3

Объектный подход в действии

Ознакомившись в первых двух главах с основами объектно-ориентированного подхода, можно рассмотреть более сложную задачу создания на его основе реального приложения.

В качестве такого приложения рассмотрим адресную книгу или контакт-менеджер, предназначенный для управления информацией о людях (индивидуумах) и организациях, и позволяющий пользователям находить и редактировать информацию о контактах. Материал этой главы познакомит читателя с основными принципами проектирования реальных объектно-ориентированных приложений. Попутно будет показано, как применять основные принципы объектно-ориентированной разработки, такие как повторное использование кода, инкапсуляция, наследование и, конечно, абстракция.

Создание менеджера контактов

Приложение для управления контактной информацией — менеджер контактов — позволяет пользователю отслеживать информацию о людях и организациях (почтовый адрес, адрес электронной почты и номер телефона), а также отношения между ними. По существу, это аналог адресной книги в Microsoft Outlook.

Приведем краткое высокоуровневое описание требований к приложению.

- Приложение будет отслеживать информацию о людях и организациях в базе данных и отображать ее на Web-странице.
- Каждому контакту может соответствовать один и более почтовых и электронных адресов, а также телефонных номеров, а может не соответствовать ни одного.
- Человек может быть связан лишь с одной организацией.
- К каждой организации относится один или несколько сотрудников (а может и ни одного).

Представим эти взаимосвязи в виде диаграммы UML. Однако, прежде чем приступить к ее построению, следует сделать одно замечание. Цель данной главы — продемонстрировать эволюцию видения системы в ходе ее создания, изменение и выявление новых решений в процессе разработки. Поэтому в данной главе описывается эволюционный процесс развития приложения, а не листинг его кода. Рассматриваемый конкретный пример позволит лучше понять, как применять на практике принципы объектно-ориентированного подхода.

Диаграммы UML для адресной книги

Запустите любое приложение, предназначенное для построения диаграмм UML, и создайте новый файл с именем `ContactManager.[`*`расширение`*`]`, где [*расширение*] — это расширение файла, присваиваемое приложением по умолчанию (например, `.dia` для диаграмм, созданных с использованием редактора Dia).

Сначала создайте классы для трех разных видов контактной информации, которая будет предоставлена в приложении. В данном случае это будут классы почтового адреса, электронного адреса и номера телефона. Можно также создать классы для любой другой контактной информации Свойства указанных классов приведены в следующей таблице.

Класс	*Свойства*
`Address`	`street1`
	`street2`
	`city`
	`state`
	`zip` (почтовый индекс)
	`type` (домашний, рабочий и т.д.)
`EmailAddress`	`email`
	`type`
`PhoneNumber`	`number`
	`extension`
	`type`

Эти классы предназначены лишь для хранения и отображения информации, поэтому их методы пока определять не нужно, и соответственно третья секция в обозначении каждого класса остается пустой. Все свойства являются открытыми, поэтому в качестве префикса в имени атрибута используется символ “плюс”. На рис. 3.1 показано текущее представление этих классов в виде обозначений UML.

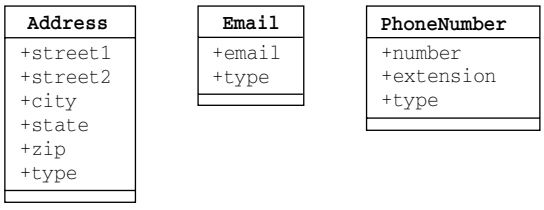

Рис. 3.1.

Далее необходимо разработать классы Individual (человек) и Organization (организация). Человек имеет имя, фамилию, уникальный идентификатор (поле id в базе данных), ему соответствует набор контактных данных (адрес электронной почты, почтовый адрес и телефонный номер), представляемый в виде коллекции, организация и должность. Необходимо также обеспечить возможность добавлять другие виды контактной информации. На рис. 3.2 изображена предыдущая диаграмма UML с добавлением класса Individual.

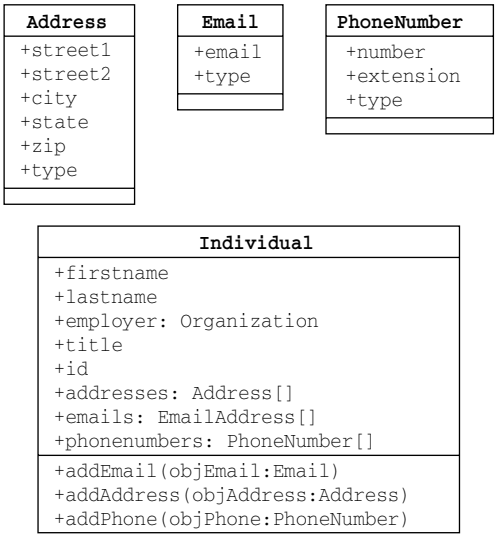

Рис. 3.2.

Организация имеет название, уникальный идентификатор, с ней связан набор контактных данных (таких же, как и у отдельного человека), методы для их добавления, а также "коллекция" индивидуумов (сотрудников). На рис. 3.3 изображена та же диаграмма UML после включения класса Organization.

Из диаграммы видно, что классы Individual и Organization имеют множество одинаковых свойств и методов. По большому счету это свидетельствует о том, что можно улучшить проектное решение с помощью наследования, значительно сэкономив трудозатраты и повысив гибкость приложения. Можно создать другой класс (в данном случае Entity), в котором объединить элементы, характерные для классов Individual и Organization, и дать возможность этим классам совместно использовать общий для них код. На диаграммах UML свойства и методы отображаются только в тех классах, в которых они фактически реализованы. Поэтому все одинаковые свойства и методы классов Individual и Organization необходимо переместить в символьное представление класса Entity (рис. 3.4).

В данном случае свойство name (имя) класса Entity перекрывается в производном классе Individual. Поэтому при извлечении значения свойства name будет возвращена строка, содержащая фамилию и имя — "lastname, firstname". Таким образом объекты классов Organization и Individual можно передавать в общую функцию, которая выводит имя, не прибегая к отдельным функциям для каждого из этих классов.

