Государственный университет – Высшая школа экономики

Факультет Бизнес-Информатики

Кафедра Основ информатики
и прикладного программного обеспечения

Объектно-ориентированный анализ и программирование на языке C# (C_Sharp)

Материалы ко 2-й лекции. Часть 2

Проф. Забудский Е.И.

Москва 2006

Лекция 2, часть 2

Тема 2. Инкапсуляция – базовое понятие объектно-ориентированного программирования.

Инкапсуляция – объектно-ориентированная характеристика модульности. Внешний интерфейс и внутренняя реализация инкапсулированного программного объекта. Характерные признаки эффективной инкапсуляции: абстракция, общедоступный интерфейс и сокрытие реализации.

(во **2-й части** сделан акцент на *теоретических* положениях связанных с понятием «Инкапсуляция». Рассмотрены **три** C#-программы (листинги **1...3**), в том числе **О**бъектно-**О**риентированный **Б**анк, л-г 3)

К Р А Т К О

ИНКАПСУЛЯЦИЕЙ называется процесс объединения некоторых взаимосвязанных (на концептуальном уровне) элементов. В результате получается КЛАСС.
На основе класса может быть создано множество ОБЪЕКТОВ – ЭКЗЕМПЛЯРОВ КЛАССА. Каждый объект имеет свои собственный ДАННЫЕ. Для их обработки используются МЕТОДЫ, которые объект получил от своего класса. При этом объект остается независимым от других объектов.
Объект сам решает, когда используются методы. (см. Материалы к Практ. зан. 1, рис. 1.3 на с. 9).

Уважаемые студенты!

Основная цель, которую необходимо достигнуть в результате изучения дисциплины Объектно-ориентированный анализ и программирование – научиться разрабатывать компьютерные модели реальных и концептуальных систем соответствующих направлению Бизнес-информатика*.*

Необходимым условием усвоения дисциплины является ВАША самостоятельная работа

Советую Вам все материалы, подготовленные мной к лекциям и практическим занятиям, распечатать и прорабатывать их! Приведенные C#-программы реализовать в среде MS VS .NET 2005 и разобраться в них.

// **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).

Данное пособие распространяется свободно. Изложенный материал, предназначенный для публикации в “твердом” варианте, дополнен и откорректирован.

© Забудский Е.И., 2006

Содержание

1. Инкапсуляция: учимся удерживать детали для себя 4
2. Три базовых понятия объектно-ориентированного программирования 4
3. Инкапсуляция: первое базовое понятие ....Рис. 1 4
3.1. Пример интерфейса и реализации ....Код класса Log 6
3.2. Общедоступный (public), частный (private) и защищенный (protected). 7
4. Зачем нужна инкапсуляция? 7
Три характерных признака эффективной инкапсуляции:
1. Абстракция; 2. Сокрытие реализации; 3. Разделение ответственности
4.1. Абстракция: учимся думать и программировать *абстрактно* 8
4.1.1. Что такое абстракция? 8
4.1.2. Два примера абстракции ....Рис. 2, рис. 3, рис. 4 9
4.1.3. Эффективная абстракция 10
4.2. Сохранение секретов с помощью сокрытия реализации 10
4.2.1. Защита объектов с помощью абстрактного типа данных. 11
4.2.2. Что такое тип? .... Листинг 1. Рис. 5 11
4.2.3. Пример абстрактного типа данных..Рис. 6 15
4.2.4. Защита пользователей от ваших секретов с помощью сокрытия реализации … 17
4.2.5. Пример сокрытия реализации из реальной жизни.. 17
4.3. Распределение ответственности: заниматься своим делом...............Листинг 2 18
5. Объектно-ориентированный Банк – компьютерная модель реального банка….. 23
Листинг 3. Рис. 7
6. Инкапсуляция: советы и типичные ошибки 27
6.1. Абстракция: советы и ошибки. 27
6.2 Советы по сокрытию реализации 27
7. Как инкапсуляция способствует достижению целей объектно-ориентированного программирования 27
8. Может ли инкапсуляция быть вредной? 28
9. Предостережения 29
Резюме 29
Вопросы студента и ответы 29
Контрольные вопросы (задание на дом) 31
Ответы на контрольные вопросы 32
Упражнения (задание на дом). 34
Ответы к упражнениям 35
Литература к курсу 36
Упражнение по программированию (задание на дом) 36

1. Инкапсуляция: учимся удерживать детали для себя

Для компьютерного моделирования реальных и концептуальных систем (в том числе бизнес-систем) необходимо прочное знание основных теоретических понятий Объектно-Ориентированного Подхода. Для применения объектно-ориентированного подхода в работе нужно быть скорее практиком, чем теоретиком. Начиная с сегодняшней лекции будем учиться применять теорию ООП: будем осваивать методы объектно-ориентированного подхода. Навыки в ООП появляются только тогда, когда теория подкрепляется практикой.

На этой лекции вы …

- познакомитесь с тремя базовыми понятиями ООП;
- научитесь эффективно применять инкапсуляцию;
- применять абстракцию при программировании;
- использовать абстрактные типы данных для инкапсуляции;
- ощутите разницу между интерфейсом и реализацией;
- узнаете о важности распределения ответственности.
- научитесь с помощью инкапсуляции достигать целей, поставленных перед объектно-ориентированным подходом

2. Три базовых понятия объектно-ориентированного программирования

Чтобы понять и научиться использовать объектно-ориентированный подход, необходимо приобрести прочные базовые знания, на которые вы будете опираться в дальнейшем обучении. Сначала освойте определения и изучите базовые понятия объектно-ориентированного подхода. Только хорошо разобравшись в базовых понятиях, вы сможете применять объектно-ориентированный подход при создании программ. Таким образом, мы подходим к трем базовым понятиям, которые должны быть представлены в настоящем объектно-ориентированном языке.

Новый термин	Тремя базовыми понятиями объектно-ориентированного программирования являются инкапсуляция, наследование (изучаем на **3**-й лекции) и полиморфизм (– на **4**-й лекции).

Эти три понятия лежат в основе ООП, поэтому его можно сравнить с башней, построенной из блоков: если из основания удалить блок, все здание рухнет. Инкапсуляция, о которой сегодня идет речь, представляет собой очень важную часть конструкции потому, что на ней основаны наследование и полиморфизм.

3. Инкапсуляция: первое базовое понятие

Инкапсуляция позволяет разбить программу на множество мелких независимых элементов, а не рассматривать ее как некую монолитную вещь. Каждый элемент представляет собой модуль, выполняющий свои функции независимо от других элементов. Именно благодаря инкапсуляции повышается степень независимости, поскольку внутренние детали, или реализация, скрываются за интерфейсом.

Новый термин	Инкапсуляция — это объектно-ориентированная характеристика модульности. С помощью инкапсуляции можно разделить программное обеспечение на модули, выполняющие определенные функции, детали реализации которых скрыты от внешнего мира.

По существу термин инкапсуляция означает "герметизированная; защищенная от внешних воздействий часть программы".

Если к некоторому программному объекту применена инкапсуляция, то такой объект можно рассматривать как черный ящик. Вы знаете, что делает черный ящик лишь постольку, поскольку вы видите его внешний интерфейс. Чтобы заставить черный ящик сделать что-либо, нужно послать ему сообщения (рис. 1). Не важно, что именно происходит внутри черного ящика; важно лишь, что он реагирует адекватно на сообщение.

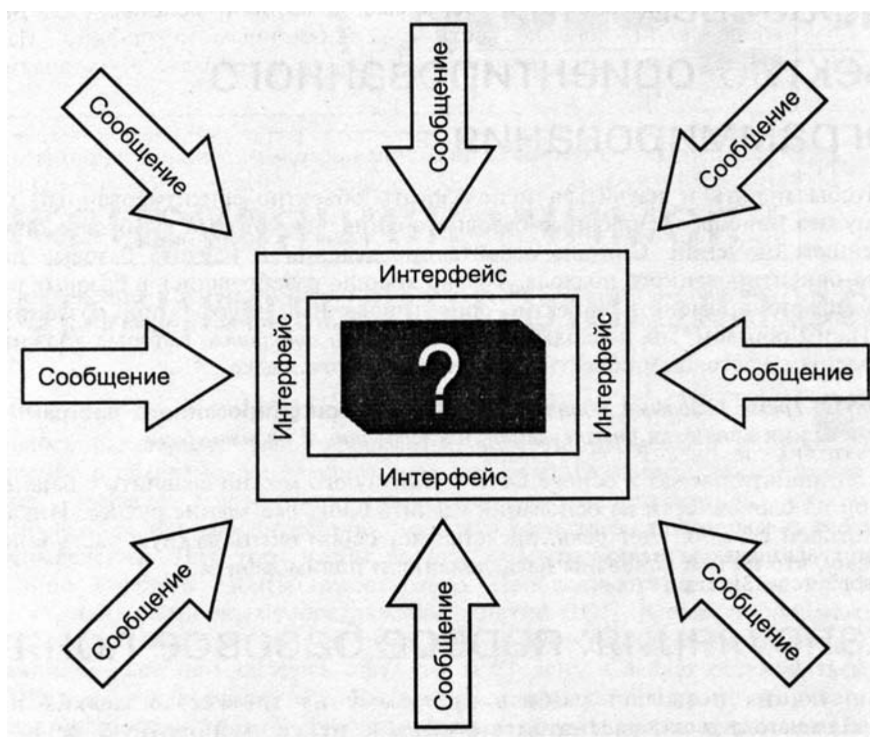


Рис. 1. Черный ящик

Новый термин	Интерфейс представляет собой список служб, предоставляемых компонентом. Интерфейс представляет собой своего рода контракт с внешним миром, в котором указано, какие запросы внешние объекты могут посылать данному объекту. Интерфейс — это пульт управления объектом.

Примечание	Интерфейс важен именно потому, что в нем сообщается, как обратиться к компоненту. Но интерфейс не сообщает, как компонент выполняет свою работу, а, наоборот, назначение интерфейса как раз и состоит в том, чтобы скрыть реализацию от внешнего мира. Именно это дает возможность изменять реализацию компонента в любое время. Изменяя реализацию, нет необходимости производить какие-либо изменения в программе, в которой используется данный класс, если, конечно, остается неизменным интерфейс. Изменения интерфейса приведут к изменениям в программе, использующей этот интерфейс.

