В. В. Мухортов, В. Ю. Рылов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, АНАЛИЗ И ДИЗАЙН

Методическое пособие

Новосибирск
2002

УДК 519.682

Мухортов В. В., Рылов В.Ю.

Объектно-ориентированное программирование, аеализ и дизайн. Методическое пособие. Новосибирск, 2002.

Учебное пособие представляет собой завершаюшую часть серии пособий по курсу информатики. Рекомендуется для студентов старших курсов, прошедших начальный курс ООП.

Разработка поддержана Фондом содействия развитию малых форм предприятий в научно-технической сфере (договор № 1296р/3029 от 27 апреля 2001г.).

Введение

Методическое пособие состоит из двух частей. Первая часть написана В. Ю. Рыловым, вторая — В. В. Мухортовым. Весь материал обсуждался обоими авторами, так что общая ответственность нераздельна.

Первая часть методического пособия призвана ознакомить читателя с теоретическими основами объектно-ориентированного программирования (ООП). За основу концепции преподавания теоретических основ ООП взята методология разработанная Г. Бучем и изложенная в его книге, посвященной объектно-ориентированному анализу и проектированию [1, §1–§3]. Книга Буча содержит наиболее полное и систематическое изложение вопросов, связанных с применением объектно-ориентированного подхода в разработке программного обеспечения, доступное на русском языке. Многие основные определения взяты из нее, что отмечается в тексте. Для более детального изучения предмета рекомендуется обратиться к этой книге.

Сам курс объектно-ориентированного программирования построен по следующей схеме: вначале рассматриваются теоретические основы объектной модели, включая эволюцию, основные принципы и рассмотрение природы классов и объектов; далее следует изучение основных инструментальных средств ООП языка программирования C++; второй семестр курса посвящен изучению инструментальных средств ООП языка программирования Java.

Вторая часть посвящена основам объектно-ориентированного дизайна (ООД).

Часть I

Объектно-ориентированное программирование

Глава 1

Эволюция методологий программирования

Прежде чем мы начнем нашу беседу, посвященную объектно-ориентированному программированию, давайте обратимся к истории развития языков и методов программирования и разработки информационных систем. Читатель наверняка уже не раз сталкивался с такими терминами, как *процедурное программирование*, или *процедурный язык программирования*, а также с терминами *объектный язык программирования*, *объектно-ориентированный язык программирования*, но многие тонкости в различии этих терминов и технологий очень часто ускользают из поля зрения. Одновременное использование разных методологий при разработке информационной системы часто ведет к серьезным трудностям при ее сопровождении и делает дальнейшее развитие практически невозможным. При выборе методологии в каждом конкретном случае нужно учитывать множество факторов, так как каждая из них имеет свои особенности и границы применимости.

То, что сейчас наиболее распространены именно объектно-ориентированное программирование и объектно-ориентированный дизайн, еще не значит, что другие методологии потеряли право на существование, а правила, заложенные в них — свою актуальность. Мир про-

граммных систем сложен и многообразен, и в разных случаях бывает удобно использовать разные подходы и методы решения возникающих задач. Данная глава призвана упорядочить уже имеющиеся сведения и проследить путь, приведший к возникновению такого мощного и удобного средства разработки программных систем, как объектный подход.

Проанализировав путь развития основных языков программирования, можно выделить следующие постоянно присутствующие, сменяющие друг друга тенденции:

- Смещение акцентов от частного (программирование деталей), к общему (программирование более крупных компонент)
- Развитие и совершенствование инструментария программиста (языков программирования высокого уровня и рабочей среды)
- Возрастание сложности программных и информационных систем.

Именно расширение области применения информационных технологий и вычислительной техники служило и продолжает служить движущей силой эволюции методов и инструментов построения программных систем. На протяжении всей истории развития информационных технологий проводилось огромное множество прикладных исследований по методологии проектирования, декомпозиции, абстрагированию и иерархиям. Следствием этих исследований стало появление все более и более выразительных языков программирования. Возникла тенденция перехода от языков, указывающих компьютеру, что делать (императивные языки), к языкам, описывающим ключевые абстракции предметной области (декларативные языки)[1, §2].