В UML также определены обозначения для отображения взаимосвязей. В данном примере необходимо показать, что классы Individual и Organization наследуются

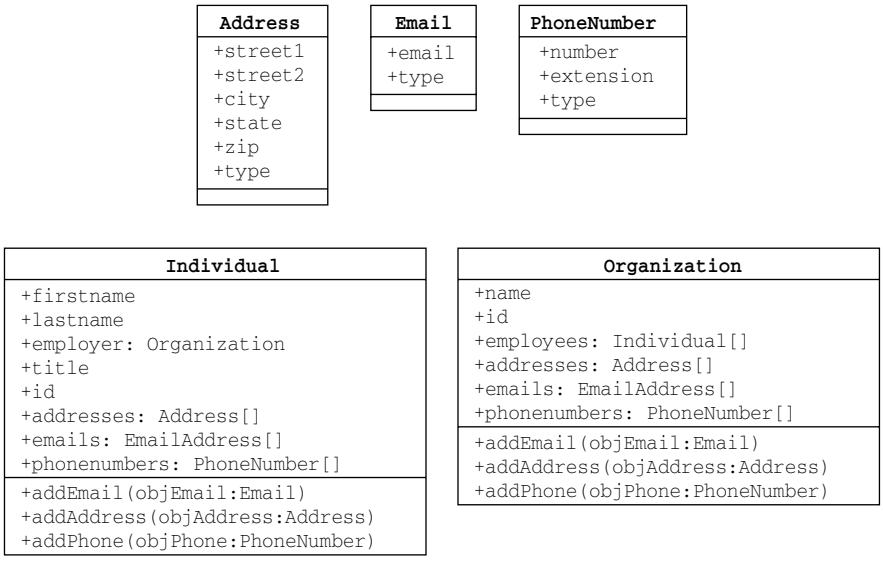
Address
+street1
+street2
+city
+state
+zip
+type
Email
+email
+type
PhoneNumber
+number
+extension
+type
Individual
+firstname
+lastname
+employer: Organization
+title
+id
+addresses: Address[]
+emails: EmailAddress[]
+phonenumbers: PhoneNumber[]
+addEmail(objEmail:Email)
+addAddress(objAddress:Address)
+addPhone(objPhone:PhoneNumber)
Organization
+name
+id
+employees: Individual[]
+addresses: Address[]
+emails: EmailAddress[]
+phonenumbers: PhoneNumber[]
+addEmail(objEmail:Email)
+addAddress(objAddress:Address)
+addPhone(objPhone:PhoneNumber)

Рис. 3.3.

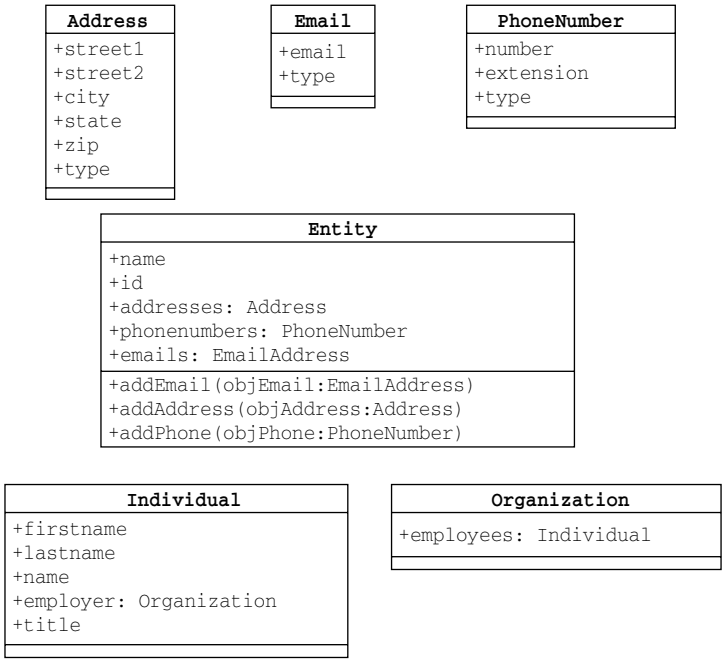
Address
+street1
+street2
+city
+state
+zip
+type
Email
+email
+type
PhoneNumber
+number
+extension
+type
Entity
+name
+id
+addresses: Address
+phonenumbers: PhoneNumber
+emails: EmailAddress
+addEmail(objEmail:EmailAddress)
+addAddress(objAddress:Address)
+addPhone(objPhone:PhoneNumber)
Individual
+firstname
+lastname
+name
+employer: Organization
+title
Organization
+employees: Individual

Рис. 3.4.

от класса Entity. В спецификации UML это отношение называется *обобщением* (generalization) и обозначается линией связи с полым треугольником на конце, направленным от дочернего класса к родительскому, как показано на рис. 3.5.

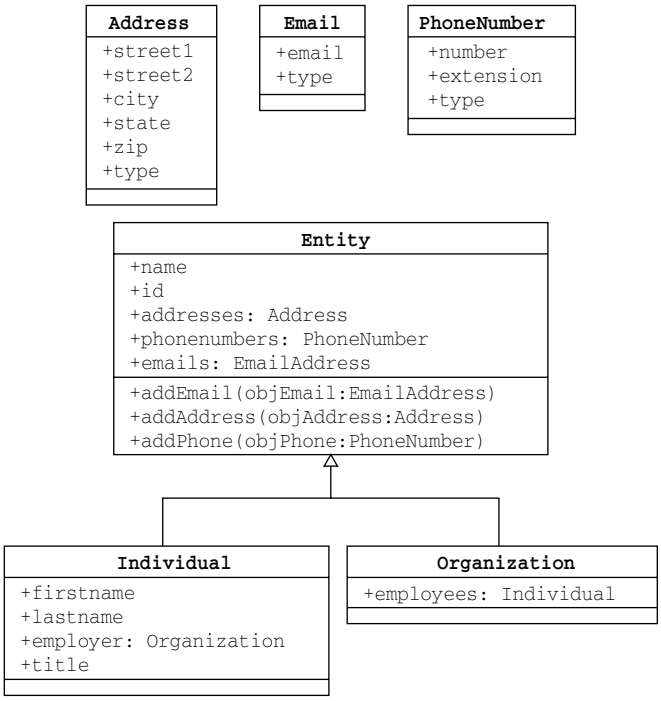

Рис. 3.5.

Теперь видно, что классы Individual и Organization наследуются от класса Entity. Использование линий связи для указания наследования позволяет при беглом взгляде на диаграмму легко определить, в каких отношениях состоят классы друг с другом.

На диаграмме следует отобразить еще один тип связи — использование классом Entity классов Address, Email и PhoneNumber. В спецификации UML это отношение называется *композицией* (composite) и обозначается с помощью линии с закрашенным ромбом на конце. Ромб изображается со стороны класса-агрегата, который является пользователем другого класса (рис. 3.6). Используемые классы обладают свойством *кратности* (multiplicity), которое указывает, какое количество объектов используется. В данном примере с каждой сущностью Entity могут быть связаны ноль, один или более экземпляров контактной информации, поэтому над соединительной линией рядом с используемым классом указывается 0..*. Это означает, что класс-композит может ссылаться на 0 или более объектов данного типа. Эти обозначения показаны на рис. 3.6. Отобразив это отношение на диаграмме, можно легко понять, какие части приложения будут затронуты при изменении некоторой его части. В данном случае изменение классов Email, Addess или PhoneNumber отразится на классах Entity, Individual и Organization.