Примечание	Известен термин программирования – интерфейс прикладной программы (**A**pplication **P**rogramming **I**nterface, **API**). Интерфейс подобен интерфейсу прикладной программы для объекта. **Интерфейс класса перечисляет все** методы **и** аргументы**, которые он принимает**

Реализация — это алгоритм исполнения компонентом определенного задания. Реализация определяет внутренние детали компонента.

3.1. Пример интерфейса и реализации

Рассмотрим следующий класс Log:

00	using System;
01	/*Класс Log демонстрирует разницу между реализацией и наследованием*/
02	public class Log
03	{
04	public void Debug(String message) // это интерфейс класса
05	{
06	Console.WriteLine ("DEBUG", message);
07	}
08	public void Info(String message) // это интерфейс класса
09	{
10	Console.WriteLine ("INFO", message);
11	}
12	public void Warning(String message) // это интерфейс класса
13	{
14	Console.WriteLine ("WARNING", message);
15	}
16	public void Error(String message) // это интерфейс класса
17	{
18	Console.WriteLine ("ERROR", message);
19	}
20	public void Fatal(String message) // это интерфейс класса
21	{
22	Console.WriteLine ("FATAL", message);
23	System.exit(0);
24	}
	// а это реализация класса
25	private void Print(String message, String severity)
26	{
27	Console.WriteLine(severity + ": " + message);
28	}
29	}

С помощью класса Log объект выводит диагностические, информационные, предупреждающие и другие сообщения во время выполнения задания. Интерфейс класса Log состоит из доступных внешней среде линий поведения (то есть **public** методов). Доступные окружающей среде линии поведения называются общедоступным интерфейсом. Общедоступный интерфейс класса Log содержит следующие методы:

04	public void Debug (String message)
08	public void Info (String message)
12	public void Warning (String message)
16	public void Error (String message)
20	public void Fatal (String message)

Все остальное в определении класса, кроме этих пяти методов, является реализацией. Реализация — это правила исполнения чего-либо. В данном случае реализацией является способ, которым **Log** выводит информацию на экран. Однако интерфейс полностью **скрывает** метод. Вместо этого интерфейс определяет **взаимодействие** объектов с окружающим миром. Например, с помощью

04	public void Debug (String message)

можно узнать, что если методу **Debug** будет передана строка (String), то будет выведено диагностическое сообщение.

Важно знать, о чем же интерфейс не сообщает. В **Debug()** не сообщается, что что-либо выводится на экран. Вместо этого сообщение обрабатывается так, как указано в реализации программы. В реализации можно предусмотреть: вывод сообщения на экран, запись его в файл или в базу данных, или же отправку его пользователю по сети.

3.2. Общедоступный (public), частный (private) и защищенный (protected)

Должно быть, вы заметили, что общедоступный интерфейс не содержит (см. выше строки **25…28**)

25	private void Print(String message, String severity)

Вместо этого объект **Log** ограничивает доступ к методу **Print**; он разрешает доступ к этому методу только себе самому.

Для того чтобы включить некий элемент в общедоступный интерфейс, или, наоборот, исключить из него, необходимо воспользоваться ключевым словом. В языке объектно-ориентированного программирования C# определен набор ключевых слов, которые определяют три уровня доступа

- Общедоступный (**public**)— разрешение доступа для всех объектов.
- Защищенный (**protected**)— разрешен доступ только для данного экземпляра и для любых производных классов (производные классы – см. Материалы к Практ. зан. № 6, с. 12,сл)
- Частный (**private**) — разрешен доступ только для данного экземпляра.

В C# предусмотрено еще два уровня доступа:

- ✓ internal (внутренний) - (см. Материалы к Прак. Зан. №4, **стр. 33, 34**)
- ✓ protected internal (внутренний защищенный) - (см. Материалы к Прак. Зан. №6, **стр. 15**).

Очень важно в проекте правильно выбрать уровень доступа. Все, что нужно сделать видимым, должно быть общедоступным (**public**). Все, что нужно скрыть, должно иметь защищенный (**protected**) или частный доступ (**private**).

4. Зачем нужна инкапсуляция?

Благодаря правильному использованию инкапсуляции с объектами можно обращаться как со сменными компонентами. Чтобы другой объект мог использовать ваш объект ему только нужно знать, как использовать общедоступный интерфейс вашего объекта. Такая независимость дает три значительных преимущества:

A. Благодаря независимости объект можно использовать повторно. При тщательной инкапсуляции объекты не будут привязаны к определенной программе. Их можно использовать везде, где это имеет смысл. Чтобы использовать объект в каком либо ином месте, нужно просто воспользоваться его интерфейсом.

B. Благодаря инкапсуляции в объекте можно осуществлять изменения, невидимые для других

объектов. Если интерфейс не менять, то все перемены останутся невидимыми для тех, кто использует объект. Инкапсуляция позволяет улучшить компонент, обеспечить эффективную реализацию, устранить ошибки, причем все это не затрагивает другие объекты программы. Пользователи объекта автоматически выигрывают от любых производимых вами усовершенствований.

C. При использовании защищенного объекта исключается возможность каких-либо непредсказуемых взаимодействий между объектом и остальной частью программы. Если объект изолирован, он может взаимодействовать с остальной частью программы только через свой интерфейс.

На этом этапе можно **обобщить** все выше сказанное об инкапсуляции. Итак, с помощью инкапсуляции можно создавать модульные программы.

Три характерных признака эффективной инкапсуляции таковы:

1. Абстракция; - раздел 4.1, стр. 8,сл.

2. Сокрытие реализации; - раздел 4.2, стр. 10,сл.

3. Разделение ответственности. - раздел 4.3, стр. 18,сл.

Рассмотрим более детально каждую из этих характеристик, чтобы знать, как выполнять инкапсуляцию наилучшим способом.

4.1. Абстракция: учимся думать и программировать *абстрактно*

Хотя объектно-ориентированные языки способствуют использованию инкапсуляции, они ее не гарантируют. Легко создать зависимый, хрупкий код. Эффективная инкапсуляция — это результат тщательной разработки, применения абстракции и опыта. Чтобы применять инкапсуляцию эффективно, сначала необходимо научиться при разработке программ применять абстракцию и связанные с ней концепции.

4.1.1. Что такое абстракция?

Абстракция — это процесс упрощения сложной задачи. Намереваясь решить определенную задачу, вы **не стремитесь** учесть все детали, а выбираете только те, которые облегчают решение.

Допустим, вам нужно составить модель дорожного движения. Очевидно, что вы создадите классы светофоров, машин, шоссе, двухсторонних и односторонних улиц, погодных условий и так далее. Каждый из этих элементов влияет на движение транспорта. Однако вы не будете создавать модели насекомых или птиц, хотя они тоже могут появиться на дороге. Более того, вы не будете выделять марки машин. Вы упрощаете реальный мир и используете только основные элементы. Машина — это важная деталь модели, однако, будет ли это «Волга» или какая либо другая марка, такие детали излишни для модели дорожного движения.

У абстракции есть два преимущества. Во-первых, она упрощает решение задачи. И, что еще более важно, во-вторых, благодаря абстракции компоненты программного обеспечения можно использовать повторно. Компоненты программы часто чрезмерно специализированы. То, что компоненты рассчитаны на решение какой-либо определенной задачи, в сочетании с их ненужной взаимозависимостью, усложняет повторное использование фрагмента программы в каком-либо другом месте. По мере возможности старайтесь создавать объекты, которые будут решать целый ряд задач. Абстракция позволяет использовать решение **одной задачи** для решения **других задач** из данной предметной области.

Хотя желательно создавать абстрактные программы, избегая чрезмерной детализации, это довольно трудно сделать, особенно если вы только начинаете практиковаться в ООП. Грань между *чрезмерной* специализацией и *недостаточной* детализацией очень тонка. Только опыт помогает распознавать ее. Однако необходимо научиться применять эту важную концепцию.

4.1.2. Два примера абстракции

Первый пример: представьте очередь людей к кассиру в банке. Когда кассир освобождается, первый в очереди клиент подходит к его окошку. Клиенты один за другим продвигаются к окошку кассира (рис. 2). Очередь продвигается согласно алгоритму "первым пришел — первым обслужен".

Второй пример: рассмотрим конвейер с гамбургерами в закусочной. Когда новый гамбургер попадает на конвейер, он занимает место рядом с последним в ряду гамбургером (рис. 3). Поэтому гамбургер, который снимают с конвейера, пролежал там дольше остальных. Можно сказать, что рестораны работают по алгоритму "первым пришел — первым обслужен".

