}
```

Несмотря на то, что код обоих методов одинаковый, класс Circle не может унаследовать метод Move у класса Point, чтобы вызывать этот метод объектами класса Circle (посылать сообщения с этим методом к объектам класса Circle), так как в теле методов Move вызываются методы Draw и Mask, имеющие одинаковое объявление, но разное определение в разных классах. Фрагмент с объектами:

Пример 4. Фрагмент основной программы с объектами классов Point и Circle

```
int main()
{Point P(300,200); // создание объекта P класса Point
P. Draw(); // вызов объектом P метода Draw класса Point
getchar(); P. Move(100,50); // вызов метода Move класса Point, сдвиг объекта P
getchar(); Circle C(295,100,70); // создание объекта C класса Circle
C. Draw (); // вызов переопределённого метода Draw класса Circle
getchar(); C. Move(40,30); // вызов переопределённого метода Move класса Circle
// объект C класса Circle будет правильно передвинут
getchar(); return 0;};
```

Если же метод Move будет унаследован от класса Point, то объявление класса Circle будет выглядеть так:

Пример 5. Объявление класса Circle без переопределения метода Move

```
class Circle: public Point //класс Circle — наследник класса Point
{public: int R;
Circle(int, int, int);
void Draw (); // переопределение унаследованного метода Draw
void Mask ();}; // переопределение унаследованного метода Mask
```

В этом случае обращение к объекту C с сообщением Move(…) в Примере 4:

```
C. Move(40,30);
```

не приведёт к смещению объекта C на новое место, а приведёт к появлению новой точки там, где должен был находиться центр сдвинутого объекта-окружности. То есть правильно использовать унаследованный метод Move в этом случае нельзя. Это происходит потому, что при вызове метода Move объектом C вызываются методы Draw и Mask класса Point, а не класса Circle.

Правильное использование унаследованного метода Move объектом класса Circle так, чтобы унаследованный метод Move обращался к методам Draw и Mask класса Circle, возможно, если в иерархии классов Point — Circle методы Draw и Mask объявить виртуальными (virtual):

Пример 6. Объявления классов Point и Circle с виртуальными функциями на C++

```
class Point //объявление класса Point
{protected: int X;
int Y;
public: Point(int, int);
int GetX(), GetY();
void PutX(int), PutY(int);
virtual void Draw (); // виртуальный метод Draw (виртуальная функция)
virtual void Mask (); // виртуальный метод Mask (виртуальная функция)
void Move(int, int);};
class Circle: public Point //класс Circle — наследник Point
{public:
int R;
Circle(int, int, int);
void Draw(); // переопределённый виртуальный метод Draw
void Mask ();}; // переопределённый виртуальный метод Mask
```

В этом случае объект C класса Circle в Примере 4 вызывает метод Move, унаследованный от класса Point, но при этом сам метод Move вызывает методы Draw и Mask из класса Circle, фактически относящиеся к этому объекту C.

Фрагменты программ в Примерах 1–6 написаны на языке C++. На этом языке программирования достаточно один раз объявить виртуальной (virtual) функцию, член класса, чтобы во всех классах этой иерархии функции, переопределяющие эту функцию, также стали бы виртуальными.

Важно заметить, что показанный приём экономии программного кода с помощью виртуальных функций возможен не только для программ на C++, но и для программ на тех объектно-ориентированных языках, которые поддерживают наследование классов, переопределение методов и виртуальные функции [6]. В то же время неважно, как именно работают методы, выполняющие изображение объекта. На текст метода Move это не влияет. При этом доля сэкономленного исходного кода увеличивается при увеличении количества классов в иерархии наследования. Разумеется, экономия кода может быть не только при программировании сдвига графического объекта. Кроме методов, выполняющих повороты графических объектов относительно разных осей координат, а также деформацию объектов относительно тех же осей, можно осуществить экономию исходного программного кода для совершенно иных ситуаций. Однако это требует определённого опыта, умения и навыка в объектно-ориентированном программировании.

Возможно и иначе экономить исходный объектно-ориентированный код [6].

Экономия исходного кода объектно-ориентированной программы за счёт использования виртуальных функций с полиморфными объектами — экземплярами классов, объединённых в иерархию наследования.

Как известно, полиморфным объектом называют объект, меняющий свой тип во время работы программы. То есть, непосредственно при функционирования объектно-ориентированной программы полиморфный объект становится экземпляром то одного, то другого класса.

При использовании полиморфных объектов возможно создавать единый код для объектов разных классов, тем самым экономя объём исходного кода программ.

Полиморфные объекты в языках C++ и Object Pascal создаются с помощью указателей [6]. В таких языках как

Java и C# все объекты являются полиморфными. Указатели в этом случае не используются.

Если взять за основу объявления классов Point и Circle в Примере 6, то для иллюстрации нового приёма следует оставить почти весь код этих классов, удалив только объявление метода Move из класса Point. Вместо этого к тексту программы следует добавить самостоятельную функцию Move (не член класса):

Пример 7. Функция Move (не член класса), аналог метода Move