1.1 Поколения языков программирования

Из огромного числа языков программирования, появившихся за период развития информационных технологий, лишь наиболее удобные и совершенные были приняты обществом разработчиков и отстояли свое

право на существование. Анализируя языки программирования и обстоятельства, сопутствующие их появлению, можно обнаружить множество общих черт. Это позволяет сгруппировать языки по основным используемым принципам и выделить поколения в их развитии. Буч[1, §2] приводит следующую классификацию Вегнера:

- Первое поколение (1954 – 1958)

FORTRAN I	Математические формулы
ALGOL-58	Математические формулы
Flowmatic	Математические формулы
IPL V	Математические формулы

- Второе поколение (1959 – 1961)

FORTRAN II	Подпрограммы, раздельная компиляция
ALGOL-60	Блочные структуры, типы данных
COBOL	Описание данных, работа с файлами
Lisp	Обработка списков, указатели, сборка мусора

- Третье поколение (1962 – 1970)

PL/1	FORTRAN+ALGOL+COBOL
ALGOL-68	Более строгий преемник ALGOL-60
Pascal	Более простой преемник ALGOL-60
Simula	Классы, абстрактные данные

Как можно видеть, многие идеи, лежащие в основе современных языков программирования, появились в том или ином виде уже к 1970 году. Все последующие языки, за редким исключением, являются потомками или результатом обобщения и развития вышеперечисленных. Этому во многом способствовало как широкое распространение мини- и

микро-ЭВМ и связанный с ним рост числа разработчиков программного обеспечения, так и многообразие операционных систем и различных сфер применения информационных технологий.

1.1.1 Начало начал, или первое поколение языков программирования

Мы начнем свое рассмотрение языков программирования со времен появления первых Цифровых ЭВМ. Еще очень несовершенные и громоздкие, электронные вычислительные машины использовались тогда исключительно для математических и статистических расчетов. Их область применения была ограничена следующими особенностями:

- Малым объем оперативной памяти.
- Несовершенством системы ввода-вывода.

Ввиду данных ограничений, а также малого количества и дороговизны этих машин, на них работали исключительно высококвалифицированные специалисты, способные управлять ими непосредственно на уровне двоичных кодов. Для облегчения процесса программирования вскоре были созданы языки первого поколения. Это были первые языки, которые приближали программирование к предметной области и отдаляли его от конкретной машины. Их словарь практически полностью был математическим. Обратите внимание на топологию[1] языков первого и начала второго поколения, приведенную на Рис 1.1.

Программы, реализованные на языках первого и начала второго поколения, имели относительно простую структуру, состоящую из подпрограмм и данных, лежащих в глобальной области видимости. Механизмы языков не поддерживали разделения разнотипных данных, что сильно осложняло написание больших программ. Основная сложность при этом заключалась в том, что ошибка или любые изменения в одной

[1]Под термином «топология» в данном контексте, мы будем понимать основные элементы языка программирования и их взаимодействие.

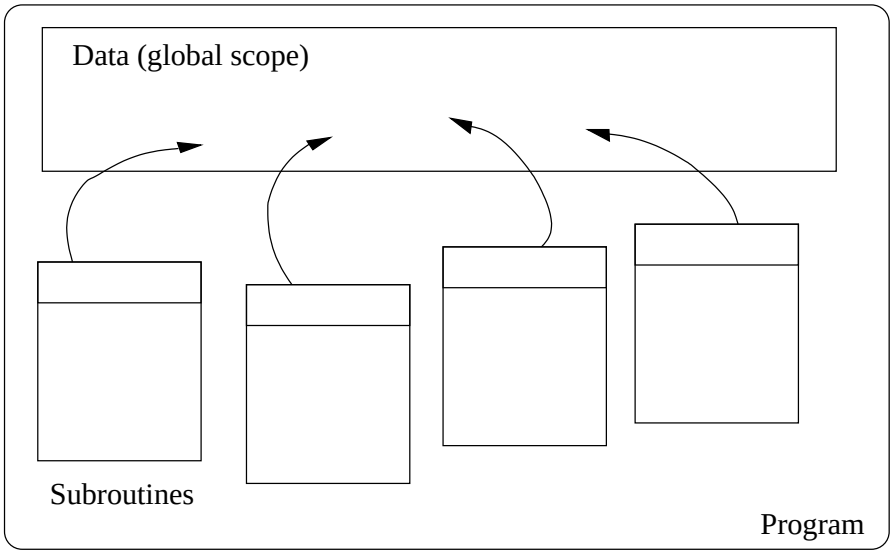

Рис. 1.1: Топология первого и начала второго поколения языков программирования

подпрограмме могли иметь губительные последствия для программной системы в целом.