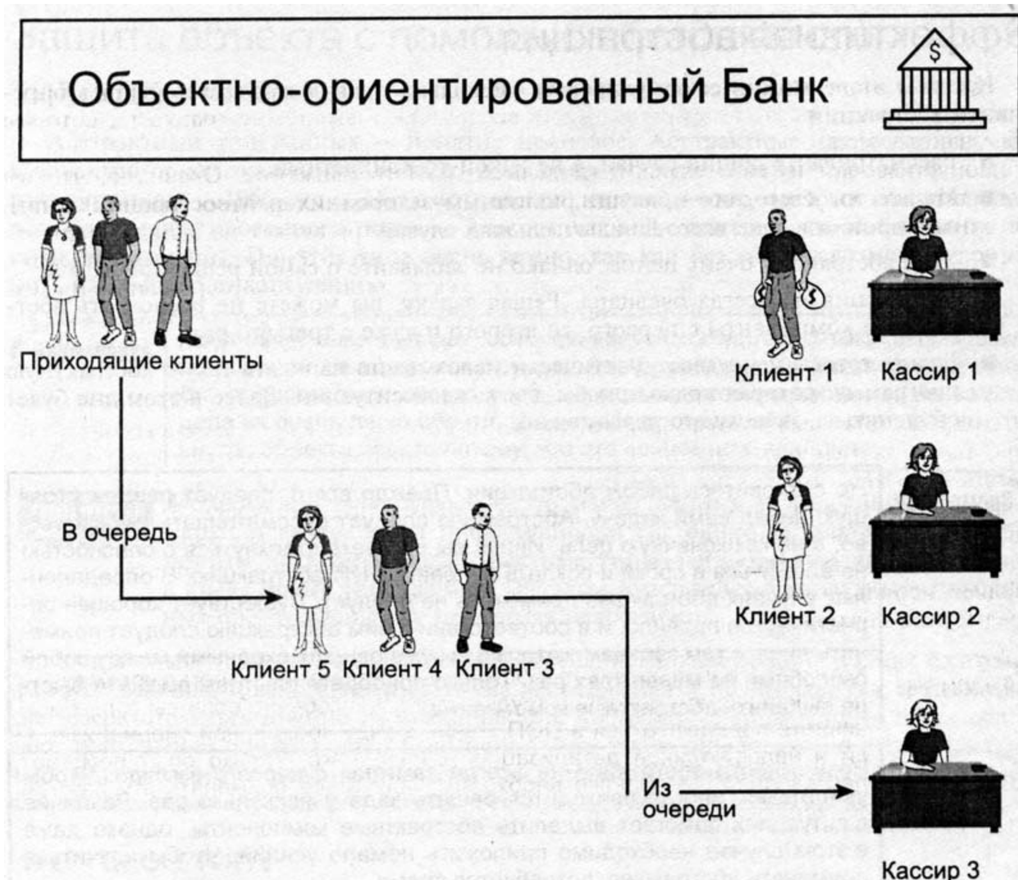


Рис. 2. Очередь в банке

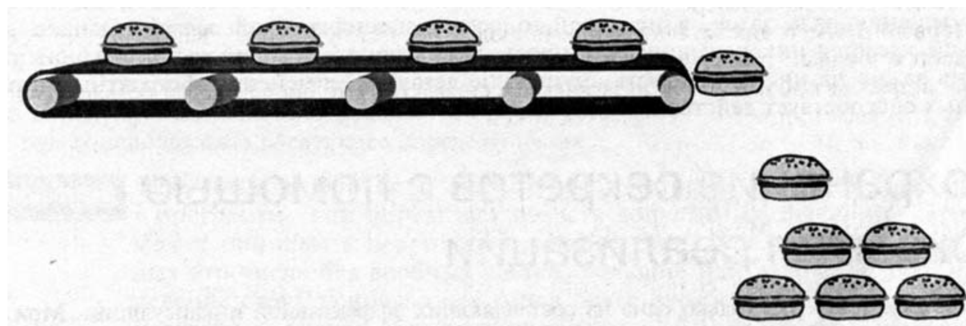


Рис. 3. Конвейер с гамбургерами

Хотя эти примеры абсолютно разные, в них используется некий общий принцип, который можно применить и в других ситуациях. Иными словами, вы приходите к абстракции.

В каждом из этих примеров используется алгоритм "первым пришел — первым обслужен". Не важно, что именно представляет собой элемент очереди. В действительности **важно лишь то**, что этот элемент присоединяется к концу очереди и покидает очередь, когда достигает ее начала, как показано на рис. 4.

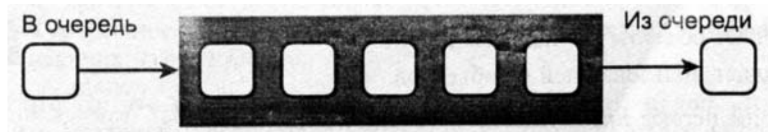


Рис. 4. Абстрактное изображение двух примеров (рис. 2. и рис. 3)

С помощью абстракции можно один раз создать очередь и использовать ее потом для написания других программ, в которых элементы обрабатываются по алгоритму "первым пришел — первым обслужен".

4.1.3. Эффективная абстракция

На этом этапе можно сформулировать несколько правил для выполнения эффективной абстракции:

1. Рассматривайте общий случай, а не какой-то конкретный.
2. Ищите то общее, что присуще различным задачам. Старайтесь увидеть основной принцип, а не всего лишь отдельный случай.
3. Хотя абстракция очень ценна, однако не забывайте о самой решаемой задаче.
4. Абстракция не всегда очевидна. Решая задачу, вы можете не распознать абстрактные компоненты с первого, со второго и даже с третьего раза.
5. Будьте готовы к неудаче. Фактически невозможно написать такую абстрактную программу, которая подходила бы для каждой ситуации.

Не становитесь *рабом* абстракции. Прежде всего, следует решать стоящую перед вами задачу. Абстракцию следует рассматривать как средство, а не как конечную цель. Иначе вы рискуете столкнуться с опасностью не вложиться в сроки и создать неправильную абстракцию. В определенных случаях абстракцию применять не следует. Существует хорошее эвристическое правило, и в соответствии с ним абстракцию следует применять лишь к тем задачам, которые вы уже решали сходными между собой способами *не менее трех раз*. Только приобретя опыт, вы сможете быстро выделять абстрактные компоненты.

Возможности абстракции не всегда заметны с первого взгляда. Чтобы увидеть их, иногда приходится решать задачу несколько раз. Различие в ситуациях помогает выделить абстрактные компоненты, однако даже в этом случае необходимо приложить немало усилий. Чтобы научиться применять абстракцию, потребуется время.

Абстрактный компонент легче использовать повторно, так как он предназначен для решения ряда задач, а не какой-то одной специфической задачи. Однако это больше касается инкапсуляции, чем простого повторного использования компонента. Очень важно научиться скрывать внутренние детали. Применение абстрактных типов данных способствует действенному применению инкапсуляции.

4.2. Сохранение секретов с помощью сокрытия реализации

Абстракция — это только одна из составляющих эффективной инкапсуляции. Можно написать абстрактную программу, которая совсем не будет защищена от внешних воздействий. Именно поэтому необходимо скрывать внутреннюю реализацию объекта.

Сокрытие реализации дает два преимущества:

1) защищает объекты от пользователей; - стр. 11,сл.: разделы 4.2.1, 4.2.2, 4.2.3

2) защищает пользователей от объектов. - стр. 17,сл.: разделы 4.2.4, 4.2.5

Рассмотрим первое преимущество — защиту объектов.

4.2.1. Защита объектов с помощью абстрактного типа данных

Абстрактный тип данных — это одна из составляющих объектно-ориентированного подхода. Его отличают две интересных особенности: абстрактность и тип. Понятие типа очень важно, так как без него невозможно применить настоящую инкапсуляцию.

Примечание	*Подлинная инкапсуляция* обеспечивается на уровне языка с помощью встроенных языковых конструкций. Все другие формы инкапсуляции просто представляют своего рода джентльменское соглашение, на самом деле их очень легко обойти. И опытные программисты смогут проникнуть внутрь объекта просто потому, что это возможно в принципе.

Новый термин	Абстрактный тип данных — это набор данных и операций над ними. Скрывая внутреннюю информацию и состояние за тщательно разработанным интерфейсом, абстрактные типы данных позволяют определить в языке новые типы данных. В таком интерфейсе абстрактные типы данных представлены как неделимая целостность.

Абстрактные типы данных облегчают применение инкапсуляции, так как благодаря им инкапсуляцию можно использовать без наследования и полиморфизма, а это позволяет сосредоточиться именно на инкапсуляции. Абстрактные типы данных также облегчают применение понятия типа. Если понять, что такое тип, то можно легко заметить, что объектно-ориентированный подход предлагает естественный способ расширения языка с помощью определения специализированных пользовательских типов.

4.2.2. Что такое тип?

При программировании создается ряд переменных и им присваиваются значения. С помощью типов определяются различные виды значений, доступные программе. Таким образом, можно сказать, что тип — один из компонентов программы. В качестве примеров обычных типов можно назвать целые (System.Int32 …), длинные (System.Int64) и плавающие (System.Double …).

Итак, типы определяют:
- виды переменных, которые можно использовать в программе;
- область допустимых значений, которые может принимать переменная данного типа
- какие операции можно выполнять над переменной
- какого типа будут получаемые результаты.

Типы — это то, что в вычислении выступает как единое целое. Это означает, что тип — это независимый элемент. Возьмем, к примеру, целое число. Складывая два целых числа, вы не думаете об операциях над битами; вы всего лишь складываете два числа. Несмотря на то, что целое число в памяти компьютера представляется в виде битов, язык программирования дает программисту возможность работать с целыми числами.

Возьмем в качестве примера класс **Item** из 1-й лекции (см. Часть 1, Листинг 5). При создании класса **Item** в словарь программы добавляется новый тип. Вместо того чтобы воспринимать код товара (**id**), описание товара (**description**) и цену (**price**) как отдельные объекты, несвязанные области памяти или переменные, при решении задачи можно использовать понятие **Item**. Другими словами, класс – тоже тип. Таким образом, понятие типа позволяет упростить сложную структуру, подняв ее на более понятный, концептуальный уровень, благодаря чему удается не вдаваться в излишние детали и при решении задачи можно всегда оставаться на уровне предметной области, а

не опускаться на уровень программной реализации.

Благодаря типам программисты могут не только не вникать в излишние детали, но и получить даже более важные преимущества. Тип гарантирует правильность, целостность и безопасность взаимодействия с объектом. Ограничения, налагаемые типом, предохраняют объекты от потери целостности и возможных деструктивных взаимодействий. Объявление типа предотвращает непреднамеренное или случайное использование типа. Именно объявление типа гарантирует его правильное применение.

Без четкого определения допускаемых операций типы могут взаимодействовать между собой любым способом. Непредвиденные взаимодействия могут привести к неправильному результату.

Вспомните класс Item, о котором шла речь в 1-й лекции (см. Часть 1, Листинг 5, строки *12...16*). Представьте, что мы немного изменили определение класса **Item (сравните** строки ***12...16*** **и строки** 06…10):

04	public class UnencapsulatedItem
05	{
06	public double price;
07	public double discount; // процентная скидка с цены
08	public int quantity;
09	public String description;
10	public String id;
	// ….
87	}

Можно заметить, что теперь все внутренние переменные общедоступны (public). А что получится, если написать следующую программу, используя новый класс UnencapsulatedItem?