```
void Move(int dX, int dY, Point *pF) {pF-> Mask();
pF-> PutX(pF->GetX()+dX);
pF-> PutY(pF->GetY()+dY);
pF-> Draw();};
```

У функции Move, не два, а три входных аргумента:

```
void Move(int, int, Point*);
```

Третий аргумент — указатель на класс Point. При вызове функции Move вместо указателя на класс Point можно подставить указатель на объект класса Point или на объект класса Circle. Продолжив иерархию наследования, можно подставить на место третьего аргумента указатель на экземпляр любого класса из этой иерархии.

Текст функции Move из Примера 7 аналогичен тексту метода Move класса Point из Примера 3. Функция Move выполняет те же действия, что и метод Move, причём при тех же условиях. Объекты классов Point и Circle движутся правильно, если методы Draw и Mask объявлены виртуальными. Иначе объект C класса Circle не будет двигаться правильно. Полиморфные объекты показаны в Примере 8.

Пример 8. Работа с полиморфными аргументами и полиморфными объектами

```
int main() {Point *pF; // объявление указателя на класс Point
Point P(1, 2); P. Draw(); // объект P класса Point создан и изображён
Circle C(3, 4, 40); C. Draw(); //объект C класса Circle создан и изображён
getchar(); Move(3, 4, &P); // сдвиг точки P
getchar(); Move(5, 6, &C); // сдвиг окружности C
getchar(); pF=&P; // полиморфный указатель
Move(7, 8, pF); // сдвиг точки P
getchar(); pF=&C; // полиморфный указатель
Move(9, 10, pF); // сдвиг окружности C
getchar(); return 0;}
```

Перемещение полиморфного объекта осуществляется правильно, если методы Draw и Mask в классах виртуальны, как показано в Примере 6. Если же эти методы не виртуальны, то полиморфный объект в Примере 8 будет двигаться неправильно. Отсутствие виртуальности делает невозможной такую экономию исходного кода.

Аналогичное использование полиморфных объектов вместе с виртуальными функциями для экономии исходного кода может иметь место быть, если функцию Move с полиморфным аргументом сделать методом класса Point (функцией, членом класса Point), определение которого выглядит так:

```
void Point:: Move(int dX, int dY, Point *pF) {pF->Mask();
pF->PutX(pF->GetX()+dX);
pF->PutY(pF->GetY()+dY);
pF->Draw();};
```

Тогда работа с полиморфным объектом осуществляется следующим образом:

Пример 9. Полиморфные объекты и полиморфные аргументы метода Move

```
int main() {Point *pF; // объявление указателя на класс Point
Point P(1, 2); P. Draw(); // объект P класса Point создан и изображён
Circle C(3, 4, 40); C. Draw() //объект C класса Circle создан и изображён
P. Move(3, 4, &P); // сдвиг точки P
getchar(); C. Move(5, 6, &C); // сдвиг окружности C
getchar(); pF=&P; // полиморфный указатель
P. Move(7, 8, pF); // сдвиг точки P
getchar(); pF=&C; // полиморфный указатель
C. Move(9, 10, pF); // сдвиг окружности C
getchar(); return 0;}
```

Как было сказано выше, объекты классов Point и Circle движутся правильно, если методы Draw и Mask объявлены виртуальными. Иначе объект C класса Circle не будет двигаться правильно. Тот же эффект от виртуальности может быть, если метод Move объявить в классе Point как статический (static):

Пример 10. Объявление статического метода (статической функции, члена класса)