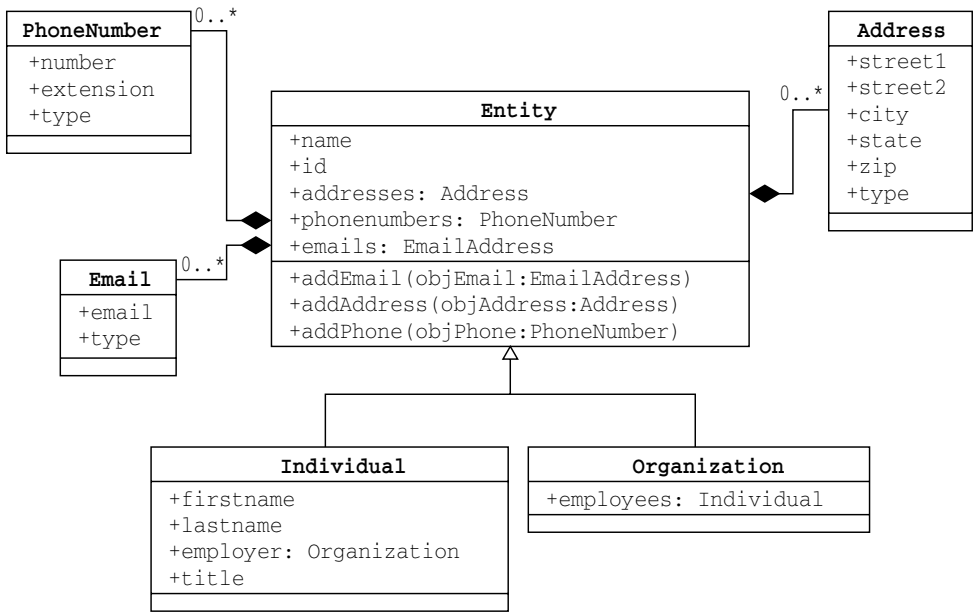

Рис. 3.6.

Класс `Entity` агрегирует объекты `PhoneNumber`, `Email` и `Address`, и в каждом из трех случаев он может включать один или несколько таких элементов либо вовсе не включать их. Так как классы `Individual` и `Organization` унаследованы от `Entity`, любой из них также может быть связан с одним или несколькими видами контактной информации.

При обсуждении инкапсуляции в главе 1 говорилось о том, что в целях безопасности данные целесообразно хранить в закрытых (private) переменных-членах, а для организации доступа к ним использовать методы доступа. Напомним также, что использование методов `__get` и `__set` несколько упрощает этот процесс. Из приведенной выше диаграммы классов видно, что требования к свойствам всех изображенных классов довольно просты, поэтому выполнить проверку корректности ввода их значений не составит труда. Однако для обеспечения общности кода и возможности его повторного использования эту функциональность целесообразно убрать из класса `Entity` и перенести в другой класс, добавив еще один уровень абстракции, реализующий общие для всех классов функции. Назовем этот родительский класс `PropertyObject`.

При работе с данными тоже целесообразно добавить еще один уровень абстракции, т.е. создать класс `DataManager`. Принимая во внимание отсутствие унифицированного средства для проверки данных, необходимо разработать интерфейс для проверки корректности ввода. Приведем код такого интерфейса.

```
<?php
  interface Validator {
    abstract function validate();
  }
?>
```

Сохраним этот код в файле `interface.Validator.php` и используем этот интерфейс для работы с новым классом `PropertyObject`, который будет рассмотрен ниже.

Класс `PropertyObject`

Следующий ниже код позволяет объединить интерфейс `Validator` с классом `PropertyObject`. Внесите код в файл под названием `class.PropertyObject.php`.

```
<?php
  require_once('interface.Validator.php');

  abstract class PropertyObject implements Validator {

    protected $propertyTable = array();
                        //содержит пары "имя-значение",
                        //которые связывают свойства с
                        //именами полей базы данных
    protected $changedProperties = array();
                        //список свойств, которые
                        //подвергались модификации
    protected $data;
                        // актуальная информация из
                        //базы данных

    protected $errors = array();
                //любые ошибки ввода информации,
                //которые могли произойти


    public function __construct($arData) {
      $this->data = $arData;
    }

    function __get($propertyName) {
      if(!array_key_exists($propertyName, $this->propertyTable))
        throw new Exception("Неверное имя свойства\"$propertyName\"!");

      if(method_exists($this, 'get' . $propertyName)) {
        return call_user_func(array($this, 'get' . $propertyName));
      } else {
        return $this->data[$this->propertyTable[$propertyName]];
      }
    }

    function __set($propertyName, $value) {
      if(!array_key_exists($propertyName, $this->propertyTable))
        throw new Exception("Неверное имя свойства\"$propertyName\"!");

      if(method_exists($this, 'set' . $propertyName)) {
        return call_user_func(
                              array($this, 'set' . $propertyName),
                              $value
                             );
      } else {

        //Если значение свойства действительно изменено
        //и оно все еще не находится в массиве
        //$changedProperties, добавить его
        if($this->propertyTable[$propertyName] != $value &&
           !in_array($propertyName, $this->changedProperties)) {
          $this->changedProperties[] = $propertyName;
        }

        //Теперь устанавливаем новое значение
        $this->data[$this->propertyTable[$propertyName]] = $value;
```

```
        }
    }

    function validate() {

    }

  }
?>
```

Рассмотрим более подробно, что получилось в результате. Созданы четыре защищенные переменных-члена. Защищенные переменные-члены видны только производным классам; они не видны во внешней части приложения, где используются эти объекты.

Массив `$propertyTable` содержит перечень имен свойств и соответствующих им имен полей в базе данных. Зачастую имена полей в базе данных содержат префикс, указывающий на тип данных поля. Например, имя `entities.sname1` может соответствовать полю в таблице сущностей Entity, имеющему тип `string` и содержащему фамилию. Однако имя `sname1` нельзя назвать подходящим для свойства объекта, поэтому необходимо обеспечить механизм для конвертирования имен полей из базы данных в осмысленные имена свойств.

Массив `$changedProperties` содержит список имен свойств, которые подвергались изменению.

Массив `$data` является ассоциативным массивом имен полей базы данных и их значений. Он формируется в конструкторе, а структура данных возвращается непосредственно функцией `pgsql_fetch_assoc()`. Такой метод, как станет ясно из дальнейшего, существенно упрощает создание нужных объектов напрямую по запросу к базе данных.

Последняя переменная-член `$errors` будет содержать массив имен полей и сообщений о возможных ошибках при выполнении метода `validate()` (объявленного в интерфейсе `Validate`).