01	using System; /*Эта программа является плохой альтернативой коду ConsAppl_OOP21_c32_35 – см. 1-я лекция (см. Часть 1, Листинг 5). **Листинг 1**
02	namespace ConsAppl_OOP21_c54_Unenc_Item /*Здесь PUBLIC-доступ к типу UnencapsulatedItem - это плохо, так другие программы могут получить доступ к этому типу и исказить данные в объекте этого класса. Далее иллюстрируется проявление ошибки */
03	{
04	public class UnencapsulatedItem
05	{
06	public double price; // цена единицы товара
07	public double discount; // скидка с цены в процентах; скидки может не быть
08	public int quantity; // количество товара
09	public String description; // описание товара
10	public String id; // код товара
11	
12	public UnencapsulatedItem(String id, String description, int quantity, double price)
13	{ ///////////// начало КОНСТРУКТОРА класса UnencapsulatedItem /////////////
14	this.id = id;

```
15         this.description = description;
16         if (quantity >= 0)
17         {
18             this.quantity = quantity;
19         }
20         else
21         {
22             this.quantity = 0;
23         }
24         this.price = price;
25     }                    ////////////// окончание КОНСТРУКТОРА класса UnencapsulatedItem //////////////
26
27     public double Adjusted_Total()        // расчет итоговой цены приобретаемого товара
28     {
29         double total = price * quantity;
30         double total_discount = total * discount;
31         double adjusted_total = total - total_discount;
32         return adjusted_total;
33     }
34
35     public double Discount                // СВОЙСТВО : Discount - set/get процентная скидка
36     {
37         set
38         {
39             if (discount <= 1.00)
40             {
41                 this.discount = value;
42             }
43             else
44             {
45                 this.discount = 0.0;                  // discount = 1.25, ЭТО ОШИБКА
46             }
46         }
47         get
48         {
49             return discount;
50         }
61     }
62
63     public int Quantity                   // СВОЙСТВО Quantity - set/get количество товара
```

64	{
65	set
66	{
67	if (quantity >= 0)
68	{
69	this.quantity = value;
70	}
71	}
72	get
73	{
74	return quantity;
75	}
76	}
77	
78	public String Id // СВОЙСТВО Id - код товара
79	{
80	get { return id; }
81	}
82	
83	public String Description // СВОЙСТВО Description - описание товара
84	{
85	get { return description; }
86	}
87	} ///////////////конец КЛАССА UnencapsulatedItem //////////////////
88	
89	//////////Начальный класс UnencapsulatedItemExample ///////////////
90	public class UnencapsulatedItemExample
91	{
92	public static void Main(String[ ] args)
93	{
94	/*конструктор UnencapsulatedItem() создает ОБЪЕКТ (товар) класса. Товар имеет следущие атрибуты: код, описание, количество, цена за единицу */
95	UnencapsulatedItem monitor = new UnencapsulatedItem("electronics-012", "17\" SVGA Monitor", 1, 299.00);
96	
97	/* применение дисконтной карты - предоставлена неправильная скидка на стоимость монитора, эта скидка присваивается непосредственно PUBLIC-переменной discount */
98	monitor.discount = 1.25; // ошибка, так как скидка должна быть меньше 100%!
99	// получаем цены на товары, рассчитанные с учетом ошибочной скидки
100	double price = monitor.Adjusted_Total();
101	Console.WriteLine("Неправильный итог: $" + price);
102	// ошибка сохраняется: тем не менее свойство Discount найдет ошибку и заблокирует ее
103	monitor.Discount = 1.25;

104	price = monitor.Adjusted_Total();
105	Console.WriteLine("Правильный итог: $" + price);
106	Console.ReadLine();
107	}
108	} //////////////// конец класса UnencapsulatedItemExample ////////////////
109	}

На **рис. 5** показано, что происходит при выполнении метода **Main()**.

Если предоставить свободный доступ к типу **UnencapsulatedItem**, другие программы могут получить доступ к этому типу и исказить данные в экземпляре **UnencapsulatedItem**. В нашем случае **Main()** создает **UnencapsulatedItem** и затем применяет неверную скидку. В результате цена будет подсчитана неверно, скорректированная цена будет отрицательной!

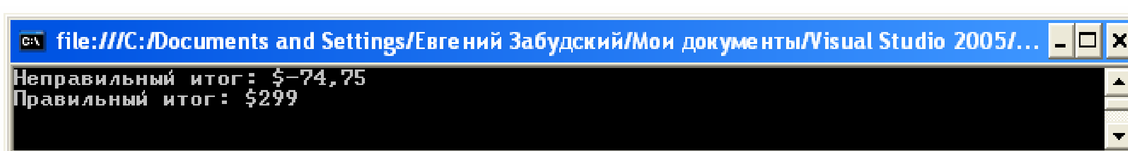


Рис. 5. Ошибочный итог

Абстрактные типы данных необходимы при инкапсуляции, так как они позволяют определить новые безопасные в употреблении языковые типы. Абстрактный тип данных позволяет создавать новые слова программирования, когда возникает необходимость ввести новое понятие.

Определив новый тип, его можно использовать, как и остальные типы. Подобно тому, как методу можно передавать целое, можно передавать и абстрактный тип данных. Такие объекты называются объектами первого класса. Объекты первого класса можно передавать в качестве параметров.

Объект **первого** класса — это объект, который можно использовать таким же образом, как и встроенный тип.

Объект **второго** класса — это тип объекта, который можно определить, однако нет необходимости использовать его как встроенный тип.

4.2.3. Пример абстрактного типа данных

Рассмотрим пример с абстрактной очередью, о котором говорилось ранее (см. стр. **9**,сл. и **рис. 4**). Существует ряд вариантов реализации очереди. Независимо от способа реализации, элементы продолжают двигаться в очереди согласно алгоритму "первым пришел — первым обслужен".

Очередь можно использовать в виде абстрактного типа данных. Для использования очереди нет необходимости знать ее *базовую* реализацию. Если очередь не будет представлена в виде абстрактного типа данных, то в каждом объекте, в котором потребуется очередь, придется повторно реализовать эту структуру данных. Каждый объект, работающий с данными в очереди, должен будет знать, как она реализована и как правильно взаимодействовать с ней. Как уже говорилось ранее, *не*предусмотренное взаимодействие очень опасно!

Поэтому очередь нужно создать в виде абстрактного типа данных. Хорошо защищенная очередь в виде абстрактного типа данных будет гарантией сохранения целостности и безопасности доступа к данным.

Приступая к созданию абстрактного типа данных, нужно хорошо разобраться в том, как функционирует этот тип данных. В нашем случае над очередью можно выполнять следующие операции:

- поместить элементы в очередь: **enqueue**;
- удалить элементы из очереди: **dequeue**;
- сделать запрос о состоянии очереди: **isEmpty**;
- взглянуть на первый элемент, не удаляя его: **peek**.

Каждая из этих операций как элемент будет представлена в виде входа в общедоступном (**public**) интерфейсе очереди **Queue**.

Необходимо также дать имя абстрактному типу данных. В этом случае именем абстрактного типа данных служит **Queue**. Абстрактный тип данных определен таким образом:

01	**public interface Queue**
02	**{**
03	**public void enqueue(Object obj);**
04	**public Object dequeue();**
05	**public bool isEmpty();**
06	**public Object peek();**
07	**}**

Рис. 6. Абстрактный тип данных **Queue**

Заметьте, что интерфейс очереди ничего не сообщает о том, как данные хранятся в очереди, и не дает свободного доступа к внутренним (**private**) данным. Все подобные детали спрятаны.

Вместо этого у вас **есть новый тип — очередь Queue**. Теперь можно использовать этот тип в любой программе (**“interface” рассмотрен в Материалах к Прак. Зан. №7, стр. 32,сл.**).

Поскольку это объект первого класса, вы можете использовать очередь как параметр. Абстрактный тип данных можно рассматривать как единое целое, потому что все его части независимы. В этом-то и состоит мощь абстракции: она дает программисту дополнительные возможности. Вместо того чтобы думать о массивах, циклах и пр., программист может обдумывать решение задачи на более высоком уровне, используя терминологию той предметной области, к которой относится решаемая задача. Он может использовать структуру высокого уровня с алгоритмом обслуживания "первым пришел — первым обслужен".

Один тип может содержать другие типы. Благодаря этому скрываются детали и увеличиваются выразительные возможности. Типы, содержащие другие типы, могут концентрировать в себе много понятий. Например, **int** в программе является простым элементом, это всего лишь одно целое число. Но использование типа **Queue** значительно увеличивает выразительные средства оператора. Ведь **Queue** содержит в себе намного больше компонентов, чем **int**.

Рассмотрим интерфейс (рис. 6) более подробно. Этот интерфейс является очень обобщенным. Вместо того, чтобы сообщать о том, что это очередь целых чисел или гамбургеров (или другое), интерфейс просто помещает объекты (**Object**) в очередь и удаляет их из нее. В C# с любыми объектами можно обращаться так, как с **Object**. Задавая параметры таким способом, в очередь можно поместить любой объект. Поэтому тип **Queue** можно использовать в различных ситуациях.

У обобщенных интерфейсов есть недостатки. Очередь (**Queue**) целых является очень конкретным объектом. Известно, что каждый элемент очереди (**Queue**) представляет собой значение типа **integer**. Но очередь (**Queue**) из произвольных объектов (**Object**) типизирована слабо. А ведь когда вы выбираете элемент, нужно знать его тип.

Для того чтобы выполнить действительно эффективную инкапсуляцию, нужно знать еще о некоторых ее особенностях. Ведь пока что мы рассмотрели только сокрытие реализации. Однако

важна и другая сторона медали — защита пользователей от объектов.

4.2.4. Защита пользователей от ваших секретов с помощью сокрытия реализации

Вы уже увидели, что интерфейс может скрывать реализацию объекта. Благодаря тому, что реализация скрыта, объект защищен от непредвиденного и деструктивного использования. Это одно из преимуществ сокрытия реализации. Однако сокрытие реализации также важно и для пользователей объектов.

Сокрытие реализации делает программу более гибкой, *так как пользователи не обязаны учитывать реализацию объекта*. Таким образом, сокрытие реализации объекта: ***1)*** не только защищает объект, ***2)*** но также помогает избежать определенных неудобств тем, кто этот объект использует, способствуя созданию слабосвязанного кода.

Слабосвязанный код — это код, не зависящий от реализации других компонентов.

Сильносвязанный код, или код с непосредственными связями — это код, тесно связанный с реализацией других компонентов.

Рассмотрим преимущества *слабосвязанного* кода.

Если в общедоступном интерфейсе объекта появляется какое-либо **свойство**, каждый, кто его использует, начинает определенным образом от него зависеть. **Если свойство неожиданно пропадает**, **нужно будет вносить изменения в программу, которая стала зависимой от него**.

Зависимый код зависит от существования определенного типа. Зависимости избежать нельзя. Однако существуют степени **допустимой** и **чрезмерной** зависимости.

Хотя полностью зависимости между объектами избежать нельзя, нужно стараться ее минимизировать. Обычно зависимость уменьшается, **если правильно определить интерфейс**. Пользователи **могут зависеть только** от интерфейса. *Но если реализация одного из объектов является частью интерфейса, пользователи объекта могут зависеть от этой реализации*. Такие сильносвязанные коды мешают изменять реализацию объекта. Небольшое изменение реализации объекта приведет к ряду изменений всех пользователей объекта.

