Практика
функционального программирования

Выпуск 1
Июль 2009

ISSN 2075-8456

Изменяемое состояние: опасности и борьба с ними

Евгений Кирпичёв
jkff@fprog.ru

Аннотация

В этой статье рассматриваются опасности использования изменяемого состояния в программах, преимущества использования неизменяемых структур и способы минимизации нежелательных эффектов от изменяемого состояния в тех случаях, когда оно все-таки необходимо.

3.1. Введение

Одно из ключевых отличий многих функциональных языков от объектно-ориентированных и процедурных — в поощрении использования неизменяемых данных; некоторые языки, в частности Haskell, даже не содержат в синтаксисе оператора присваивания! Апологеты функционального программирования объясняют это решение, в частности, тем, что отказ от изменяемых данных резко повышает корректность программ и делает их значительно более удобными для анализа с помощью формальных методов. Это действительно так, и в данной статье мы в этом убедимся. Однако полный отказ от изменяемых данных зачастую не оправдан по следующим причинам:

1) Некоторые техники программирования, применяющиеся в функциональных языках без присваиваний (к примеру, ленивые вычисления), применимы в более традиционных языках, таких как Java или C++, лишь с огромным трудом.

2) Для некоторых алгоритмов и структур данных не известно или не существует столь же эффективных аналогов без использования присваиваний (к примеру, для хэш-таблиц и систем непересекающихся множеств).

3) Многие предметные области по своей сути содержат изменяемые объекты (например, банковские счета; элементы систем в задачах имитационного моделирования, и т. п.), и переформулировка задачи на язык неизменяемых объектов может «извратить» задачу.

В данной статье мы поговорим о том, как пользоваться изменяемыми данными, не жертвуя простотой и корректностью кода.

3.2. Опасности изменяемого состояния

Перед тем, как перейти к техникам нейтрализации опасностей изменяемых данных, перечислим сами эти опасности.

3.2.1. Неявные изменения

Необходимое условие корректности программы — целостность ее внутреннего состояния, выполнение некоторых инвариантов (к примеру, совпадение поля `size` у объекта типа «связный список» с реальным числом элементов в этом списке). Код пишется так, чтобы в моменты, когда состояние программы наблюдаемо, инварианты не нарушались: каждая отдельная процедура начинает работать в предположении, что все инварианты программы выполняются и гарантирует, что после ее завершения инварианты выполняются по-прежнему.

Инварианты могут охватывать сразу несколько объектов: к примеру, в задаче представления ненаправленных графов логично требовать инварианта «если узел A связан ребром с узлом B, то и узел B связан ребром с узлом A».

Сохранение такого инварианта представляет собой непростую задачу: всякая процедура, меняющая один из составляющих его объектов, обязана знать не только о существовании всех остальных составляющих этого инварианта, но и обо всех составляющих *всех* инвариантов, зависящих от этого объекта! В противном случае, процедура может, сама того не ведая, нарушить инвариант.

Добиться такого знания порой чрезвычайно сложно; еще сложнее сделать это без нарушения модульности. Поэтому программисты стремятся делать инварианты охватывающими как можно меньше объектов и зависящими от как можно меньшего числа их изменяемых свойств.

Рассмотрим классический пример, иллюстрирующий данную проблему.

Пример: Обходчик интернета. Предположим, что мы разрабатываем программу — обходчик интернета. Она ходит по графу некоторого подмножества интернета и собирает данные со встречаемых страничек. В графе интернета узлами являются страницы, ребрами — ссылки с одних страниц на другие. В результате работы программа записывает в базу ссылки на некоторые из найденных страничек вместе с определенной дополнительной информацией.

Структура классов выглядит примерно так:

```
public class Address {
    private String url;
    public String getUrl() {
        return url;
    }
    public void setUrl(String u) {
        this.url = u;
    }
    int hashCode() {
        return url.hashCode();
    }
    boolean equals(Address other) {
        return url.equals(other.url);
    }
}

public class Node {
    Address address;
    List<Node> inLinks, outLinks;
}

public class Graph {
    Map<Address,Node> addr2node = new HashMap<Address,Node>();
}
```

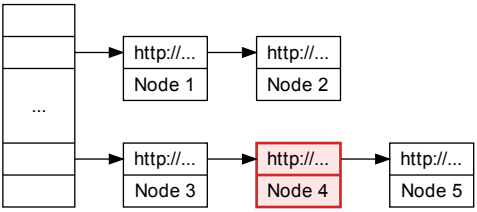


Рис. 3.1. Организация объекта класса HashMap

Во время разработки программы в один прекрасный момент выясняется, что доступ к некоторым страничкам приводит к перенаправлению (*redirect*) на другой адрес. Если в базе оказывается записан исходный адрес, то когда другая программа будет считывать адреса из базы и загружать соответствующие странички, она потратит лишнее время на перенаправление — поэтому лучше записать в базу новый адрес, полученный после перенаправления. В код добавляется следующее небольшое изменение:

```
Page download(Address address) {
    ...
    if(response.isRedirect()) {
        address.setUrl(response.getRedirectedUrl());
    }
    ...
}
```

И тут про некоторые адреса, определенно обязанные содержаться в графе, `addr2node.containsKey(address)` вдруг начинает отвечать **`False`**! Опытные читатели, скорее всего, заметят здесь проблему; однако, будучи встречена впервые, она может потребовать для решения пары часов отладки и сомнений в собственном душевном здоровье и качестве стандартной библиотеки. На деле проблема очень проста: метод `download` модифицировал объект `address`, но не учел, что его состояние является частью инварианта объекта `addr2node`.

Вспомним, как устроен класс `HashMap` в языке Java (рис. 3.1). Он реализует хэш-таблицу с закрытой адресацией: каждому хэш-коду (по модулю выделенной длины хэш-таблицы) соответствует «корзина» — список элементов, чей ключ обладает таким хэш-кодом.

Отмеченный на рисунке элемент соответствует адресу, измененному в методе `download`. В результате изменения поменялся и его хэш-код, однако элемент остался в корзине, соответствующей старому хэш-коду! В результате методом `download` оказывается нарушен инвариант класса `HashMap` — «Хэш-код всех ключей в одной корзине по модулю длины таблицы равен номеру этой корзины».

Теперь, к примеру, при попытке проверить наличие нового адреса в графе, поиск будет производиться в корзине, соответствующей хэш-коду нового адреса — конечно же, ключа там не окажется, т.к. он находится в другой корзине. При попытке прове-

рить наличие в графе старого адреса, поиск будет производиться в корзине, соответствующей старому адресу — однако самого адреса там также не окажется. Таким образом, после выполнения метода download в графе будут «отсутствовать» и старый, и новый адреса.

