Статья опубликована в журнале «Информационно-управляющие системы». 2003. №6, с. 38–49.

УДК 62-507

Искусство программирования лифта. Объектно-ориентированное программирование с явным выделением состояний

Л. А. Наумов,
магистрант
А. А. Шалыто,
д-р техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий механики и оптики

Уже около сорока лет, одним из примеров, на котором Д. Кнут обучает «Искусству программирования», является задача управления лифтом. Он на основании невнятного технического задания, используя только словесное описание алгоритмов, приводит в окончательной форме программу на языке низкого уровня. В предисловии к третьему изданию упомянутой книги приводятся слова Б. Гейтса: «За последние двадцать лет мир изменился». Настоящая работа призвана показать справедливость этого высказывания на примере задачи управления лифтом.

Near forty years, one of the examples, which was used by D. Knuth for teaching “The Art of Computer programming”, is the task about lift control. Basing on indistinct description, using only verbal algorithms definitions, he gives source code of the program on the low-level language. In the preface to the third edition of mentioned book there are words of B. Gates: “During last twenty years the world has changed”. The purpose of the present article is to show the correctness of last statement on the example of the task about lift control.

Когда б Вы знали,
Из какого сора
Растут стихи
Не ведая стыда.
Анна Ахматова

Введение

О книге Дональда Кнута «Искусство программирования» Билл Гейтс сказал: «Если вы считаете себя действительно хорошим программистом..., прочитайте «Искусство программирования»... Если Вы сможете прочесть весь этот труд, то вам определенно следует отправить мне резюме» [1]. Математики говорят, что «в «Корне» есть все!» [2]. Программисты могут то же самое сказать о Кнуте.

Д. Кнут, кроме математических основ программирования, пытается обучать также и искусству прикладного программирования. С этой целью он создает виртуальную машину *MIX*, для программирования которой разработал язык ассемблера. Одним из самых «больших» и подробно рассмотренных примеров, демонстрирующих «искусство программирования по Кнуту», является разработка программного обеспечения для системы управления лифтом (разд. 2.2.5 «Дважды связанные списки» в работе [1]).

Из этого примера следует, что искусством программирования Д. Кнут обладает, так как настоящее искусство не предполагает обоснования процесса создания произведения. Автор на основании технического задания, о качестве которого сказано выше, используя только словесное описание алгоритмов, на многих страницах книги приводит в окончательной форме программу на упомянутом языке низкого уровня, откомментированную не более внятно, чем было сформулировано задание.

В этом случае имеет место ситуация, не удовлетворявшая Э. Дейкстру, который говорил, «что программы часто приводятся в форме готовых изделий, почти без упоминания тех рассуждений, которые проводились в процессе разработки и служили обоснованием для окончательного вида завершенной программы» [3].

Переходя к науке программирования, которая, в отличие от искусства, должна объяснять, как создавалось произведение, отметим, что будем понимать ее существенно шире, чем Д. Грис [4], трактовавший ее только, как верификацию программ. В настоящее время наука программирования включает в себя также проектирование [5], реализацию [6], документирование [7] и отладку [8].

Программное обеспечение для системы управления лифтом можно разработать с помощью различных подходов, один из которых связан с использованием конечных автоматов [9–11].

Настоящая работа призвана продемонстрировать преимущества подхода, названного «автоматное программирование с явным выделением состояний» [9, 12–14], на примере задачи управления лифтом. Кроме того, приводится методика преобразования объектной программы, написанной в рамках предлагаемого подхода, в процедурную программу для выполнения ее на микроконтроллере.

Необходимо отметить, что в настоящей работе, для того, чтобы не загромождать исходный код не относящимися к делу функциями, инкапсуляция данных не используется. Основная причина отсутствия инкапсуляции связана с тем, что разработанная объектно-ориентированная программа впоследствии преобразуется в процедурную программу для микроконтроллера. При этом, если бы инкапсуляция применялась, то, в результате указанного преобразования, она перестала бы выполнять свою основную задачу – сокрытие данных.

1. Особенности объектно-ориентированного программирования с явным выделением состояний

Автоматное программирование может использоваться в одном из двух вариантов: как процедурное программирование с явным выделением состояний [12] или как объектно-ориентированное программирование с явным выделением состояний [13].

В настоящей работе, как и в работе [13], используется второй подход, основанный на двух парадигмах: объектной и автоматной. При этом отметим, что в указанной статье автоматы реализуются, как методы классов, в то время как в настоящей работе предлагается реализовывать их, как классы. Это позволяет в полной мере совместить гибкость объектно-ориентированного программирования с наглядностью и ясностью автоматного подхода.

Проектирование каждого автомата состоит в создании по словесному описанию (декларации о намерениях) схемы связей, описывающей его интерфейс, и графа переходов, определяющего его поведение. По этим двум документам формально и изоморфно может быть построен модуль программы, соответствующий автомату.

Используя объектную парадигму, автоматы предлагается разрабатывать, как наследники базового класса `Automat`. Этот класс реализует типовые функции автоматов (основные и

вспомогательные). В наследниках определяются только функции, специфические для конкретных автоматов.

Перечислим основные функции автоматов, реализованные в базовом классе [14]:

- организация выполнения действий в вершинах графа переходов (для автоматов Мура), на его дугах и петлях (для автоматов Мили), а также в вершинах, на дугах и петлях (для автоматов Мура-Мили) [15];
- организация взаимодействия автоматов:
  - вызов автоматов с определенными событиями;
  - реализация вложенных автоматов;
  - обмен номерами состояний.

Отметим, что если взаимодействие по вложенности возможно только «сверху вниз» в иерархии автоматов, то остальные два способа могут осуществляться в обе стороны, как «сверху вниз», так и «снизу вверх».

Из вспомогательных функций автоматов в классе `Automat` реализована поддержка протоколирования. При этом возможно:

- автоматическое протоколирование:
  - при начале работы автомата в определенном состоянии с определенным событием;
  - при переходах из состояния в состояние;
  - при завершении работы автомата в определенном состоянии;
- добавление описаний входных и выходных воздействий автомата.

В классах наследниках переопределяется ряд функций базового класса, и добавляются входные воздействия (события и переменные), внутренние переменные, выходные воздействия, объекты управления, а также вложенные и вызываемые автоматы.

В настоящей работе, как отмечалось выше, предлагаемый подход иллюстрируется примером моделирования лифта. При этом с помощью среды *Microsoft Visual C++ 6* была разработана программа *Lift*. Эта программа размещена на сайте http://is.ifmo.ru в разделе «Статьи».

Как отмечалось выше, программа *Lift* является объектно-ориентированной. Такую программу удобно разрабатывать на персональном компьютере и легко переносить на *PC*-подобные контроллеры. Однако, кроме таких контроллеров, в системах управления используются также микроконтроллеры, для которых отсутствуют компиляторы с объектно-ориентированных языков. Поэтому для микроконтроллеров применяется процедурное программирование.

В настоящей работе предлагается методика преобразования ядра объектно-ориентированной программы с явным выделением состояний на языке *C++* в процедурную программу с явным выделением состояний на языке *C*. В данном случае, под ядром программы понимается ее фрагмент, в котором отсутствует интерфейсная часть и не реализованы функции входных и внутренних переменных, а также выходных воздействий. Методика иллюстрируется примером переноса ядра программы *Lift* на микроконтроллер *Siemens SAB 80C515*. При этом использовалась среда *Keil µVision 2*. Полученная в результате программа также размещена на сайте http://is.ifmo.ru в разделе «Статьи».

2. Формулировка задачи о лифте

Попытаемся из потока слов, описаний «сопрограмм», протокола работы и исходного кода программы на языке машины *MIX* [1], извлечь формулировку задачи.

Дано пятиэтажное здание с одним лифтом. Этаж с номером ноль – подвальный, один – цокольный, два – первый, три – второй, а четыре – третий. Будем считать, что этажи пронумерованы числами от нуля до четырех.

На каждом этаже – две кнопки для вызова лифта на движение вверх и вниз. На нулевом этаже кнопка «Вниз» заблокирована, как и кнопка «Вверх» на четвертом этаже.

В кабине лифта имеется панель с пятью кнопками для перемещения на конкретный этаж. В ней также размещены два индикатора движения (вверх и вниз). Будем рассматривать их, как единое устройство, формирующее одно из трех значений: *GoingUp* (лифт движется вверх), *GoingDown* (лифт движется вниз) или *Neutral* (лифт находится в режиме ожидания).