Инкапсуляция и сокрытие реализации — это не волшебство. При изменении интерфейса нужно обновить код, который зависит от старого интерфейса. Если при написании программы детали скрываются в интерфейсе, то в результате вы получите слабосвязанную программу

В сильносвязанной программе теряются преимущества инкапсуляции: создание независимых, повторно используемых объектов невозможно.

4.2.5. Пример сокрытия реализации из реальной жизни

Это пример, описывающий ситуацию из повседневной жизни. Рассмотрим следующее определение класса:

```
01  public class Customer
02    {
03   // ... различные методы покупателя ...
04        public Item[ ] items;
05   // в массиве items хранятся все выбранные товары
06    }
```

В классе **Customer** хранятся все выбранные товары. Здесь **Customer** создает массив **Item** как часть его внешнего (**public**) интерфейса:

01	public static void Main(String [ ] args)
02	{
03	Customer customer = new Customer();
04	// ... выберите некоторые товары ...
05	// узнаем цену товаров
06	double total = 0.0;
07	for (int i = 0; i < customer.items.length; i++)
08	{
09	Item item = customer.items[i];
10	total = total + item. Adjusted_Total();
11	}
12	}

Main() принимает покупателя, прибавляет несколько товаров и подсчитывает стоимость заказа. Все работает, но что произойдет, если изменить способ хранения товаров в классе **Customer** (Покупатель)? Возможно, вы захотите ввести класс **Basket**. Если изменить реализацию, нужно будет изменить весь код, который получает непосредственный доступ к массиву **Item**.

Без сокрытия реализации невозможно усовершенствовать объекты. Например, в классе **Customer** массив **Item** должен быть частным (**private**). Доступ к объектам должен осуществляться с помощью **средства доступа** (например, свойства – **см. Материалы к Практ. Зан. №2, дополнение 1, стр. 12…19**).

Примечание	**Сокрытие реализации имеет свои недостатки**. Иногда требуется информации больше, чем можно получить с помощью интерфейса. В мире программ нужны черные ящики, которые работают с определенным допуском, т.е. с некоторым подходящим количеством разрядов. Например, может случиться так, что вам потребуются **64**-битовые целые числа, так как вы выполняете действия над очень большими числами. Определяя интерфейс, важно не только предоставить его, но и *документировать* специфику типов, использованных при реализации. Однако подобно любой другой части общедоступного интерфейса после определения поведения **изменять его нельзя**.

Скрывая реализацию, можно написать независимую слабо связанную с другими компонентами программу. Слабосвязанная программа более устойчива, причем ее легче модифицировать. А благодаря этому ее проще использовать повторно и усовершенствовать, так как изменения в одной части системы не затрагивают остальных, независимых частей.

Как научиться эффективно скрывать реализацию и создавать слабосвязанные программы**?**
Вот несколько советов:

1. Доступ к данным абстрактного типа должен осуществляться только через методы интерфейса. Такой интерфейс обеспечивает сокрытие информации о реализации.
2. Следует исключить возможность бесконтрольного доступа к структурам данных.
3. Не следует строить догадок об используемых типах. Если поведение не описано в интерфейсе или документации, **не полагайтесь на свои догадки** о нем.
4. Следует соблюдать осторожность при написании двух тесно связанных типов. При программировании избегайте случайного использования допущений и зависимостей.

4.3. Распределение ответственности: заниматься своим делом

Сокрытие реализации естественно связано с понятием ответственности. В предшествующем разделе говорилось о том, как с помощью сокрытия деталей реализации ослабить связи программы.

Но сокрытие реализации — это только первый шаг к написанию слабосвязанной программы.

Чтобы создать слабосвязанную программу, нужно также должным образом распределить ответственность. При надлежащем распределении ответственности **каждый объект выполняет одну функцию-метод**, за которую он несет ответственность и выполняет эту функцию хорошо. Это означает также, что объект образует единое целое. Другими словами, нет смысла в инкапсуляции случайного набора функций и переменных. Между инкапсулируемыми объектами должна быть тесная **концептуальная** связь. Все функции должны выполнять общую задачу.

Примечание	**Сокрытие реализации** и **ответственность** идут рука об руку. Без сокрытия реализации ответственность может просочиться за пределы объекта. Но именно объект должен знать, как решать свою задачу, т.е. именно объект должен содержать алгоритм выполнения своей задачи. Если оставить реализацию открытой (**public**), пользователь может напрямую использовать ее, разделяя таким образом ответственность. **Если два объекта выполняют одно и то же задание**, **значит, нет должного разделения ответственности**. Когда в программе есть избыточные логические схемы, ее нужно переделать. Это обычное явление при создании объектно-ориентированной программы.

Рассмотрим распределение ответственности на примере из жизни: **отношения между менеджером и программистом**.

Представьте, что к вам приходит ваш менеджер, вручает вам технические требования к вашей части проекта и уходит, оставляя вас работать. Он уверен, что вы получили задание, и вы знаете, как выполнить его наилучшим образом.

Теперь представьте, что ваш босс не так умен. Он объясняет, в чем состоят ваши обязанности и говорит, что вы можете обращаться к нему, если возникнут вопросы. Однако как только вы приступаете к работе, он тут как тут, и начинает давать подробные инструкции к каждому оператору программы.

Хотя пример описывает крайность, программисты очень часто допускают подобные ошибки при написании программ. Инкапсуляцию можно сравнить с умелым менеджером. Как и в жизни, знания и ответственность могут быть делегированы тому, кто знает, как выполнить работу наилучшим образом. Многие программисты структурируют свою программу подобно тому, как чересчур властные начальники обращаются со своими подчиненными. Данный пример нетрудно проиллюстрировать конкретной программой. Рассмотрим следующий фрагмент кода:

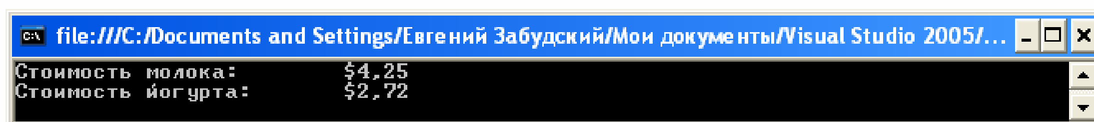
Листинг 2

1	`using System;`
2	
3	`namespace ConsAppl_OOP21_c59_61`
4	`{`
5	`// класс BadItem иллюстрирует, что происходит, если неправильно делить ответственность`
6	`    public class BadItem      ///////////////////////////// начало класса BadItem ////////////////////////////////`
7	`    {`
8	`        private double unit_price;`
9	`        private double adjusted_price;`
10	`        private double discount;   // скидка к цене товара, ее может и не быть`
11	`        private int quantity;`
12	`        private String description;`
13	`        private String id;`
14	
15	`        public BadItem(String id, String description, int quantity, double price) // конструктор класса`

16	{
17	this.id = id;
18	this.description = description;
19	
20	if (quantity >= 0)
21	{
22	this.Quantity = quantity;
23	}
24	else
25	{
26	this.Quantity = 0;
27	}
28	
29	this.unit_price = price;
30	}
31	
32	public double Unit_Price // цена единицы товара
33	{
34	get
35	{
36	return unit_price;
37	}
38	}
39	public double Discount // скидка к цене товара
40	{
41	set
42	{
43	if (discount <= 1.00)
44	{
45	this.discount = value;
46	}
47	}
48	get
49	{
50	return discount;
51	}
52	}
53	
54	public int Quantity // количество единиц товара
55	{
56	set
57	{
58	this.quantity = value;
59	}
60	get
61	{
62	return quantity;
63	}
64	}
65	
66	public String Id // код товара

67	{
68	get
69	{
70	return id;
71	}
72	}
73	
74	public String Description // описание товара
75	{
76	get
77	{
78	return description;
79	}
80	}
81	
82	public double Adjusted_Price // окончательная цена товара
83	{ // см. для сравнения строки 27…33 в листинге 1
84	set
85	{
86	this.adjusted_price = value;
87	}
88	get
89	{
90	return adjusted_price;
91	}
92	
93	}
94	} ////////////////////////////// конец класса BadItem //////////////////////////////////////
95	
96	/*класс BadItemExmple ПОКАЗЫВАЕТ: если объект *неправильно* делит ответственность,
97	то получается процедурная программа (а не объектно-ориентированная) */
98	public class BadItemExample ////////////////// начало класса BadItemExample //////////////
99	{
100	public static void Main(String [] args)
101	{
102	/* создается товар: код товара, описание товара, количество единиц товара (литров), цена 1-го литра молока*/
103	BadItem milk = new BadItem("молочный-011", "Молоко (1 литр)", 2, 2.50);
104	BadItem yogurt = new BadItem("молочный-032", "Персиковый йогурт", 4, 0.68);
105	
106	// размер скидки на цену товара
107	milk.Discount=0.15;
108	
109	// расчет конечной цены молока (см. для сравнения строки 27…33 в листинге 1)
110	double milk_price = milk.Quantity * milk.Unit_Price;
111	double milk_discount = milk.Discount * milk_price;
111	milk.Adjusted_Price = milk_price - milk_discount;
112	// расчет конечной цены йогурта (см. для сравнения строки 27…33 в листинге 1)
113	double yogurt_price = yogurt.Quantity * yogurt.Unit_Price;

114	double yogurt_discount = yogurt.Discount * yogurt_price;
115	yogurt.Adjusted_Price = yogurt_price - yogurt_discount;
116	
117	Console.WriteLine("Стоимость молока:\t $" + milk.Adjusted_Price);
118	Console.WriteLine("Стоимость йогурта:\t $" + yogurt.Adjusted_Price);
119	Console.ReadLine();
120	}
121	} ////////////////// конец класса BadItemExample //////////////
122	}



Класс **BadItem** (см. выше строки **6…94**) больше не отвечает за вычисление скорректированной цены (**см. Листинг 5 в Материалах к лекции №1, часть 1, стр. 33…39**). Каким же образом вычисляется скорректированная цена**?** Рассмотрим метод **Main():**

Теперь вместо того, чтобы запрашивать скорректированную цену у **Item**, вам приходится поступать, как неумелому руководителю. Вам приходится давать пошаговые объяснения объекту (см. выше строки **109…111** и **112…115**). А ведь объект должен самостоятельно выполнять все этапы работы!