Как видно, объекты класса `Address` можно изменять только если известно, что они не содержатся ни в каком контейнере!

Единственное, по-видимому, решение данной проблемы — никогда не использовать изменяемые поля в качестве ключей контейнеров, в частности, в методах `equals`, `hashCode`, `compareTo`. Эта проблема настолько часта и опасна, что некоторые среды разработки генерируют предупреждение, если в одном из этих методов используется изменяемое поле.

В рассматриваемой задаче компромиссное решение таково: иметь в классе `Address` два поля: одно, неизменяемое, соответствует изначальному адресу странички, без учета перенаправлений, и именно им индексируются узлы в графе; второе, изменяемое, соответствует конечному адресу с учетом перенаправлений, и именно оно записывается в базу, но не используется в качестве ключа.

3.2.2. Кэширование и разделение

Следующая опасность изменяемых данных заключается в том , что их наличие существенно усложняет корректное кэширование. Рассмотрим один из классических примеров этой проблемы, широко известный в сообществе Java-программистов.

Пример: Геометрические классы GUI-библиотеки AWT. AWT содержит классы `Point`, `Dimension`, `Rectangle`, обозначающие соответственно: точку на плоскости, размеры прямоугольника и прямоугольник. Методы-аксессоры у классов окон возвращают объекты этих классов на запросы о положении и размере окна. Все три класса изменяемы:

```
class Point {
    int x, y;
}
class Dimension {
    int width, height;
}
class Rectangle {
    int x, y, width, height;
}
```

Как должен выглядеть метод получения размеров окна, возвращающий Dimension?

```
class Component {
    private Dimension size;
    ...
    Dimension getSize() {
```

```
        return size;
    }
    ...
}
```

Этот код, конечно же, неверен! Клиент *может* изменить возвращенный объект, тем самым нарушив инвариант класса `Component` — «Всякое изменение размеров объекта типа `Component` оповещает всех клиентов, подписавшихся на это изменение (с помощью метода `addComponentListener`)». Заметим, что клиент может и не иметь никакого злого умысла при изменении такого возвращенного объекта — например, его может интересовать центр окна, который он станет вычислять таким образом:

```
Point center = w.getLocation();
center.x += w.getSize().width/2;
center.y += w.getSize().height/2;
```

Такая реализация `getSize` недопустима. Правильная реализация обязана возвращать объект, изменение которого не может повлиять на окно.

```
Dimension getSize() {
    return new Dimension(size.width, size.height);
}
```

Однако такая реализация обладает другим недостатком — низкой производительностью: всякий раз при вызове `getSize` создается новый объект. В ситуации, к примеру, вызова менеджера размещения окон для сложного интерфейса методы `getSize`, `getLocation`, `getBounds` могут вызываться десятки тысяч раз, и издержки на создание объекта становятся совсем не безобидными.

Точно такие же проблемы возникают при возвращении массивов методов.[1]

Рассмотрим еще один пример.

Пример: Корзина в интернет-магазине. В программе, реализующей интернет-магазин, есть класс «корзина». Общая стоимость продуктов зависит от стоимости каждого продукта и скидки, вычисляющейся по некоторым сложным правилам, зависящим от самих продуктов, от покупателя и т. п. Правила настолько сложные, что всякий раз вычислять стоимость корзины заново — неэффективно, поэтому она кэшируется и сбрасывается при изменении набора продуктов.

```
class Cart {
    private Customer customer;
    private List<Product> products;
    private int totalPrice = -1;
```

[1]При возвращении списков и других коллекций проблемы несколько меньше, поскольку они допускают *инкапсуляцию* изменений, давая возможность переопределить изменяющие методы (`add`, `set`, …) и, к примеру, запретить изменения.

```
    private int computeTotalPrice() {
        // Scary code here
    }

    public int getTotalPrice() {
        if(totalPrice == -1)
            totalPrice = computeTotalPrice();
        return totalPrice;
    }
    public void addProduct(Product p) {
        products.add(p);
        totalPrice = -1;
    }
    public void removeProduct(Product p) {
        products.remove(p);
        totalPrice = -1;
    }
}
```

Стоимость продуктов может изменяться во время работы магазина, поэтому в классе `Product` есть метод `setPrice`.

Близится праздник, и наш герой (назовем его Петром) подбирает подарки для своей семьи; это нелегкое дело отнимает у него 2 дня. С приближением праздника в интернет-магазине начинается распродажа, и некоторые товары дешевеют. К концу второго дня корзина Петра полна подарков, и он уже готов нажать «Checkout», но тут он замечает, что — о ужас! — указанная в корзине сумма заказа не соответствует суммарной стоимости товаров. Пётр негодует: закэшированное значение `totalPrice` не было обновлено при изменении цен продуктов — нарушился инвариант «`totalPrice` равно либо –1, либо истинной суммарной цене содержащихся в корзине продуктов», поскольку метод `setPrice` действовал лишь над объектом класса `Product`, ничего не зная об объекте `Cart`, в чьем инварианте этот `Product` присутствовал.

Для решения этой проблемы придется либо отказаться от кэширования цены вовсе, либо сделать так, чтобы класс `Product` позволял подписываться на изменения цены. Оба решения одинаково плохи: первое неэффективно, второе — сложно и подвержено ошибкам: класс `Product`, бывший обычной структурой данных, обрастает всевозможными оповещателями, а все его пользователи обязаны на эти оповещения подписываться. Легко представить, какая путаница будет в коде, учитывая, что бизнес-область содержит множество взаимосвязанных объектов с изменяемыми свойствами — гораздо больше, чем просто `Product` и `Cart`.

3.2.3. Многопоточность

Подавляющее большинство багов в многопоточных программах связано с изменяемыми данными, а именно — с тем, что две (или более) корректных последователь-

ности изменений, переплетаясь в условиях многопоточности, вместе образуют некорректную. Вот классический пример такой ошибки.

Пример: Банковские транзакции. Пусть есть класс `BankAccount`, поддерживающий операции `deposit` (положить деньги на счет) и `withdraw` (снять деньги со счета).

```
class BankAccount {
    void deposit(int amount) {
        setMoney(getMoney() + amount);
    }
    void withdraw(int amount) {
        if(amount > getMoney())
            throw new InsufficientMoneyException();
        setMoney(getMoney() − amount);
    }
}
```

Предположим, у супругов Ивана да Марьи есть общий семейный счет, на котором лежит 100 рублей. Иван решает положить на счет 50 рублей, а Марья в это же время решает положить на счет 25 рублей.