Первоначально лифт находится на втором этаже в режиме ожидания (состояние *Neutral*). Ни одна кнопка вызова не нажата.

При моделировании необходимо ввести масштаб времени. Будем измерять его в десятых долях секунды. Зададим продолжительность характерных для системы действий. В табл. 1 приведены имена временных параметров, их значения и комментарии к ним.

Таблица 1

Имя	Значение	Комментарий
TInactive	300	Если лифт находится на каком-либо этаже без движения в течение этого времени, то он должен быть автоматически направлен на второй этаж. Для определения действий в этом случае необходимо выполнить четвертый шаг приводимого после табл. 2 словесного описания работы системы
TDoorsTimeout	76	Если после открытия дверей прошло *TDoorsTimeout* десятых долей секунды, то необходимо попытаться их закрыть
TDoors	20	Время открытия и закрытия дверей лифта
TWaitTimeout	40	Время, через которое система пытается закрыть двери лифта. Входящий/выходящий человек может помешать этому. Описываемая задержка требуется для того, чтобы после вхождения последнего человека в кабину, двери, через некоторое время, закрылись
TStarting	15	Время ускорения лифта при начале движения
TUp	51	Время равномерного подъема лифта на один этаж
TUpBraking	14	Время замедления лифта перед остановкой при подъеме
TDown	61	Время равномерного спуска лифта на один этаж
TDownBraking	23	Время замедления лифта перед остановкой при спуске
TAfterRest	20	Время перед стартом лифта после выхода из режима ожидания
THuman	25	Время, требуемое человеку, чтобы войти или выйти из кабины
TWaitLimit	600	Максимальное время, которое человек согласен ждать лифт

Отметим, что время перемещения лифта из состояния покоя на один этаж вверх (вниз) с остановкой на этом этаже определяется соотношением *TStarting* + *TUp* + *TUpBraking* (*TStarting* + *TDown* + *TDownBraking*). Прохождение этажа «транзитом» происходит быстрее, так как при этом лифт не должен замедляться.

Если с этажа, на котором сейчас находится лифт, поступил вызов или один из пассажиров лифта желает выйти на данном этаже, то производится открытие дверей. Это занимает *TDoors* единиц времени. По истечении *TDoorsTimeout* единиц времени после открытия

дверей на этаже необходимо автоматически попытаться их закрыть. Если это не удалось, то далее лифт будет пробовать закрывать двери каждые *TWaitTimeout* единиц времени.

Если лифт выполнил все вызовы и остался стоять на этаже, то, по истечении *TInactive* единиц времени, он должен вернуться на второй этаж, так как считается, что там наиболее вероятно появление новых пассажиров.

Для реализации временных задержек в систему введены три таймера: таймер бездействия *C1* (для отсчета *TInactive* единиц времени), таймер *C2* (для остальных временных операций, не связанных с работой дверей) и таймер *C3* (для временных операций, связанных с работой дверей).

Для моделирования описываемой системы используются переменные и структуры данных, перечисленные в табл. 2.

Таблица 2

Имя	Комментарий
Floor	Номер этажа, на котором находится лифт. Эта переменная получает новое значение при начале движения с этажа
State	Состояние движения лифта (*GoingUp*, *GoingDown* или *Neutral*)
Queue[*i*]	Список людей, ожидающих лифт на *i*-м этаже
Elevator	Список пассажиров, находящихся в кабине лифта
CallCar[*i*]	Вектор, *i*-ая ячейка которого содержит единицу, если от какого-либо из пассажиров кабины поступал вызов для движения на *i*-й этаж, и ноль – в противном случае
CallUp[*i*]	Вектор, *i*-ая ячейка которого содержит единицу, если с *i*-го этажа поступал вызов на движение вверх
CallDown[*i*]	Вектор, *i*-ая ячейка которого содержит единицу, если с *i*-го этажа поступал вызов на движение вниз

Опишем работу лифта по шагам.

0. **Ожидание вызова.** *Floor* = 2, *State = Neutral*. Если поступает вызов со второго этажа, то перейти к шагу 1. Если вызов – с другого этажа, то перейти к шагу 4.
1. **Открытие дверей.** Сбросить таймер *C1*. Открыть двери (*TDoors* единиц времени) и перейти к шагу 2. Все пассажиры считаются дисциплинированными, и поэтому они не будут входить до полного открытия дверей.
2. **Вход и выход пассажиров.**
   a. Если пройдет *TInactive* единиц времени (сработает таймер *C1*), то перейти к шагу 4.
   b. Сбросить таймер *C2*. Каждого пассажира кабины (элемент списка *Elevator*), который ехал до данного этажа, выпустить из лифта. Это займет *THuman* единиц времени на одного пассажира.
   c. Пока на этаже есть люди, ждущие лифт для движения в том же направлении, в котором он двигался (их надо искать в списке *Queue*[*Floor*]), люди впускаются внутрь кабины по одному. Это займет *THuman* единиц времени на каждого пассажира. Как только человек входит в кабину лифта, он сразу же нажимает на кнопку целевого этажа (изменяет значение ячейки вектора *CallCar*[*i*]). Если вызов на этот этаж был произведен раньше, то фиксировать его не требуется.
   d. Если по таймеру *C2* прошло *TDoorsTimeout* единиц времени, то перейти к шагу 3.
3. **Закрытие дверей.**

a. Если пройдет *TInactive* единиц времени (сработает таймер *C1*), то перейти к шагу 4.
b. Попытаться закрыть дверь. Если не получилось, то повторять попытки каждые *TWaitTimeout* единиц времени.
c. Когда двери закроются – выполнить присваивания: *CallUp*[*Floor*] = 0 (если *State* *GoingDown*), *CallDown*[*Floor*] = 0 (если *State* *GoingUp*) и *CallCar*[*Floor*] = 0. Присвоения удаляют информацию об обработанных вызовах. Перейти к шагу 4.

4. **Принятие решения о дальнейшем движении.**
   a. Если *State* = *GoingUp* (*GoingDown*) и есть еще вызовы на движение вверх (вниз), то значение переменной *State* не изменится. Это условие означает, что существует значение *i* большее (меньшее) значения переменной *Floor*, для которого значение из ячеек *CallCar*[*i*], *CallUp*[*i*] или *CallDown*[*i*] отлично от нуля. Перейти к подпункту d.
   b. Если вызовов на движение вверх (вниз) нет, но есть вызовы на движение вниз (вверх), то присвоить переменной *State* значение *GoingDown* (*GoingUp*). Перейти к подпункту d.
   c. Если вызовов вообще нет и при этом значение *Floor* меньше (больше) двух, то присвоить переменной *State* значение *GoingUp* (*GoingDown*). При *Floor* = 2 присвоить переменной *State* значение *Neutral*. Перейти к подпункту d.
   d. В результате, если *State* = *GoingUp*, то перейти к шагу 5, если *State* = *GoingDown*, то – к шагу 6, а если *State* = *Neutral*, то – к шагу 0.
5. **Подняться на этаж.**
   a. Увеличить значение переменной *Floor* на единицу и начать движение. Это займет *TStarting* + *TUp* единиц времени.
   b. Если есть вызов для движения на этаж *Floor*, то необходимо, чтобы лифт притормозил. Это займет *TUpBraking* единиц времени. Перейти к шагу 1.
   c. Если нет вызовов для движения на этаж *Floor*, то повторить шаг 5.
6. **Спуститься на этаж.**
   a. Уменьшить значение переменной *Floor* на единицу и начать движение. Это займет *TStarting* + *TDown* единиц времени.
   b. Если есть вызов для движения на этаж *Floor*, то необходимо, чтобы лифт притормозил. Это займет *TDownBraking* единиц времени. Перейти к шагу 1.
   c. Если нет вызовов для движения на этаж *Floor*, то повторить шаг 6.

Человек, в рамках решаемой задачи, представляет собой сущность, характеризуемую тремя атрибутами:

- *CurFloor* – номер этажа, на котором он появляется;
- *TgtFloor* – номер этажа, на который он хочет попасть (*CurFloor C TgtFloor*);
- *WaitFor* – максимальное время, которое человек согласен ждать лифт. Если по истечении этого времени он не попадет в кабину, то человек пойдет пешком (покинет этаж *CurFloor*). При этом его вызов остается в силе. Начальное значение атрибута *WaitFor* равно *TWaitLimit*.