```
class Point {… static void Move (int, int, Point*); …};
```

Этот частный случай очень важен с точки зрения реализации представленных приёмов программиро-

вания на других объектно-ориентированных языках. На языке Object Pascal удобно создать самостоятельную процедуру Move (не метода) с полиморфным аргументом, аналогичную рассмотренной выше самостоятельной функции Move на C++. Но на чисто объектно-ориентированных языках Java и C# нет самостоятельных функций, только методы классов. В качестве аналогов самостоятельных процедур и функций можно использовать типичные для Java и C# статические методы с полиморфными аргументами.

Увеличение затрат машинного времени при работе с виртуальными функциями, связанное с компиляцией и подключением виртуальных функций в процессе выполнения объектно-ориентированных программ

Вызов виртуального метода (виртуальной функции, члена класса в C++) выполняется опосредовано через таблицу виртуальных методов, что, как известно, замедляет выполнение программы [5]. Это можно проверить экспериментально, используя имеющиеся в языках программирования возможности по измерению времени работы программы. В языке C++ библиотека time.h содержит известную функцию clock(), которая возвращает текущее время, измеряемое в машинных тиках. Несмотря на то, что компьютер выполняет миллионы операций в секунду, пользователю доступна гораздо более низкая точность вычислений. Минимальный интервал времени между двумя моментами вычислений на компьютере есть так называемый **тик**, описанный в [7] на стр.302. Всего происходит 91 тик за 5 секунд.

Максимальная частота, доступная измерению функцией clock(), задана константой CLK_TCK из библиотеки time.h и равна CLK_TCK =18,2 Гц. То есть время одного тика точно 0,(054945) с, а округлённо 0,054945 секунд. Это очень низкая точность. Неопытные программисты порой имеют ошибочное представление о точности вычисления времени, так как некоторые функции возвращают время не в тиках, а в секундах, с точностью до 10^{-6}. Отсюда мнение о минимальном интервале времени 10^{-6} с. Но ненулевые значения текущего времени в секундах в шестом разряде после десятичной запятой имеются потому, что 5 не делится нацело на 91 и 91 не кратно 5-и. Чтобы измерить расход машинного время на вычисление фрагмента программы, следует вычесть из времени, измеренного в конце исследуемого фрагмента программы, время, измеренное в начале исследуемого фрагмента программы. Но если этот интервал времени меньше времени одного тика, то окажется, что этот интервал равен нулю. Чтобы повысить точность измерения времени, следует применить способ, известный со времён первых компьютеров. Фрагмент программы, время выполнения которого требуется измерить, заключают в цикл с большим количеством повторов, чтобы разность между временем конца работы фрагмента и временем начала стала больше нуля. Полученное значение следует поделить на количество шагов цикла, что и будет искомым числом. Для расчёта затрат машинного времени намеренно взят не самый мощный компьютер: Intel ® Celeron ® M processor 1.60 GHe. Язык программирования Borland C++ 3.1:

Пример 11. Вычисление времени выполнения тестового фрагмента программы.

```
class A {private: long double a, b, c, d, e, f, g, h, i, j, k, l, m,
n, o, p, q, r, s, t, u, v, w, x, y, z, A1, B, C, D, E, F, G, H, I, J, K, L, M,
N, O, P, Q, R, S, T, U, V, W, X, Y, Z;
public: A(long double xx){a=b=c=d=e=f=g=h=i=j=k=l
=m=n=o=p=q=r=s=t=u=v=w=x=
y=z=A1=B=C=D=E=F=G=H=I=J=K=L=M=N=O=P=Q=
R=S=T=U=V=W=X=Y=Z=xx;;
virtual // подключение или снятие виртуально-
сти с методов
void pa(long double), pb(long double), pc(long
double), pd(long double),
/* и так далее, всего 52 объявления простейших
методов */
pW(long double), pX(long double), pY(long double),
pZ(long double);};
void A:: pa(long double xx) {a=xx;}; void A:: pb(long
double xx) {b=xx;};
/* и так далее, всего 52 определения простейших ме-
тодов */
void A:: pY(long double xx) {Y=xx;}; void A:: pZ(long
double xx) {Z=xx;};
void main() {A a1(1); unsigned long i;
cout<<endl<<" wait non virtual…"<<endl;
clock_t start=clock();
for (i=0; i<42949672; i++)
{a1.pa(1); a1.pb(2); a1.pc(3); a1.pd(4); a1.pe(5); a1.pf(6);
a1.pg(7); a1.ph(8);
/* и так далее, всего 52 вызова объектом a1 создан-
ных в классе A методов */
a1.pT(46); a1.pU(47); a1.pV(48); a1.pW(49); a1.pX(50);
a1.pY(51); a1.pZ(52);}
clock_t end=clock();
cout<<"start_tick= "<<start<<" end_tick=
"<<end<<endl;
cout<<" delta="<<end-start<<" non virtual time:
"<<(end-start)/CLK_TCK<<endl;
getch();}
```

Пример 12. Результаты вычисления времени выполнения с виртуальностью и без.