Класс объявлен как абстрактный (`abstract`) по двум причинам. Во-первых, класс `PropertyObject` сам по себе не очень полезен. Прежде чем использовать его методы, необходимо реализовать классы, расширяющие `PropertyObject`. Во-вторых, не реализован требуемый метод `validate()`. Поскольку этот метод помечен в классе `PropertyObject` как абстрактный, абстрактным необходимо объявить и сам класс. Тогда указанный метод придется реализовать во всех производных классах. Попытка использовать класс, расширяющий класс `PropertyObject`, не реализовав в нем метод `validate()`, приведет к ошибке во время выполнения программы.

В приведенном примере реализован весьма упрощенный конструктор. Он попросту принимает в качестве параметра ассоциативный массив, который наиболее вероятно будет получен в результате выполнения запроса к базе данных, и записывает его в защищенную переменную-член `$data`. При создании большинства подклассов `PropertyObject` этот конструктор потребуется перекрыть, сделав его более содержательным.

И наконец, обратите внимание на методы доступа `__get()` и `__set()`. Так как данные хранятся в элементе `$data`, необходимо иметь возможность преобразовывать имена свойств в фактические имена полей базы данных. Строка, содержащая код `$this->data[$this->propertyTable[propertyName]]` делает именно это.

Не беспокойтесь, если принципы использования массивов `$data` и `$propertyTable` вам еще не совсем ясны. Все станет ясно, как только мы рассмотрим конкретный пример.

Типы контактной информации

Разработав класс `PropertyObject`, можно начинать его использовать. Ниже приводится описание классов `Address`, `EmailAddress` и `PhoneNumber`.

В коде встречаются ссылки на класс `DataManager`, который является классом-оболочкой для работы с базой данных. Эта оболочка полностью инкапсулирует код взаимодействия с данными. Класс `DataManager` будет рассмотрен несколько позже.

Внесите следующий код (класс `Address`) в файл под названием `class.Address.php`.

```
<?php
  require_once('class.PropertyObject.php');

  class Address extends PropertyObject {

    function __construct($addressid) {
      $arData = DataManager::getAddressData($addressid);

      parent::__construct($arData);

      $this->propertyTable['addressid'] = 'addressid';
      $this->propertyTable['id'] = 'addressid';
      $this->propertyTable['entityid'] = 'entityid';
      $this->propertyTable['address1'] = 'saddress1';
      $this->propertyTable['address2'] = 'saddress2';
      $this->propertyTable['city'] = 'scity';
      $this->propertyTable['state'] = 'cstate';
      $this->propertyTable['zipcode'] = 'spostalcode';
      $this->propertyTable['type'] = 'stype';
    }

    function validate() {
      if(strlen($this->state) != 2) {
        $this->errors['state'] = 'Пожалуйста, введите правильно штат';
      }

      if(strlen($this->zipcode) != 5 &&
         strlen($this->zipcode) != 10) {
          $this->errors['zipcode'] = 'Пожалуйста, введите пяти- или
девятизначный ZIP-код';
      }


      if(!$this->address1) {
            $this->errors['address1'] = 'Адрес 1 является обязательным полем';
      }

      if(!$this->city) {
            $this->errors['city'] = 'Город является обязательным полем';
      }

      if(sizeof($this->errors)) {
        return false;
      } else {
        return true;
      }
    }

    function __toString() {
      return $this->address1 . ', ' .
             $this->address2 . ', ' .
             $this->city . ', ' .
             $this->state . ' ' . $this->zipcode;
```

```
    }
  }
?>
```

Так как бо́льшую часть работы выполняет класс `PropertyObject`, в классе `Address` остается реализовать только два метода (причем метод `__toString()` добавлен только для удобства). В конструкторе впервые используется массив `$propertyTable`. Список необходимых свойств этого класса определен на диаграмме UML, созданной на первом этапе разработки приложения (в начале этой главы). Учитывая перечень свойств объекта, можно принять несколько решений, касающихся структуры таблицы базы данных. Вообще говоря, на каждое свойство требуется по одному полю. Однако поскольку этот класс связан с классом `Entity`, надо также хранить ссылку на этот родительский класс. Для создания таблицы `Address` можно использовать следующий SQL-запрос.

```
CREATE TABLE "entityaddress" (
  "addressid" SERIAL PRIMARY KEY NOT NULL,
  "entityid" int,
  "saddress1" varchar(255),
  "saddress2" varchar(255),
  "scity" varchar(255),
  "cstate" char(2),
  "spostalcode" varchar(10),
  "stype" varchar(50),
  CONSTRAINT "fk_entityaddress_entityid"
    FOREIGN KEY ("entityid") REFERENCES "entity"("entityid")
);
```

В имени каждого поля создаваемой таблицы тип содержащихся в нем данных отражается при помощи односимвольного префикса. Это позволяет легко узнать, какой тип данных хранится в поле. Соглашения об именовании важны как с точки зрения проектирования базы данных, так и для кода приложения.

Массив `propertyTable` используется в классе `Address` для поддержки соответствия дружественных для разработчика названий свойств (таких, как `city`, `state` и `zipcode`) и менее дружественных имен полей базы данных (такие как `scity`, `sstate` и `spostalcode`). Заметим, что в массиве `propertyTable` одному полю базы данных может соответствовать несколько имен свойств. Эта таблица соответствий позволяет обращаться к первичному ключу адреса двумя способами: `$objAddress->addressed` либо `$objAddress->id`.

Преобладающая часть кода класса `Address` обеспечивает реализацию бизнес-логики и проверку корректности данных. Здесь практически нет лишнего кода. Единственной обязанностью этого класса является предоставление информации о себе и проверка корректности своих данных. Все остальные функции выполняют классы `DataManager` (который вскоре будет рассмотрен более детально) и `PropertyObject`.

Далее приведен код класса `Email`, который очень похож на класс `Address`. Внесите его в файл под названием `class.EmailAddress.php`.

```
<?php
  require_once('class.PropertyObject.php');

  class EmailAddress extends PropertyObject {

    function __construct($emailid) {
      $arData = DataManager::getEmailData($emailid);

      parent::__construct($arData);
```

```
      $this->propertyTable['emailid'] = 'emailid';
      $this->propertyTable['id'] = 'emailid';
      $this->propertyTable['entityid'] = 'entityid';
      $this->propertyTable['email'] = 'semail';
      $this->propertyTable['type'] = 'stype';
    }

    function validate() {
      if(!$this->email) {
       $this->errors['email'] ='Необходимо указать адрес электронной почты.';
      }

      if(sizeof($this->errors)) {
        return false;
      } else {
        return true;
      }
    }

    function __toString() {
      return $this->email;
    }
  }
?>
```

В этом файле содержится очень мало вспомогательного кода. В конструкторе просто делается выборка из базы данных и заполняется массив `propertyTable`. Все остальное — это проверка корректности данных. Свойства класса `Email` и структура соответствующей таблицы базы данных определяются приведенной выше диаграммой UML.