1.1.2 Развитие алгоритмических абстракций. Второе поколение языков программирования

На момент своего появления, подпрограммы расценивались лишь как средство облегчающее процесс написания программ. Будучи минимальными единицами переиспользования, они стали «кирпичиками» для построения первых библиотек. Постепенно они стали играть фундаментальную роль в декомпозиции программных систем.

Выделение подпрограмм, как механизм абстрагирования, имело следующие важные последствия:

- Были разработаны различные механизмы передачи параметров.
- Были заложены основания для структурного программирования, что выражалось в появлении языковой поддержки механизмов вложенности подпрограмм и научной разработке структур управления и областей видимости.
- Возникли методы структурного проектирования, основой которых служило использование процедур или подпрограмм в качестве отдельных строительных блоков.

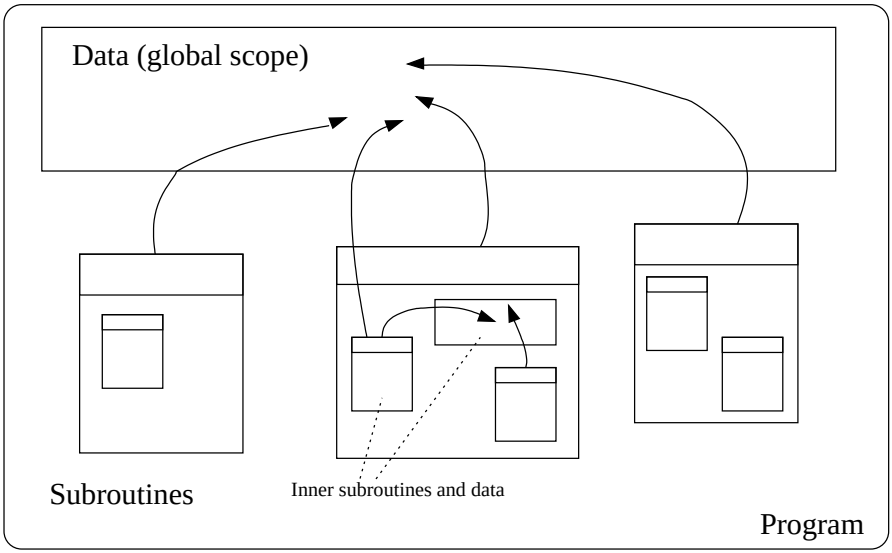

Рис. 1.2: Топология языков программирования конца второго и начала третьего поколения

1.1.3 Модульность, как единица построения программных систем, или третье поколение языков программирования

Разрастание программных проектов требовало увеличения коллективов разработчиков и появления механизмов, позволяющих этим коллективам независимо работать над разными частями проекта. Так появились модули.

Модуль — отдельно компилируемая части программы, состоящая из наборов данных и подпрограмм.

В модули, как правило, собирались подпрограммы, которые, как предполагалось, должны разрабатываться и изменяться совместно. Туда же собирались и данные, которые использовались этими подпрограммами. Постепенно модули стали новым, более крупным механизмом абстракции программных систем.

Первоначально языки программирования не имели достаточно развитых механизмов защиты данных одного модуля от использования их процедурами другого. Во многом эта задача ложилась на коллективы разработчиков. Появившиеся различные подходы в разработке программных систем благоприятствовали возникновению огромного количества языков, имеющих те или иные сильные и слабые стороны в реализации этих принципов. Одним из наиболее развитых представителей языков третьего поколения стал язык ALGOL-68. Будучи универсальным и реализуя почти все разработанные к тому времени механизмы в алгоритмических языках, он был достаточно труден для первоначального освоении, однако позволял разрабатывать системы корпоративного масштаба для больших ЭВМ.