Действия Ивана	Действия Марьи	Деньги на счете
$deposit(50)$	$deposit(25)$	100
$getMoney() \to 100$		100
	$getMoney() \to 100$	100
$setMoney(100 + 50)$		150
	$setMoney(100 + 25)$	125
Итого		125

В результате деньги Ивана оказываются выброшенными на ветер.

Причина этого — переплетение трасс: каждая из операций по отдельности работает правильно, однако лишь в предположении, что состояние системы во время ее работы контролируется только ею; это предположение оказывается неверным. Такая проблема может возникнуть не только в условиях многопоточности, но в этих условиях она проявляется особенно часто и ярко. Возможные пути решения — использование обыкновенных примитивов синхронизации или специальных средств, таких как транзакции.

3.2.4. Сложный код

С введением изменяемых данных в коде появляется измерение времени как на высоком уровне (взаимодействия компонентов), так и на низком — уровне последовательности строк кода. Вслед за ним приходит дополнительная сложность: необходимо

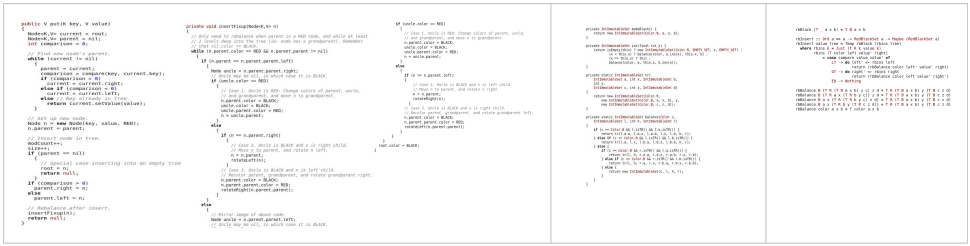
не только решить, *что* надо сделать, но и *в каком порядке*. Она проявляется, в основном, в реализациях сложных структур данных и алгоритмов.

Пример: Двусвязный список. Корректная реализация этой структуры данных — на удивление трудное дело: немногие могут реализовать двусвязный список правильно с первой попытки. Вот (слегка сокращенный) фрагмент кода, осуществляющего вставку в двусвязный список в GNU Classpath. Он выглядит невинно, но можете ли вы *в уме* убедиться в его корректности, не рисуя на бумаге диаграмм для трех возможных случаев и не отслеживая последовательно влияние каждой строки кода на диаграмму?

```
public void add(int index, Object o) {
    Entry e = new Entry(o);
    if (index < size) {
        Entry after = getEntry(index);
        e.next = after;
        e.previous = after.previous;
        if (after.previous == null)
            first = e;
        else
            after.previous.next = e;
        after.previous = e;
    } else if (size == 0) {
        first = last = e;
    } else {
        e.previous = last;
        last.next = e;
        last = e;
    }
    size++;
}
```

Пример: Красно-черные деревья. Более радикальный пример — реализация красно-черных деревьев: на рис. 3.2 представлен вид «с высоты птичьего полета» на процедуры вставки элемента в такую структуру данных: в изменяемое дерево на Java (из GNU Classpath), в неизменяемое на Java (из библиотеки functionaljava) и в неизменяемое на Haskell.

Даже использование диаграмм на бумаге не решает проблемы наличия времени: для отражения изменений во времени приходится в каждой строке кода либо перерисовывать диаграмму заново на чистом участке листа, либо зачеркивать ее части, увеличивая путаницу.



Java, mutable Java, immutable Haskell

Рис. 3.2. Вставка в красно-черное дерево

3.2.5. Наследование

Классы с изменяемым состоянием плохо поддаются наследованию. Класс-наследник, согласно принципу подстановки Лисков,[2] должен быть пригоден к использованию вместо базового класса в любом контексте — в частности, должен поддерживать все его операции и сохранять все его инварианты и спецификации операций. По отношению к операциям, способным изменять состояние объекта базового класса, это означает, что *класс-наследник не имеет права накладывать дополнительные ограничения на это изменяемое состояние*, т.к. тем самым он нарушит спецификацию и инварианты изменяющих методов. Рассмотрим классическую иллюстрацию этой проблемы.

Пример: Геометрические фигуры.

```
class Rectangle {
    private int width, height;
    public Rectangle(int w,h) {
        this.width = w;
        this.height = h;
    }
    int getWidth()              {...}
    int getHeight()             {...}
    void setWidth(int width)    {...}
    void setHeight(int height) {...}
}
class Square extends Rectangle {
    public Square(int side) {
        super(side,side);
    }
}
```

[2]Имеется в виду «принцип подстановки Барбары Лисков», также известный как LSP (Liskov Substitution Principle), гласящий «Если тип S унаследован от типа T, то должно быть возможным подставить объект типа S в любом месте программы, ожидающем тип T, без изменения каких-либо желаемых свойств программы — в т. ч. корректности» [3].

В классе `Rectangle` спецификация операций `setWidth` и `setHeight` такова:

- `r.getWidth() == w && r.getHeight() == h`
  ⇒ после `r.setWidth(w2)` верно
  `r.getWidth() == w2 && r.getHeight() == h`

- `r.getWidth() == w && r.getHeight() == h`
  ⇒ после `r.setHeight(h2)` верно
  `r.getWidth() == w && r.getHeight() == h2`

Вызов `setWidth` или `setHeight` на объекте класса `Square` обязан также удовлетворять этим спецификациям, однако при этом, очевидно, будет разрушен инвариант класса `Square` «`getWidth() == getHeight()`».

Правило стоит повторить еще раз: *Класс-наследник не имеет права накладывать дополнительные ограничения на изменяемое состояние базового класса.*

3.3. Круги ада

Ознакомившись с некоторыми недостатками изменяемого состояния, приступим к организации борьбы с ними. Первый этап борьбы — подробное изучение врага. Выполним классификацию вариантов изменяемого состояния по степени их «вредности».

Прекрасная классификация предложена Скоттом Джонсоном в [2]; приведем ее с небольшими изменениями и добавлениями. Чем больше номер круга ада, тем больше опасностей подстерегает нас. Избавление от опасностей будет зачастую заключаться в переходе с большего номера к меньшему.

1) **Отсутствие изменяемого состояния.** Этот «нулевой» круг ада абсолютно безопасен с точки зрения вышерассмотренных проблем, но достижим лишь в теории.