Таблица базы данных для таблицы `entitymail` создается следующим образом.

```
CREATE TABLE "entityemail" (
  "emailid" SERIAL PRIMARY KEY NOT NULL,
  "entityid" int,
  "semail" varchar(255),
  "stype" varchar(50),
  CONSTRAINT "fk_entityemail_entityid"
    FOREIGN KEY ("entityid") REFERENCES "entity"("entityid")
);
```

Класс `PhoneNumber` работает аналогично `Address` и `Email`. Приведем его код, который нужно сохранить в файле `class.PhoneNumber.php`.

```
<?php
  require_once('class.PropertyObject.php');

  class PhoneNumber extends PropertyObject {

    function __construct($phoneid) {
      $arData = DataManager::getPhoneNumberData($phoneid);

      parent::__construct($arData);

      $this->propertyTable['phoneid'] = 'phoneid';
      $this->propertyTable['id'] = 'phoneid';
      $this->propertyTable['entityid'] = 'entityid';
      $this->propertyTable['number'] = 'snumber';
      $this->propertyTable['extension'] = 'sextension';
      $this->propertyTable['type'] = 'stype';
    }
```

```
    function validate() {
      if(!$this->number) {
       $this->errors['number'] = 'Нужно указать номер телефона.';
      }

      if(sizeof($this->errors)) {
        return false;
      } else {
        return true;
      }
    }

    function __toString() {
      return $this->number .
             ($this->extension ? ' x' . $this->extension : '');
    }
  }
?>
```

Приведем SQL-запрос для создания таблицы `entityphone`.

```
CREATE TABLE "entityphone" (
  "phoneid" int SERIAL PRIMARY KEY NOT NULL,
  "entityid" int,
  "snumber" varchar(20),
  "sextension" varchar(20),
  "stype" varchar(50),
  CONSTRAINT "fk_entityemail_entityid"
    FOREIGN KEY ("entityid") REFERENCES "entity"("entityid")
);
```

Класс `DataManager`

Теперь можно обратиться к классу `DataManager`. В нем и в других примерах кода, связанного с работой базы данных в этой главе, используется база данных PostgreSQL, хотя подобный класс может успешно работать и с другими реляционными СУБД, в том числе Oracle.

Основной обязанностью класса `DataManager` является обеспечение доступа к данным. Сосредоточение всего кода для работы с базой данных в одном классе в дальнейшем существенно упростит изменение вида базы данных или параметров соединения. Все методы класса объявлены как статические, потому что класс не содержит ни одной переменной-члена. Обратите внимание на использование статической переменной в функции `getConnection()`. Это сделано для того, чтобы при запросе каждой страницы было открыто лишь одно соединение с базой данных. При установке соединения с базой данных используется много системных ресурсов, поэтому предотвращение бесполезных соединений помогает улучшить качество функционирования системы. Создайте файл под названием `class.DataManager.php` и введите в него следующий код.

```
<?php
require_once('class.Entity.php'); //Это понадобится позже
require_once('class.Individual.php');
require_once('class.Organization.php');

class DataManager
{
   private static function _getConnection() {
      static $hDB;
```

```
      if(isset($hDB)) {
         return $hDB;
      }

      $hDB = pg_connect("host=localhost port=5432 dbname=sample_db
                         user=phpuser password=phppass");
         or die("Не удается установить соединение с базой данных!");
      return $hDB;
   }

   public static function getAddressData($addressID) {
      $sql = "SELECT * FROM \"entityaddress\" WHERE \"addressid\" = $addressID";
      $res = pg_query(DataManager::_getConnection(), $sql);
      if(! ($res && pg_num_rows($res))) {
         die("Невозможно получить данные адреса для $addressID");
      }
      return pg_fetch_assoc($res);
   }

   public static function getEmailData($emailID) {
      $sql = "SELECT * FROM \"entityemail\" WHERE \"emailid\" = $emailID";
      $res = pg_query(DataManager::_getConnection(), $sql);

      if(! ($res && pg_num_rows($res))) {
         die("Невозможно получить данные электронного адреса для $emailID");
      }

      return pg_fetch_assoc($res);
   }

   public static function getPhoneNumberData($phoneID) {

      $sql = "SELECT * FROM \"entityphone\" WHERE \"phoneid\" = $phoneID";
      $res = pg_query(DataManager::_getConnection(), $sql);
      if(! ($res && pg_num_rows($res))) {
         die("Невозможно получить номер телефона для $phoneID");
      }

      return pg_fetch_assoc($res);
   }
}
?>
```

Класс `DataManager` предоставляет структуры данных, используемые для заполнения элемента `$data` подкласса `PropertyObject`. Каждая функция возвращает данные одного из типов. Позже в этот класс придется добавить несколько новых функций.

Все методы этого класса объявлены как статические. При использовании статических методов все переменные-члены тоже должны быть статическими. Для использования методов статических классов не требуется создавать экземпляров (выполнять инстанцирование). Это имеет смысл в нескольких случаях. Рассмотрим класс `Math` (математика), который предоставляет методы `squareRoot()`(извлечь квадратный корень), `power()` (возвести в степень) и `cosine()` (вычислить косинус) и имеет свойства, включающие математические константы `e` и `pi`. Каждый экземпляр этого класса представляет собой одну и ту же математику. Квадратный корень из 2 не меняется, 4 в кубе всегда будет 64 и две константы, ясное дело, являются постоянными. Нет нужды создавать отдельный экземпляр этого класса, так как его состояние и свойства никогда не меняются. При такой реализации класса `Math` его функции могут быть объявлены статическими.

Класс DataManager во многом аналогичен. Все его функции самодостаточны. Для взаимодействия с функциями не используются нестатические переменные-члены. В этом классе отсутствуют свойства. В результате методы класса можно вызывать, используя оператор вызова статического метода ::. Поскольку все методы являются статическими, вам никогда не придется инициализировать объект с помощью оператора $obj=new DataManager(), а затем вызывать методы, используя оператор $obj->getEmail(). Вместо этого можно использовать простой синтаксис DataManager::getEmail().

Классы Entity, Individual и Organization

Создав все вспомогательные классы, можно перейти к основе приложения: классу Entity и его подклассам.

Не забывайте обновлять диаграммы UML по ходу изменения иерархии объектов, отслеживая таким образом ход разработки. К настоящему моменту создан класс PropertyObject и все его наследники, а также класс DataManager, который не наследуется, а попросту представляет набор функций для работы с данными. Класс PropertyObject реализует абстрактный интерфейс Validator. На рис. 3.7 изображена обновленная диаграмма.