Взаимодействие людей с лифтом осуществляется через списки *Queue*[*i*] и *Elevator*, а также вектора *CallUp*[*i*], *CallDown*[*i*] и *CallCar*[*i*].

3. Проектирование автоматов и классов

На основании словесного описания, приведенного в предыдущем разделе, можно построить схему алгоритма, которая будет весьма громоздкой. Поэтому в дальнейшем будем использовать автоматный подход.

Построим три автомата, каждый из которых определяет поведение соответствующей компоненты системы. Они обеспечивают:

- принятие решений (автомат *A0*);
- управление двигателем, перемещающим лифт между этажами (автомат *A1*);
- управление пятью двигателями, открывающими/закрывающими двери на этажах (автомат *A2*).

Схема связей автомата *A0* приведена на рис. 1, а его граф переходов – на рис. 2. Схема связей автомата *A1* – на рис. 3, а его граф переходов – на рис. 4, и, наконец, схема связей автомата *A2* – на рис. 5, а его граф переходов – на рис. 6.

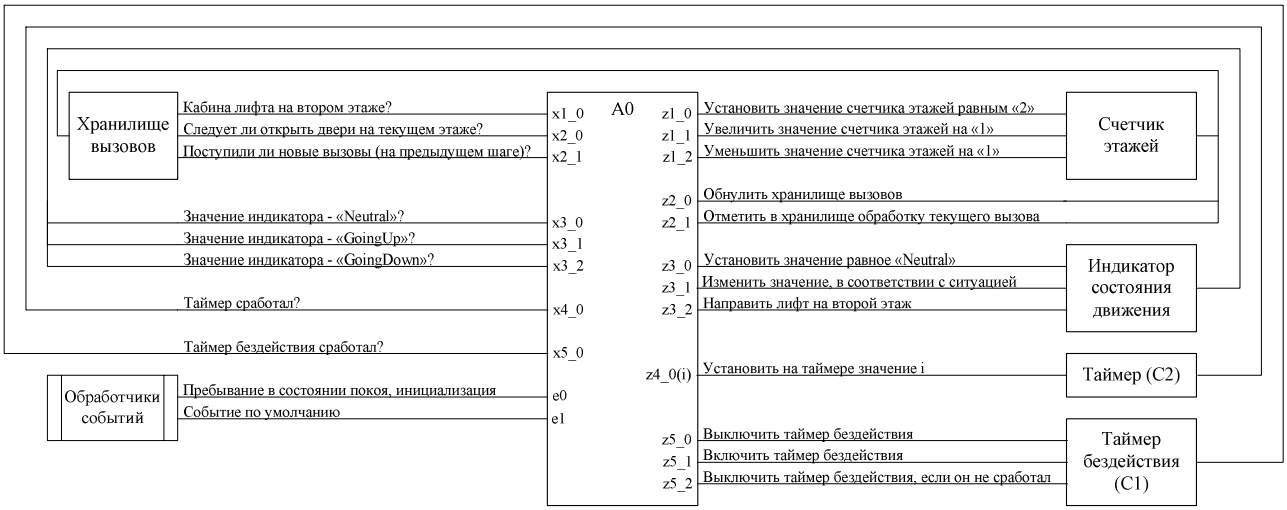

Рис. 1. Схема связей автомата *A0*. Принятие решений

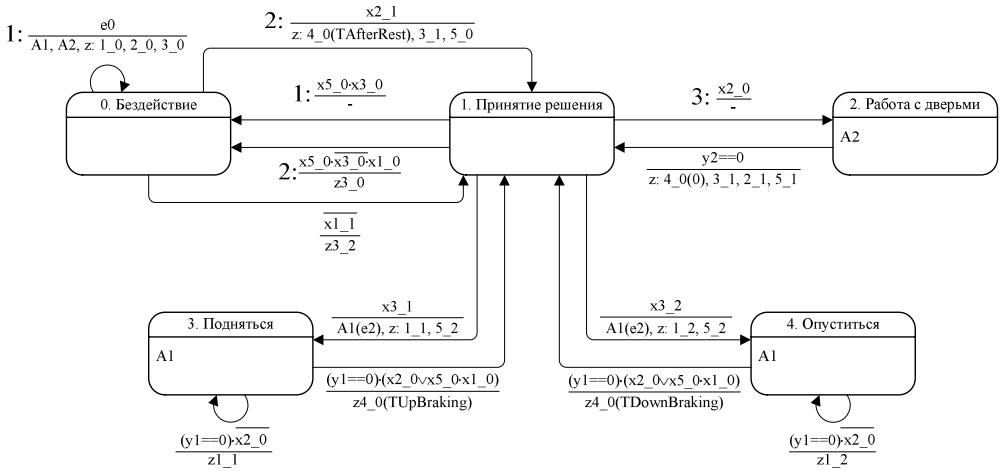

Рис. 2. Граф переходов автомата *A0*. Принятие решений

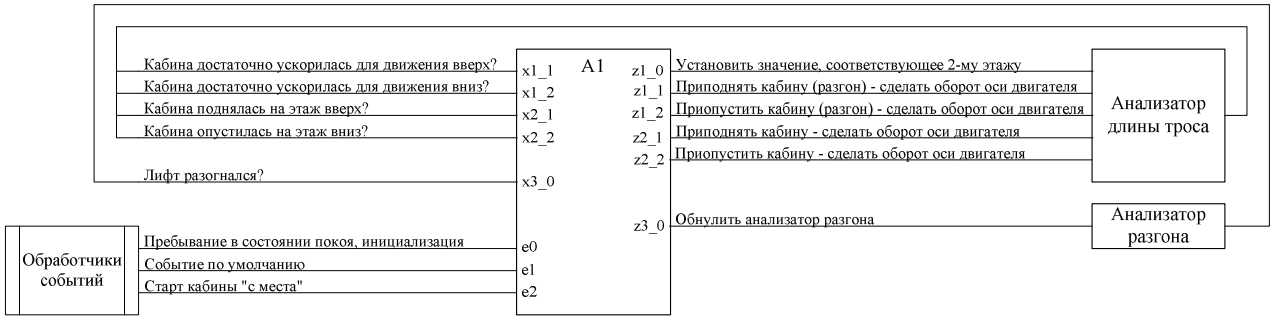

Рис. 3. Схема связей автомата *A1*.
Управление двигателем, перемещающим лифт между этажами

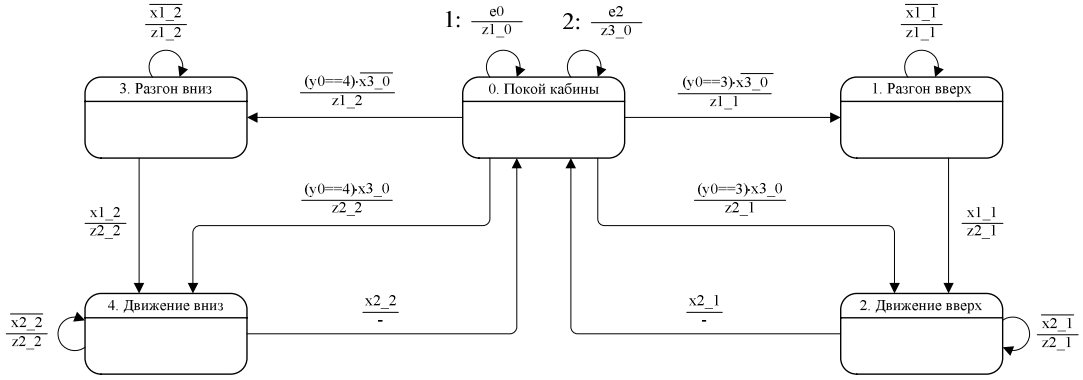

Рис. 4. Граф переходов автомата *A1*.
Управление двигателем, перемещающим лифт между этажами