2) **Невидимое программисту изменяемое состояние** — код алгоритмов, не использующих изменяемое состояние, компилируется в машинный код, использующий изменяемые регистры, стек, память, что при правильной реализации компилятора заметить невозможно. Этот круг так же безопасен с практической точки зрения, как и предыдущий.

3) **Невидимое клиенту изменяемое состояние** — скажем, локальные переменные-счетчики внутри процедуры: изменение таких переменных ненаблюдаемо извне самой процедуры.[3] Этот круг безопасен с точки зрения клиента.

[3] В некоторых языках используются *системы эффектов* [1], позволяющие компилятору делать подобные суждения автоматически.

Рис. 3.3. Круги ада

4) **Монотонное изменяемое состояние** — переменные, присваивание которых происходит не более 1 раза: переменная вначале не определена, а затем определена. Это довольно безобидный тип изменяемого состояния, поскольку у переменной всего 2 состояния, лишь одно из которых не целостно (к тому же, *перевести* переменную в неопределенное состояние невозможно!), и обычно легко обеспечить, чтобы в нужный момент такая переменная оказалась определена. Монотонное изменяемое состояние часто используется для реализации ленивых вычислений.

5) **Двухфазный цикл жизни.** Это разновидность п. 4, при которой состояний у объекта более двух, при этом жизнь объекта поделена на две фазы: инициализация («наполнение»), при которой к нему происходит доступ только на запись, и мирная жизнь, при которой доступ производится только на чтение. Например, система сначала собирает статистику, а затем всячески анализирует ее. Необходимо гарантировать, что во время фазы чтения не будет производиться запись, и наоборот. Позднее будет рассмотрен прием («заморозка»), позволяющий давать такую гарантию.

6) **Управляемое изменяемое состояние** — такое, как внешняя СУБД: в системе присутствуют специальные средства для координации изменений и ограничения их опасностей, например, транзакции.

7) **Инкапсулированное изменяемое состояние** — переменные, доступ к которым производится только из одного места (скажем, private поля объектов). Контроль за целостностью состояния лежит целиком на реализации объекта, и если реализация правильная, то не существует способа привести объект в не-целостное состояние (инварианты самого объекта не нарушаются). Достаточно правильно реализовать сам объект. Тем не менее, для контроля инвариантов, охватывающих несколько объектов, по-прежнему необходимы специальные средства; либо же необходимо инкапсулировать весь контроль за состоянием этих несколь-

ких объектов в другом объекте.

8) **Неинкапсулированное изменяемое состояние** — глобальные переменные. Всем известно, что это — страшное зло. В этом случае нельзя сказать *ничего* о том, кто и когда изменяет глобальную переменную, не изменяет ли ее кто-нибудь прямо сейчас, между *вот этими* двумя строками кода, и т. п.

9) **Разделяемое между несколькими процессами изменяемое состояние.** В условиях многопоточности управление изменяемым состоянием превращается в ад во всех случаях, кроме тривиальных. Огромное количество исследований посвящено разработке методик, позволяющих хоть как-то контролировать корректность многопоточных программ, но пока что главный вывод таков: хотите избежать проблем с многопоточностью — минимизируйте изменяемое состояние. Основная причина трудностей заключается в том, что если имеется N потоков, каждый из которых проходит K состояний, то количество возможных последовательностей событий при одновременном выполнении этих потоков имеет порядок K^N. Техники, позволяющие минимизировать подобные эффекты, рассмотрены ниже.

3.4. Борьба

Прежде чем перейти к обсуждению техник обезвреживания изменяемого состояния, обсудим следующие глобальные идеи:

- **Минимизация общего числа состояний:** Чем меньше у системы состояний, тем меньшее количество случаев надо учитывать при взаимодействии с ней. Следует помнить и о том, что чем больше состояний, тем больше и *последовательностей* состояний, а именно непредусмотренные *последовательности* состояний зачастую являются причинами багов.

- **Локализация изменений:** Чем более локальные (обладающие меньшей областью видимости) объекты затрагивает изменение, тем из меньшего числа мест в коде изменение может быть замечено. Чем менее изменения размазаны между несколькими объектами во времени, тем легче логически сгруппировать их в несколько крупных и относительно независимых изменений объектов. К примеру, при сложном изменении узла структуры данных лучше сначала вычислить все новые характеристики этого узла, а затем произвести изменения, нежели производить изменения сразу в самом узле.

- **Разграничение права на наблюдение и изменение состояния (инкапсуляция):** Чем меньше клиентов могут изменить состояние, тем меньше действия этих клиентов нужно координировать. Чем меньше клиентов могут прочитать состояние, тем легче его защищать от изменений во время чтения.

- **Исключение наблюдаемости промежуточных не-целостных состояний:** Если систему невозможно застать в таком состоянии, то снаружи она всегда выглядит целостной и корректно работающей.
- **Навязывание эквивалентности состояний и их последовательностей:** Если некоторые состояния или их последовательности в каком-то смысле эквивалентны, то клиент избавлен от необходимости учитывать их частные случаи.

И, наконец, сами техники.

3.4.1. Осознание или отказ

Самый первый и самый важный шаг, который следует предпринять, разрабатывая систему, использующую изменяемое состояние — понять, чем в действительности обусловлено его наличие. Действительно ли в предметной области есть понятие объектов, изменяющихся во времени? Действительно ли в предметной области меняются *именно те* объекты, которые вы собираетесь сделать изменяемыми?

К примеру, неудачное архитектурное решение об изменяемости геометрических классов `java.awt` было бы отброшено уже на этом этапе: в геометрии не бывает изменяемых точек и прямоугольников! Авторы библиотеки неверно определили изменяющиеся сущности: в действительности меняются не координаты точки — левого верхнего угла окна, а меняется то, *какая именно* точка является левым верхним углом окна.

Аналогично этому примеру, множества и словари, в их математическом понимании, также не являются сами по себе изменяемыми объектами — во многих задачах оправдано использование *чисто функциональных структур данных* (структур данных, не использующих присваивания). К примеру, в интерфейс чисто функционального множества вместо операции «добавить элемент» входит операция «получить множество, отличающееся от данного наличием указанного элемента». Такие структуры допускают реализацию, совершенно не уступающую по эффективности изменяемым структурам, а в некоторых задачах и существенно превосходящую их (как ни странно, по потреблению памяти) — см. [5].

Если предметная область диктует наличие изменяющихся во времени объектов, то необходимо признать этот факт и помнить о нем постоянно. Особенно важно это в условиях многопоточности.