Благодаря распространению малых ЭВМ, огромную популярность приобрели более простые потомки ALGOL-60 — язык Pascal, до сих пор наиболее широко используемый в академический и учебной среде, и появившийся во второй половине 70-х годов язык C, который приобрел наибольшую популярность среди профессиональных программистов. Pascal изначально служил средством обучения структурному программированию, а язык C был разработан для написания операци-

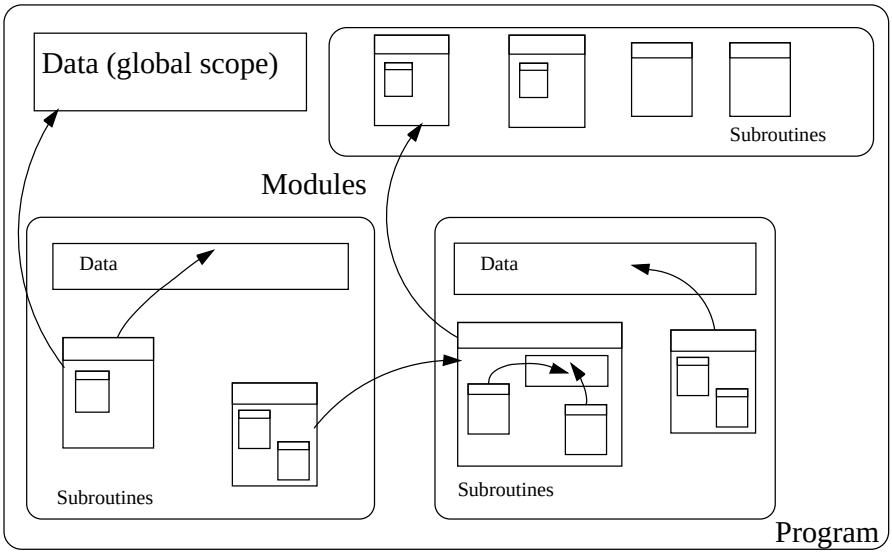

Рис. 1.3: Топология языков конца третьего поколения

онной системы Unix.

Классическим потомком языка Pascal, предназначенным для профессионального написания сложных программных комплексов, стал язык Ada, имеющий встроенную поддержку модульности и абстрактных типов данных.

Язык C, имеющий гибкий лаконичный синтаксис и простую алгоритмическую структуру (в нем отсутствуют вложенные процедуры), стал очень популярным языком как системного, так и прикладного программирования. Наличие возможности низкоуровневого управления памятью (преобразование типов, наличие указателей на данные и функции), препроцессор и поддержка макросов делают его языком системного программирования, предоставляя средства, ранее доступные для малых ЭВМ только на уровне языка ассемблера. Простота в освоении и чрезвычайная гибкость делают его многофункциональным средством прикладного программирования.

1.2 Зарождение объектной модели

Процедурно-ориентированные языки мало подходят для написания программных систем, где центральным место занимают данные, а не алгоритмы. С определенного момента возникла острая необходимость в языковой поддержке описания произвольных объектов окружающего мира. Вводятся абстрактные типы данных. Как хорошо сказал Шанкар: «Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи».

Первым языком, в котором нашли свое выражение идеи построения программ на основе данных и объектов стал язык Simula 67. Концепции, заложенные в языке Simula получили свое развитие в серии языков Smalltalk-72,-74,-76,-80, а также в языках C++ и Objective C. При внесении объектно-ориентированного подхода в язык Pascal появился язык Object Pascal. В 90-х годах компания Sun представила миру язык Java, как воплощение идеи платформенной независимости и наиболее полную реализацию концепций объектно-ориентированного программирования, положенных в основу языков Simula 67, Smalltalk, C++.

1.2.1 Объектные языки программирования

Основные принципы и концепции объектной модели в программировании развивались в процессе эволюции множества разных объектных и объектно-ориентированных языков, что привело к некоторой путанице в терминологии.

Можно очень хорошо продемонстрировать трансформацию и развитие модульной структуры в объектную на основе классического примера из книги Страуструпа [**?**, §2]. Представим себе, что нам нужно

реализовать такой известный программный механизм как стек. Совершенно естественно, что при этом мы преследуем несколько целей. Во-первых, обеспечить некоторую универсальность (чтобы использовать данный механизм во многих приложениях). И во-вторых, обеспечить защиту внутренних деталей реализации от потенциального воздействия других компонент программ.