Когда программа построена таким образом, что для подсчета скорректированной цены нужно вызывать многочисленные функции, **ответственность, которую должен нести объект, переходит к пользователю**. Такое перемещение ответственности так же плохо, как и *ра*скрытие внутренней реализации. В результате ответственность будет возложена на вашу программу. В каждом объекте, вычисляющем скорректированную цену, придется повторить ту же логику, что и в **Main()**.

При написании интерфейса убедитесь, что ваш интерфейс не сводится к раскрытию реализации под различными именами. Если вернуться к примеру с очередью (раздел 4.2.3, стр. 15,сл), можно заметить, что там не нужны методы **addObjectToList()**, **updateEndListPointer()** и т.д., так как эти методы раскрывают реализацию. Ведь они демонстрировали бы поведение, специфическое для данной реализации. Вместо этого нужно скрыть реализацию на более высоком уровне поведения, предоставляемом методами **enqueue()** и **dequeue()**, хотя внутри вы можете добавлять объект к списку. Имея такой класс как **BadItem**, вам не обязательно вызывать метод **calculateAdjusted_Price()** для того, чтобы вывести скорректированную цену с помощью метода **Adjusted_Price()**. Вместо этого **Adjusted_Price()** может знать, как самому выполнить вычисление.

Если ответственность плохо распределена между объектами, программа будет **процедурно-ориентированной**, а основное внимание, естественно, будет уделено **обработке данных**. Программа **Main**, например, слишком большое внимание уделяет обработке данных. Если программа **Main** дает пошаговые инструкции объекту **Queue** для выполнения постановки в очередь (процесс **enqueue()**), то ее нельзя ***не*** признать процедурно-ориентированной. А вот если вы просто посылаете сообщение объекту и уверены, что он справится с заданием, то вот это и есть настоящее объектно-ориентированное программирование.

С помощью инкапсуляции скрываются детали. В соответствии с принципом распределения ответственности знание конкретных деталей передается только ответственным за надлежащую часть обработки объектам. На один объект следует возлагать ответственность **за одну** (во вся-

ком случае, **небольшое количество) задачу**. *Если на один объект возложена ответственность за большое количество задач, его реализация становится слишком сложной, **ее будет трудно сопровождать и усовершенствовать***. Изменять ответственность будет рискованно, так как при этом придется изменить другие линии поведения, если объект имеет несколько линий поведения. В результате очень большое количество информации концентрируется в одном месте, а ее следует распределять более равномерно. Когда объект становится слишком большим, он фактически становится самостоятельной программой и может не только воспользоваться преимуществами процедурного программирования, но и угодить во все его ловушки. В итоге вы столкнетесь со всеми проблемами, которые возникают в программе, где инкапсуляция не используется вообще.

Обнаружив, что объект отвечает больше чем за одну задачу, нужно перенести часть ответственности в другой объект.

Сокрытие реализации — это только один шаг к эффективной инкапсуляции. **Без должного распределения ответственности** вы в итоге получите список процедур.

На этом этапе можно расширить определение инкапсуляции.

Эффективная инкапсуляция = **абстракция** + **сокрытие реализации** + **ответственность**.

Убрав абстракцию, программу нельзя будет использовать повторно. Убрав сокрытие реализации, вы получите сильносвязанную программу. Если убрать ответственность, то результатом будет процедурная, ориентированная на обработку данных, децентрализованная сильносвязанная программа.

5. Объектно-ориентированный Банк – компьютерная модель реального банка

На рис. 2 представлен **О**бъектно-**о**риентированный **Б**анк. В этом **ООБ**анке потребители становятся в очередь и ждут кассира. Однако не огорчайтесь: класс **Queue** (очередь) писать не придется. В нашем случае придется программировать класс **Account** (**Счет**).

Конечно, счета бывают разными. Различают, например:

1. Счет комиссионного вознаграждения,
2. Специальный счет, с которого снимаются деньги по чекам клиента,
3. Счет, позволяющий в любой момент вносить и снимать деньги
(так называемый счет до востребования),
4. Текущий счет,
5. Депозитный счет денежного рынка.

Но все счета имеют несколько общих свойств:

1. Все счета имеют баланс.
2. На любой счет можно вносить деньги,
3. С любого счета можно их снимать,
4. Можно узнать остаток.

Напишем класс **Account** (**счет**), класс **Teller** (**кассир**). Класс **Teller** (кассир) содержит функцию **Main()**, которую вы можете использовать для тестирования реализации класса **Account** (**счет**).

Для класса **Teller** (**кассир**) необходим специальный общедоступный интерфейс. Вот правила, которые необходимо соблюдать при разработке класса **Account** (**счет**):

1. Вы должны назвать этот класс **Account**;

2. Класс должен иметь следующие два конструктора:

```
public Account()
public Account(double initial_deposit)
```

Конструктор без аргументов установит начальный остаток равным **0.00**. Второй конструктор установит начальный баланс равным **initial_deposit;**

3. Класс должен иметь два метода и одно свойство.

Первый метод выполняет кредитование счета с указанием денежных средств (**funds**):

```
public void depositFunds(double funds)
```

Второй метод служит для проведения по счету денежных средств (**funds**):

```
public double withdrawFunds(double funds)
```

Однако метод **withdrawFunds()** не должен допускать превышение кредита. Это значит, что если **funds** больше, чем денег на балансе, то записать в дебет разрешается только имеющийся на балансе остаток. Функция **withdrawFunds()** должна возвращать фактическое количество денег, снимаемое со счета.

Свойство (**get**-блок) выдает текущий остаток на счете:

```
public double Balance
```

Коль скоро указанные правила соблюдены, можно добавить любые другие необходимые методы. Однако убедитесь, что каждый из вышеперечисленных методов работает именно так, как описано выше. В противном случае кассир (**Teller**) не сможет выполнить своей работы!

1	using System; // Листинг 3
2	// Инкапсуляция. Объектно-Ориентированыый Банк
3	namespace ConsAppl_OOP21_c73_75
4	{
5	public class Account //////////////////////// начало класса Account //////////////////////
6	{
7	private double balance; // данные частного (private) характера
8	
9	public Account(double init_deposit) // Конструктор с одним параметром
10	{
11	balance = init_deposit;
12	}
13	
14	public Account() // Конструктор по умолчанию
15	{
16	// не нужно ничего делать, баланс по умолчанию равен нулю, balance = 0
17	}
18	
19	public void depositFunds(double amount) // Метод - внесение денежных сумм на счет
20	{
21	balance += amount; // Console.WriteLine("balance=" + Balance);
22	}
23	
24	public double Balance // Свойство: запрос остатка (balance есть private-переменная)

25	{
26	get
27	{
28	return balance;
29	}
30	}
31	
32	public double withdrawFunds(double amount) // Метод - снятие денег со счета
33	{
34	if (amount > balance)
35	{ // исправить сумму
36	amount = balance; // Console.WriteLine("amount=" + amount);
37	}
38	
39	balance -= amount; //Console.WriteLine("balance=" + Balance);
40	// Console.WriteLine("amount=" + amount);
41	return amount;
42	}
43	} ///////////////////////// конец класса Account /////////////////////////////////
44	public class Teller ///////////////////////// начало класса Teller ///////////////////
45	{
46	
47	public static void Main(String [ ] args)
48	{
49	
50	// открыть счета
51	Account account1 = new Account();
52	Account account2 = new Account();
53	Account account3 = new Account(5000.00);
54	// положить на счета сумму
55	account1.depositFunds(500.00); // balance = 500.00
56	account2.depositFunds(1000.00); // balance = 1000.00
57	account3.depositFunds(5000.00); // balance = 10000.00
58	// снять со счетов сумму
59	account1.withdrawFunds(600.00); // balance = 0.00
60	account2.withdrawFunds(1.00); // balance = 999.00
61	account3.withdrawFunds(1150.00); // balance = 8850.50
62	
63	Console.WriteLine("Я создал три счета.");
64	
65	double balance1 = account1.Balance;
66	double balance2 = account2.Balance;
67	double balance3 = account3.Balance;
68	
69	if(balance1 != 300.00)
70	{
71	Console.WriteLine("\nСчет 1 не имеет баланс, соответствующий 300.00. Баланс счета " + balance1);

72	}
73	else
74	{
75	Console.WriteLine("\nСчет 1 имеет баланс, соответствующий 300.00.");
76	}
77	
78	if(balance2 != 999.00)
79	{
80	Console.WriteLine("\nСчет 2 не имеет баланс, соответствующий 999.00. Баланс счета " + balance2);
81	}
82	else
83	{
84	Console.WriteLine("\nСчет 2 имеет баланс, соответствующий 999.00.");
85	}
86	
87	if(balance3 != 8850.00)
88	{
89	Console.WriteLine("\nСчет 3 не имеет баланс, соответствующий 8850.00. Баланс счета " + balance3);
90	}
91	else
92	{
93	Console.WriteLine("\nСчет 3 имеет баланс, соответствующий 8850.00.");
94	}
95	
96	Console.ReadLine();
97	} ////////////////////////// конец метода Main ////////////////////////////////
98	} ///////////////////////////// конец класса Teller (кассир) //
99	}

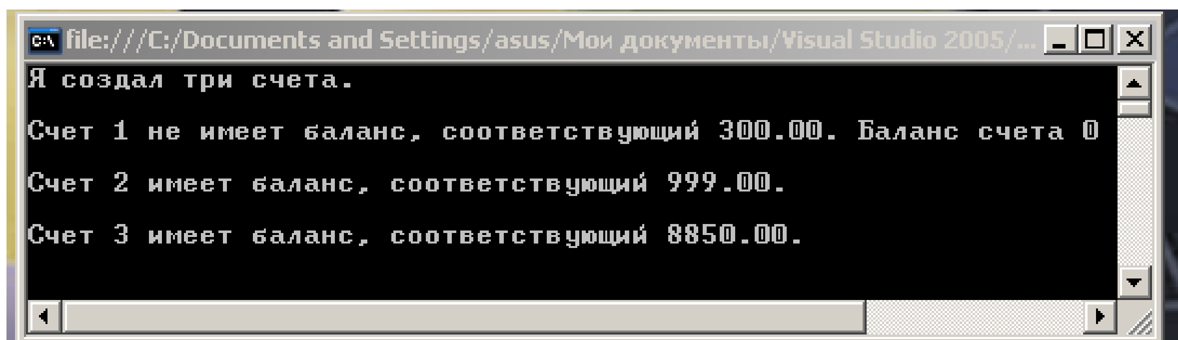


Рис. 7. Правильные выходные данные, полученные от класса Teller

Класс Account демонстрирует важные концепции инкапсуляции:

1. Он довольно **абстрактный**. Он может быть **принят в качестве основы при реализации счетов многих конкретных типов.**
2. Класс Account **прячет свою реализацию** за правильно определенным интерфейсом.
3. Наконец, в классе Account правильно распределена ответственность, **поскольку он содержит в себе все знания, необходимые для внесения денежных сумм на счет и снятия денег со счета.** За пределы объекта информация о том, как это делается, не "просачивается".