Необходимо осознать, что код — это больше не спецификация работы алгоритма, а разворачивающаяся во времени последовательность событий с ненулевой длительностью: ни один вызов метода, ни одно присваивание не проходит мгновенно, и между любыми двумя строками кода есть временной «зазор».

Это кажется тривиальным, но если постоянно помнить об этих правилах, то многие глупые (и не только) ошибки многопоточного программирования или неявного взаимодействия становятся отчетливо видны на этапе написания, а не на этапе отладки продакшн-системы.

3.4.2. Инкапсуляция

Большинство описанных ранее проблем с изменяемым состоянием возникает из-за того, что слишком многие клиенты (функции, классы, модули) изменяют состояние, и слишком многие его читают. При этом скоординировать чтение и изменение корректным образом не удается без нарушения модульности и сильного усложнения кода. Минимизировать проблемы такого рода можно, разграничив доступ к состоянию — предоставив клиентам как можно меньше способов прочтения и изменения состояния. Вместе с этим крайне важно, чтобы доступ был согласован с разбиением системы на компоненты. Так компонент, читающий состояние в процессе своей работы, должен либо быть тесно связан с компонентами, могущими его записывать, либо получать доступ к состоянию только в моменты, когда оно не может изменяться, либо проектироваться с учетом того, что состояние может измениться в любой момент.

Вот несколько советов, связанных с инкапсуляцией:

Не возвращайте изменяемые объекты из «читающих» методов Если метод, по своей сути, предназначен для *выяснения* какого-то свойства объекта (скажем, даты создания), а не для *получения доступа* к этому свойству, то он должен возвращать неизменяемый объект. В крайнем случае, он должен возвращать «свежий» объект (как сделано в `java.awt`), но через возвращенный объект должно быть невозможно повлиять на исходный объект.

- Возвращать дату создания в виде объекта класса `Calendar`, возвращая `private`-поля, недопустимо, т.к. объекты класса `Calendar` изменяемы. Гораздо лучше возвращать дату создания в виде `long` — например, в виде числа миллисекунд с 1 января 1970 года (число, возвращаемое функцией **`System.`**`currentTimeMillis()` в Java).
- Возвращая коллекцию, возвращайте ее неизменяемую обертку.
- Возвращая коллекцию, позаботьтесь о том, чтобы составляющие ее объекты были неизменяемы.

Не возвращайте массивы — возвращайте коллекции Если необходимо *получить доступ* к свойству типа «набор объектов», то возвращайте его в виде коллекции, а не в виде массива. Доступ к массивам не инкапсулирован: имея в своем распоряжении массив, всякий клиент может прочитать или перезаписать какие-то из его элементов; при этом невозможно гарантировать атомарность доступа в условиях многопоточности, тогда как коллекции позволяют переопределять операторы чтения и записи и делать их атомарными (`synchronized`).

Предоставляйте интерфейс изменения, соответствующий предметной области Скажем, класс `BankAccount` должен предоставлять не метод `getMoney`/`setMoney`, а методы `deposit` и `withdraw`. Благодаря этому реализация атомарности и контроль

за инвариантами банка (скажем, ненулевое количество денег на счете и сохранение общего числа денег в банке) ляжет на реализацию `BankAccount`, а не на клиентов, вызывающих `getMoney`/`setMoney`. Сюда же можно отнести атомарные операции: `AtomicInteger.addAndGet`, `ConcurrentMap.putIfAbsent`, и т. п. Об этой технике речь пойдет ниже, в разделе «Многопоточные техники».

3.4.3. Двухфазный цикл жизни и заморозка

Довольно часто изменяемые данные требуются для того, чтобы накопить информацию и проинициализировать некоторый объект, который затем будет использоваться уже без изменений. Таким образом, цикл жизни объекта делится на две фазы: фаза записи (накопления) и фаза чтения; причем во время фазы накопления не производится доступ на чтение *извне*, а во время фазы чтения не производится запись. В результате все, кто читают объект, видят его неизменным. Для корректной работы системы достаточно обеспечить, чтобы чтение и запись не пересекались во времени.

В точке перехода от фазы накопления к фазе чтения можно подготовить информацию (закэшировать какие-то значения, построить индексы), что сделает чтение более эффективным.

Существует два подхода к организации двухфазного цикла жизни: статический и динамический.

Статический подход предполагает, что интерфейсы объекта в фазе чтения и фазе записи отличаются.

```
public interface ReadFacet {
    Foo getFoo(int fooId);
    Bar getBar();
}
public interface WriteFacet {
    void addQux(Qux qux);
    void setBaz(Baz baz);

    ReadFacet freeze();
}

class Item implements WriteFacet {
    ...
    ReadFacet freeze() {
        return new FrozenItem(myData);
    }
}
class FrozenItem implements ReadFacet {
    FrozenItem(ItemData data) {
        this.data = data.clone();
        prepareCachesAndBuildIndexes();
```

```
    }
    ...
}
```

Клиент получает объект типа `WriteFacet`, производит в него запись, а затем, когда запись окончена, замораживает объект и в дальнейшем пользуется для чтения полученным `ReadFacet`. Конечно, необходимо, чтобы созданный объект `ReadFacet` был полностью независим от замораживаемого, в противном случае дальнейший вызов записывающих методов испортит его.

Зачастую организовывать два лишних интерфейса неудобно, или же такой возможности может просто не оказаться: скажем, клиент ожидает библиотечный интерфейс, содержащий и методы чтения, и методы записи. При этом все же желательно иметь возможность гарантировать отсутствие записей после начала чтения, и возможность построить кэши после фазы чтения с гарантией их будущей целостности. Тут пригодится второй подход к заморозке.

Динамический подход предполагает, что формально интерфейсы объекта в фазе чтения и записи одинаковы, однако фактически поддерживаемые операции отличаются, как и в случае статического подхода.

```
class Item implements QuxAddableAndFooGettable {
    private boolean isFrozen = false;
    ...
    void addQux(Qux qux) {
        if(isFrozen) throw new IllegalStateException();
        ...
    }

    Foo getFoo(int fooId) {
        if(!isFrozen) throw new IllegalStateException();
        // Efficiently get foo using caches/indexes
    }

    void freeze() {
        prepareCachesAndBuildIndexes();
        isFrozen = true;
    }
}
```

Здесь статическая проверка заменяется на динамическую, однако общая идея остается той же: на фазе записи объект предоставляет только интерфейс записи, на фазе чтения — только интерфейс чтения.

В качестве иллюстрации такого подхода рассмотрим пример из практики автора.