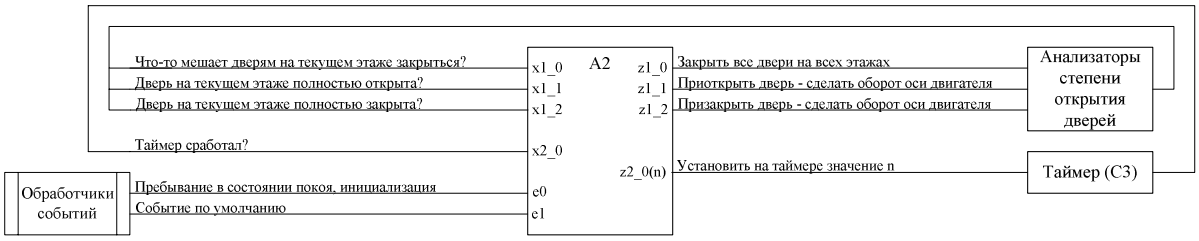

Рис. 5. Схема связей автомата *A2*.
Управление пятью двигателями, открывающими/закрывающими двери на этажах

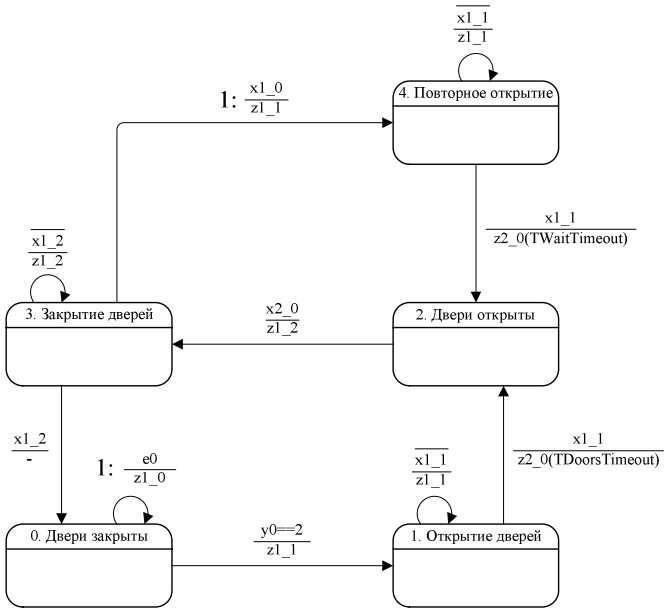

Рис. 6. Граф переходов автомата *A2*.
Управление пятью двигателями, открывающими/закрывающими двери на этажах

Все три автомата являются автоматами Мили, и поэтому каждый из них будет реализован с помощью одного оператора *switch*. Отметим, что для автомата *A0* характерно, что в состояние «Работа с дверьми» вложен автомат *A2*, а в состояния «Подняться» и «Опуститься» – автомат *A1*. На переходах из состояния «Принятие решения» в состояния «Подняться» и «Опуститься» осуществляется вызов автомата *A1* с событием *e2*.

Класс *A0* реализует автомат принятия решений и содержит пять объектов, управляемых им: хранилище вызовов (реализуемое векторами *CallCar*[*i*], *CallUp*[*i*] и *CallDown*[*i*]), счетчик этажей (*Floor*), индикатор состояния движения (*State*), таймер (*C2*) и таймер бездействия (*C1*). Кроме того, он содержит объекты классов *A1* и *A2*, реализующих автоматы *A1* и *A2*, соответственно.

Автоматы *A1* и *A2* не содержат вложенных автоматов и не вызывают другие автоматы с какими-либо событиями.

Класс *A1* реализует автомат управления двигателем, перемещающим кабину лифта между этажами, и содержит этот двигатель.

Основной характеристикой положения лифта является длина части троса между кабиной и точкой перегиба троса. Будем измерять эту величину в «расстояниях между этажами». Таким образом, данная характеристика – число с плавающей точкой от нуля до четырех. Скорость перемещения лифта зависит от степени его разгона/торможения, которая определяется анализатором разгона. Анализаторы длины троса и разгона, в совокупности, моделируют двигатель, перемещающий кабину между этажами.

Класс *A2* реализует автомат управления пятью двигателями, открывающими/закрывающими двери на этажах, а также содержит эти двигатели и таймер (*C3*).

Основной параметр, на который влияет двигатель, открывающий/закрывающий двери – степень ее открытия (вещественное число от нуля до единицы). Он равен отношению числа сделанных оборотов оси двигателя к общему числу оборотов, необходимых для того, чтобы

дверь открылась полностью. Анализаторы степени открытия дверей моделируют соответствующие двигатели.

Кроме того, системе управления дверьми необходим таймер для определения времени закрытия дверей.

У каждого из рассмотренных автоматов имеется, так называемое, «событие по умолчанию», с которым он вызывается при отсутствии прочих событий.

В заключение раздела отметим, что включение объектов управления в классы автоматов не является обязательным.

4. Описание базового класса `Automat` и вспомогательных макроопределений

Рассмотрим базовый класс `Automat`, реализующий функциональность, описанную во введении:

```
class Automat
{
public:
    int y;          // Переменная, хранящая состояние автомата
    int y_old;      // Переменная, хранящая состояние, в котором автомат начал
                    // последнюю итерацию
    int BaseIndent; // Минимальный базовый сдвиг для всех записей в протоколе
    bool DoLog;     // Писать ли протокол? Вот в чем вопрос! J
    Automat* Nest;  // Указатель на автомат, запустивший данный

    Automat(bool DoLog=false); // Конструктор
    void Main(int e);  // Главная функция автомата

// Основные функции
protected:
    // Шаг автомата. Необходимо переопределять в наследниках
    virtual void Step(int e)=0;

    // Пост-шаг автомата, выполняемый только в случае перехода из состояния в
    // состояние (только для автоматов Мура и Мура-Мили). Необходимо переопределять
    // в наследниках
    virtual void PostStep(int e)=0;

public:
    // Выполняет шаг автомата A с событием e
    void DoNested(int e, Automat* A);

// Вспомогательные функции
public:
    // Записать в протокол строку s с отступом indent. Необходимо
    // переопределять в наследниках
    virtual void WriteLog(CString s, int indent=0)=0;

protected:
    // Записать в протокол информацию о том, что автомат запущен с событием e
    // (indent – величина отступа). Необходимо переопределять в наследниках
    virtual void WriteLogOnStart(int y,int e,int indent=0)=0;

    // Записать в протокол информацию о переходе автомата из состояния y_old в
```

```
    // состояние y (indent - величина отступа). Необходимо переопределять в
    // наследниках
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0)=0;

    // Записать в протокол информацию о том, что автомат завершил работу в
    // состоянии y (indent - величина отступа). Необходимо переопределять в
    // наследниках
    virtual void WriteLogOnFinish(int y,int indent=0)=0;

// Объекты управления. Добавить в наследниках
protected:

// Вызываемые автоматы. Добавить в наследниках
protected:

// События. Добавить в наследниках
protected:

// Входные переменные. Добавить в наследниках
protected:

// Внутренние переменные. Добавить в наследниках
private:

// Выходные воздействия. Добавить в наследниках
protected:
};
```

В конце рассмотренного кода приведён состав членов класса, которые необходимо добавлять в наследники класса `Automat` для реализации конкретных автоматов.

Кратко опишем функции-члены класса `Automat`. Одной из них является функция `Main` – главная функция автомата, обеспечивающая выполнение действий в вершинах графа переходов, на его дугах и петлях:

```
void Automat::Main(int e)            // Главная функция автомата
{
    y_old=y;                         // Запомнить состояние перед началом итерации
    WriteLogOnStart(y,e);            // Протоколировать факт начала итерации
    Step(e);                         // Выполнить шаг
    WriteLogOnStateChanged(y,y_old); // Протоколировать факт перехода или
                                     // сохранения состояния
    if (y!=y_old) PostStep(e);       // Если переход был - выполнить пост-шаг
    WriteLogOnFinish(y);             // Протоколировать факт окончания итерации
}
```

Параметром этой функции является идентификатор события, который передается функциям `Step` и `PostStep`. При реализации автоматов Мили (Мура) единственный оператор `switch` необходимо разместить в переопределенной в наследнике функции `Step` (`PostStep`). Для автоматов Мура-Мили требуется переопределять обе эти функции.

Макроопределения `DECLARE_NO_STEP` и `DECLARE_NO_POSTSTEP` позволяют реализовать функции `Step` и `PostStep`, как пустые.

Протоколирование поведения автомата реализуется посредством переопределения функций `WriteLog`, `WriteLogOnStart`, `WriteLogOnStateChanged` и

WriteLogOnFinish. Последние три функции, как правило, используют первую. Функцию WriteLog следует вызывать дополнительно, например, из функций входных и внутренних переменных, а также функций выходных воздействий.

