Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

C#

Объектно-ориентированный язык программирования

Пособие к практическим занятиям - №3

Проф. Забудский Е.И.

Москва 2005

Тема 3. Объектно-ориентированное программирование: практический пример

Три практических занятия
(6 часов)

Изучаются две ключевые концепции ООП: *абстракция* и *инкапсуляция*. Выполняется объектно-ориентированный анализ модели лифтовой системы. Рассматривается объектно-ориентированная C#-программа модели лифтовой системы и полная UML диаграмма классов этой программы.

Информация, полученная от программы, моделирующей лифт, нужна для ответа на вопрос: "Выполняет ли моделируемая система лифта свою задачу по перевозке пассажиров между этажами должным образом?"

Другими словами, компьютерная модель с соответствующими характеристиками позволяет оценить, насколько хорошо будет работать реальная система.

Unified Modeling Language (UML) – популярный язык графического моделирования, используемый для представления объектно-ориентированных программ (**www.omg.org/uml**).

// **КОММЕНТАРИЙ**. На сайте http://www.firststeps.ru/ “Первые шаги” представлено много интересных обучающих материалов по различным интегрированным средам и языкам программирования, в том числе представлены C# и платформа .NET (step by step).

Содержание

1. Абстракция и инкапсуляция

В основе C# лежит концепция объектно-ориентированного программирования (ООП). Практически, все в C# является объектом. Прежде чем обращаться к первому примеру программы C# познакомимся с двумя ключевыми концепциями ООП: абстракцией (*abstraction*) и инкапсуляцией (*encapsulation*).

1.1. Абстракция

Рассмотрим, например, самолет. Сначала задача представить самолет компьютерной программой может показаться невыполнимой: он содержит огромное число деталей (например, двигатели или чрезвычайно сложные бортовые системы). Однако, рассмотрев роль, которая отводится самолету в приложении, можно значительно уменьшить число представляемых свойств. Возможно, самолет требуется лишь как точка на карте. Или же нужно рассчитать его аэродинамические характеристики — в этом случае достаточно ограничиться представлением лишь его формы. Возможно, также, что дизайнер создает трехмерную презентацию интерьера самолета, тогда программа должна представить лишь внутренние поверхности самолета.

В каждом из приведенных случаев можно указать ключевые характеристики, важные для конкретного приложения. *Таким образом, абстрагируясь от определенных деталей, можно понизить сложность самолета*.

Абстракция — одна из фундаментальных концепций, используемых для упрощения сложных задач.

Когда объект (или система) описывается более простым, менее детализованным образом, чем в реальности, такой способ описания **называется абстракцией**. Выделение свойств, важных для реальной задачи, и усечение неважных является полезной абстракцией в ООП.

При абстрагировании важно включить в рассмотрение лишь те свойства, которыми обладает описываемый объект. Поведение объекта не должно выходить за пределы ожидаемого. Например, создания абстрактного автомобиля со способностью создавать архитектурные планы следует избегать. Это вносит путаницу и для автора, и для программистов, пытающихся понять логику исходного кода.

// **ПРИМЕЧАНИЕ**

Обратимся к обсуждению модели лифта (стр. 7, 8, Пособие к практическому занятию 1). При создании моделирующей программы следует, прежде всего, игнорировать (посредством абстракции) все несущественные детали, относящиеся к реальному миру. Например, цвет каждого лифта или прическу пассажира можно при рассмотрении отбросить. С другой стороны, важной характеристикой является скорость лифта и число людей, которые его ждут, а также этажи, на которые они хотят переместиться.

Модель позволяет подсчитать статистику, связанную со временем ожидания, и оценить количество и типы лифтов, необходимые для обслуживания здания.

1.2. Инкапсуляция

В то время как *абстракция служит для снижения уровня сложности*, с которой объ екты в программе представляют реальные объекты, инкапсуляция служит непосредственно разработке кода C#. Это *мощный механизм*: ***1)*** снижения сложности и ***2)*** защиты данных отдельных объектов на программном уровне.

Инкапсуляция — это процесс объединения данных и действий (методов) в единый элемент (см. рис. 1.2 и рис. 1.3, Пособие к практическому занятию 1). **В ООП, а значит, и C#, такой единицей является *объект*.**

Инкапсуляция — это механизм *сокрытия*:***1)*** переменных экземпляра и ***2)*** неважных методов класса от других объектов. Открытыми являются лишь необходимые другим **методы объекта класса**.

Далее приведен полезный пример, вновь связанный с лифтами.

1.2.1. Инкапсуляция в реальной системе лифтов

Любой лифт обладает механизмом (кнопками), который позволяет пассажиру выбрать этаж назначения. Нажатие кнопки —простое действие для пользователя, но не для лифта, так как к последнему часто поступают одновременные противоречивые запросы от различных пассажиров. Поэтому не так просто определить, на какой этаж следует переместиться, и выполнить все запросы надлежащим образом. Разработку лифта затрудняют многие условия; несколько примеров этому приведено в примечании.

// **ТЯЖЕЛАЯ ЖИЗНЬ ЛИФТА**

Современные лифты пользуются достаточно сложными алгоритмами, управляющими каждым их перемещением. Вот несколько причин, по которым сложно выбрать, куда должен перемещаться лифт.

Пассажир **A** вошел в лифт на этаже **3** и запросил перемещение на этаж **30**. Пассажир **B** до этого нажал кнопку на первом этаже, запросив лифт. Если лифт сразу пойдет на этаж **30** с пассажиром **A** (без **B**), первый доберется до точки назначения быстрее всего. Пассажир **B** будет вынужден ждать. Возможно, пройдя всего два этажа вниз, лифт мог бы доставить обоих пассажиров на верхние этажи, экономя, таким образом, время и ресурсы. Вместо этого лифт движется на **30**, а затем возвращается за пассажиром **B**. При большем числе людей задача усложняется еще сильнее.

Как минимум, два аспекта затрудняют работу лифта:

1. Для доставки людей на различные этажи работает несколько лифтов. Им требуется "совместный" подход, чтобы выполнить наибольшее возможное число запросов. Плотно заполненные здания с высоким трафиком требуют примерно два лифта на каждые три этажа, т.е., например, 60-этажное здание требует около 40 лифтов.

2. Некоторые этажи посещаются и присыпают запросы чаще, чем другие. Вероятность посещения может быть различной в течение дня. Если лифты могут каким-либо образом реализовать подобные схемы, эффективность их перемещения между этажами возрастет.

Для управления перемещением лифтов и доставки наибольшего возможного числа пассажиров разработаны *сложные алгоритмы*.

Здесь можно сделать важное наблюдение.

Пассажир, нажимая кнопку, использует **скрытые** от него сложные службы лифта (алгоритмы, двигатели, гидравлику и т.д.).

Такое скрытие данных, алгоритмов и механизмов лифта имеет достоинство: ***1)*** не только для пассажиров, которым не нужно заботиться о сложных вопросах, не связанных с их деятельностью, ***2)*** но и для самого лифта, так как оно повышает его надежность. Неуполномоченное и некомпетентное лицо не имеет доступа к внутренним механизмам управления. В ином случае, если бы пассажиры могли каким-либо образом влиять на алгоритм движения, ситуация в скором времени стала бы хаотической.

Заключительное достоинство отделения кнопок от базового механизма лифта — возможность вносить изменения в механизм, не затрагивая кнопки (а значит, и методику

управления). Таким образом, каждый лифт содержит привычный всем набор кнопок, тогда как механизмы и алгоритмы, естественно, с годами улучшались.

1.2.2. Инкапсуляция в компьютерной модели лифта

Рассмотрим, как все обсуждавшиеся ранее вопросы связаны с объектно-ориентированной программой моделирования лифта на C#.

Такая программа может содержать, например, объекты **Elevator** и **Person**. Объект **Person** может выполнять эквивалент действия "нажатие кнопки" над объектом **Elevator**, посылая, таким образом, запрос нужного этажа.

Объект **Elevator** модели, как и настоящий лифт, может содержать множество сложных методов и важных данных. Многие объекты **Elevator**, чтобы модель была более реалистичной, оснащены, например, алгоритмами (подобными реальным) *для вычисления наиболее эффективной последовательности этажей.*

Следует обратить внимание на важную аналогию: каждому объекту **Person** модели (и программисту, создающему его) нет необходимости "знать" о внутренней структуре (ис ходном коде) объекта **Elevator**. Все, о чем ему нужно "заботиться" (как и реальному пассажиру), — это перемещение с этажа на этаж. Ему не разрешено работать с внутренними структурами (данными и исходным кодом) объекта **Elevator**. А программисты,создающие объект лифта, имеют возможность изменять исходный код, не влияя на то, как используется объект **Elevator**.

В мире ООП такое скрытие внутренних структур называют **инкапсуляцией базовых данных и исходного кода**, которые требуются только объектам класса **Elevator**.

При создании объектов **Elevator** непосредственно на языке C# требуется способ указания программе и другим объектам, включая **Person**, что данные и исходный код скрыты. Для этой цели в C# зарезервировано ключевое слово **private**. Таким образом, когда данные и блок исходного кода объявлены закрытыми (**private**), никакие объекты кроме **Elevator** не имеют доступа к ним.

1.2.3. Ключевые слова private и public

Слово **private** имеет специальное значение для компилятора C#. Оно используется для сокрытия методов и переменных экземпляра (расположенных внутри объекта) от других объектов.

Аналогично, слово **public** также имеет специальное (противоположное по смыслу) значение для компилятора C#. **public** открывает другим объектам доступ к методам и переменным экземпляра текущего объекта.

Поскольку ключевые слова **private** и **public** управляют доступностью методов и переменных экземпляра объекта, их называют *спецификаторами доступности* (**access modifiers**).

Однако если спрятать все переменные экземпляра и методы объекта, он станет совершенно бесполезным, так как им нельзя будет воспользоваться. Например, объекту **Person** требуется возможность "запросить" нужный этаж. Метод, реализующий эту функциональность, называется **NewFloorRequest**. Что, если он объявлен как **private**? Объект **Elevator** с такими характеристиками, который "загрузил" "пассажира" (объект **Person**), не сможет обратиться к методу **NewFloorRequest**, представляющему единственный способ определить этаж, на который хочет переместиться "пассажир". Это эквивалентно тому, что реальный пассажир просто не может нажать кнопку. В исходном коде C# требуется способ

снабдить объект **Person** руками, а **Elevator** — кнопками, чтобы эти объекты можно было "привести в контакт".

Для этого методы наподобие **NewFloorRequest** делаются открытыми: вместо **private** применяется ключевое слово **public**.

Теперь объект **Person** может "зайти" в объект **Elevator**. При переходе в состояние готовности объект **Elevator** может послать сообщение объекту **Person** и, посредством метода **NewFloorRequest**, получить запрос на нужный этаж от **Person**. Такая ситуация иллюстрируется рис. 3.1. Следует обратить внимание на “жирную” стрелку — она обозначает сообщение, пересланное объекту **Person** (вызов метода). “Тонкая” стрелка, указывающая в обратном направлении, представляет значение (запрос этажа), возвращаемое объекту **Elevator** объектом **Person**. “Жирная” стрелка отображает вызов метода. Программисты в таких случаях говорят, что объект **Elevator** вызывает метод **NewFloorRequest**.