```
wait virtual…
start_tick= 0 end_tick= 1960
```

```
delta=1960 virtual time: 107.692308
wait non virtual…
start_tick= 0 end_tick= 1957
delta=1957 non virtual time: 107.527473
```

В этом примере отсутствует время вычисления за один шаг цикла, так как для сравнения достаточно знать лишь относительное время вычисления. Поэтому удобнее сравнивать время непосредственно в тиках. Из Примера 12 видно, что виртуальные функции при вызове их программой затрачивают немного больше времени, чем такие же функции, но не виртуальные. Разница весьма мала, но она устойчиво проявляется в таких же запусках других программ. Количественная величина этой разницы не столь важна — важен факт её наличия.

Снижение затрат машинного времени при работе с виртуальными функциями может происходить тогда, когда виртуальные методы (функции, члены класса), объявленные и определённые в классе, но не вызванные в объектно-ориентированной программе не будут даже откомпилированы (фактически не существуют в программе). В отличие от этого невиртуальные функции, объявленные в классе, будут обязательно откомпилированы и подключены к объектно-ориентированной программе, даже если ни разу не будут вызваны во время выполнения этой программы.

У объектно-ориентированных программ есть важное отличие от программ на императивных языках, состоящее в том, что при создании классов в них могут быть объявлены и определены методы, которые не будут вызваны при работе основной программы. Такое не считается ошибкой, так как класс, воплощающий принцип модульности программ, может быть перенесён из одной программы в другую, будучи создан для использования в разных программах. В то же время при создании программ на императивных языках обычно не создают функции, нигде в программе не используемые. Любая функция, присутствующая в программе, будет обязательно откомпилирована вместе с программой, даже если это и не нужно. Далее в примере для усиления обнаруженного эффекта взята иерархия простых классов с большим количеством методов-акцессоров:

Пример 13. Вычисление времени выполнения другого тестового фрагмента.

```
class A{protected: long double a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z;
public: A(long double xx)
{a=b=c=d=e=f=g=h=i=j=k=l=m=n=o=p=q=r=s=t=u=v=w=x=y=z=xx;};
// virtual // подключение или снятие виртуальности с методов
void pa(long double), pb(long double), pc(long double), pd(long double),
/* и так далее, всего 26 объявления простейших методов */… pz(long double);
// virtual // подключение или снятие виртуальности с методов
long double ga(), gb(), gc(), gd(), ge(), gf(), gg(), gh(), gi(), gj(), gk(), gl(), gm(), gn(),
go(), gp(), gq(), gr(), gs(), gt(), gu(), gv(), gw(), gx(), gy(), gz();};
class B: A
{private: long double A1, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z;
public: B(): A(1){A1=B=C=D=E=F=G=H=I=J=K=L=M=N=O=P=Q=R=S=T=U=
V=W=X=Y=Z=a=b=c=d=e=f=g=h=i=j=k=l=m=n=o=p=q=r=s=t=u=v=w=x=y=z=1;};
// virtual // подключение или снятие виртуальности с методов
void pA(long double), pB(long double), pC(long double), pD(long double),
/* и так далее, всего 26 объявлений простейших методов */ … pZ(long double);
// virtual // подключение или снятие виртуальности с методов
long double gA(), gB(), gC(), gD(), gE(), gF(), gG(), gH(), gI(), gJ(), gK(), gL(), gM(),
gN(), gO(), gP(), gQ(), gR(), gS(), gT(), gU(), gV(), gW(), gX(), gY(), gZ();};
void A:: pa(long double xx) {a=xx;}; void A:: pb(long double xx) {b=xx;};
/* и так далее, всего 52 определения простейших методов */
long double A:: gy(){return y;}; long double A:: gz() {return z;};
void B:: pA(long double xx) {A1=xx;}; void B:: pB(long double xx) {B=xx;};
/* и так далее, всего 52 определения простейших методов */
long double B:: gY(){return Y;}; long double B:: gZ() {return Z;};
void main() {A a1(0); B b1; unsigned long i; cout<<endl<<" wait non virtual…";
clock_t start=clock();
for(i=0; i<429496729; i++) {a1.pa(4294967295); b1.pB(4294967295);}
clock_t end=clock(); cout<<"start_tick= "<<start<<" end_tick= "<<end<<endl;
cout<<" delta="<<end-start<<" non virtual time: "<<(end-start)/CLK_TCK<<endl;}
```

Пример 14. Результаты вычисления времени выполнения с виртуальностью и без.