Макроопределения DECLARE_NO_LOG, DECLARE_NO_LOG_ON_START, DECLARE_NO_LOG_ON_STATE_CHANGED и DECLARE_NO_LOG_ON_FINISH позволяют реализовать указанные выше функции, как пустые.

Все четыре функции протоколирования содержат необязательный параметр indent – отступ при размещении информации в протоколе. Сумма значений параметра indent и переменной BaseIndent определяют количество пробелов, добавляемых перед записью в протокол, что позволяет наглядно отразить в нем вложенность автоматов.

Для включения/выключения режима протоколирования используется булева переменная DoLog. Поэтому рекомендуется перечисленные выше функции протоколирования (или только функцию WriteLog) начинать со следующей строки:

```
if (!DoLog) return;
```

Для временного отключения протоколирования используются макроопределения SWITCH_LOG_OFF и SWITCH_LOG_ON. Необходимо, чтобы они применялись только парами, и при этом первое макроопределение должно всегда предшествовать второму! Приведем пример их использования в программе *Lift*:

```
void A0::z5_2()
{
    WriteLog("z5_2: Выключить таймер бездействия, если он не сработал",3);
    SWITCH_LOG_OFF // Отключить протоколирование
    if (!x5_0()) z5_0();
    SWITCH_LOG_ON  // Включить протоколирование
}
```

В этом фрагменте программы временное отключение протоколирования используется для того, чтобы при вызове функции z5_2() в протоколе не отражались вызовы функций x5_0() и z5_0().

Конструктор класса Automat инициализирует необходимые переменные:

```
Automat::Automat(bool DoLog)
{
    y=0;
    Nest=NULL;
    BaseIndent=0;
    this->DoLog=DoLog;
}
```

Используемая в тексте конструктора переменная Nest требуется для работы с вызываемыми автоматами. Вызов автомата выполняется с помощью функции DoNested, реализованной следующим образом:

```
void Automat::DoNested(int e, Automat* A)
{
    A->Nest=this;
```

```
    A->Main(e);
}
```

Так, например, вызов автомата `B` с событием `e` из автомата `A` должен быть осуществлен, как показано ниже:

```
DoNested(e, &B);
```

При этом автомат `A`, при необходимости, сможет определить состояние автомата `B`, обратившись к переменной `B.y`, а автомат `B` – узнать состояние автомата `A`, получив значение переменной `Nest->y`.

5. Разработка автоматов – потомков класса `Automat`

При разработке классов наследников необходимо:

1. Переопределить функции `Step` и/или `PostStep`.
2. Переопределить функции протоколирования `WriteLog`, `WriteLogOnStart`, `WriteLogOnStateChanged` и `WriteLogOnFinish`.
3. Объявить экземпляры структур данных, соответствующие объектам управления.
4. Объявить объекты, реализующие автоматы, которые вызывает данный автомат.
5. Объявить целочисленные статические константы, соответствующие обрабатываемым событиям.
6. Реализовать функции входных переменных, возвращающие значения для анализа.
7. Реализовать функции внутренних переменных. Эти функции могут возвращать значения произвольных типов.
8. Реализовать функции выходных воздействий, не возвращающие значений.

При решении задачи о лифте, классы, соответствующие автоматам *A0*, *A1* и *A2*, разработаны по этому плану. Приведём в качестве примера объявление класса `A0`.

```
//////////////////////////////////////////////////////////////
// Автомат, реализующий функциональность кабины лифта

class A0 : public Automat
{
protected:
    virtual void Step(int e); // Шаг автомата
    DECLARE_NO_POSTSTEP;      // Пост-шаг не нужен

public:
    virtual void WriteLog(CString s,int indent=0);
protected:
    virtual void WriteLogOnStart(int y,int e,int indent=0);
    virtual void WriteLogOnStateChanged(int y,int y_old,int indent=0);
    virtual void WriteLogOnFinish(int y,int indent=0);

// Вспомогательные определения
public:
    // Значения индикатора движения
    typedef enum StateValuesType {NEUTRAL, GOINGUP, GOINGDOWN} StateValues;

    // Тип структуры, реализующей хранилище вызовов
```

```
    typedef struct CallsStoreType
    {
        int up[5],down[5],car[5];
        bool newcalls;
    } CallsStore;

// Устройства
public:
    unsigned floor;      // Счётчик этажей
    unsigned timer;      // Таймер
    int inactivitytimer; // Таймер бездействия
    CallsStore calls;    // Хранилище вызовов
    StateValues state;   // Индикатор движения

// Автоматы, к которым будет обращаться данный
public:
    A1 aA1; // Автомат A1
    A2 aA2; // Автомат A2

// События
public:
    const static int e0; // Пребывание в состоянии покоя, инициализация
    const static int e1; // Событие по умолчанию

// Входные переменные
protected:
    // Переменные, получаемые от счётчика этажей
    int x1_0();          // Кабина лифта находится на втором (базовом) этаже?

    // Переменные, получаемые от хранилища вызовов
    int x2_0();          // Нужно ли открыть двери на текущем этаже?
    int x2_1();          // Поступили ли новые вызовы (на предыдущем шаге)?

    // Переменные, получаемые от индикатора движения
    int x3_0();          // Значение индикатора - «Neutral»
    int x3_1();          // Значение индикатора - «GoingUp»
    int x3_2();          // Значение индикатора - «GoingDown»

    // Переменные, получаемые от таймера
    int x4_0();          // Таймер сработал?

    // Переменные, получаемые от таймера бездействия
    int x5_0();          // Таймер бездействия сработал?

// Выходные воздействия
protected:
    // Воздействия на счётчик
    void z1_0();          // Установить значение счётчика этажей равное двум
    void z1_1();          // Увеличить значение счётчика этажей на один
    void z1_2();          // Уменьшить значение счётчика этажей на один

    // Воздействия на хранилище вызовов
    void z2_0();          // Обнулить хранилище вызовов
    void z2_1();          // Отметить в хранилище обработку текущего вызова

    // Воздействия на индикатор движения
    void z3_0();          // Установить значение «Neutral»
```

```
    void z3_1();           // Изменить значение, в соответствии с текущей ситуацией
    void z3_2();           // Направить лифт на второй (базовый) этаж

    // Воздействия на таймер
    void z4_0(unsigned n); // Установить значение таймера равное n

    // Воздействия на таймер бездействия
    void z5_0();           // Выключить таймер бездействия
    void z5_1();           // Включить таймер бездействия
    void z5_2();           // Выключить таймер бездействия, если он не сработал

// Внутренние переменные
private:
    // Переменные, связанные с индикатором движения
    StateValues i3_0();    // Вычислить состояние индикатора движения, адекватное
                           // текущей ситуации
};
```

Большую часть исходного кода программы *Lift* занимает реализация интерфейса. Отметим, что вызовы главной функции автомата *A0* из интерфейсной части программы происходят только в двух случаях: при повторной инициализации автомата и при выполнении шага. Они инициируются нажатием пользователем на кнопки «*Reset*», «*Step*», «*Pass*» или «*Auto*». Другие автоматы из интерфейсной части не вызываются. Подробно интерфейс программы будет описан в следующем разделе.

6. Интерфейс программы

Опишем внешний вид окна программы (рис. 7).