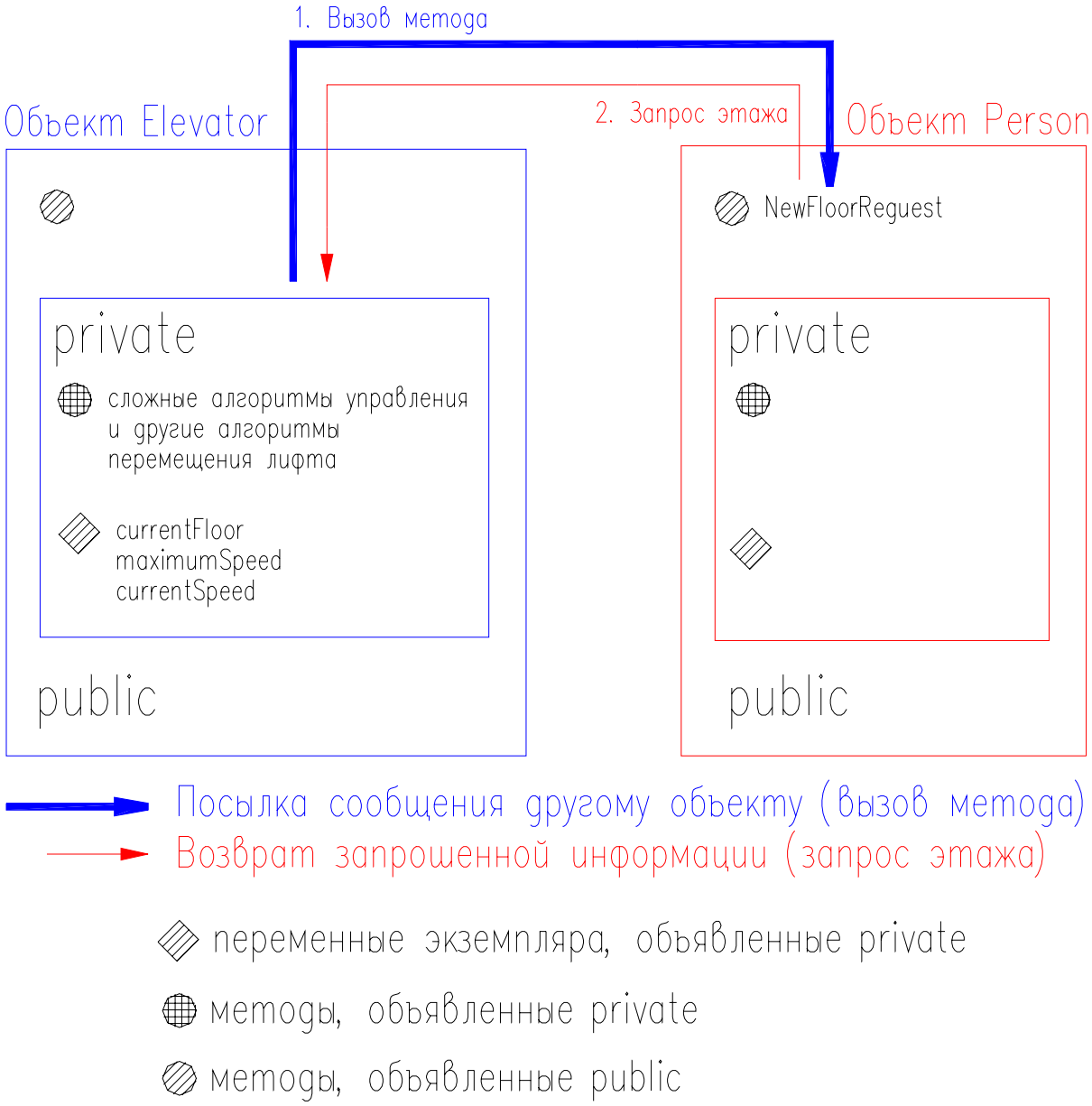


Рис. 3.1. *Объект* Elevator запрашивает информацию у *объекта* Person

Объекту **Elevator**, как и реальному лифту, может потребоваться информация о текущей и максимальной скорости, — она хранится, как упоминалось ранее, в переменных экземпляра (данных) объекта. Как и для лифтов в реальном мире, для объектов **Elevator** в модели C# какое-либо взаимодействие со сторонними объектами, приводящее к изменению значений переменных экземпляра, нежелательно. Поэтому они объявлены в объекте **Elevator** как **private**, что показано на рис. 3.1. Такое объявление делает эти переменные недоступными (currentFloor, maximumSpeed, currentSpeed) для других объектов.

1.2.4. Инкапсуляция в типичном классе

Обратимся вновь к объектам, изображенным на рис. 1.2 (см. Пособие к практическому занятию 1). Они эквивалентны объектам, приведенным на рис. 3.1. На рис. 3.2 показан обобщенный объект, использующий спецификаторы доступности и инкапсулирующий открытый (**public**) слой. Следует обратить внимание на то, что открытая часть объекта видна, а закрытая — нет. Объект содержит инструкции, исполняемые компьютером (потоком исполнения или алгоритмами). Теперь они разделены на две части: открытую и закрытую. Все данные (или переменные экземпляра) спрятаны в закрытой части.

// **КЛАСС И ОБЪЕКТ**

Класс представляет собой шаблон для создания объектов. Для конструктора лифтов чертеж является тем же, чем класс — для программиста.

Все объекты одного класса содержат одинаковые методы и переменные экземпляра. Таким образом, класс является логической конструкцией. Объект же обладает способностью предпринимать действия, заданные в классе.

Все объекты одного класса называются его **экземплярами**. Когда объект формируется в процессе исполнения программы, говорят, что он создается или ***порождается*** **его** ***экземпляр***.

Теперь типичный класс можно представить так, как показано на рис. 3.3. В упрощенном рассмотрении класс состоит из данных (переменных экземпляра) и алгоритмов (методов). При написании исходного кода класса указываются все методы и переменные экземпляра. В результате их содержат все объекты, производные от этого класса. Все переменные экземпляра и методы в совокупности называются членами (*members*) или полями (*fields*) класса.

Инкапсуляция определяет слой, предназначенный для взаимодействия с внешним миром. Только посредством этого слоя внешние объекты могут работать с данными. *Внутри слоя содержится скрытая часть.* **Этот защитный слой называют интерфейсом**. **Он должен состоять *только из методов***.

// **ВСПОМОГАТЕЛЬНЫЕ МЕТОДЫ**

Методы, объявленные **private**, могут вызываться только методами, принадлежащими тому же объекту. Они обеспечивают функциональность и поддержку, необходимую другим методам (зачастую объявленным как **public**). Поэтому закрытые методы нередко называют вспомогательными методами.

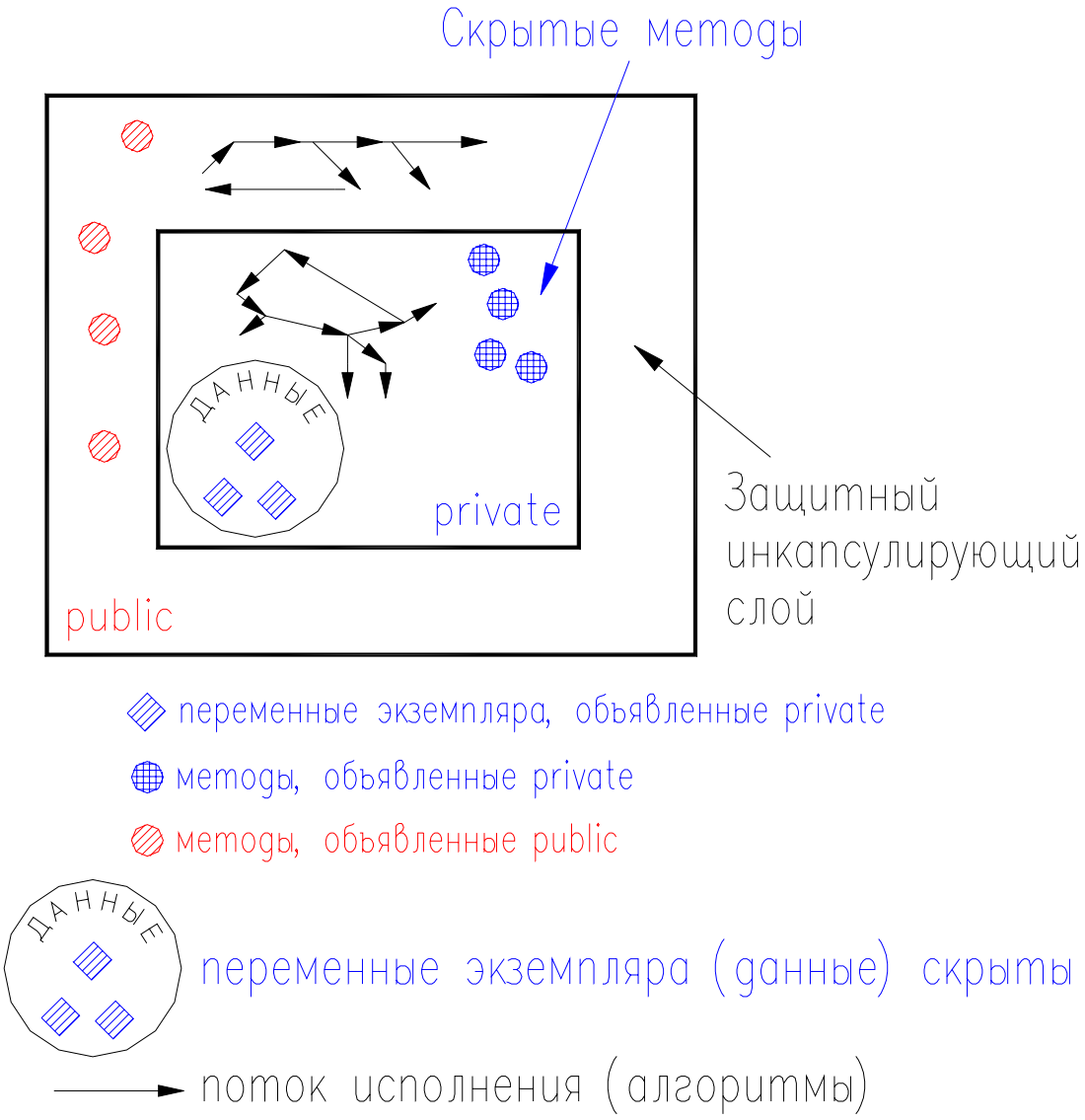


Рис. 3.2. Объект общего вида

1.2.5. Основные достоинства инкапсуляции

1. *Она обеспечивает слой абстракции*. Инкапсуляция освобождает программиста, использующего класс, от необходимости подробно знать, как он реализован.

2. *Она разделяет:* ***а)*** *интерфейс объекта и* ***б)*** *его реализацию*. Поэтому можно улучшить, например, реализацию объекта, оставив его интерфейс без изменений.

3. *Она покрывает объект защитным слоем*. Инкапсуляция защищает данные внутри объекта от нежелательного доступа, который может привести к их повреждению или некорректному использованию.

// **ИНТЕРФЕЙС ОБЪЕКТА ДОЛЖЕН СОДЕРЖАТЬ ТОЛЬКО МЕТОДЫ, НО НЕ ПЕРЕМЕННЫЕ ЭКЗЕМПЛЯРА**

Переменные экземпляра, размещенные в интерфейсе (это возможно в C#) нарушают защитную оболочку и позволяют другим объектам работать с этими открытыми данными. Если требуется доступ к определенным данным, он должен быть реализован только посредством метода (или свойств и индексаторов, которые обсуждаются в последующем).