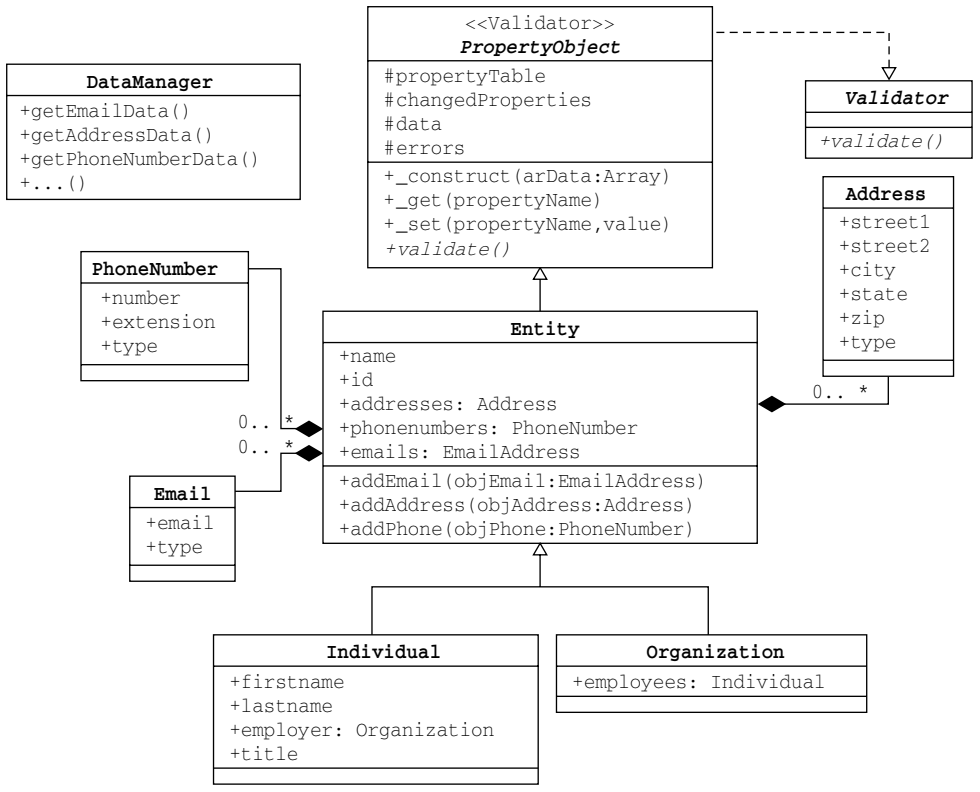

Рис. 3.7.

Теперь можно приступать к разработке классов `Entity`, `Individual` и `Organization`. Приведенный ниже код описывает полностью реализованный класс `Entity`. Его нужно поместить в файл `class.Entity.php`.

```
<?php

  require_once('class.PropertyObject.php');
  require_once('class.PhoneNumber.php');
  require_once('class.Address.php');
  require_once('class.EmailAddress.php');

  abstract class Entity extends PropertyObject {

    private $_emails;
    private $_addresses;
    private $_phonenumbers;

    public function __construct($entityID) {
      $arData = DataManager::getEntityData($entityID);

      parent::__construct($arData);

      $this->propertyTable['entityid'] = 'entityid';
      $this->propertyTable['id'] = 'entityid';
      $this->propertyTable['name1'] = 'sname1';
      $this->propertyTable['name2'] = 'sname2';
      $this->propertyTable['type'] = 'ctype';

      $this->_emails = DataManager::getEmailObjectsForEntity($entityID);
      $this->_addresses = DataManager::getAddressObjectsForEntity($entityID);
      $this->_phonenumbers =
                  DataManager::getPhoneNumberObjectsForEntity($entityID);

    }

    function setID($val) {
      throw new Exception('Вы не можете изменять значение поля ID!');
    }

    function setEntityID($val) {
      $this->setID($val);
    }

    function phonenumbers($index) {
      if(!isset($this->_phonenumbers[$index])) {
        throw new Exception('Задан некорректный номер телефона!');
      } else {
        return $this->_phonenumbers[$index];
      }
    }

    function getNumberOfPhoneNumbers() {
      return sizeof($this->_phonenumbers);
    }

    function addPhoneNumber(PhoneNumber $phone) {
      $this->_phonenumbers[] = $phone;
    }

    function addresses($index) {
      if(!isset($this->_addresses[$index])) {
        throw new Exception('Задан некорректный адрес!');
      } else {
```

```
        return $this->_addresses[$index];
      }
    }

    function getNumberOfAddresses() {
      return sizeof($this->_addresses);
    }

    function addAddress(Address $address) {
      $this->_addresses[] = $address;
    }

    function emails($index) {
      if(!isset($this->_emails[$index])) {
        throw new Exception('Задан некорректный адрес электронной почты!');
      } else {
         return $this->_emails[$index];
      }
    }

    function getNumberOfEmails() {
      return sizeof($this->_emails);
    }

    function addEmail(Email $email) {
      $this->_emails[] = $email;
    }

    public function validate() {
      //Добавьте процедуры проверки
    }

  }
?>
```

Перемещая всю функциональность методов доступа в родительский класс `PropertyObject`, вы упрощаете класс `Entity` (сущность) и этим обеспечиваете сужение его обязанностей до реализации самой сущности.

Класс `Entity` объявлен как абстрактный, так как он сам по себе бесполезен. Все сущности относятся либо к классу `Individual`, либо к `Organization`. Объекты класса `Entity` инстанцировать не придется. Если класс объявлен абстрактным, для него нельзя создать экземпляров.

Для создания таблицы `entities` в базе данных PostgreSQL выполните следующий код.

```
CREATE TABLE "entities" (
  "entityid" SERIAL PRIMARY KEY NOT NULL,
  "name1" varchar(100) NOT NULL,
  "name2" varchar(100) NOT NULL,
  "type" char(1) NOT NULL
);
```

В класс `DataManager` необходимо добавить несколько новых функций: `getEntityData()` и `get[x]ObjectsForEntity`. Функция `geEntityData()` возвращает данные, необходимые для инициализации сущности, подобно аналогичным функциям для отдельных типов контактной информации. Далее следует код этой новой функции в файле `class.DataManager.php`.

```
    //верхняя часть файла опущена для краткости
   ...
      die("Не удалось получить информацию о телефонном номере для $phoneID");
```

```
    }

    return pg_getch_assoc($res);
  }
```

```
  public static function getEntityData($entityID) {
    $sql = "SELECT * FROM \"entities\" WHERE\"entityid\" = $entityID";
    $res = pg_query(DataManager::_getConnection(), $sql);
    if(! ($res &&pg_num_rows($res))) {
      die("Не удалось получить сущность $entityID");
    }
    return pg_fetch_assoc($res);
  }
```

```
?>
```

