Снижение риска ошибок

Инкапсуляция также помогает программистам обеспечить правильность работы их кода. В качестве примера можно привести еще одну известную историю: на президентских выборах 2000 года Альберт Гор получил –16 022 голоса на электронной машине для голосования в округе Волузия (штат Флорида). Переменная-счетчик неправильно инициализировалась в программном обеспечении машины! Чтобы понять суть проблемы, рассмотрим класс `Counter` (листинг 3.3.2), реализующий простой счетчик в соответствии со следующим API:

`public class Counter`	
`Counter(String id, int max)`	создает счетчик и инициализирует его 0
`void increment()`	увеличивает счетчик, пока его значение не достигнет max
`int value()`	возвращает значение счетчика
`String toString()`	строковое представление

API типа данных счетчика (листинг 3.3.2)

Эта абстракция может использоваться в разных контекстах, в том числе и на электронных машинах для голосования. Она инкапсулирует одно целое число и гарантирует, что с этим числом может выполняться только одна операция — *увеличение на 1*. Следовательно, счетчик ни при каких условиях не станет отрицательным. Целью абстракции данных является ограничение операций с данными. Кроме того, она обеспечивает *изоляцию* операций с данными. Например, можно добавить новую реализацию с поддержкой ведения журнала, чтобы метод `increment()` сохранял временную метку каждого голоса, или другую информацию, которая могла бы использоваться для проверки целостности данных. Но без модификатора `private` где-то в коде машины для голосования может присутствовать команда вида:

```
Counter c = new Counter("Volusia", VOTERS_IN_VOLUSIA_COUNTY);
c.count = -16022;
```

С модификатором `private` этот код не будет компилироваться; без него счетчик голосов может быть инициализирован отрицательным значением. Возможно, инкапсуляция не обеспечивает полного решения всех проблем безопасности голосования, но она станет хорошей отправной точкой.

Ясность кода

Точное описание типа данных также является признаком качественного проектирования, потому что клиентский код более ясно выражает выполняемые вычисления. Многочисленные примеры такого кода встречались в разделах 3.1 и 3.2; кроме того, проблема упоминалась в обсуждении `Histogram` (листинг 3.2.3). Клиенты этой программы более понятны, чем код, не использующий ее, потому что вызов метода экземпляра `addDataPoint()` четко говорит, что требуется клиенту. Код, написанный на основе правильно выбранных абстракций, практически является

Листинг 3.3.2. Счетчик

```
public class Counter
{
   private final String name;
   private final int maxCount;
   private int count;

   public Counter(String id, int max)
   {  name = id; maxCount = max;  }

   public void increment()
   {  if (count < maxCount) count++;  }

   public int value()
   {  return count;  }

   public String toString()
   {  return name + ": " + count;  }

   public static void main(String[] args)
   {
     int n = Integer.parseInt(args[0]);
     int trials = Integer.parseInt(args[1]);
     Counter[] hits = new Counter[n];
     for (int i = 0; i < n; i++)
        hits[i] = new Counter(i + „", trials);
     for (int t = 0; t < trials; t++)
        hits[StdRandom.uniform(n)].increment();
     for (int i = 0; i < n; i++)
        StdOut.println(hits[i]);
   }
}
```

`name`	имя счетчика
`maxCount`	максимальное значение
`count`	текущее значение

Класс инкапсулирует простой целочисленный счетчик: он присваивает ему строковое имя и инициализирует 0 (инициализация по умолчанию в языке Java). Счетчик увеличивается каждый раз, когда клиент вызывает increment(), выдает текущее значение при вызове value() и создает строку с именем и текущим значением при вызове toString().

```
% java Counter 6 600000
0: 100684
1: 99258
2: 100119
3: 100054
4: 99844
5: 100037
```

самодокументируемым. Некоторые сторонники объектно-ориентированного программирования могут возразить, что сам код `Histogram` станет более понятным, если в нем будет использоваться класс `Counter` (см. упражнение 3.3.3), но, пожалуй, это утверждение спорно.