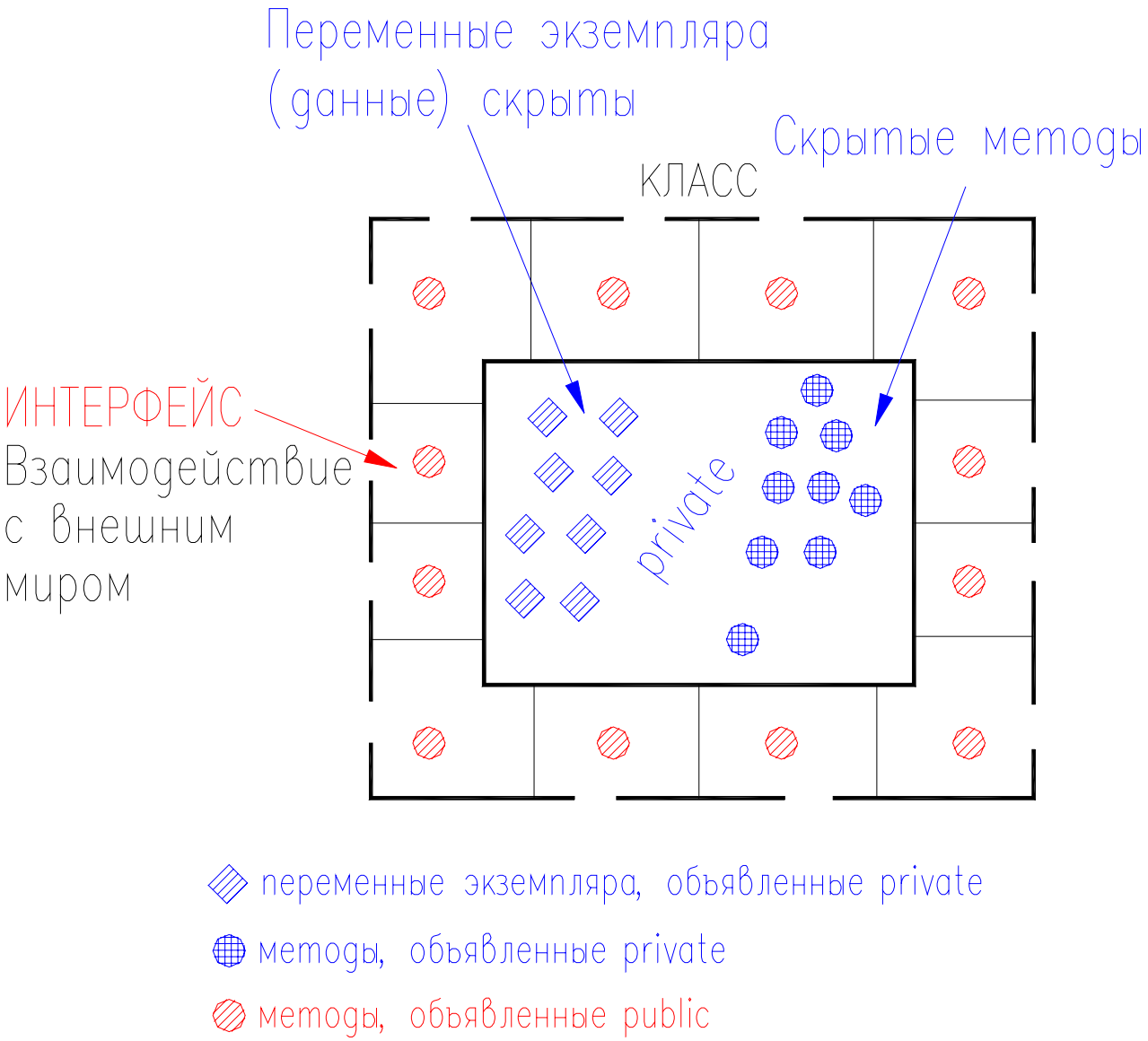


Рис. 3.3. Иллюстрация инкапсуляции *для класса* общего вида

2. Некоторые соображения по моделированию лифта

Представленная ниже программа на C# является первой попыткой реализации объектно-ориентированной модели лифта. Проведенный выше анализ подготовил для восприятия этого практического примера.

2.1. Концепции, цели и решения в программе, моделирующей лифт: сбор важной статистики

Информация, полученная от программы, моделирующей лифт, нужна для ответа на вопрос: "Выполняет ли моделируемая система лифта свою задачу по перевозке пассажиров между этажами должным образом?" Другими словами, компьютерная модель с соответствующими характеристиками позволяет оценить, насколько хорошо будет работать реальная система.

// **СТАТИСТИКА**

Числовые данные (значения), полученные от некоторого образца, называют статистикой.
Статистика, как наука, занята сбором, обработкой, классификацией, анализом и интерпретацией числовых данных, а также выявлением закономерностей сводных показателей с использованием методов теории вероятности.
Термин "статистика" часто используется для указания на собственные собранные данные.

Два статистических показателя считаются важными при оценке лифта:

1) Время, которое пассажир затрачивает на ожидание после нажатия кнопки вызова лифта.

2) Среднее время, необходимое для перемещения на один этаж.

Идея получить эти значения посредством моделирующей программы выглядит следующим образом:

объекты класса **Person** "путешествуют" по системе, именно их обслуживают объекты **Elevator**. Следовательно, объекты **Person** могут собрать статистику и "узнать", насколько хорошо работает система. Аналогичным подходом в реальном мире был бы опрос пассажиров лифта.

В каждом объекте **Person** из программы, моделирующей лифт, следует реализовать переменные экземпляра, в которых будет храниться общее время ожидания вне лифта (ответ на **первый** пункт) и среднее время перемещения на один этаж (**второй** пункт). Эти величины будут частью статистики, собранной в модели, и дадут оценку работе системы. Переменные называются **totalWaitingTime** и **averageFloorTravelingTime**.

Вычисление **totalWaitingTime** требует наличия внутри объекта **Person** метода, запускающего секундомер каждый раз, когда пассажир "нажал кнопку" вызова лифта. Как только объект **Elevator** "прибудет" к месту назначения (то есть к пассажиру), секундомер останавливается, и полученное время добавляется к текущему значению **totalWaitingTime**.

Аналогично **averageFloorTravelingTime** вычисляется другим методом внутри **Person**, который запускает секундомер, как только **Person** "входит в лифт" (объект **Elevator**). Когда лифт достигает "пункта назначения", секундомер останавливается, и полученное время делится на число этажей, чтобы узнать среднее время перемещения на один этаж. Этот результат сохраняется в списке. При определении окончательной статистики метод подсчитает среднее всех чисел **averageFloorTravelingTime**, сохраненных в списке.

Все запуски и остановки секундомеров, сложение чисел, вычисление средних значений и прочие подробности не нужны остальным участникам модели, к тому же они могут чрезмерно усложнить восприятие для других программистов. **Следовательно, все перечисленные методы нужно спрятать, объявив их private.**

Каждый объект **Person** должен уметь сообщить **totalWaitingTime** и **averageFloorTravelingTime**. Для этого применяются два метода, объявленных как **public**: **GetTotalWaitingTime** и **GetAverageFloorTravelingTime**. Любой объект, вызывающий эти методы, получит соответствующую статистику.

К примеру, над этим проектом может работать другой программист, который пишет класс, собирающий статистку по каждому сеансу моделирования. Называя этот класс **StatisticsReporter**, он должен быть уверен, что все объекты **Person** "опрошены" по завершении моделирования путем сбора величин **totalWaitingTime** и **averageFloorTravelingTime**. В результате, **StatisticsReporter** может просто вызвать методы **GetTotalWaitingTime** и **GetAverageFloorTravelingTime** каждого объекта **Person**, занятого в конкретном сеансе моделирования.

В итоге:

1. **GetTotalWaitingTime** и **GetAverageFloorTravelingTime** являются частью **интерфейса**, который прячет (инкапсулирует) всю ненужную внутреннюю сложность от программистов, разрабатывающих класс **StatisticsReporter**.
2. Таким же образом, переменные экземпляра объекта **Person** скрыты путем объявления их как **private**. Это предохраняет их значения от ошибочного изменения любым другим объектом, включая **StatisticsReporter**. Другими словами, методы **GetTotalWaitingTime** и **GetAverageFloorTravelingTime** "скрывают" переменные экземпляра **totalWaitingTime** и **averageFloorTravelingTime**, позволяя только читать эти значения, но не устанавливать их.

2.2. Объектно-ориентированное программирование: практический пример

До сих пор были только теоретические рассуждения об объектно-ориентированном программировании. Обсуждалась разница между классами и объектами, а также, как следует создавать и инициализировать объекты до того, как они смогут выполнить какое-либо действие.

Чтобы подтвердить предыдущие теоретические рассуждения об объектно-ориентированном программировании, ниже приводится пример, иллюстрирующий важные концепции C#.

2.2.1. Программа SimpleElevatorSimulation (листинг 3.1)

Листинг 3.1 содержит исходный текст простой программы моделирования лифта. Ее цель — продемонстрировать, как в C# выглядит объект, реализуемый из спроектированного программистом класса. В частности, здесь показано, как создается объект **Elevator**, который вызывает (строка 21) метод **NewFloorRequest** (строки 44 – 47) объекта **Person** для получения числа "этажей", на которые необходимо переместить лифт.

При разработке программы из листинга 3.1 был достигнут высокий уровень абстракции, что позволило сделать программу простой и сконцентрироваться на важнейших объектно-ориентированных частях исходного текста.

Ниже приведено описание общей конфигурации модели лифта с учетом основных отличий от реальной системы.

1. Класс **Building** имеет один объект класса **Elevator** по имени **elevatorA** (строка 53).

2. Один объект **Person** находится внутри переменной **passenger** (строка 16), "использующей" **elevatorA**.

3. Объект **Elevator** может "путешествовать" на любой "этаж", который находится в пределах типа **int** (от -2147483648 до 2147483647). Однако, объект **Person** запрограммирован выбирать случайное значение этажа в интервале от 1 до 30 (строка 47).

4. Лифт "следует" непосредственно на "этаж назначения".

5. "Перемещение" лифта **elevatorA** отображается на консоли.

6. После того как пассажир "вошел" в **elevatorA**, он остается в нем на всем протяжении сеанса моделирования и выбирает очередной этаж, когда выполнен предыдущий запрос.

7. В этой модели используются только классы **Elevator**, **Person** и **Building.**

8. В конце каждого сеанса моделирования общее число этажей, пройденных **elevatorA**, выводится на консоль (строка 31). Это значение является важной характеристикой в серьезной моделирующей программе (переменная экземпляра **totalFloorsTraveled** класса **Elevator,** соответствующая объекту **elevatorA**).

Несмотря на простоту и значительный уровень условности, из данной модели можно извлечь необходимую статистику. Это *пример того, как без лишней сложности можно создать небольшое приложение, позволяющее пользователю получить важные сведения о реальной системе лифта.*

Рекомендуется внимательно ознакомиться с листингом 3.1. Попробовать сначала выявить большие **процедуры**) Следует обратить внимание на описание трех классов (**Elevator**, **Person** и **Building**), лежащее в основе всей программы, а также отметить методы и переменные экземпляра, определенные в каждом из трех классов.

// **ПРИМЕЧАНИЕ**
Порядок, в котором методы класса написаны в исходном коде, не влияет на исполнение программы. То же самое справедливо и для последовательности классов в программе. В листинге 3.1 метод **Main** расположен последним, хотя выполняется он, очевидно, первым.