В его левой части находится область визуализации системы, предназначенная для отображения лифта, этажей, кнопок вызова на этажах и людей. Справа расположены средства управления моделью, а сверху – главное меню. Перечислим упомянутые средства управления:

- кнопка *Auto* предназначена для перехода в режим автоматического выполнения итераций и выхода из него. В этом режиме происходит постоянное выполнение пассов (наборов итераций) одного за другим. Паузу между ними можно настроить;
- флажок *Live* предназначен для включения/выключения режима случайного автоматического добавления людей на этажи;
- поле ввода *Pause between passes in seconds* определяет паузу (в секундах) между пассами при работе системы в автоматическом режиме;
- кнопка *Pass* позволяет выполнить набор итераций;
- поле ввода *Number of steps per pass* определяет количество итераций в пассе;
- кнопка *Step* позволяет выполнить итерацию;
- кнопка *Reset* обеспечивает выполнение сброса системы (перевод каждого автомата и их объектов управления в начальные состояния). В результате, лифт пуст, находится на втором этаже, все двери закрыты, людей на этажах нет и вызовов нет;
- кнопка *View log* позволяет открыть/закрыть окно отображения протокола;
- кнопка *Add human* позволяет добавить человека на этаж. При этом появляется диалоговое окно для задания его параметров, описанных ниже;
- поле *Time* отображает номер текущей итерации;
- поле *Men on the floors* отображает число людей на этажах;

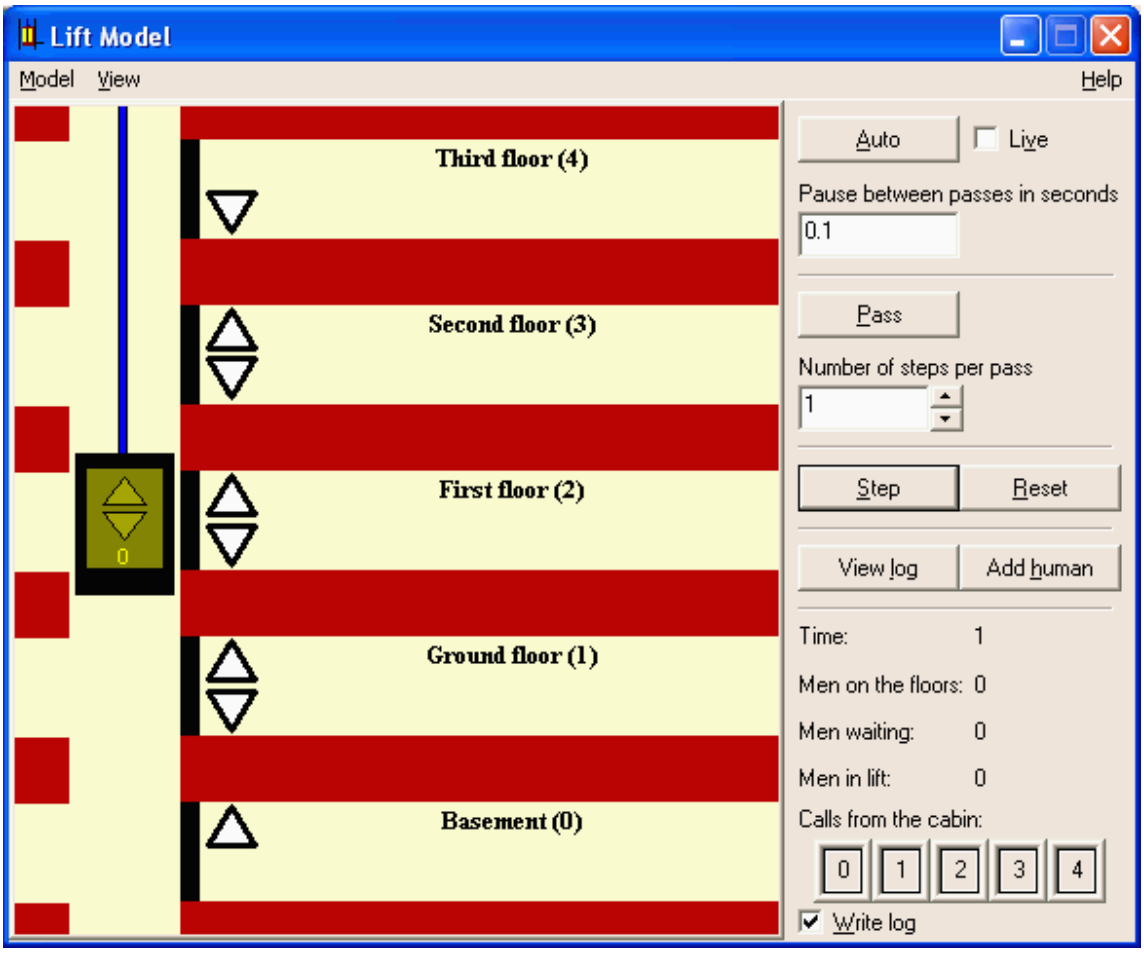

Рис. 7. Окно программы *Lift*

- поле *Men waiting* отображает число людей в очереди ожидания, размещение которых на этажах было запланировано на будущее;
- поле *Men in lift* отображает число людей в лифте. Это число также изображается на кабине лифта;
- панель *Calls from the cabin* с пятью кнопками представляет собой пульт управления в кабине лифта. При этом нажатые кнопки подсвечиваются;
- флажок *Write log* предназначен для включения/выключения режима протоколирования.

На рис. 8 изображено окно программы после появления людей на этажах. На нем также приведено окно настройки свойств добавляемого человека, открывающееся при нажатии кнопки *Add human*, которое позволяет изменять следующие параметры «личности»:

- *Name* – имя человека. Имя вида «*Human*#...» генерируется автоматически, но, при необходимости, может быть изменено вручную. Этот параметр используется только при протоколировании;
- *Current floor* – этаж, на котором следует разместить человека;
- *Target floor* – этаж, на который должен попасть человек;
- *Wait for* – время ожидания лифта (в итерациях), по истечении которого человек покидает этаж, не дождавшись лифта;

- *Enter model at* – итерация, на которой человека следует разместить на этаже. Значение по умолчанию – текущая итерация. Если указанное значение не редактировать, то человек будет размещен незамедлительно.

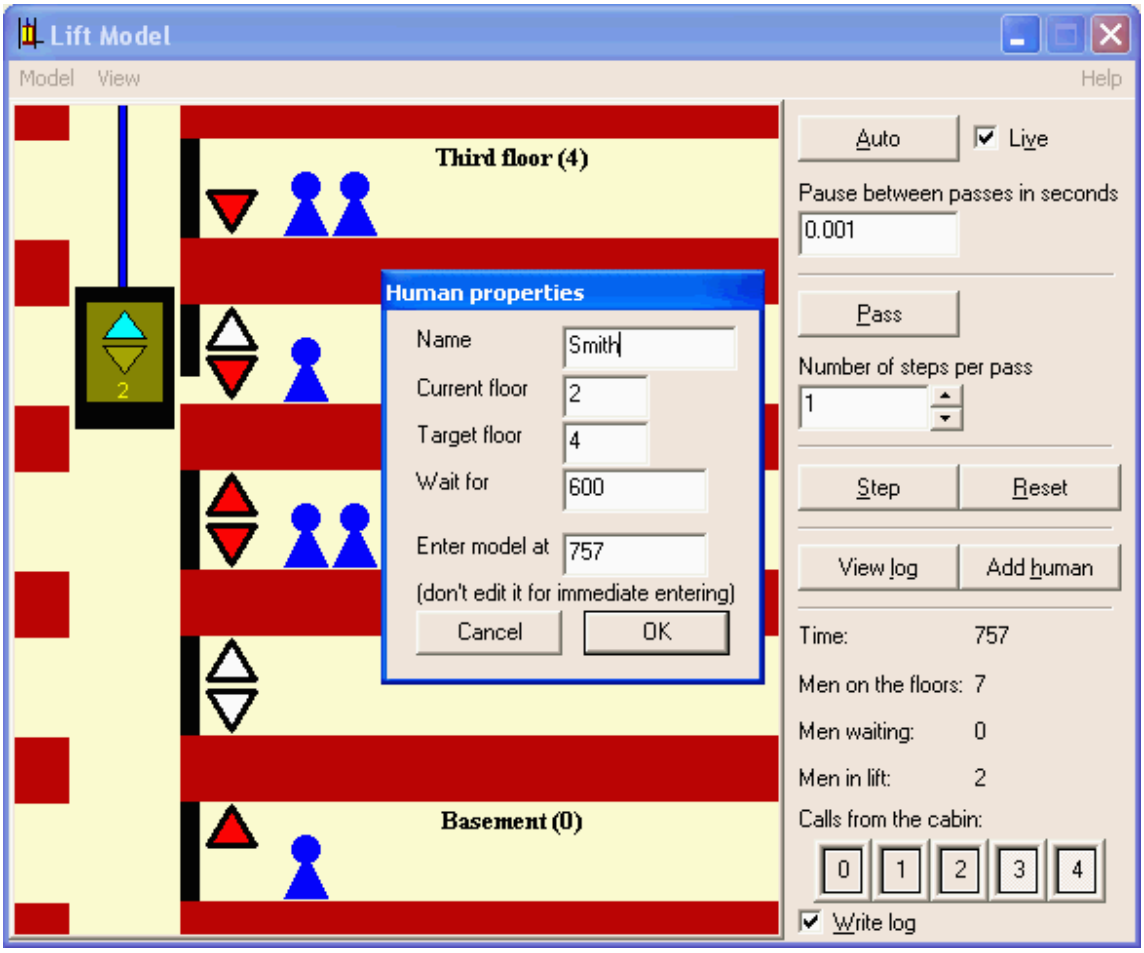

Рис. 8. Окно программы *Lift* после добавления людей

Нажатие правой кнопки мыши на изображении человека приводит к отображению окна просмотра его параметров.