Несмотря на это, класс **Account** можно существенно усовершенствовать. Для краткости, в версии класса **Account**, использованной в решении, опущены какие-либо проверки аргумента, кроме простой проверки превышения кредита. В реальности нужно добавить код, который будет проверять все параметры метода

6. Инкапсуляция: советы и типичные ошибки

Применяя инкапсуляцию, следует не только выполнять множество правил, но и избегать множества ошибок.

6.1. Абстракция: советы и ошибки

Чрезмерное использование абстракции может вызвать определенные проблемы при написании класса. Невозможно написать класс, подходящий всем пользователям и во всех ситуациях. Допустим, нужно написать объект **Person** для системы платежных ведомостей какого-то предприятия. Этот объект **Person** будет значительно отличаться от объекта **Person** в модели транспортного движения.

Внутри класса обычно находятся: ***1)*** методы и ***2)*** внутренние переменные — данные. Доступ к этим переменным и методам предоставляется функциями доступа (например, свойства – **см. Материалы к Практ. Зан. №2, дополнение 1, стр. 12…19**). Интерфейс абстрактного типа данных должен быть частью общедоступного (**public**) интерфейса объекта.

Абстрактный тип данных не является прямым аналогом объектно-ориентированного класса. Абстрактному типу данных не хватает возможностей наследования и полиморфизма.

6.2. Советы по сокрытию реализации

Не всегда просто решить, что в интерфейсе оставлять на виду (**public**), а что прятать (**private**). Однако можно сделать несколько замечаний о доступе, которые не зависят от языка. В общедоступном интерфейсе должны быть только те методы (**public**), которые необходимы другим пользователям. Методы, которые будет использовать только данный тип, должны быть спрятаны (**private**). В примере с очередью методы **dequeue** и **enqueue** (см. раздел 4.2.3) должны быть в общедоступном интерфейсе. Однако вспомогательные методы, такие как **updateFrontPointer()** и **addToList()**, надо спрятать.

Следует **всегда** скрывать (private) внутренние переменные кроме тех случаев, когда они являются константами. Важно, чтобы они были не только скрыты, но и чтобы доступ к ним мог получить только класс. При разрешении доступа к внутренним переменным *открывается реализация*.

Если пользователи имеют доступ к методам и значениям, не зная, что имеют дело именно со значением, то в этом случае значение можно открыть. В таком языке открытая внутренняя переменная будет выглядеть так же, как метод без параметров.

7. Как инкапсуляция способствует достижению целей объектно-ориентированного программирования

Целью объектно-ориентированного программирования является создание программы, обладающей следующими свойствами:

1) естественность;
2) надежность;
3) возможность повторного использования;
4) удобство в сопровождении;
5) способность совершенствоваться;
6) удобство периодического выпуска (издания) новых версий.

Этих целей можно достичь с помощью инкапсуляции.

1) Естественность: с помощью инкапсуляции можно распределить ответственность таким способом, который кажется естественным с точки зрения человека. Применяя абстракцию, **решение задачи можно выразить в терминах той предметной области, к которой относится решаемая задача**, **а не в терминах реализации**. Абстракция позволяет выделить в задаче главное.
2) Надежность: изолируя ответственные участки кода и **скрывая реализацию**, **можно проверить правильность каждого отдельного компонента**. Когда используется проверенный компонент, это позволяет провести тщательную проверку каждого модуля, так что нет причин волноваться. И все же нужна общая проверка, чтобы убедиться в том, что программа работает правильно.
3) Возможность повторного использования: с помощью абстракции **можно создать легко поддающуюся изменениям программу**, пригодную для использования в различных ситуациях.
4) Удобство в сопровождении: защищенную программу легче сопровождать. **Можно вносить любые необходимые изменения в реализацию класса, не изменяя зависимый код**. Эти изменения могут включать как изменения в реализации, так и добавление новых методов в интерфейс. Только изменения семантики *интерфейса* требуют изменений зависимого кода.
5) Совершенствование: **можно изменить реализацию, не разрушая программу**. Другими словами, можно усовершенствовать функциональные характеристики, сохраняя работоспособность существующего кода. Даже более того, поскольку реализация скрыта, эксплуатационные характеристики кода, использующего усовершенствованный компонент, улучшатся автоматически — ведь код, хотя он и не изменился(**!**), будет использовать усовершенствованные компоненты**!** Однако после внесения изменений следует снова произвести проверку модуля. Изменение объекта может вызвать эффект домино во всем коде, использующем объект.
6) Удобство периодического выпуска (издания) новых версий: разбивая программу на самостоятельные модули, **задание по разработке кода можно распределить между несколькими разработчиками**, и таким образом ускорить процесс разработки.

Разработав и проверив компоненты, их не придется переделывать заново. Таким образом, **программист может просто повторно использовать эти компоненты**, а не тратить время на создание их с "нуля".

8. Может ли инкапсуляция быть вредной?

Действительно, инкапсуляция может быть вредной. Предположим, что есть компонент, который выполняет математические расчеты, причем необходимо добиться определенной точности этих расчетов. К сожалению, компонент может полностью инкапсулировать количество сохраняемых разрядов. Если в расчетах сохранялось недостаточное количество разрядов, то результат может быть неправильным. Если кто-то будет изменять данный компонент, вы можете обнаруживать странные сбои в работе.

Значит, инкапсуляция может быть вредной, если вам надо полностью контролировать все методы, с помощью которых объект обрабатывает ваши запросы.

Единственным средством зашиты является *полная документация*. Поэтому нужно документировать все важные детали реализации и допущения. Создав документацию, уже не так легко будет внести изменения в детали и допущения. Если, например, сделать изменения в рассмотренном выше математическом компоненте, то можно разрушить всех пользователей объекта.

9. Предостережения

Для применения **абстракции** и **инкапсуляции** при разработке программы вовсе не обязательно применять объектно-ориентированный подход. Абстрактные типы данных сами по себе не являются объектно-ориентированными. Применять инкапсуляцию позволяет практически любой язык.

Однако есть одна проблема. В языках других типов (*не* объектно-ориентированных) часто необходимо создавать свой собственный механизм инкапсуляции. Так как в языке нет ничего такого, что бы помогало вам отслеживать вами же установленные стандарты, надо быть внимательным и самому неусыпно следить за соблюдением этих стандартов. Вам придется также при написании каждой программы повторно воспроизводить ваши руководящие принципы и соответствующий механизм.

Это подходит для одного разработчика. А если разработчиков будет двое**?** Десять**?** Большая группа**? Чем больше разработчиков, тем сложнее выдержать единый стиль**.

Настоящий объектно-ориентированный язык **предоставляет свой механизм инкапсуляции**. Таким образом, вам не приходится этого делать. Язык инкапсулирует детали механизма инкапсуляции от пользователя. В объектно-ориентированном языке для этого **предусмотрен набор ключевых слов.** Программист использует только эти ключевые слова, а обо всех деталях заботится компилятор, реализующий объектно-ориентированный язык программирования.

Когда программисты используют средства языка, **то все они используют один и тот же механизм**, **а поскольку механизм один и тот же, то вопрос о совместимости механизмов**, **используемых разными разработчиками**, даже не возникает!

Резюме

Разобравшись с тем, что такое инкапсуляция, можно начать использовать объекты при программировании. **Прибегая к инкапсуляции, можно воспользоваться**: ***1)*** преимуществами абстракции, ***2)*** сокрытия реализации и ***3)*** возможностью распределения ответственности при создании профессионального кода, соответствующего современному уровню развития науки программирования.

С помощью абстракции можно создать объекты, **пригодные для использования в различных ситуациях**. Если подходящим образом **скрыть реализацию объекта**, то перед вами откроется возможность **свободного усовершенствования программы**. И, наконец, если надлежащим образом распределить ответственность между объектами, то **можно избежать дублирования логики и процедур в программе**.

Вопросы студента и ответы

1. Как узнать, какие методы включать в интерфейс?

Узнать, какие методы нужно включать, совсем не трудно. Следует включать только те методы, **которые делают объект полезным**; т.е. только те методы, которые необходимы для того, чтобы **другие объекты могли выполнять свою работу**.

Желательно, чтобы интерфейс был как можно меньше и удовлетворял всем потребностям. **Интерфейс класса нужно делать как можно проще**. Не следует включать методы, которые "возможно" понадобятся. Их можно будет добавить тогда, когда они действительно будут нужны.

Следует остерегаться некоторых так называемых "удобных методов". Если объект содержит другие объекты, то методы, которые просто содержат вызов метода одного из содержащихся объектов, не нужны.

Скажем, к примеру, у вас есть объект "тележка для магазинов самообслуживания", содержащий товары (см. Лекция 1, часть 1, листинг 5). Не следует добавлять к объекту "тележка для магазинов самообслуживания" удобный метод, который будет запрашивать цену товара и возвращать ее. Вместо этого нужен метод, который позволит получить товар. Получив товар, вы сможете сами узнать его цену.

2. Существуют ли модификаторы доступа, отличные от тех, которые задаются ключевыми словами public, protected и private?

В языке C# предусмотрены еще два модификатора доступа: internal (внутренний) (см. Материалы к Практич. занятию №4, **стр. 33, 34**); protected internal (внутренний защищенный) (см. Материалы к Практич. занятию №6, **стр. 15**).

3. Дублируют ли модификаторы доступа механизм защиты?

Нет. Модификаторы доступа всего лишь ограничивают взаимодействия между данным объектом и другими объектами. Модификаторы не имеют ничего общего с защитой компьютера.