Типичный вывод программы приведен после листинга 3.1. Поскольку номера запрошенных этажей генерируются случайно, этажи " **floor** – этаж отправления", "**Traveling to** – перемещаться на этаж" (исключая начальный этаж отправления, который всегда равен **1**) и "**Total floors traveled** – общее число пройденных этажей" будут разными при каждом запуске программы.

Листинг 3.1. Исходный код. Файл class.cs. Project**SimpleElevatorSimulation** (Среда C#Builder)

```
01: // Простая программа моделирования лифта
02:
03: using System;
04:
05: namespace ProjectSimpleElevatorSimulation
06: {
07: class Elevator  // Пользовательский класс
08: {
09:     private int currentFloor = 1 ;
10:     private int requestedFloor = 0;
```

```
11:     private int totalFloorsTraveled = 0;
12:     private Person passenger;
13:
14:     public void LoadPassenger()
15:     {
16:     passenger = new Person(); // Определение специал-го метода-конструктора.
17:     }                                  // Он вызывается автоматически при создании
18:                                        // нового объекта класса Person.
19:     public void InitiateNewFloorRequest()
20:     {
21:     requestedFloor = passenger.NewFloorRequest();
22:     Console.WriteLine("Этаж отправления: " + currentFloor
23:                                + " Этаж назначения: " + requestedFloor);
24:     totalFloorsTraveled = totalFloorsTraveled +
25:                                Math.Abs(currentFloor - requestedFloor);
26:     currentFloor = requestedFloor; // Лифт переместился с текущего этажа на запрошенный
27:     }                                      // этаж, то есть перемещение лифта закончилось.
28:
29:      public void ReportStatistic()
30:     {
31:     Console.WriteLine("Общее число пройденных этажей: " + totalFloorsTraveled);
32:    }
33: }
34:/*************************************************************************/
35: class Person       // Пользовательский класс
36: {
37:      private System.Random randomNumberGenerator;
38:
39:      public Person()
40:     {
41:     randomNumberGenerator = new System.Random();
42:     }
43:
44:      public int NewFloorRequest()
45:     {
46:     // Возвращает сгенерированное случайное число
47:     return randomNumberGenerator.Next(1,30);
48:     }
49:}
50:/*************************************************************************/
51: class Building    // Пользовательский класс
52: {
53:      private static Elevator elevatorA;
54:
55:      public static void Main()
56:     {
57:     elevatorA = new Elevator();
```

```
58:      elevatorA.LoadPassenger();
59:      elevatorA.InitiateNewFloorRequest();
60:      elevatorA.InitiateNewFloorRequest();
61:      elevatorA.InitiateNewFloorRequest();
62:      elevatorA.InitiateNewFloorRequest();
63:      elevatorA.ReportStatistic();
64:    }
65:}
66: }
```

```
Этаж отправления: 1          Этаж назначения: 2
Этаж отправления: 2          Этаж назначения: 24
Этаж отправления: 24         Этаж назначения: 15
Этаж отправления: 15         Этаж назначения: 10
Этаж отправления: 10         Этаж назначения: 21
Общее число пройденных этажей: 48
```

2.2.2. Общая структура программы

Перед тем как продолжить детальный анализ программы, рассмотрим рис. 3.4. Он связывает рис. 3.1 с конкретным листингом 3.1 программы на C#.

Классы **Elevator** и **Person** из листинга 3.1 — абстрагированные версии объектов реального мира. Они изображены рядом с их представлением в C# на рис. 3.4. Каждая часть классов **Elevator** и **Person**, написанных на C# (в фигурных скобках), связана с их графическим представлением стрелками. Следует обратить внимание, как **методы public** двух классов (**интерфейс**) инкапсулируют, скрытые переменные **private**. В этом случае нет необходимости в методах, объявленных **private**.

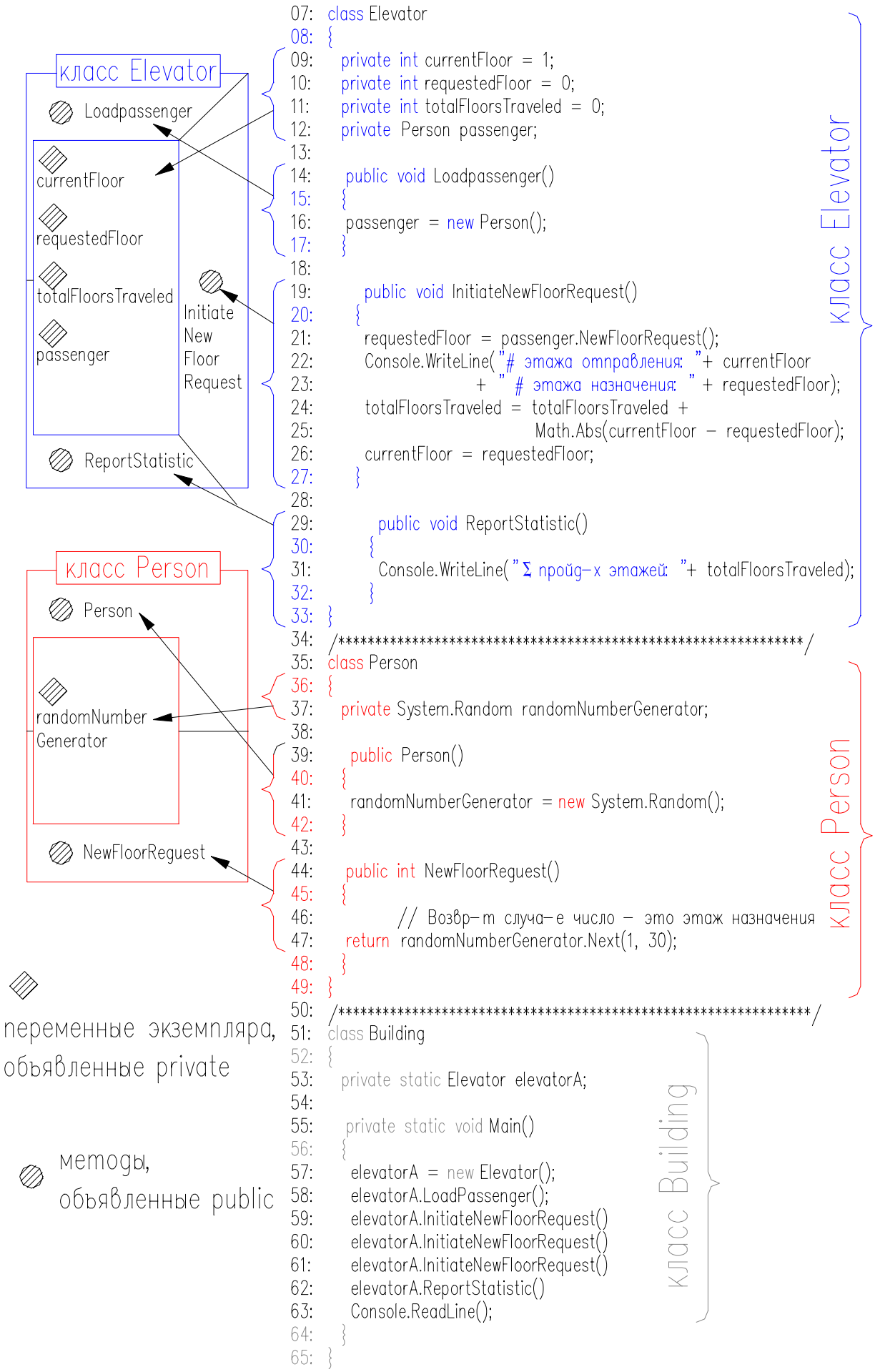


Рис. 3.4. Связь рис. 3.1 с реальной программой на C#

Класс **Building** имеет один объект класса **Elevator**, который представлен в его переменной экземпляра **elevatorA**, объявленной в строке 53. Он также содержит метод **Main**. Этот класс (**Building**) никогда не реализуется в программе как объект. Он используется механизмом выполнения **.NET**-платформы для доступа к **Main** и запуска программы.

Ниже приводится краткое пояснение некоторых строк исходного текста.

2.2.3. Листинг 3.2. Краткий анализ листинга 3.1

07: Начало определения класса **Elevator**

09: Объявление переменной экземпляра типа **int currentFloor** (этаж отправления – текущий этаж) со спецификатором доступности **private** и начальным значением 1.

12: Объявление переменной **passenger**, которая может содержать объект класса **Person**. Класс **Person** играет роль пассажира (по отношению к классу **Elevator**).

14: Начало определения метода **LoadPassenger**. Он является частью интерфейса класса **Elevator** и объявлен как **public** (интерфейс класса **Elevator** – public–методы **LoadPassenger, InitiateNewFloorRequest, ReportStatistic**).

16: Порождение (создание) нового объекта класса **Person** с использованием ключевого слова **new**. Этот объект присваивается переменной **passenger**.

19: Начало определения метода **InitiateNewFloorRequest**. Он является частью интерфейса класса **Elevator** и объявлен как **public**.

21: Вызов метода **NewFloorRequest** (см. рис. 3.1) объекта **passenger**; число, возвращенное этим методом (строка 47) присваивается переменой **requestedFloor** – этаж назначения.

22-23: Информация о "перемещениях" выводится на консоль.

24-25: Вычисление количества этажей, которое проехал лифт во время конкретного перемещения (строка 25), и прибавление его к общему числу пройденных этажей (строка 24 - totalFloorsTraveled).

26: Лифт перемещается на запрошенный этаж, когда **currentFloor** (текущий этаж – этаж отправления) присваивается значение **requestedFloor** (этаж назначения)**.**

31: При каждом вызове (строка 62) метода **ReportStatistic** (строки 29 - 32) выводится значение переменной **totalFloorsTraveled -** общее число пройденных этажей**.**

35: Начало определения класса **Person**.

37: Объявление переменной **randomNumberGenerator** для хранения объекта класса **System.Random**.

39: Начало *определения* специального метода (конструктора), *который вызывается автоматически* при создании нового объекта класса **Person**.

41: Создание нового объекта класса **System.Random** с использованием ключевого слова **new**; присваивание этого объекта переменной **randomNumberGenerator**.

44: Определение метода **NewFloorRequest**. Благодаря описанию **public** он является частью интерфейса класса **Person**. Тип возвращаемого значения — **int**.

47: **Person** (пассажир) выбирает этаж назначения, возвращая случайное число в интервале от 1 до 30.

53: В классе **Building** объявлена переменная типа **Elevator**, по имени **elevatorA**. Класс **Building** взаимосвязан с классом **Elevator** композицией.

55: Начало определения метода **Main**.

57: Создание объекта класса **Elevator** с помощью ключевого слова **new**; присваивание этого объекта переменной **elevatorA**.

58: Вызов метода **LoadPassenger** объекта **elevatorA**.

59-61: Вызов метода **InitiateNewFloorRequest** объекта **elevatorA** (три раза).

62: Вызов метода **ReportStatistic** объекта **elevatorA**.

2.2.4. Более глубокий анализ Project**SimpleElevatorSimulation**

В этом разделе обсуждаются важные моменты, связанные с листингом 3.1

2.2.4.1. Определение класса, который будет реализован в программе

Определение класса представлено на рис. 1.5 (см. Пособие к практическому занятию 1). В строке 07 приведенной программы применяется пользовательский класс (для создания конкретного экземпляра объекта **elevatorA** в программе).