```
wait virtual… start_tick= 0 end_tick= 783
delta=783 virtual time: 43.021978
wait non virtual… start_tick= 0 end_tick= 838
delta=838 non virtual time: 46.043956
```

Из результатов видно, что **даже на этапе выполнения** откомпилированные, но нигде не используемые невиртуальные функции, члены класса (методы), могут дополнительно занимать вычислительные ресурсы компьютера, в отличие от нигде не использованных виртуальных функций, которых в этом случае как бы нет.

Эта особенность работы виртуальных функций известна разработчикам языков программирования. В чисто объектно-ориентированном языке Java все методы по умолчанию виртуальные (см.[2, 8]). Также виртуальными по умолчанию являются методы в гибридном языке PascalABC [9]. Фактически Java и PascalABC — интерпретируемые языки, для которых машинное время на трансляцию программы больше чем время на компиляцию аналогичного кода, например, в C++.

В целом с точки зрения экономии вычислительных затрат на тех языках, где программист может сам назначать виртуальность у функции или у метода (C++, C#, Delphi, Turbo Pascal и т.п.), можно предложить следующие две рекомендации.

1. Если создаётся программа, классы в которой не предназначены к использованию в других программах, то следует избегать того, чтобы назначать виртуальность методам, если того не требуют, например, соображения экономии исходного кода, изложенные выше (не следует назначать виртуальность с самого начала подряд всем методам). Но важно помнить: программа, верно работающая с виртуальными методами, может начать работать неверно, если эту виртуальность отменить.
2. Если пользовательский класс создаётся как библиотечный, когда предполагается широкое использование методов этого класса в разных прикладных программах, то разумно назначить виртуальность всем его функциям (методам), за исключением конструкторов. В этом случае сначала надо объявить виртуальными функции (методы), необходимые для экономии вычислительных затрат, упростив программу, как в (1). Затем, когда все методы уже отлажены, можно объявить виртуальными уже все остальные методы, за исключением конструкторов. При таком порядке невиртуальные методы, став новыми виртуальными, не повлияют на результат работы программы (хотя и повлияют на вычислительный процесс).

ЛИТЕРАТУРА

1. Новиков П.В. Увеличение объёма используемой оперативной памяти компьютера при наследовании классов в объектно-ориентированном программировании // Современная наука: актуальные проблемы теории и практики. Серия «Естественные и технические науки». 2018. № 6, С. 116–122.
2. Бадд Т. «Объектно-ориентированное программирование в действии». — СПб, Питер, 1997. — 464 с.
3. Епанешников А.М., Епанешников В.А. Программирование в среде Turbo Pascal 7.0. — М.: ДИАЛОГ-МИФИ, 2014. — 367 с.
4. Березин Б.И., Березин С.Б. Начальный курс С и С++. — М.: ДИАЛОГ-МИФИ, 2005. — 288 с.
5. Павловская Т.А. C#. Программирование на языке высокого уровня. — СПб.: Питер, 2007. — 432 с.
6. Новиков П.В. «Объектно-ориентированное программирование». Учебное пособие. — М.: Издательство МАИ, 2019. — 124 с.
7. Сафронов И.К. Бэйсик в задачах и примерах. — 2-е изд., перераб. и доп. — СПб, БХВ-Петербург, 2006. — 320 с.
8. Гослинг Дж., Арнольд К. «Язык программирования Java». — СПб: Питер, 1997, 304 с.
9. Долинер Л.И. Основы программирования в среде PascalABC.NET — Екатеринбург: Издательство «Уральский федеральный университет», 2014. — 128 с.

Журнал «Современная наука: актуальные проблемы теории и практики»