Пример: Кластеризация документов. Каждый документ в программе описывается длинным разреженным битовым вектором свойств. В начале программа анализи-

рует документы, составляя их векторы свойств и пользуясь изменяющим интерфейсом битового вектора (установить/сбросить бит). Затем, при вычислении матрицы попарных расстояний между документами выполняется лишь две операции: вычисление мощности пересечения и мощности объединения двух векторов. Эти операции можно реализовать очень эффективно, если «подготовить» векторы, построив на них «пирамиду» (не будем углубляться в подробности), однако обновлять пирамиду при вставках в битовый вектор слишком дорого. Поэтому пирамида строится для каждого вектора сразу после вычисления векторов и перед вычислением матрицы попарных похожестей, и матрица затем строится очень быстро (ускорение в рассматриваемой задаче достигло двух порядков).

Поскольку объекты с двухфазным циклом жизни легко использовать корректно, то стоит попытаться разглядеть их в своей задаче или свести ее к ним; вовсе не обязательно при этом даже использовать заморозку — само наличие двухфазности вносит в программу стройную структуру.

3.4.4. Превращение изменяемой настройки в аргумент

Довольно часто бывает так, что изменяемые данные используются для «настройки» объекта: объект настраивается с помощью нескольких сеттеров, а затем выполняет работу. Это похоже на двухфазный цикл жизни, однако процесс настройки сконцентрирован в одном месте кода и в одном коротком промежутке времени, известное количество настроек подаются одна за другой.

В некоторых случаях это не является проблемой, однако представим себе такой класс:

Пример: Подсоединение к базе данных.

```
class Database {
    void setLogin(String login);
    void setPassword(String password);
    Connection connect() throws InvalidCredentialsException;
}
```

Наличие такого класса может поначалу выглядеть оправданным, если он используется, скажем, из графического интерфейса, который сначала запрашивает у пользователя логин (и, запросив, вызывает `setLogin`), затем запрашивает пароль (и вызывает `setPassword`), а затем по нажатию кнопки «Connect» вызывает `connect`.

Однако если объект `Database` окажется разделяемым между несколькими пользователями, то вызовы `setLogin`, `setPassword`, `connect` начнут путаться между собой, пользователи будут получать чужие соединения, и т. п. — такая ситуация совершенно недопустима.

Гораздо лучше избавиться от изменяемости в интерфейсе `Database`:

```
class Database {
    Connection connect(String login,String password)
```

```
        throws InvalidCredentialsException;
}
```

При этом, конечно, клиенту придется управлять хранением значений `login` и `password` самостоятельно, однако путаница будет устранена.

Для ситуаций, когда хранение данных нежелательно (например, долгое хранение пароля в памяти может быть нежелательно по соображениям безопасности), пригодится, в частности, паттерн «Curried Object» («Каррированный объект»), о котором речь пойдет позже.

3.4.5. Концентрация изменений во времени

Родственный предыдущему метод — концентрирование изменений во времени. Он заключается в том, чтобы вместо последовательности мелких изменений выполнять одно большое, тем самым убивая сразу двух зайцев:

1) Избавление от промежуточных состояний между мелкими изменениями.

2) Избавление от необходимости устанавливать протокол последовательности внесения изменений.

П.1 позволяет решить проблемы, сходные с описанными в предыдущем разделе (путаницу между клиентами).

П.2 полезен в случаях, когда мелкие изменения в определенном смысле взаимозависимы, и не всякая последовательность мелких изменений является корректной, поэтому приходится навязывать клиенту протокол, предписывающий, в каком порядке нужно вносить изменения. Это может быть чрезвычайно неудобно.

Пример: Библиотека бизнес-правил. Рассмотрим библиотеку, позволяющую пользователю задать ряд бизнес-правил для вычисления некоторых величин. Правила (формулы) могут быть взаимозависимы.

Интерфейс библиотеки выглядит так:

```
interface RuleEngine {
    void addRule(String varName, String formula)
        throws ParseException, UndefinedVariableException;

    double computeValue(String varName);
}
```

В каждый момент правила должны быть целостными, поэтому `addRule` бросает `UndefinedVariableException` в случае, когда `formula` ссылается на переменную, для которой еще нет правила.

Большой минус такой библиотеки — в том, что если клиенту необходимо задать сразу несколько взаимозависимых правил (скажем, считать их из файла или электронной таблицы), то клиент должен *сам* заботиться о том, чтобы подавать правила в порядке топологической сортировки, т.е. добавлять зависимое правило только после добавления всех, от которых оно зависит!

Лучше перепроектировать интерфейс так:

```
interface RuleParser {
    RuleSet parseRules(Map<String,String> var2formula)
        throws ParseException, CircularDependencyException;
}
interface RuleSet {
    double computeValue(String varName);
}
```

Теперь конструирование набора взаимозависимых правил инкапсулировано в методе `parseRules`. Он сам выполняет топологическую сортировку передаваемых ему пар «переменная/формула» и детектирует циклические зависимости.

3.4.6. Концентрация изменений в пространстве объектов

Эта методика призвана устранить необходимость в поддержании инвариантов, охватывающих сразу несколько объектов: чем меньше объектов охватывает инвариант, тем проще его сохранять, и тем меньше шансов, что кто-то разрушит инвариант, изменив один из составляющих его объектов, но не изменив и ничего не зная о другом.

К примеру, сложность реализации двусвязных списков в значительной степени проистекает из того, что каждая операция, затрагивающая определенный узел, должна позаботиться о двух его соседних узлах и о самом списке (ссылках на первый/последний узел).

Методика состоит в том, чтобы минимизировать количество объектов, охватываемых инвариантами. Зачастую она сводится к разрыву зависимостей между объектами и, особенно, разрыву циклических зависимостей. К сожалению, с двусвязными списками, по-видимому, ничего не поделать; мы рассмотрим другой пример.

Пример: Многопользовательская ролевая игра. В рассматриваемой игре есть понятие «артефакта», и характеристики артефакта могут зависеть от характеристик игрока, носящего его. Поэтому в артефакт включается информация о его владельце:

```
class Artifact {
    Player getOwner();
    void setOwner(Player player);

    Picture getPicture();

    int getStrengthBoost();
    int getHealthBoost();
    int getDexterityBoost();
    int getManaBoost();
}
class Player {
    List<Artifact> getArtifacts();
    void addArtifact(Artifact a);
```

```
    void dropArtifact(Artifact a);
    void passArtifact(Artifact a, Player toWhom);
}
```

Должен выполняться следующий инвариант: «если `a.getOwner()==p`, то `p.getArtifacts().contains(a)`, и наоборот».