Для добавления функции `get[x]ObjectsForEntity` вставьте следующий код в конец файла `class.DataManager.php` сразу после функции `getEntityData`.

```
public static function getAddressObjectsForEntity($entityID) {
  $sql = "SELECT \"addressid\" FROM \"entityaddress\" WHERE \"entityid\" =
          $entityID";
  $res = pg_query(DataManager::_getConnection(), $sql);

  if(!$res) {
    die("Невозможно получить данные для сущности $entityID");
  }

  if(pg_num_rows($res)) {
    $objs = array();
    while($rec = pg_fetch_assoc($res)) {
      $objs[] = new Address($rec['addressid']);
    }
    return $objs;
  } else {
    return array();
  }
}

public static function getEmailObjectsForEntity($entityID) {

$sql = "SELECT \"emailid\" FROM \"entityemail\"
        WHERE \"entityid\" = $entityID";
  $res = pg_query(DataManager::_getConnection(), $sql);
  if(!$res) {
  die("Невозможно получить данные электронной почты для сущности $entityID");
  }

  if(pg_num_rows($res)) {
    $objs = array();
    while($rec = pg_fetch_assoc($res)) {
      $objs[] = new EmailAddress($rec['emailid']);
    }
    return $objs;
  } else {
    return array();
  }
}

public static function getPhoneNumberObjectsForEntity($entityID) {
```

```
$sql = "SELECT \"phoneid\" FROM \"entityphone\"
        WHERE \"entityid\" = $entityID";
  $res = pg_query(DataManager::_getConnection(), $sql);

  if(!$res) {
    die("Невозможно получить номер телефона для сущности $entityID");
  }

  if(pg_num_rows($res)) {
    $objs = array();
    while($rec = pg_fetch_assoc($res)) {
      $objs[] = new PhoneNumber($rec['phoneid']);
    }
    return $objs;
  } else {
    return array();
  }
}
```

В качестве параметра этим функциям передается значение идентификатора ID сущности. Они делают запрос к базе данных и определяют, существует ли указанный адрес электронной почты, почтовый адрес либо номер телефона для данной сущности. Если да, то создается массив объектов `EmailAddress`, `Address` либо `PhoneNumber`. Затем этот массив передается обратно объекту `Entity`, где он хранится в соответствующей закрытой переменной-члене.

Вся основная работа по созданию класса `Entity` выполнена, осталось выполнить только простые действия по реализации классов `Individual` и `Organization`. Создайте файл `class.Individual.php` и введите следующий код.

```
<?php
  require_once('class.Entity.php');
  require_once('class.Organization.php');

  class Individual extends Entity {

    public function __construct($userID) {
      parent::__construct($userID);

      $this->propertyTable['firstname'] = 'name1';
      $this->propertyTable['lastname'] = 'name2';

    }

    public function __toString() {
      return $this->firstname . ' ' . $this->lastname;
    }

    public function getEmployer() {
      return DataManager::getEmployer($this->id);
    }

    public function validate() {
      parent::validate();

      //проверка данных для индивидуума

    }

  }
?>
```

Коротко и ясно. Процесс создания класса `Individual` существенно упрощается благодаря наследованию. В этом классе задаются несколько новых свойств, упрощающих доступ к имени и фамилии контактного лица, не входящих в перечень свойств, определенных в классе `Entity`. Тут также определен новый метод `getEmployer()`, для работы которого потребуется новая функция в классе `DataManager`. Мы вернемся к этой функции после рассмотрения класса `Organization`, код которого приведен ниже. Создайте файл `class.Organization.php` и введите в него следующий код.

```
<?php
  require_once('class.Entity.php');
  require_once('class.Individual.php');

  class Organization extends Entity {

    public function __construct($userID) {
      parent::__construct($userID);

      $this->propertyTable['name'] = 'name1';

    }

    public function __toString() {
      return $this->name;
    }

    public function getEmployees() {
      return DataManager::getEmployees($this->id);
    }

    public function validate() {
      parent::validate();
      //проверка корректности данных для организации
    }

  }
?>
```

Благодаря наследованию структура этого класса довольно проста. В нем объявлено свойство `name`, упрощающее задание единственного имени для организации (свойство `sname2` для организации не используется).

Чтобы добавить функции `getEmployer()` и `getEmployee()` в класс `DataManager`, поместите следующий код в конец `class.DataManager.php`.

```
public static function getEmployer($individualID) {
  $sql = "SELECT \"organizationid\" FROM \"entityemployee\" " .
         "WHERE \"individualid\" = $individualID";
  $res = pg_query(DataManager::_getConnection(), $sql);
  if(! ($res && pgsql_num_rows($res))) {
    die("Невозможно получить информацию об организации для сотрудника $individualID");
  }

  $row = pgsql_fetch_assoc($res);

  if($row) {
    return new Organization($row['organizationid']);
  } else {
      return null;
    }
  }

  public static function getEmployees($orgID) {
```

```
    $sql = "SELECT \"individualid\" FROM \"entityemployee\" " .
           "WHERE \"organizationid\" = $orgID";
    $res = pgsql_query(DataManager::_getConnection(), $sql);
    if(! $res && pgsql_num_rows($res)) {
      die("Невозможно получить информацию о сотруднике для организации $orgID");
    }

    if(pgsql_num_rows($res)) {
      $objs = array();
      while($row = pgsql_fetch_assoc($res)) {
        $objs[] = new Individual($row['individualid']);
      }
      return $objs;
    } else {
      return array();
    }
  }
```

Эти две функции предполагают наличие таблицы `entityemployee`, запрос для создания которой приведен ниже. Эта таблица связывает сотрудников с соответствующими организациями. К примеру, сотрудники некоторой компании могут иметь различные индивидуальные идентификаторы, но один общий идентификатор организации.

```
CREATE TABLE "entityemployee" (
  "individualid" int NOT NULL,
  "organizationid" int NOT NULL,
  CONSTRAINT "fk_entityemployee_individualid"
    FOREIGN KEY ("individualid") REFERENCES "entity"(entityid),
  CONSTRAINT "fk_entityemployee_organizationid"
    FOREIGN KEY ("organizationid") REFERENCES "entity"("entityid")
);
```

Последняя функция, необходимая для функционирования всей системы в целом, — это метод класса `DataManager`, предназначенный для отображения всех сущностей в базе данных. Эта функция называется `getAllEntitiesAsObjects()` и завершает список действий, выполняемых над объектами.

```
public static function getAllEntitiesAsObjects() {
  $sql = "SELECT \"entityid\", \"type\" FROM \"entities\"";
  $res = pgsql_query(DataManager::_getConnection(), $sql);

  if(!$res) {
    die("Невозможно получить все сущности");
  }

  if(pgsql_num_rows($res)) {
    $objs = array();
    while($row = pgsql_fetch_assoc($res)) {
      if($row['type'] == 'I') {
        $objs[] = new Individual($row['entityid']);
      } elseif ($row['type'] == 'O') {
        $objs[] = new Organization($row['entityid']);
      } else {
        die("Неизвестный тип сущности {$row['type']}");
    }
    return $objs;
  } else {
    return array();
  }
}
```

Класс DataManager позволяет обрабатывать всю контактную информацию в системе. Он проверяет значение поля ctype таблицы entity и определяет, к какому типу относится контакт — Individual или Organization, — затем инициализирует объект соответствующего типа и добавляет его в возвращаемый массив.