Когда человек появляется на этаже, он «нажимает» на кнопку вызова для движения вверх или вниз, а при входе в кабину он «нажимает» на кнопку на пульте управления.

Отметим, что пользователь программы может нажимать кнопки вызовов с этажей и кнопки пульта управления с помощью мыши.

Обратим внимание, что нажатие на кнопку отменить невозможно и вызов, рано или поздно, будет обработан. Так, например, если человек появился на этаже, вызвал лифт и ушел, не дождавшись его прибытия, то лифт все равно приедет на этаж, и двери откроются.

На кабине лифта, в виде двух стрелок, изображен индикатор направления движения лифта.

В случае, если на этаже пытаются разместиться более десяти человек, то изображаются только первые десять.

Как отмечалось выше, в системе имеются средства для ведения и просмотра протокола работы модели. Окно просмотра изображено на рис. 9.

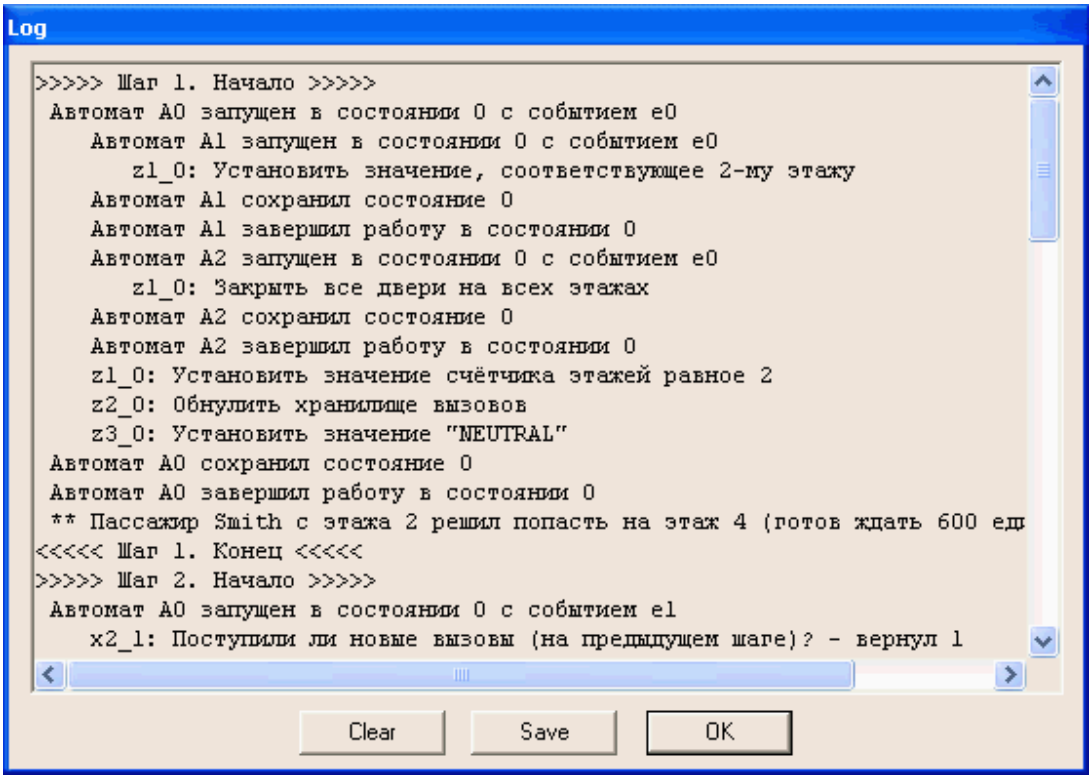

Рис. 9. Окно просмотра протокола

В протоколе итерации отделены друг от друга специальными скобками. Вложенность автоматов отражается отступами. Информация о входных и внутренних переменных, а также выходных воздействиях добавляется с еще большими отступами.

Строка, начинающаяся с двух звездочек, содержит информацию, являющуюся внешней для системы управления лифтом, и может представлять собой одно из следующих сообщений:

- «Пассажир … с этажа … решил попасть на этаж … (готов ждать … единиц времени)»;
- «Пассажир … вошел в лифт на этаже …»;
- «Пассажир … приехал на этаж …»;
- «Пассажир … решил пойти пешком».

Информация о нажатии кнопок вызова пользователем фиксируется в протоколе с помощью сообщений:

- «Поступил вызов на движение (вверх|вниз) с … этажа»;
- «Из кабины лифта поступил вызов на движение на … этаж».

Опишем назначение кнопок в окне протоколирования. Кнопка *Clear* предназначена для очистки протокола, кнопка *Save* позволяет записать протокол в текстовый файл, а кнопка *OK* – закрыть окно.

Завершим описание интерфейса рассмотрением главного меню. В нем пункт *Reset* подменю *Model* повторяет функциональность кнопки *Reset*, а пункт *Exit* того же подменю служит для выхода из программы.

Подменю *View* содержит три пункта:

- *Log window* аналогичен кнопке *View log*, описанной выше;
- *Timeouts tuning window* открывает окно, позволяющее настраивать временные параметры, описанные в табл. 1 (кроме параметра *TWaitLimit*). По умолчанию эти параметры настроены, как в работе [1];
- *Add human window* аналогичен кнопке *Add human*, описанной выше.

Подменю *Help* содержит единственный пункт *About*, отображающий информацию о программе.

На этом завершается описание созданной модели лифта, функционирующей на персональном компьютере. Ее работоспособность и соответствие техническому заданию (разд. 2) подтверждаются использованием.

7. Переход от объектного варианта программы к процедурному

При переходе от модели (разработанной выше программы) к реальному управлению лифтом на базе PC-подобного контроллера достаточно создать наследников классов, реализующих автоматы модели, переопределив лишь виртуальные функции входных и внутренних переменных, а также выходных воздействий. Таким образом, реализацию реальной системы на указанном вычислительном устройстве следует наследовать от модели!

В случае построения системы на микроконтроллерах, такой подход неприменим, так как они, обычно, программируются процедурно. Изложим методику, позволяющую перенести программу, написанную в рамках предлагаемого подхода на языке *C++* на язык *C*, и проиллюстрируем ее примером переноса ядра программы *Lift* на микроконтроллер *Siemens SAB 80C515*. При этом в качестве среды разработки программного обеспечения для микроконтроллеров используется продукт *Keil µVision 2*.

Опишем эту методику.

1. Создать каталог, в котором будет размещен переносимый проект (в рассматриваемом примере он назван *Hard*). Скопировать в каталог файлы, реализующие автоматы (*A0.h*, *A0.cpp*, *A1.h*, *A1.cpp*, *A2.h* и *A2.cpp*). Изменить расширение файлов с «*cpp*» на «*c*». Создать файл «*Common.h*» и файл проекта *µVision* (*Lift.uv2*) для требуемого микроконтроллера, добавив в последний все упомянутые (семь) файлов.
2. Из каждого файла с исходным кодом, имеющим расширение «*c*», удалить директивы `#include`, кроме тех, которые включают одноименные заголовочные файлы или заголовочные файлы стандартных библиотек. Во все файлы с исходным кодом следует добавить директиву «`#include "Common.h"`», а в файл *Common.h* скопировать необходимые общие определения макросов, структур данных из файла *stdafx.h* и т.п.
3. Преобразовать определения классов в заголовочных файлах. Для этого удаляются ключевые слова `class`, а методы преобразуются в функции с именами вида «`<имя класса>_<имя функции>`». Например, методы `Step` для трех рассматриваемых автоматов будут преобразованы в функции `A0_Step`, `A1_Step` и `A2_Step`. Кроме того, необходимо удалить из определений макросы «`DECLARE_NO_...`» и удалить или преобразовать функции протоколирования, перенаправив их вывод;
4. События, устройства и другие члены классов преобразуются по аналогии с методами, как указано в предыдущем пункте. В результате они превратятся в константы и статические переменные с определенными префиксами в названии. При условии сохранения уникальности имен, префиксы можно опустить. Таким образом, для того, чтобы обратиться к некой сущности, бывшей до преобразования членом класса, следует писать

имя класса, за ним – подчеркивание (вместо точки) и имя члена класса. Кроме того, для автомата *Ai* необходимо добавить в заголовочный файл переменную «`static int Ai_y`», в которой будет храниться номер состояния автомата. Объявления объектов, реализующих вложенные автоматы, необходимо поменять на объявления их главных функций, как внешних (*extern*).