Все 4 метода `get*Boost()` учитывают расу игрока (`getOwner().getRace()`): у великанов умножается на 2 значение `getStrengthBoost` любого артефакта, у гномов — `getHealthBoost`, у эльфов — `getDexterityBoost`, у друидов — `getManaBoost`.

Пока артефакт лежит на земле, его `getOwner()` равен **`null`**. Когда игрок подбирает, роняет или передает артефакт, вызывается `setPlayer`.

Для каждого артефакта, присутствующего на карте мира, нужен отдельный экземпляр класса `Artifact`, хранящий в себе `Picture`, `strengthBoost`, `healthBoost`, `dexterityBoost` и `manaBoost` — использовать один и тот же экземпляр даже для совершенно одинаковых артефактов нельзя, т.к. экземплярами могут владеть разные игроки. Учитывая, что артефактов на карте может быть *очень* много, а характеристик у них может быть гораздо больше, чем указано здесь — имеет место лишний расход памяти.

Ситуацию можно улучшить, если сделать класс `Artifact` неизменяемым и разорвать зависимость от игрока (убрать методы `getPlayer`/`setPlayer`), тем самым избавившись полностью от необходимости поддерживать указанный инвариант, позволив использовать один и тот же экземпляр класса `Artifact` в рюкзаках разных игроков и существенно сократив потребление памяти.

Встает, конечно, вопрос — как же теперь быть с методами `get*Boost()`? Ведь метод без аргументов у артефакта больше не может давать правильный ответ.

Ответов несколько, и все они очень просты:

- Добавить аргумент типа `Player` к методам `get*Boost()`.
- Перенести методы в класс `Player`: `Player.getStrengthBoost(Artifact a)`.
- Переименовать методы в `getBaseStrenghBoost` и т. п., и вынести логику определения действия артефакта в отдельный класс `ArtifactRules`, в методы `getStrengthBoost(Artifact a, Player p)`. К таким методам можно будет легко добавить обработку и других условий: погоды, наличия вокруг других игроков и т. п.

Этот прием — избавление от изменяемых данных благодаря разрыву зависимостей — используется в чисто функциональных структурах данных и позволяет разделять значительную часть информации между двумя мало отличающимися структурами. Благодаря этому бывает возможна огромная экономия памяти (в задаче из практики автора, где требовалось хранить большое количество не очень сильно различающихся целочисленных множеств, переход от изменяемого красно-черного дерева к неизменяемому позволил сократить потребляемую память примерно на два порядка).

3.4.7. Каррирование

Название этой методики связано с аналогичным понятием из функционального программирования: при каррировании функции с несколькими аргументами некоторые из этих аргументов фиксируются, и получается функция от оставшихся.

Методика описана в статье [4] и предназначена для упрощения сложных протоколов, где каждый вызов требует большого количества аргументов, некоторые из которых меняются редко или не меняются вовсе.

Эта методика прекрасно подходит для разрыва зависимости по состоянию между несколькими клиентами, использующими объект, и попутно дает еще несколько преимуществ.

Можно сказать, что паттерн *Curried Object* — это частный случай инкапсуляции состояния; более точно, он предписывает выделить часть объекта, позволяющую хорошую инкапсуляцию состояния, в самостоятельный объект.

Пример: Улучшение безопасности подсоединения к БД. Рассмотрим упомянутый выше пример с подсоединением к базе данных. Проблема изначальной версии заключалась в том, что вызовы разных клиентов `setLogin`, `setPassword`, `connect` переплетались, а исправленной — в том, что программа могла долго хранить в памяти пароль. Фиксированным (хотя и неявным) аргументом в данном случае является то, какой клиент производит вызовы. Применим паттерн *Curried Object* и выделим каждому клиенту его личный объект для обработки протокола соединения.

```
class Database {
    Connector makeConnector();
}
class Connector {
    void sendLogin(String login);
    void sendPassword(String password);
    Connection connect()
        throws InvalidCredentialsException;
}
```

В такой реализации `makeConnector` будет создавать новый независимый объект, производящий сетевое соединение с базой данных и в методах `sendLogin`, `sendPassword` сразу отсылающий логин и пароль по сети. У каждого клиента объект `Connector` будет свой, поэтому клиенты не будут мешать друг другу.

Рассмотрим еще одну иллюстрацию паттерна «Curried Object».

Пример: Загрузчик данных. Программа предназначена для загрузки в базу большого количества данных от разных клиентов: клиент подключается к программе и загружает большое количество данных, затем отключается. Программа обязана обеспечить транзакционность загрузки данных от каждого клиента. Изначально API программы проектируется так:

```
class DataLoader {
    ClientToken beginLoad(Client client);
```

```
    void writeData(ClientToken token, Data data);
    void commit(ClientToken token);
    void rollback(ClientToken token);
}
```

Клиент вызывает `beginLoad` и с помощью полученного `ClientToken` многократно вызывает `writeData`, затем вызывает `commit` или `rollback`. Эта программа избавлена от проблем переплетения запросов между клиентами, однако код `DataLoader` довольно сложен: он хранит таблицу соответствия клиентов и транзакций и соединений БД, и в каждом из методов пользуется соответствующими элементами таблицы.

В некоторых задачах клиенты сами по себе могут быть многопоточными: скажем, клиент может в несколько потоков вычислять данные и выполнять их запись. Если метод `writeData` не обладает потокобезопасностью при фиксированном `token`, то код еще усложнится: добавится таблица соответствия «клиент / объект синхронизации», и метод `writeData` будет синхронизироваться по соответствующему ее элементу.

Нельзя забывать и о том, что доступ к таблицам также должен быть синхронизирован.

Существенно упростить код можно, если перепроектировать `DataLoader`, применив *Curried Object*:

```
class DataLoader {
    PerClientLoader beginLoad(Client client);
}
class PerClientLoader {
    void writeData(Data data);
    void commit();
    void rollback();
}
```

Теперь класс `DataLoader` полностью избавлен от изменяемого состояния! В классе `PerClientLoader` нет никаких таблиц, и синхронизация выполняется тривиально — достаточно объявить все три метода как `synchronized`. Клиенты также никак не могут помешать друг другу. Код получился простым и безопасным.

3.4.8. Многопоточные техники

Как уже было сказано, трудности с написанием корректных многопоточных программ проистекают от большого количества возможных совместных трасс выполнения нескольких процессов. Корректность необходимо гарантировать на всех трассах, в связи с чем приходится рассматривать большое количество возможных переплетений трасс и проверять на корректность каждое из них. Напомним, что количество трасс при N потоках, у каждого из которых K состояний, можно оценить как K^N.