Рассмотрим решение этой задачи средствами процедурного языка C, далее решим задачу с использованием средств поддержки модульности языка C++, и, наконец, предложим объектную реализацию стека на C++. Для простоты наш стек будет хранить нетипизированные указатели на void*. Это, безусловно не лучшее решение, но в нашем случае цель состоит в демонстрации разных подходов, а не написании профессионального кода для практического использования. Для того, чтобы не загромождать пример лишними подробностями, мы опустим обработку ошибок выделения памяти.

В случае использования механизмов языка C, стратегия может быть следующей — мы объявляем набор операций для работы со стеком и заводим некоторую структуру для хранения его содержимого. Ниже приведен пример заголовочного файла для стека, а также файл модуля, где реализованы функции работы со стеком, объявленные в заголовке.

```
/* file simple_stack.h */
#ifndef _SIMPLE_STACK_H
#define _SIMPLE_STACK_H
#define MAX_STACK_SIZE 200

typedef struct  {
   int top;
   void* content[MAX_STACK_SIZE];
} SimpleStack;

/* returns -1 if overflow */
int push(SimpleStack *pstack, void *pobj);
/* returns NULL if empty */
void* pop(SimpleStack *pstack);
```

```
SimpleStack* allocateStack();
void freeStack(SimpleStack *pstack);
#endif /*_SIMPLE_STACK_H */

/* file simple_stack.c */
#include <malloc.h>
#include <stdlib.h>
#include "simple_stack.h"

SimpleStack* allocateStack()
{
    SimpleStack* pstack =
        (SimpleStack*)malloc(sizeof (SimpleStack));
    pstack->top = 0;
    return pstack;
}

void freeStack(SimpleStack *pstack)
{
    free(pstack);
}

int push(SimpleStack *pstack, void *pobj)
{
    if (pstack->top < MAX_STACK_SIZE) {
        pstack->content[(pstack->top)++] = pobj;
        return 0;
    } else {
        return -1; /* overflow */
    }
}

void* pop(SimpleStack *pstack)
{
    if(pstack->top > 0) {
        return pstack->content[–(pstack->top)];
    } else {
        return NULL; /* underflow */
```

```
    }
}
```

Данная реализация имеет множество недостатков, хотя и может сгодиться для небольших программ. Главными являются следующие два:

- Данные экземпляров структуры не защищены от недобросовестного использования. Следующий код может непоправимо нарушить работу программы:

  ```
  SimpleStack pStack = allocateStack();
  pStack->top = -10;
  ```

- Стек реализован в виде массива, и нет простого способа заменить его реализацию в программах, уже использующих данную версию.

Мы можем существенно усовершенствовать наш стек, если воспользуемся средствами модульного программирования, имеющимися в языке C++.

```
// Файл advanced_stack.h
// Усовершенствованный вариант стека, использующий
// механизм модульности языка C++
#ifndef _ADVANCED_STACK_H
#define _ADVANCED_STACK_H

// Таким образом мы определяем функциональный
// контракт модульной реализации стека
namespace advanced_stack {
    struct Stack_impl;
    typedef struct Stack_impl AdvancedStack;

    // returns -1 if overflow
    int push(AdvancedStack *pstack, void *pobj);
    // returns NULL if empty
    void* pop(AdvancedStack *pstack);
```

```
    AdvancedStack* allocateStack();
    void freeStack(AdvancedStack* pstack);
}
#endif // _ADVANCED_STACK_H
```

```
// Файл advanced_stack.cpp
// Реализация контракта модульной реализации стека
#include "advanced_stack.h"

namespace advanced_stack
{
    const int MAX_STACK_SIZE = 200;
    struct Stack_impl {
        int top;
        void* content[MAX_STACK_SIZE];
    };

    Stack_impl* allocateStack()
    {
        Stack_impl *pstack = new Stack_impl();
        pstack->top = 0;
        return pstack;
    }

    void freeStack(Stack_impl *pstack)
    {
        delete pstack;
    }

    int push(Stack_impl *pstack, void *pobj)
    {
        if (pstack->top < MAX_STACK_SIZE) {
            pstack->content[(pstack->top)++] = pobj;
            return 0;
        } else {
            return -1; // overflow
```

```
        }
    }

    void* pop(Stack_impl *pstack)
    {
        if(pstack->top > 0) {
            return pstack->content[–(pstack->top)];
        } else {
            return 0; // underflow
        }
    }
}
```