07: class **Elevator**

2.2.4.2. Инициализация переменных

Новый элемент, представленный в строке 09 — это комбинация операторов объявления и присваивания. Такую запись называют инициализацией. Конструкция **private int currentFloor** — объявление, а запись **= 1** в конце строки присваивает 1 переменной **currentFloor** (этаж отправления или текущий этаж) во время создания объекта, которому принадлежит эта переменная экземпляра.

```
09:     private int currentFloor = 1;
```

Строка 09 является хорошим примером того, что зачастую переменная должна иметь начальное значение до использования в вычислениях. **Такое значение (1) отвечает движению лифта с первого этажа.**

В таких случаях говорят, что переменная инициализируется. Существует два важных метода инициализации переменной экземпляра. Можно:

1. Присвоить переменной начальное значение в ее объявлении (как в строке 09).
2. Использовать элемент C#, называемый ***конструктором***. Исходный код, содержащийся в конструкторе, исполняется **в момент создания объекта** (что является идеальным временем для любой инициализации). Конструкторы представлены далее.

Переменные экземпляра, которые не инициализированы в исходном тексте явно, автоматически получают значения по умолчанию от компилятора C#. Например, если бы переменной **currentFloor** не присваивалось значение 1 в строке 09, она получила бы значение 0 по умолчанию.

// ВСЕ ПЕРЕМЕННЫЕ СЛЕДУЕТ ИНИЦИАЛИЗИРОВАТЬ ЯВНО

При инициализации не следует полагаться на компилятор C#. Явная инициализация делает код яснее. Кроме того, код не зависит от значений по умолчанию, которые могут в дальнейшем измениться, что приведет к появлению ошибок.

// СОВЕТ

В объявлении элемента класса можно не указывать ключевые слова **public** или **private**. По умолчанию будет использоваться **private**.

Тем не менее, рекомендуется явно указывать спецификатор доступности **private**, чтобы повысить читаемость кода.

2.2.4.3. Объявление переменной, представляющей объект определенного класса

В строке 12 показано, что переменная экземпляра **passenger** может содержать объект класса **Person**.

```
12:     private Person passenger;
```

В ней еще не размещен конкретный объект **Person**, — для этого нужно использовать оператор присваивания (речь о котором пойдет ниже). Пока можно сказать только то, что объект **Elevator** может "перевозить" одного пассажира (переменная **passenger**), который относится к классу **Person**.

Перейдем к строкам 35-49. Определение класса **Person** здесь, фактически, создает новый тип. Таким образом, тип **Person** можно использовать для объявления переменной так же, как **int** или **string** (именно это и делается в строке 12).

// **ПРИМЕЧАНИЕ**

int и **string** являются встроенными, предопределенными типами. Классы, определяемые в исходном коде пользователем, являются его собственными типами.

Следует обратить внимание: строка 12 тесно связана со строками 16, 21 и 35-49. В строке 16 новый объект типа **Person** присваивается переменной **passenger**; строка 21 использует некоторую функциональность переменной **passenger** (и, таким образом, объекта Person), применяя один из его методов, а строки 35-49 определяют класс **Person**.

// **СОВЕТ**

Последовательность объявлений элементов класса может быть любой. Несмотря на это, желательно разбивать их на разделы, имеющие общие спецификаторы доступности.

Пример общепринятого стиля:

```
class ИмяКласса
{
   объявления закрытых переменных экземпляра
   определения закрытых методов
   определения открытых методов
}
```

2.2.4.4. Переменные экземпляра, представляющие состояние объекта Elevator

Строки 09-12 содержат список переменных экземпляра, описывающих объект **Elevator**.

```
09:     private int currentFloor = 1 ;
10:     private int requestedFloor = 0;
11:     private int totalFloorsTraveled = 0;
12:     private Person passenger;
```

Итак, состояние объекта **Elevator** задается такими переменными:

- *Строка 09* — переменная **currentFloor** содержит текущий этаж, на котором расположен объект **Elevator**.
- *Строка 10* — запрос нового этажа сохраняется в переменной **requestedFloor**. Объект **Elevator** выполняет этот запрос при первой возможности (строка 26), — в зависимости от скорости компьютера.

• *Строка 11* — сразу после создания объект **Elevator** еще не перемещен ни вверх, ни вниз, поэтому **totalFloorsTraveled** инициализируется значением **0**. Число пройденных этажей добавляется к **totalFloorsTraveled** (строки 24—25) непосредственно перед окончанием перемещения лифта (строка 26).

• *Строка 12* — объект **passenger**, находящийся внутри лифта, выбирает этаж назначения. Ответ на запрос присваивается переменной **requestedFloor** в строке 21.

// **АБСТРАКЦИЯ И ВЫБОР ПЕРЕМЕННЫХ ЭКЗЕМПЛЯРА**

Обратимся вновь к обсуждению абстракции. **Цель абстракции** — идентифицировать характеристики класса объектов, существенные для программы.

Решая, какие переменные экземпляра включить в определение класса, и объявляя их в исходном коде, программист применяет концепцию абстракции на практике.

Например, в класс **Elevator** можно было бы включить переменную экземпляра **color** типа **string**, объявив ее так:

private string color; <——— вероятно, бесполезно в программе

Вопрос в том, где ее использовать? Например, можно было бы присвоить ей строку **red** и создать метод, который выводил бы на консоль:

My color is: **red**

Однако это не имеет никакого отношения к предназначению модели лифта и только без необходимости усложняет класс **Elevator**.

Другой программист мог бы включить в **Elevator** переменную, которая подсчитывает число перемещений, выполненных лифтом, назвать ее **totalTrips** и объявить как:

private int totalTrips; <——— потенциально полезно

Класс **Elevator** мог бы тогда содержать метод, прибавляющий единицу к **totalTrips** при каждом перемещении лифта. Такая переменная позволила бы получать новую статистику и пригодилась бы в моделирующей программе.

Как видим, существует множество разных способов представить объекты реального мира выбором переменных экземпляра. Естественно, выбор зависит от конкретной задачи и пути ее решения.

2.2.4.5. Даем возможность объекту Elevator загружать нового пассажира

Объект **Elevator** должен уметь "загружать" новый объект **Person**. Это достигается методом **LoadPassenger**, находящимся в объекте **Elevator** (строка 14). К **LoadPassenger**() требуется доступ снаружи объекта, поэтому метод объявлен **public**. Вызов **LoadPassenger** происходит в строке 58 листинга 3.1.

```
14:    public void LoadPassenger()
```

// **ТЕРМИНОЛОГИЯ ПРИ ОБСУЖДЕНИИ КЛАССОВ И ОБЪЕКТОВ**

Рассмотрим следующий оператор объявления:

```
private Person passenger;
```

Существует несколько способов описать эту строку разговорным языком. Следующее описание звучит довольно удачно:
"**passenger** является переменной, объявленной для хранения объекта класса **Person**". Это предложение несколько длинно, поэтому часто используется и более короткая форма: "**passenger** — переменная для хранения объекта **Person**."
Наиболее корректным (и самым длинным) является следующее описание (об этом далее):
"**passenger** — это переменная, объявленная для хранения ссылки на объект класса **Person** ".

2.2.4.6. Создание нового объекта при помощи ключевого слова new

Строка 16 используется непосредственно со строкой 12.

```
16:          passenger = new Person ();
```

new — это ключевое слово C# для порождения нового объекта. Оно создает новый экземпляр (объект) класса **Person**. Затем последний присваивается переменой **passenger**. Теперь к нему можно обращаться и выполнять над ним необходимые действия.

// **ПРИМЕЧАНИЕ**

Когда порождается объект, часто необходима инициализация экземпляра. Конструктор — это специальный метод, который выполняет задачу инициализации.

Чтобы указать, что метод является конструктором, он должен иметь такое же имя, как и класс. Таким образом, конструктор класса **Person** называется **Person**() (строка 39). При создании всякого нового объекта **Person** посредством ключевого слова **new**, конструктор **Person**() вызывается автоматически, чтобы выполнить необходимую инициализацию. Это объясняет наличие круглых скобок в строке 16 — они всегда необходимы в вызове метода или конструктора.

Person() является конструктором класса **Person**, он *вызывается* по ключевому слову **new**

```
16:      passenger = new Person();
```

Person() конструктор класса *Person*

имена класса и конструктора должны совпадать

```
35: class Person
36: {
37:      private System.Random randomNumberGenerator;
38:
39:      public Person()  // конструктор класса Person
40:     {
41:       randomNumberGenerator = new System.Random();
42:     }
```

фрагмент кода, где инициализируется новый объект

2.2.4.7. Даем возможность объекту requestedFloor принимать запросы пассажира

Вызов метода **InitiateNewFloorRequest** заставляет объект **Elevator** сделать следующее

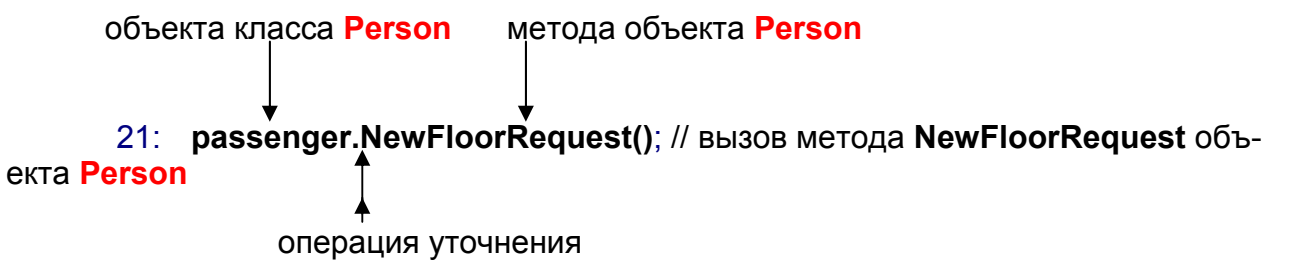
- Получить новый запрос от пассажира **passenger** (строка 21)
- Вывести номера этажей отправления и прибытия (строки 22-23)
- Обновить статистику **totalFloorsTraveled** (строки 24-25)
- Выполнить запрос пассажира **passenger** (строка 26).

```
19:     public void InitiateNewFloorRequest()
20:     {
21:    requestedFloor = passenger.NewFloorRequest();
```

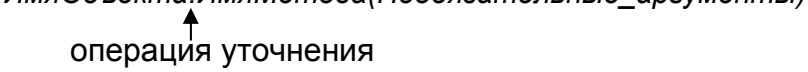
```
22:    Console.WriteLine("Этаж отправления: " + currentFloor
23:    + " Этаж назначения: " + requestedFloor);
24:    totalFloorsTraveled = totalFloorsTraveled +
25:                                    Math.Abs(currentFloor - requestedFloor);
26:    currentFloor = requestedFloor;
27:     }
```

Начнем со строки 21:

В строке 21 метод **NewFloorRequest** объекта **passenger** вызывается с указанием:



Здесь используется синтаксис, введенный ранее:



где операция уточнения (**.**) используется для указания метода, находящегося внутри объекта. Эта операция уже неоднократно применялась, например, при вызове метода **WriteLine** из **System.Console**: **System.Console.WriteLine**("Bye Bye!"). Однако, на этот раз, вместо вызова существующего метода библиотеки классов **.NET Framework**, вызывается пользовательский метод, описанный в классе **Person**.

// **ПРИМЕЧАНИЕ**

Класс — это статическое описание в исходном тексте. С другой стороны, объект — динамическая сущность, которая появляется только во время выполнения программы.