Использование системы

Теперь вы можете оценить реальную мощь объектно-ориентированного подхода. Следующий код будет подробно отображать все записи о контактах из базы данных. Этот код нужно поместить в файл test.php.

```
<?php
  require_once('class.DataManager.php');

  function println($data) {
    print $data . "<br>\n";
  }

  $arContacts = DataManager::getAllEntitiesAsObjects();
  foreach($arContacts as $objEntity) {

    if(get_class($objEntity) == 'individual') {
      print "<h1>Частное лицо - {$objEntity->__toString()}</h1>";
    } else {
      print "<h1>Организация - {$objEntity->__toString()}</h1>";
    }

    if($objEntity->getNumberOfEmails()) {
      //У нас есть электронная почта! Выведите заголовок
      print "<h2>Электронная почта</h2>";

      for($x=0; $x < $objEntity->getNumberOfEmails(); $x++) {
        println($objEntity->emails($x)->__toString());
      }
    }

    if($objEntity->getNumberOfAddresses()) {
      //У нас есть адреса!
      print "<h2>Адреса</h2>";

      for($x=0; $x < $objEntity->getNumberOfAddresses(); $x++) {
        println($objEntity->addresses($x)->__toString());
      }
    }

    if($objEntity->getNumberOfPhoneNumbers()) {
      //У нас есть телефоны!
      print "<h2>Телефоны</h2>";

      for($x=0; $x < $objEntity->getNumberOfPhoneNumbers(); $x++) {
        println($objEntity->phonenumbers($x)->__toString());
      }
    }

    print "<hr>\n";
  }
?>
```

Читателю в качестве упражнения предлагается самостоятельно ввести данные в таблицы; это поможет составить представление о том, как работает созданное приложение. Пример результатов приложения показан на рис 3.8. Для его получения запустите в браузере сценарий test.php.

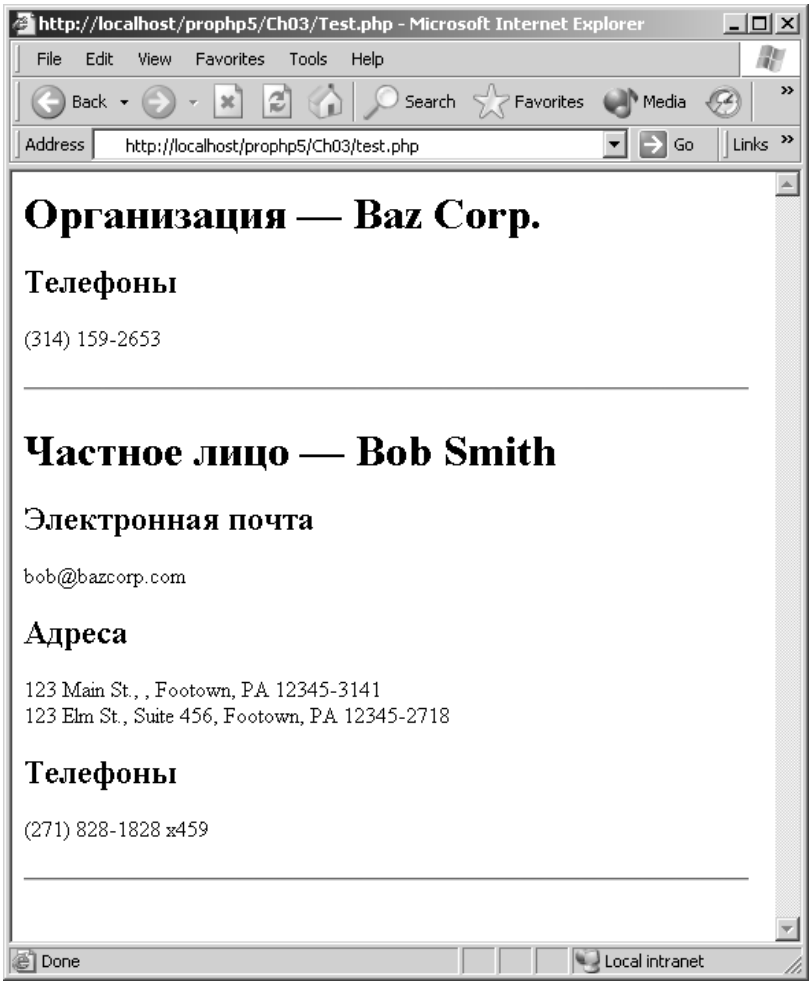

Рис. 3.8.

В 36 строках кода можно отобразить почти всю информацию о сущностях системы. Работодатель и работник здесь не показаны. Отображение информации о них снова оставим в качестве упражнения. Строка вызова метода `get_class()` в `test.php` может натолкнуть на мысль о том, с каким классом приходится иметь дело в данный момент, и определить, какую функцию вызывать: метод `getEmployer()` класса `Individual` либо метод `getEmployer()` класса `Organization`.

Резюме

Язык UML — это важнейший инструмент для моделирования сложных (и не очень) приложений. Разработанные должным образом диаграммы позволяют документировать серьезные системы гораздо проще и яснее, чем с помощью обычного текста. Использование диаграмм классов в процессе разработки программного обеспечения позволяет существенно облегчить проектирование классов и таблиц базы данных.

Использование преимуществ объектно-ориентированного подхода в PHP 5 помогает ускорить разработку приложения и создать библиотеку кода, которую легко использовать и расширять, а также сокращает общий объем кода, требуемый для реализации требований приложения.

Разделяя архитектуру приложения на уровень бизнес-логики, включающий класс `Individual`, и уровень работы с данными, представленный классом `DataManager`, вы упрощаете изменение источников данных, структуры таблиц или запросов без лишнего влияния на остальные части приложения. Объекты, отвечающие за реализацию бизнес-логики, не должны вмешиваться в механизм доступа к данным. Это может привести к путанице в алгоритме реализации и сделать его сложным для понимания.