5. В файлах с исходным кодом необходимо заменить подстроку «::» на символ «_». В результате функции-члены класса получат требуемые имена. Внутри файла, реализующего автомат *Ai*, следует добавить префикс «`Ai_`» для обращений к переменной состояния, событиям, функциям входных и внутренних переменных, а также функциям выходных воздействий. Это, почти всегда, можно сделать, выполнив в текстовом редакторе замену подстроки «`y`» на подстроку «`Ai_y`» (в режиме «только слово целиком»), а также произведя следующие замены: «`e`» на «`Ai_e`», «`x`» на «`Ai_x`», «`t`» на «`Ai_t`», а «`z`» на «`Ai_z`».
6. Обращение к переменной состояния головного автомата, которое в объектном случае осуществляется посредством переменной `Nest`, следует преобразовать в обращение к внешней переменной, объявленной в заголовочном файле. При вызове вложенного автомата, обращение к функции-члену «`DoNested(e, &Ai)`» следует заменить на вызов функции «`Ai_Step(e)`». В общем случае, при необходимости обратиться к устройству, событию, состоянию или главной функции другого автомата, они должны быть объявлены с помощью директивы `extern`. Поэтому в рассматриваемом примере в файле *A0.h* должны присутствовать объявления следующих внешних переменных:

```
extern const int A1_e2;
extern int A1_y;
extern int A2_y;
```

7. Необходимо привести все конструкции в соответствие с особенностями микроконтроллера. Например, для рассматриваемого микроконтроллера не предусмотрен тип данных `bool`. Поэтому в файле *Common.h* должны появиться следующие строки:

```
typedef int bool;
static const int true=1;
static const int false=0;
```

На этом завершается преобразование ядра объектно-ориентированной программы в процедурную. Одним из важнейших свойств изложенной методики является то, что все указанные выше замены и преобразования могут быть выполнены автоматически.

Для окончательного построения программы необходимо изменить или переписать функции входных и внутренних переменных, а также функции выходных воздействий, в которых осуществляются такие операции, как, например, взаимодействие с реальными устройствами, обращения к портам ввода-вывода и т.п.

Выводы

В заключение, отметим, что предложенный подход к построению системы управления лифтом обеспечивает создание хорошо документированного и легко модифицируемого проекта.

Методика преобразования объектно-ориентированной программы в процедурную позволяет, отладив первую из них, достаточно легко перенести ее в управляющее устройство, например, на базе микроконтроллера. На сайте http://is.ifmo.ru в разделе «Статьи» размещена программа *Lift*, а также ядро программного обеспечения системы управления лифтом для микроконтроллера *Siemens SAB 80C515*, полученное из программы *Lift* с помощью описанной выше методики переноса.

Кроме работы [1], использованной авторами в качестве прототипа, известны работы [16, 17], в которых также рассмотрена разработка программного обеспечения для задачи управления лифтом.

Как и в настоящей статье, в них используются объектно-ориентированное проектирование и программирование. Однако в работе [16] лифт проектируется для трехэтажного здания, а в работе [17] – для двухэтажного. В настоящей работе, как и в прототипе, моделируется лифт для пятиэтажного здания, что является более сложным.

Кроме того, в работах [16, 17] на этаже в каждый момент времени может находиться не более одного человека, что резко упрощают логику поведения системы. В описанном проекте число людей на этажах практически не ограничено.

К недостаткам работы [16] относится также то, что состояния автомата «Управление лифтом» выделяются на основании нескольких несвязанных между собой характеристик (наличие пассажиров, степень открытия дверей и направление движения лифта). Это привело к построению весьма сложного автомата, содержащего одиннадцать состояний, взаимодействующего с рядом других автоматов. В настоящей работе выделяются только три автомата (по пять состояний), причем каждый из них отвечает за определенную составляющую системы.

Еще один недостаток работы [16] состоит в том, что при реализации автоматов объектный подход не используется в должной мере. При этом процедурная реализация каждого автомата «обертывается» в метод одного и того же класса.

Недостатком работы [17] является то, что в ней *UML*-проектирование и программирование на *C++* не связаны формально.

Предлагаемый подход лишен указанных «минусов», за счет совместного использования преимуществ объектного и автоматного подходов.

В заключение работы, отметим, что в настоящее время все шире применяется технология объектно-ориентированного проектирования, названная *Rational Unified Process* [18]. Существуют и другие технологии, например, изложенные в работах [17, 19], а также в настоящей статье.

Авторы надеются, что, ознакомившись с предлагаемым подходом, читатели согласятся со словами Б. Гейтса, что «за последние двадцать лет мир изменился», по крайней мере, в области искусства программирования лифта.

Д. Кнут тоже не «стоит на месте». Он модернизировал модель *MIX*, разработав новую архитектуру *MMIX* [20]. Однако эти усовершенствования не коснулись области проектирования программ.

Литература

1. **Кнут Д.** Искусство программирования. Т. 1: Основные алгоритмы. М.: Вильямс, 2001. – 712 с.
2. **Корн Г., Корн Т.** Справочник по математике для научных работников и инженеров. М.: Наука, 1978. – 820 с.

3. **Дейкстра Э.** Дисциплина программирования. М.: Мир, 1979. – 239 с.
4. **Грис Д.** Наука программирования. М.: Мир, 1984. – 416 с.
5. **Буч Г.** Объектно-ориентированный анализ и проектирование. М.: Бином, СПб.: Невский диалект, 1998. – 558 с.
6. **Страуструп Б.** Язык программирования C++. М.: Бином, СПб.: Невский диалект, 2001. – 1098 с.
7. **Шалыто А.А.** Новая инициатива в программировании. Движение за открытую проектную документацию // Мир ПК. 2003. № 9. – С. 52–56. (http://is.ifmo.ru, раздел «Статьи»).
8. **Тэллес М., Хсих Ю.** Наука отладки. М.: Кудиц-образ, 2003. – 556 с.
9. **Шалыто А.А.** Switch-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. – 628 с.
10. **Любченко В.С.** Конечно-автоматная технология программирования // Труды международной научно-методической конференции «Телематика'2001». СПб.: СПбГИТМО(ТУ), 2001. – С. 127–128.
11. **Сацкий С.** Дизайн шаблона конечного автомата на C++ // RSDN Magazine. 2003. № 1. – С. 20–24.
12. **Шалыто А.А., Туккель Н.И.** Программирование с явным выделением состояний // Мир ПК. 2001. № 8. – С. 116–121. № 9. – С. 132–138. (http://is.ifmo.ru, раздел «Статьи»).
13. **Шалыто А.А., Туккель Н.И.** Танки и автоматы // BYTE/Россия. 2003. № 2. – С. 69–73. (http://is.ifmo.ru, раздел «Статьи»).
14. **Шалыто А.А.** Технология автоматного программирования // Мир ПК. 2003. № 10. – С. 74–78. (http://is.ifmo.ru, раздел «Статьи»).
15. **Брауэр В.** Введение в теорию конечных автоматов. М.: Радио и связь, 1987. – 392 с.
16. **Наумов А.С., Шалыто А.А.** Система управления лифтом. Проектная документация. http://is.ifmo.ru, раздел «Проекты». – 51 с.
17. **Дейтел Х.М., Дейтел П.Дж.** Как программировать на C++. Третье издание. М.: Бином, 2003. – 1152 с.
18. **Крачтен Ф.** Введение в Rational Unified Process. М.: Вильямс, 2002. – 198 с.
19. **Шопырин Д.Г., Шалыто А.А.** Объектно-ориентированный подход к автоматному программированию. http://is.ifmo.ru, раздел «Проекты». – 57 с.
20. **Еремин ЕА.** MMIX – учебный RISC-процессор нового тысячелетия от Дональда Кнута // Информатика. 2002. № 40. – С. 18–27.