Вместо вызова метода, находящегося внутри текущего объекта, вызывается метод, содержащийся в другом объекте. В этом случае говорят, что объект **Elevator** посылает сообщение объекту **Person**. Верхняя часть рис. 3.5 показывает вызов **NewFloorRequest** — это символизирует шаг 1 (см. также рис.3.1). **NewFloorRequest** не имеет формальных параметров, поэтому ему не передаются никакие аргументы. Этот шаг эквивалентен шагу 1 в нижней части рисунка. По завершении метод **NewFloorRequest** возвращает значение типа **int**, как показывает стрелка 2. Графический эквивалент шага 2 показан в нижней части. После его выполнения можно заменить **passenger.NewFloorRequest()** значением типа **int** и присвоить его переменной **requestedFloor** (строка 21).

При получении запроса от **passenger** в строке 21, объект **Elevator** сохраняет текущую позицию в переменной **currentFloor** и новый этаж в **requestedFloor**. Сразу после создания объекта **Elevator** значение **currentFloor** равно 1. Если же объект использовался ранее, **currentFloor** равно номеру этажа назначения из последней поездки. Строка 22 выводит значение **currentFloor** при помощи метода **WriteLine**. Запрос, полученный от **passenger**, выводится в строке 23.

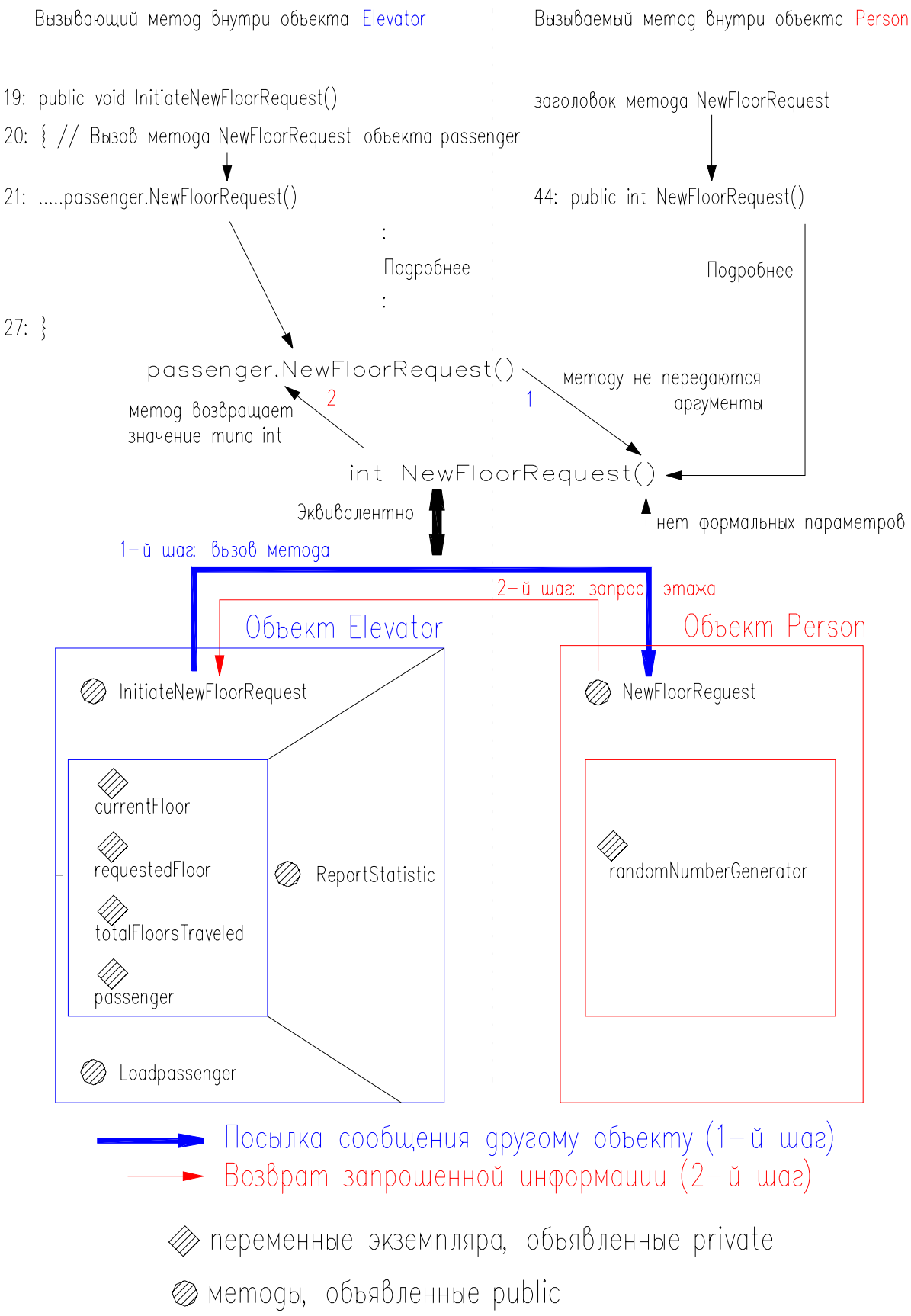


Рис. 3.5. Вызов метода другого объекта

2.2.4.8. Использование Math.Abs()

// **АБСОЛЮТНОЕ ЗНАЧЕНИЕ**
Абсолютное значение положительного числа — само число.
Абсолютное значение отрицательного числа — число без знака минус.
Например,
Абсолютное значение (-12) равно 12.
Абсолютное значение 12 равно 12.
В математике абсолютное значение (модуль) обозначают двумя вертикальными чертами, заключающими в себя число или переменную: |-12| = 12

В библиотеке **.NET** содержится метод **Abs** из класса **Main**. Он возвращает абсолютное значение переданного ему аргумента. Вызов **Main.Abs**(99) возвращает, очевидно, 99, а вызов **Main.Abs**(-34) возвращает 34 и т.д.

При вычислении расстояния, пройденного лифтом, требуется подсчитать число этажей (положительное) вне зависимости от того, двигался ли лифт вниз или вверх. Если применяется формула:

currentFloor - requestedFloor

то отрицательное значение, естественно, соответствует случаю, когда **requestedFloor** больше, чем **currentFloor**. Если его добавить к **totalFloorsTraveled**, число этажей будет ошибочно уменьшено. Поэтому в строке 25 вычисляется модуль выражения (**currentFloor** - **requestedFloor**):

```
Math.Abs(currentFloor - requestedFloor);
```

Все выражение увеличивает общее число этажей на количество, пройденное в текущей поездке (**totalFloorsTraveled** является "счетчиком" лифта, содержащим общее число пройденных этажей).

Оператор присваивания в строке 26 представляет "поездку" лифта. Именно здесь выполняется запрос пассажира. Присваивание **requestedFloor** переменной **currentFloor** говорит о том, что лифт "переместился" с текущего этажа **currentFloor** на новый этаж **requestedFloor.**

2.2.4.9. Класс Person: шаблон для пассажира лифта

В строке 35 начинается определение второго класса, используемого для создания объекта.

```
35: class Person
```

Как было упомянуто выше объект **Person** создается и хранится внутри объекта **Elevator**, из которого вызывается метод **NewFloorRequest**, сообщая объекту **Elevator** этаж назначения.

В строке 37 в объекте класса **Person** вводится объект, содержащий метод, генерирующий случайное число. Последнее сохраняется затем в переменной **randomNumberGenerator**. Соответствующий класс находится в библиотеке классов **.NET** и **System.Random**.

```
37:     private System.Random randomNumberGenerator;
```

Поскольку **randomNumberGenerator** является переменной экземпляра класса **Person**, она объявлена как **private**. Следует отметить, что после этого объявления **randomNumberGenerator** все еще не содержит значения — этой переменной не присвоен ни

один объект. До того как генерировать какое-либо число, ей необходимо присвоить значение, что и было произведено в строках 39—42.

Концепция конструктора была кратко описана ранее. Фактически, конструктор соответствующего класса вызывается в строке 41, когда новый объект **System.Random** присваивается переменной **randomNumberGenerator**.

```
39:     public Person()
40:    {
41:     randomNumberGenerator = new System.Random();
42:    }
```

Объявленный в строках 44-48 метод **NewFloorRequest** при вызове генерирует и возвращает случайное число (требуемый этаж). В строке 47 вычисляется случайное число от 1 до 30 (это указано в скобках — **.Next(1,30)**). По ключевому слову **return** случайное число возвращается в точку вызова (в методе **InitiateNewFloorRequest** объекта **Elevator**). Подробнее ознакомиться с классом **System.Random** можно в документации по **.NET**

```
44:     public int NewFloorRequest()
45:    {
46:   // Возвращает сгенерированное случайное число
47:   return randomNumberGenerator.Next(1,30);
48:    }
```

// **ПРИМЕЧАНИЕ**

Программист, разрабатывающий класс **Elevator**, использует метод **NewFloorRequest** объекта **Person**. Для этого ему нужно знать определение метода. Однако для него не имеет значения, как объект **Person** получает число. Заголовок метода и краткое описание — вот и все, что нужно знать.

Заголовок метода осуществляет взаимосвязь между разработчиком и пользователем метода. Он определяет имя и число аргументов, а также тип возвращаемого значения (если метод не объявлен как **void**).

Поэтому, не изменяя заголовка метода (имя, число и тип формальных параметров, тип возвращаемого значения - стр. 44), можно по-разному реализовывать алгоритм выбора следующего этажа в пределах метода.

2.2.4.10. Метод Main(): запуск процесса моделирования

*В строках 55-64 содержится етод **Main()**.*

```
55:     public static void Main()
56:     {
57:     elevatorA = new Elevator();
58:     elevatorA.LoadPassenger();
59:     elevatorA.InitiateNewFloorRequest();
60:     elevatorA.InitiateNewFloorRequest();
61:     elevatorA.InitiateNewFloorRequest();
62:     elevatorA.ReportStatistic(),
63:     Console.ReadLine();
64:    }
```

Хотя **Main()** размещен в конце программы, он, как всегда; выполняется первый после запуска. Это метод представляет собой "диспетчерский пункт" всей программы: здесь применяется функциональность других классов программы.

Итак, метод **Main(): *1)*** создает объект **Elevator** и присваивает его переменной **elevatorA** (строка 57); ***2)*** загружает пассажира (строка 58); ***3)*** с помощью **elevatorA** запрашивает этаж назначения три раза подряд (строки 59—61); ***4)*** и, наконец, посредством метода объекта **Elevator** (строки 29–32) выводит статистику, собранную за эти три поездки (строка 63).

Важно отметить, что моделирование выглядит очень простым в пределах метода **Main** — все подробности скрыты в классах **Elevator** и **Person**.

2.2.5. Взаимосвязь классов и UML

Три класса, определенных пользователем, и класс **System.Random** из листинга 3.1 работают совместно, моделируя лифт. Класс **Building** содержит объект **Elevator** и вызывает его методы, объект **Elevator** использует объект **Person**, чтобы управлять своим движением, а объект **Person** использует объект **System.Random**, чтобы выбрать следующий этаж. В хорошо спроектированных, объектно-ориентированных программах, классы работают аналогичным образом — каждый вносит уникальную функциональность, обеспечивая работу программы в целом.

Если два класса действуют совместно, возможны различные взаимосвязи, определяемые разработчиком конкретной программы. Эта фаза разработки приходится на определение классов программы. В следующем разделе обсуждаются несколько общеизвестных отношений.