Контрольные вопросы

1. Каким образом использование инкапсуляции помогает достичь целей объектно-ориентированного программирования?
2. Дайте определение понятию "абстракция" и приведите пример применения абстракции.
3. Дайте определение понятию "реализация".
4. Дайте определение понятию "интерфейс".
5. Объясните разницу между интерфейсом и реализацией.
6. Почему для достижения эффективной инкапсуляции важно четко распределить ответственность?
7. Определите понятие типа.
8. Что такое абстрактный тип данных?
9. Как можно получить эффективное сокрытие реализации в сильносвязанной программе?
10. Какие опасности таит абстракция?
11. В классе **Account** (**листинг 3**): какой метод называется **мутатором**? какой метод называется **средством доступа**?
12. Назовите два типа конструкторов. Найдите в **листинге 3** пример конструктора каждого типа.

Ответы на контрольные вопросы

1. Инкапсуляция естественна. Инкапсуляция позволяет моделировать программное обеспечение в терминах поставленной задачи, а не в терминах, определяемых способом решения этой задачи.

   Она делает программное обеспечение более надежным. Инкапсуляция скрывает методы обработки данных и гарантирует, что доступ к ним осуществляется корректно. Инкапсуляция позволяет распределить и закрепить обязанности каждого объекта. Если доказано, что какой-либо компонент работает подобающим образом, его без опасений можно использовать в других программах.

   Инкапсуляция делает программное обеспечение универсальным. Так как компоненты программного обеспечения независимы, то их можно многократно использовать во многих других ситуациях.

   Инкапсуляция облегчает поиск и исправление ошибок. Изменения, внесенные в один компонент, не влияют на другие. Таким образом, значительно облегчается исправление ошибок и усовершенствование программы.

   Инкапсуляция делает программное обеспечение модульным. Изменение одной части программы не повлияет на другие ее части. Модульность позволяет исправлять ошибки и усовершенствовать программу, не затрагивая не относящуюся к этим изменениям часть программного кода.

   Инкапсуляция позволяет ускорить создание программ, так как устраняет излишние зависимости в программном коде. Очень часто такие скрытые зависимости приводят к появлению ошибок, которые трудно найти и исправить.

2. Абстракция — процесс упрощения сложной задачи. Когда программист приступает к решению какой-либо задачи, он не позволяет себе останавливаться на рассмотрении каждой детали. Наоборот, он обращает внимание лишь на детали, имеющие определяющее значение для решения задачи.

   "Рабочий стол" на экране монитора компьютера — пример абстракции. Он полностью скрывает детали устройства файловой системы.

3. Реализация определяет, каким образом компонент выполняет какое-либо действие. Она определяет внутренние детали компонента.

4. Интерфейс определяет, какие действия может выполнить компонент. Интерфейс полностью скрывает детали реализации.

5. Интерфейс определяет, что делает компонент; реализация определяет, как он это делает.

6. Без четкого разделения обязанностей начинается неразбериха. Подобная неразбериха приводит к двум взаимосвязанным проблемам.

   Во-первых, не удается централизовать, т.е. собрать в одном месте программы, код, который должен быть централизованным. И поэтому не удается сконцентрировать в одном месте выполнение необходимых функций, т.е. приходится повторять код, реализующий их выполнение, иными словами, реализовать повторно в каждом месте, где это понадобится. Вернемся к примеру с **BadItem**, приведенному в разделе **4.3** (листинг **2**).

   Легко видеть, что каждый пользователь должен иметь реализацию кода для вычисления общей стоимости товара. Каждый раз, когда переписывается программный код, существует риск допустить ошибку. Более того, создается возможность некорректного использования компонента, так как на компонент нельзя возложить обязанность сохранять свое внутреннее состояние. Наоборот, эта обязанность перекладывается на другие части системы.

7. Тип — элемент языка, представляющий некоторый элемент вычислений или поведения. Если сравнить строку кода с предложением, типы окажутся словами. Обычно типы трактуются как независимые, самодостаточные, элементарные блоки.

8. Абстрактный тип данных (Abstract Data Type, ADT) — набор данных и операций, осуществляемых с этими данными. Абстрактный тип данных позволяет определять новые типы языка, скрывая данные и состояние за четко определенным интерфейсом. Интерфейс представляет абстрактный тип данных как единичный элементарный блок.

9. Существует несколько способов создания скрытого и слабосвязанного кода. Простым способом является использование инкапсуляции. Однако эффективность этого способа зависит от программиста. Приведем несколько советов по эффективному применению инкапсуляции:

- Получать доступ к абстрактному типу данных следует только через метод или интерфейс; никогда не делайте внутренние структуры данных частью общедоступного интерфейса.
- Не предоставляйте доступ к внутренним структурам данных; используйте только операции, определенные для абстрактного типа данных.
- Никогда не используйте типы, в которых нет полной уверенности. Не используйте поведение, если оно не определено в интерфейсе или не описано в документации.
- Соблюдайте осторожность при реализации двух тесно связанных типов. Не допускайте даже случайно неоправданных предположений и зависимостей.

10. При абстракции старайтесь избегать следующих ловушек. Избегайте остановок при проведении абстракции. Решайте появляющиеся проблемы последовательно. Главной целью программиста является решение задачи. Смотрите на абстракцию как на приз, а не как на конечную цель. В противном случае вы можете не уложиться в конечный срок и допустить ошибки при выполнении абстракции. Бывает, что необходимо применить абстракцию, но случается, что абстракция не нужна.

Абстракция может представлять опасность. Даже если некоторый элемент получен в результате абстракции, он не всегда может работать правильно. Очень сложно реализовать класс, который будет удовлетворять всем требованиям пользователя.

Закладывайте в класс только те возможности, которые необходимы для решения поставленной задачи.

Не пытайтесь решить все проблемы разом. Решите наиболее насущную проблему, а затем уже попытайтесь применить абстракцию к полученному решению.

11. Класс **Account** (**листинг 3**) имеет **два мутатора**: **depositFunds()** (см. выше строки **19…22**) и **withdrawFunds()** (см. выше строки **32…37**). Класс **Account** имеет **один метод доступа**: свойство **Balance** (см. выше строки **24…30**).

12. В классе **Account** (**листинг 3**) есть два типа конструкторов: один вызывается с аргументами (см. выше строки **9…12**), другой без них (см. выше строки **14…17**).

Упражнения

1. Рассмотрите классическую структуру данных — **стек**. Стек представляет собой структуру типа "последним пришел — первым обслужен". В отличие от очереди типа "последним пришел — первым обслужен" в стек можно добавлять элементы и убирать их только *с одного и того же* конца. Так же как и очередь, стек можно проверить, чтобы узнать, не пуст ли он, а, кроме того, еще и выбрать первый элемент, который затем можно удалить (см. рис. 6).
2. Определите абстрактный тип данных для класса **стек**.
3. Используя абстрактный тип данных **стек** из первого упражнения, опишите еще одну его реализацию.
4. Вернитесь к первому и второму упражнениям. Подошел ли интерфейс, разработанный в упражнении 1, для обеих реализаций, которые вы рассматривали во втором упражнении? Если да, то какие преимущества предлагает интерфейс? Если нет, то какими недостатками обладал первоначальный интерфейс?

Ответы к упражнениям

1. Возможный вариант реализации стека как абстрактного типа данных (ср. с рис. 6):

```
01  public interface Stack
02  {
03      public void push(Object obj);
04      public Object pop();      // выталкивание объекта
05      public bool isEmpty();
06      public Object peek();
07  }
```

2. Стек проще всего реализовать как односвязный список с указателем на начало списка. Добавляя элемент в стек или удаляя (выталкивая) элемент из стека, указатель на начало списка используется для поиска первого элемента.

3. Сравнив приведенные ответы для упражнений 1 и 2, можно убедиться, что описанный интерфейс удовлетворяет заданным требованиям. Он обладает достоинствами, которыми обладает каждый корректно созданный интерфейс. Приведем короткий список таких достоинств:

- Интерфейс определяет **стек** как тип. Просмотрев описание интерфейса, можно точно сказать, что способен делать **стек**.
- Интерфейс полностью скрывает внутреннее представление данных в **стеке**.
- Интерфейс четко определяет возможности **стека**.

Литература к курсу

Базовый учебник

1. Мейер Б. Объектно-ориентированное конструирование программных систем. М.: Русская Редакция, 2005.

Основная

2. Буч Г., Якобсон А., Рамбо Дж. **UML**. С.-Петербург: Питер, 2006.
3. Гамма Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования. С.-Петербург: Питер, 2006.
4. Забудский Е.И. Учебно-методические материалы по дисциплине «Объектно-ориентированный анализ и программирование». М.: Кафедра ОИиППО ГУ-ВШЭ, 2005,

   Internet-ресурс – http://new.hse.ru/C7/C17/zabudskiy-e-i/default.aspx .
5. Кватрани Т. Визуальное моделирование с помощью Rational Rose 2002 и UML. М.: Вильямс, 2003.
6. Лафоре Р. Объектно-ориентированное программирование в С++. С.-Петербург: Питер, 2005.
7. Троелсен Э. C# и платформа .NET. С.-Петербург: Питер, 2006.
8. Синтес А. Освой самостоятельно объектно-ориентированное программирование за 21 день. Москва; С.-Петербург; Киев: Вильямс, 2002.

Дополнительная – **Internet-ресурсы**

9. Новые книги раздела **C#** – http://books.dore.ru/bs/f6sid16.html
10. **C#** и **.NET** по шагам – http://www.firststeps.ru .
11. **UML** – язык графического моделирования – http://www.uml.org/ .
12. **JUnit** – каркас тестирования для испытания *Java*-классов – http://www.junit.org .
13. Пакет объектного моделирования **Rational Rose** – http://www-306.ibm.com/software/rational/

Дополнительная – книги

14. Мэтт Вайсфельд. Объектно-ориентированный подход: Java, .NET, C++. М.: КУДИЦ-ОБРАЗ, 2005.
15. Дж. Кьоу, М. Джеанини. Объектно-ориентированное программирование. С.-Петербург: Питер, 2005.

Упражнение по программированию (задание на дом)

1. Реализовать в среде MS VS .NET 2005 и проанализировать рассмотренные программы: **листинги 1…3**.

2. Напишите программу основа которой приведена в разделе 4.2.5.

В классе **Customer** хранятся все выбранные товары (включите несколько товаров). В классе **Customer** предусмотрите массив **Item** как часть закрытого (**privare**) интерфейса класса. Доступ к объектам должен осуществляться с помощью **средства доступа. Main()** принимает покупателя, прибавляет несколько товаров и подсчитывает стоимость заказа. Проявите инициативу – фантазируйте и дерзайте.