Возможные пути уменьшения числа трасс, которые необходимо учитывать, таковы:

Использование критических секций для синхронизации. Охватывание критической секцией блока кода, содержащего M состояний, уменьшает число состояний на $M - 1$, превращая весь блок в один переход между двумя состояниями. Благодаря этому соответственно уменьшается количество совместных трасс.

Использование атомарных операций. Это двоякий совет. С одной стороны, речь идет о том, чтобы пользоваться эффективными аппаратно реализованными операциями, такими как «прочитать-и-увеличить» (get-and-add), «сравнить-и-поменять» (compare-and-exchange) и т. п. С другой стороны, что более важно, речь идет о том, чтобы предоставлять API в терминах атомарных, соответствующих предметной области, операций:

- «Перевести деньги с одного счета на другой» (помимо «снять деньги, положить деньги»).
- «Добавить элемент, если он еще отсутствует» (помимо «добавить элемент, проверить наличие») — кстати, сюда же относятся SQL-команды MERGE и INSERT IGNORE.
- «Выполнить соединение с данным логином и паролем» (помимо «установить логин, установить пароль, выполнить соединение»).
- и т. п.

Локализация изменяемого состояния. Вместо применения нескольких изменений к глобально видимому объекту — вычисление большого изменения в локальной области видимости и его атомарное применение. Этот прием уже был рассмотрен выше в разделе «Концентрация изменений во времени».

Навязывание эквивалентности трасс. Кардинально иной подход к уменьшению числа различных трасс — стирание различий между некоторыми из них. Для этого можно использовать следующие два алгебраических свойства некоторых операций (а если операции ими не обладают — попытаться перепроектировать их так, чтобы обладали):

- **Идемпотентность:** Операция, будучи выполненной несколько раз, имеет тот же эффект, что и при однократном выполнении.
- **Коммутативность:** Порядок последовательности из нескольких операций различного типа или с различными аргументами не имеет значения.

Такие операции особенно важны в распределенном и асинхронном программировании, когда синхронизация может быть крайне неэффективна или просто-напросто невозможна.

Пример: Покупка в интернет-магазине. Рассмотрим простой интерфейс оформления покупки в интернет-магазине: пользователь логинится, выбирает продукт, количество, и жмет кнопку «купить». При этом на сервере вызывается следующий API:

```
class Shop {
    void buy(Request request, int productId, int quantity) {
        Session session = request.getSession();
        Order order = new Order(
            session.getCustomer(), productId, quantity);
        database.saveHistory(order);
        billing.bill(order);
    }
}
```

Представим себе ситуацию, когда закупиться хочет пользователь с нестабильным соединением с интернетом. Он нажимает кнопку «Купить», однако ответный TCP-пакет от сервера теряется и браузер в течение 5 минут показывает «Идет соединение…». Наконец, пользователю надоедает ждать и он нажимает кнопку «Купить» еще раз. На этот раз все проходит нормально, однако через неделю пользователю приходит два экземпляра товара и он с удивлением обнаруживает, что деньги с его кредитной карты также сняты два раза.

Это произошло потому, что операция `buy` не была идемпотентна — многократное ее выполнение не было равносильно однократному. Конечно, нельзя говорить об идемпотентности операции покупки товара — ведь купить телевизор два раза — не то же самое, что купить его один раз! Можно, однако, говорить об идемпотентности операции нажатия кнопки «Купить» на некоторой странице.

```
class Session {
    int stateToken;

    void advance() {
        ++stateToken;
    }
    int getStateToken() {
        return stateToken;
    }
}
class Shop {
    Page generateOrderPage(Request request) {
        Session session = request.getSession();
        session.advance();
        ...<INPUT type='hidden'
                  value='"+session.getStateToken()+"'>...
    }

    void buy(Request request, int productId, int quantity) {
        Session session = request.getSession();
```

```
            if(session.currentStateToken() !=
               request.getParamAsLong("stateToken"))
            {
                return;
            }
            Order order = new Order(
                session.getCustomer(), productId, quantity);
            database.saveHistory(order);
            billing.bill(order);

            session.advance();
        }
    }
```

Таким образом, каждая страница заказа помечается целым числом; при каждой загрузке страницы заказа или покупке это число увеличивается. Нажать кнопку «Купить» несколько раз с одной страницы нельзя — это обеспечивается тем, что после того, как покупка уже была произведена, `stateToken` у сессии увеличивается и перестает совпадать со `stateToken` в запросе.

Операция стала идемпотентной, а ситуации, подобные описанной, невозможными.

3.5. Заключение

Тот факт, что обеспечение корректности программ, использующих изменяемое состояние, особенно многопоточно — нелегкая задача, осознают почти все программисты. Программистский фольклор включает множество описанных и неописанных приемов рефакторинга таких программ в более простую и корректную форму. В этой статье автор предпринял попытку собрать вместе и описать эти приемы, а также выделить их общие идеи.

Идеи, как оказалось, сводятся к избавлению от изменяемого состояния, приведению кода в более декларативную и близкую к предметной области форму, инкапсуляции и локализации объектов с состоянием, налаганию на код определенных алгебраических свойств — все то, что знакомо функциональным программистам, ценящим возможность легко рассуждать о свойствах программ математически.

Автор выражает надежду, что данная статья послужит как полезным справочником для программистов на объектно-ориентированных и процедурных языках, так и мотивирует их погрузиться в интереснейший мир функциональных языков, откуда многие из описанных идей почерпнуты и где изменяемое состояние — скорее исключение, чем правило, благодаря чему код легко тестируем и настолько корректен, что бытует лишь наполовину шуточное утверждение «компилируется — значит работает».

Литература

[1] Effect system. — Статья в Wikipedia, URL: http://en.wikipedia.org/wiki/Effect_system (дата обращения: 20 июля 2009 г.).

[2] *Johnson S.* — Комментарий на форуме Lambda-the-Ultimate, URL: http://lambda-the-ultimate.org/node/724#comment-6621 (дата обращения: 20 июля 2009 г.).

[3] *Liskov B. H., Wing J. M.* Behavioural subtyping using invariants and constraints // Formal methods for distributed processing: a survey of object-oriented approaches. — New York, NY, USA: Cambridge University Press, 2001. — Pp. 254–280.

[4] *Noble J.* Arguments and results // In PLOP Proceedings. — 1997.

[5] *Okasaki C.* Purely Functional Data Structures. — Cambridge University Press, 1998.