Building-**Elevator** и **Elevator**-**Person** формируют два вида взаимосвязи, которые используются здесь как примеры.

2.2.5.1. Отношение Building-Elevator

Обычное здание состоит из многих частей — полы, стены, потолки, иногда — лифты. В данном случае, моделируемое здание **Building** содержит лифт **Elevator** как свою часть. Такое отношение называют отношением "**целое-часть**". Именно оно реализовано листинге 3.1, когда переменная экземпляра **elevatorA** типа **Elevator** объявлена внутри класса **Building** в строке 53:

```
53:     private static Elevator elevatorA;
```

Переменная **elevatorA** хранит объект **Elevator** и позволяет вызывать **его** методы. Класс может содержать **множество переменных, содержащих другие классы**. К примеру, класс **Building** мог бы содержать переменную, хранящую число этажей. Такая концепция конструирования классов (**Building**) из других классов (**Elevator**, **Floors** и других) называется **агрегацией**, а соответствующее отношение — **отношением агрегации**.

Отношение агрегации (как в случае **Building**-**Elevator**), когда один из классов является частью другого, *называется* **композицией**. (Из изложенного далее станет понятно, почему отношение **Elevator**-**Person** является агрегацией, но не композицией.) Отношение композиции можно проиллюстрировать диаграммой классов **Unified Modeling Language (UML)**, как показано на рис. 3.6. Два прямоугольника символизируют классы, а линия, соединяющая их с закрашенным ромбом (указывающим на "**целый**" класс), показывает отношение композиции между классами. Оба класса помечены числом 1 (один объект **Building** содержит один объект **Elevator**).

UNIFIED MODELING LANGUAGE (UML):
ЯЗЫК ОБЪЕКТНО-ОРИЕНТИРОВАННОГО МОДЕЛИРОВАНИЯ

Псевдокод полезен для выражения алгоритмов, которые реализуются в пределах одного метода, поскольку он читается сверху вниз — так же, как выполняется программа, не имея при этом жесткой

структуры языка программирования (точек с запятыми, скобок и т.д.). Однако классы могут состоять из нескольких методов, а большие программы — из нескольких классов. Псевдокод не подходит для иллюстрации модели, согласно которой классы связаны в программе, **так как сами классы не соответствуют последовательному процедурному способу мышления** (каждый класс потенциально способен взаимодействовать с любым другим, определенным в программе), реализуемому форматом псевдокода.

Чтобы эффективно моделировать отношения классов и общую архитектуру программы, необходим язык, позволяющий абстрагироваться от внутренних деталей методов и обеспечить средства для выражения всех отношений на подходящем уровне детализации. Для этой цели, на сегодняшний день большинство программистов ООП, вне зависимости от их языка программирования, используют язык графических диаграмм — **Unified Modeling Language (UML)**. **UML** — очень богатый язык и для полного изложения требует целой книги, поэтому здесь приведено лишь его небольшое подмножество.

Более подробную информацию о **UML** можно получить на **Web**-странице некоммерческой организации **Unified Modeling Language (UML)** (**www.omg.org**) по адресу **www.omg.org/uml** . Кроме того, о **UML** написано немало хороших книг, включая **The Unified Modeling Language User Guide**, авторами которой стали основатели **UML** — *Грэди Буч* (Grady Booch), *Джеймс Рамбо* (James Rumbaugh) и *Ивар Джакобсон* (Ivar Jacobson)

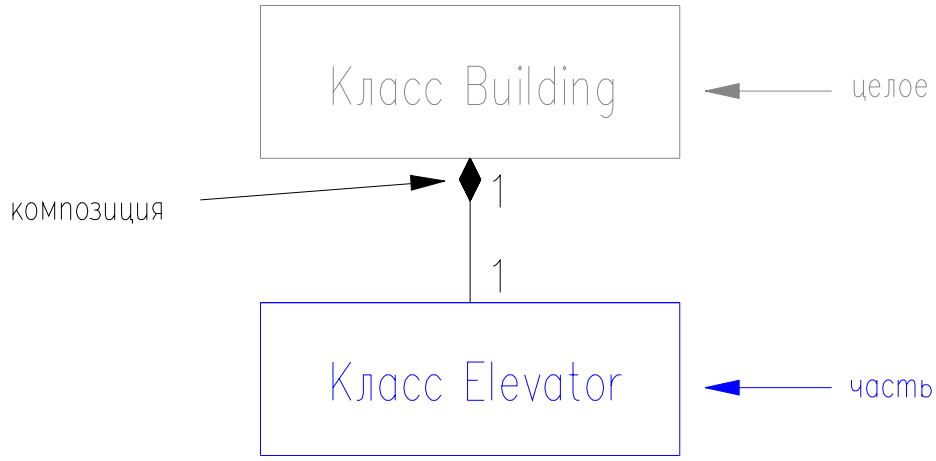


Рис. 3.6. Диаграмма UML: композиция

2.2.5.2. Отношение Elevator-Person

Кнопка является неотъемлемой частью лифта, а пассажир — нет. (**Лифт работоспособен без пассажира, но не без своих кнопок.**) Хотя в данной реализации пассажир и сделан постоянной частью лифта (один и тот же пассажир остается внутри лифта на протяжении всего процесса моделирования), это **отношение** не композиции, а **агрегации**. Оно показано на рис. 3.7. Пустой ромб, в отличие от закрашенного на рис. 3.6, символизирует агрегацию.

2.2.5.3. Полная UML диаграмма классов для программы из листинга 3.1

Полная **UML** диаграмма классов для программы из листинга 3.1 показана на рис. 3.8. **UML** позволяет разделить прямоугольник, представляющий класс, на три отдела: **1)** верхний содержит имя класса, **2)** средний — переменные экземпляра (или атрибуты), **3)** а нижний — методы (поведение), принадлежащие классу.

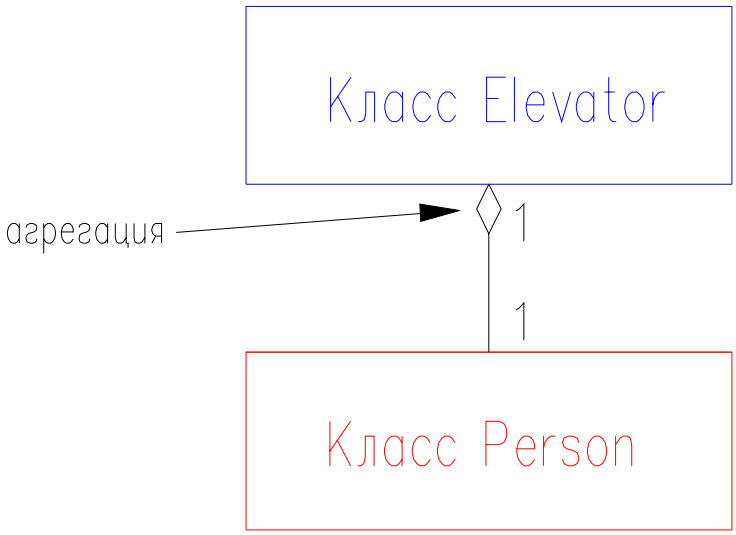


Рис. 3.7. Диаграмма UML: агрегация

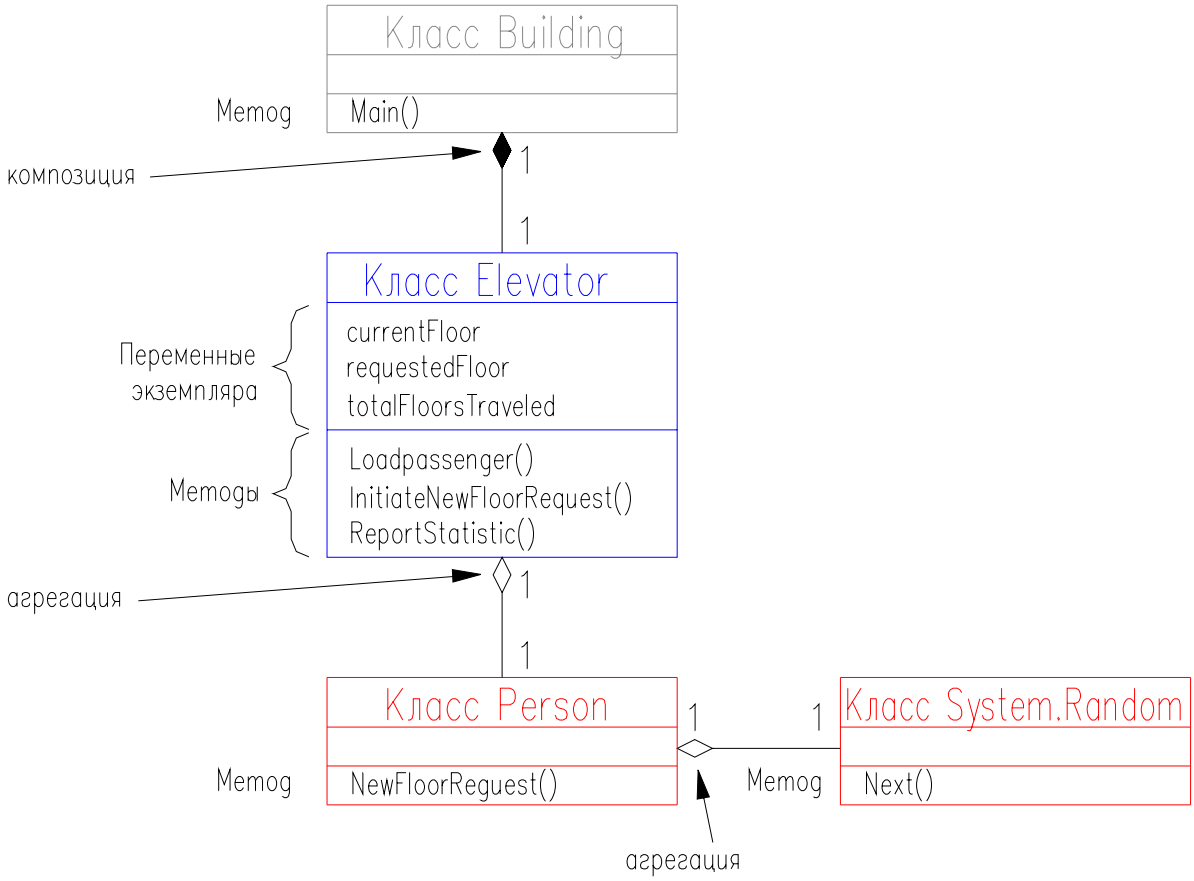


Рис. 3.8. UML диаграмма классов для C#-программы на листинге 3.1

2.2.5.4. Ассоциации

Постоянные отношения между классами (наподобие рассмотренных выше агрегации и композиции) называются *структурными отношениями*, или более формально — *ассоциациями*. Кроме агрегации и композиции существуют и другие виды ассоциаций. В качестве примера рассмотрим такую ситуацию: чтобы сделать модель лифта более реалистичной, в исходный список абстракций внесены изменения, позволяющие нескольким пассажирам входить, перемещаться и выходить из лифта. Ни один объект **Person** теперь не может быть назван частью объекта **Elevator**. О таком взаимоотношении можно сказать лишь, что класс **Elevator** ассоциируется с классом **Person**. Ассоциация показана на рис. 3.8 простой линией.