Преимущества инкапсуляции неоднократно подчеркивались в книге. Сейчас мы снова кратко повторим их в контексте проектирования типов данных. Инкапсуляция делает возможным модульное программирование, позволяя нам:

- независимо разрабатывать код клиента и код реализации;
- совершенствовать реализации без вреда для клиента;
- поддерживать еще не написанные программы (клиент пишется для API).

Инкапсуляция также изолирует операции с типами данных, благодаря чему:

- вы можете добавлять в реализации проверки целостности и другие средства отладки;
- клиентский код становится более ясным.

Правильно реализованный тип данных (с инкапсуляцией) расширяет язык Java и может использоваться любой клиентской программой.

Неизменность

Как упоминалось в конце раздела 3.1, объект некоторого типа данных называется *неизменяемым*, если его значение не может быть изменено после создания. Неизменяемым типом данных называется тип, все объекты которого являются неизменяемыми. Напротив, изменяемым типом данных называется тип, значения объектов которого могут изменяться. Из типов данных, представленных в этой главе, `String`, `Charge`, `Color` и `Complex` являются неизменяемыми, а `Turtle`, `Picture`, `Histogram`, `StockAccount` и `Counter` — изменяемыми. Стоит ли делать тип данных неизменяемым? Это важное решение из области проектирования, которое зависит от конкретной ситуации.

Неизменяемые типы

Многие типы данных предназначены для инкапсуляции значений, которые не изменяются, и своим поведением напоминают примитивные типы. Например, для программиста, реализующего клиента `Complex`, разумно ожидать, что команда `z = z0` для двух переменных `Complex` будет работать так же, как и для переменных типа `double` или `int`. Но если бы объекты `Complex` были изменяемыми, а *после* присваивания `z = z0` значение `z` вдруг изменилось, то оказалось бы, что значение `z0` *тоже* изменилось (обе переменные являются ссылками на один и тот же объект!). Этот неожиданный результат, известный как *совмещение имен*, нередко преподносит неприятные сюрпризы новичкам в объектно-ориентированном программировании. Очень важная причина для реализации неизменяемых типов заключается в том, что неизменяемые типы можно использовать в командах присваивания (или в аргументах и возвращаемых значениях методов), не беспокоясь о непредвиденном изменении их значений.

неизменяемые	изменяемые
String	Turtle
Charge	Picture
Color	Histogram
Complex	StockAccount
Vector	Counter
	Массивы Java

Изменяемые типы

Для многих типов данных главной целью абстракции является инкапсуляция значений при их изменении. Типичным примером служит программа `Turtle` (листинг 3.2.4). Мы используем `Turtle` для того, чтобы избавить клиента от обязанности следить за изменениями значений. Аналогичным образом мы ожидаем, что у типов данных `Picture`, `Histogram`, `StockAccount` и `Counter`, а также у массивов Java значения будут изменяться. Передавая `Turtle` методу в качестве аргумента (как в программе `Koch`), мы ожидаем, что значение объекта `Turtle` будет изменяться.

Массивы и строки

Различие между изменяемыми и неизменяемыми объектами уже встречалось вам при написании клиентского кода, когда вы использовали массивы Java (изменяемые) и тип данных Java `String` (неизменяемый). Когда вы передаете `String` методу, вам не нужно беспокоиться о том, что метод изменит исходную цепочку символов `String`; с другой стороны, при передаче методу массива метод может изменять значения элементов в массиве. Тип данных `String` является неизменяемым, потому что обычно изменение строковых значений *нежелательно*, а массивы Java являются изменяемыми, потому что значения элементов массива обычно *должны* изменяться. При этом в некоторых ситуациях желательно иметь изменяемые аналоги строк (для этого предназначен тип данных Java `StringBuilder`) и неизменяемые аналоги массивов (для этого предназначен тип данных `Vector`, который будет рассмотрен далее в этом разделе).

Преимущества неизменяемости

Как правило, неизменяемые типы проще в использовании и создают меньше риска для ошибок, потому что область видимости кода, способного влиять на их значения, намного меньше, чем для изменяемых типов. Код, использующий неизменяемые типы, проще отлаживается — разработчику проще следить за тем, что переменные клиентского кода, использующие эти типы, сохраняют корректные значения. При использовании изменяемых типов всегда приходится думать о том, где и как могут изменяться их значения.