Другие примеры ассоциаций, не являющихся агрегациями

- **Employee** (Служащий) работает в **Company** (Компании)
- **BankCustomer** (Клиент) обслуживается у **BankTeller** (Клерка)

// **ПРИМЕЧАНИЕ**

Ассоциации, связывающие только два класса, называют бинарными. Это самый распространенный вид ассоциаций.

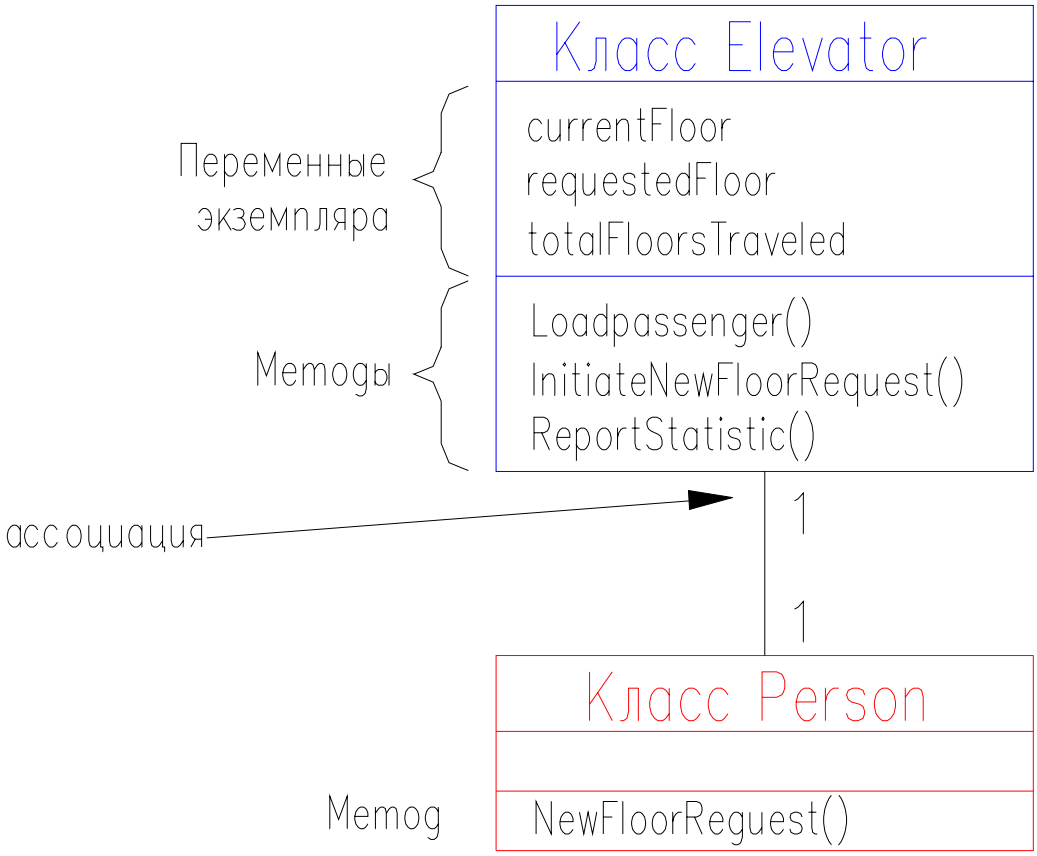


Рис. 3.9. Ассоциация классов Elevator/Person

Контрольные вопросы

1. Как абстракция позволяет программисту справиться со сложностью проблемы?
2. Каковы достоинства инкапсуляции в создании ПО?
3. Какие два важных ключевых слова C# реализуют концепцию инкапсуляции?
4. В чем разница между классом и его объектами?
5. В чем значение интерфейса класса?
6. Кратко охарактеризуйте объектно-ориентированный стиль программирования. В чем его преимущества в сравнении с процедурно-ориентированным стилем?
7. Почему псевдокод не очень хорошо подходит для описания общей структуры объектно-ориентированной программы?
8. Какой вид отношений между объектами **BankCustomer** (Клиент) и **BankTeller** (Клерк)? Как это показать при помощи **UML**?
9. Какой вид отношений между объектами **Heart** (Сердце) и **HumanBody** (ТелоЧело-века)? Как выразить его средствами **UML**?
10. Какой вид отношений между объектами **LightBulb** (Лампочка) и **Lamp** (Настольная Лампа). Как это выражается в **UML**?
11. Каким образом в C# реализовано отношение ассоциации?
12. Как переменная экземпляра может быть инициализирована при создании объекта?
13. Опишите переменную **passenger**, объявленную в строке:

```
private Person passenger;
```

Упражнение по программированию

Внеся изменения в исходный код, добавьте в листинг 3.1 следующую функциональность:

1. Вывод текста "**The simulation has commenced - Моделирование началось**" на консоль сразу после запуска.

2. Вывод текста "**The simulation has ended** – **Моделирование закончилось**" на консоль непосредственно перед завершением программы.

3. Пусть вместо выбора этажа между 1 и 30, класс **Person** выбирает между 0 и 50.

4. Пусть первая поездка **Elevator** начинается с этажа 0, а 1.

5. Объект **Elevator** должен сделать 10 поездок вместо 3.

6. Объект **Elevator** содержит общее число пройденных этажей в переменной **totalFloorsTraweled**. Объявите переменную экземпляра **totalTripsTraweled** внутри класса **Elevator**, в которой будет храниться общее число выполненных поездок. После каждой поездки **Elevator** должен добавлять к **totalTripsTraweled** 1. Доработайте метод **ReportStatistics** класса **Elevator** так, чтобы он выводил не только **totalFloorsTraweled**, но и **totalTripsTraweled** с объяснением этих значений.

7. Добавьте в класс **Elevator** переменную экземпляра **myName**, в которой будет храниться имя лифта. Какой тип нужно использовать для этой переменной? Следует ли применить спецификатор доступности **private** или **public**? Напишите *конструктор*, устанавливающий значение этой переменной при создании объекта **Elevator** при помощи ключевого слова **new**. (Подсказка: *конструктор* является методом и должен иметь то же имя, что и его класс. В заголовке *конструктора* нужно указать формальный параметр типа **string**). Исправьте вызов *конструктора*, вставив имя объекта **Elevator** в качестве аргумента (т.е. вместо **new Elevator()** применяется синтаксис **new Elevator("ElevatorA")**).

Когда **Elevator** выполняет поездку, он должен указывать свое имя рядом с этажом отправления и назначения. Другими словами, вместо вывода

Этаж отправления: 2 Этаж назначения: 24

он должен выводить

ElevatorA: Этаж отправления: 2 Этаж назначения: 24 ,

где **ElevatorA** — имя лифта, хранящееся в **myName**.

Список литературы

1. Микелсен Клаус. Язык программирования C#. Лекции и упражнения. Учебник: пер. с англ./ Клаус Микелсен –СПб.: ООО «ДиаСофтЮП», 2002. – 656 с.

2. Джо Майо. C#Builder. Быстрый старт. Пер. с англ. – М.: ООО «Бином-Пресс», 2005 г. – 384 с.

3. Основы Microsoft Visual Studio .NET 2003 / Пер. с англ. - М.: Издательско-торговый дом «Русская Редакция», 2003. – 464 с. Брайан Джонсон, Крэйт Скибо, Марк Янг.

4. Герберт Шилдт. Полный справочник по C# . / Пер. с англ./ Издательство: Вильямс, 2004 г. 752 с.

5. Чарльз Петцольд. Программирование в тональности C# / Пер. с англ. Издательство: Русская Редакция, 2004 г. - 512 стр.

C# & .NET по шагам (Web-ресурс)

1 | 2 | 3 | 4

Шаг 1 - Разработка приложений в .NET (основы). (24.09.2001 - 2.3 Kb)
Шаг 2 - Как будет распространяться приложение (основы). (24.09.2001 - 3.8 Kb)
Шаг 3 - Нам нужен .Net Framework SDK. (24.09.2001 - 3.8 Kb)
Шаг 4 - Hello Word C#. (25.09.2001 - 2.4 Kb)
Шаг 5 - Hello Word VB. (25.09.2001 - 1.7 Kb)
Шаг 6 - Hello Word VC++. (25.09.2001 - 1.6 Kb)
Шаг 7 - Пространство имен. (26.09.2001 - 2.7 Kb)
Шаг 8 - Net ассемблер и дизассемблер. (26.09.2001 - 3.5 Kb)
Шаг 9 - Просмотр класса в EXE проекте ILDasm.exe. (26.09.2001 - 1.6 Kb)
Шаг 10 - Две основы Net. (27.09.2001 - 2 Kb)
Шаг 11 - Отладка. (27.09.2001 - 33 Kb)
Шаг 12 - ADO.NET (27.09.2001 - 10 Kb)
Шаг 13 - Попробуем OLEDB. (27.09.2001 - 6 Kb)
Шаг 14 - Типы данных - системные и языка программирования. (28.09.2001 - 3 Kb)
Шаг 15 - Windows Form. (28.09.2001 - 7 Kb)
Шаг 16 - Где взять редактор C#. (28.09.2001 - 21 Kb)
Шаг 17 - Избавляемся от консольного окна. (28.09.2001 - 9 Kb)
Шаг 18 - Создаем окно. (28.09.2001 - 6 Kb)
Шаг 19 - Добавляем меню. (28.09.2001 - 6 Kb)
Шаг 20 - Свойства (properties). (28.09.2001 - 3 Kb)
Шаг 21 - Обработка событий на форме. (30.09.2001 - 5 Kb)
Шаг 22 - Изменение размера формы. (30.09.2001 - 2 Kb)
Шаг 23 - Изменение положения формы. (30.09.2001 - 2 Kb)
Шаг 24 - Override. (30.09.2001 - 2 Kb)
Шаг 25 - Встраиваем элемент управления в окно. (30.09.2001 - 5 Kb)
Шаг 26 - Обработка сообщений элемента классом элемента. (30.09.2001 - 6 Kb)
Шаг 27 - Еще один редактор C#. (30.09.2001 - 30 Kb)
Шаг 28 - Создание меню подробнее. (01.10.2001 - 6 Kb)
Шаг 29 - Одномерные Массивы. (01.10.2001 - 3 Kb)
Шаг 30 - foreach. (01.10.2001 - 2 Kb)
Шаг 31 - Интерфейсы. (01.10.2001 - 3 Kb)
Шаг 32 - Коллекции. (01.10.2001 - 6 Kb)
Шаг 33 - Создаем обработчик событий меню. (01.10.2001 - 6 Kb)
Шаг 34 - Сохраняем данные в файл. (01.10.2001 - 7 Kb)
Шаг 35 - Добавляем строку состояния. (02.10.2001 - 5 Kb)
Шаг 36 - Панели на строке состояния. (02.10.2001 - 6 Kb)
Шаг 37 - Икона формы. (02.10.2001 - 9 Kb)
Шаг 38 - Диалог открытия файлов. (02.10.2001 - 14 Kb)
Шаг 39 - Отображаем картинку. (02.10.2001 - 12 Kb)
Шаг 40 - Создаем панель инструментов. (02.10.2001 - 6 Kb)
Шаг 41 - Net Classes первые вывод. (02.10.2001 - 6 Kb)
Шаг 42 - XML документация кода. (02.10.2001 - 6 Kb)
Шаг 43 - XML notepad. (02.10.2001 - 16 Kb)
Шаг 44 - Заголовок формы и пункт меню выход. (03.10.2001 - 4 Kb)
Шаг 45 - Создаем файл с ресурсами строк. (03.10.2001 - 5 Kb)

……..

1 | 2 | 3 | 4