В данной реализации решены обе упомянутые выше проблемы. Примером использования модульной организации стека может стать следующая программа:

```
#include <stdio.h>
#include "advanced_stack.h"

int main()
{
    using namespace advanced_stack;
    AdvancedStack *pstack = allocateStack();
    //pstack->top = 10; // Данная строка вызвала бы ошибку компиляции.
    char *letters [3] = {"one","two","three"};
    int i;
    push(pstack, (void*) (letters[0]));
    push(pstack, (void*) (letters[1]));
    push(pstack, (void*) (letters[2]));
    for (i = 0; i < 3; i++) {
        printf("\n%s\n", (char*)pop(pstack));
    }
    freeStack(pstack);
}
```

Если в будущем потребуется заменить реализацию стека, то это можно сделать изменив реализацию структуры *Stack_impl* и функции работы

с ней, реализованные в файле *advanced_stack.cpp*. Также можно объявить новое пространство имен (namespace), например *dynamic_stack*, в котором будет реализован стек изменяемого размера. В прежней программе, использующей модульную реализацию стека, достаточно будет заменить строку

```
using namespace advanced_stack;
```

на строку

```
using namespace dynamic_stack;
```

Это можно сделать при условии, что новое пространство имен имеет контракт, совпадающий с предыдущим. Опытные программисты на C могут заявить, что реализацию модульного подхода можно осуществить без использования механизма namespace языка C++, используя исключительно возможности препроцессора. Но на самом деле, дело обстоит не так просто как кажется на первый взгляд, и в дальнейшем, при детальном рассмотрении C++, мы увидим, какие преимущества несет использование внутренней языковой поддержки модульности в этом языке средствами namespace.

Таким образом, мы вплотную подошли к идее объединения данных и кода, предназначенного для обработки этих данных, в одну концептуально целостную единицу. Следующим шагом является объектная реализация стека.

```
//Файл object_stack.h
#ifndef _OBJECT_STACK_H
#define _OBJECT_STACK_H

// Объектная модель стека
class ObjectStack
{
private:
   static const int MAX_STACK_SIZE;
   int top;
   void **content;
```

```
public:
    //Вместо allocateStack(...), используется конструктор
    ObjectStack();
    //Вместо freeStack() используется деструктор
    ~ObjectStack();
    int push(void* pobj);
    void* pop();
};

#endif //_OBJECT_STACK_H

// Файл object_stack.cpp
#include "object_stack.h"

// Реализация объектной модели стека
const int ObjectStack::MAX_STACK_SIZE = 200;

ObjectStack::ObjectStack()
    :top(0)
{
    content = new void*[MAX_STACK_SIZE];
}

ObjectStack::~ObjectStack()
{
    delete[] content;
}

void* ObjectStack::pop()
{
    if(top > 0) {
        return content[–top];
    } else {
        return 0; // underflow
    }
}

int ObjectStack::push(void* pobj)
```

```
{
    if (top < MAX_STACK_SIZE) {
        content[top++] = pobj;
        return 0;
    } else {
        return -1; // overflow
    }
}
```

Примером использования такого стека может служить следующая программа:

```
int main()
{
    ObjectStack *pstack = new ObjectStack();
    char *letters [3]= {"one","two","three"};
    int i;
    //Доступ к членам данным в принципе невозможен извне
    //функций-членов класса ObjectStack
    // pstack->top = -10;  //Ошибка времени компиляции

    pstack->push ((void*) letters[0]);
    pstack->push ((void*) letters[1]);
    pstack->push ((void*) letters[2]);
    for (i= 0;i<3;i++) {
        printf("\n%s\n",(char*)pstack->pop());
    }
    delete pstack;
}
```

Пользоваться такой реализацией стека гораздо удобней, кроме того, она надежней предыдущих. Код лаконичен и прост для понимания.

Важным шагом к объектно-ориентированному программированию стало появление языков, поддерживающих объектный взгляд на написание программных систем.

Основной идеей объектного подхода является объединение данных и производимых над этими данными операций в одно концептуально

замкнутое понятие — *класс*. Данные класса не должны изменяться извне его, доступ к данным стоит осуществлять только через *функции-члены* (методы класса).

Программа, написанная на *объектном* языке, представляет собой совокупность объектов, каждый из которых принадлежит к определенному абстрактному типу данных (классу) и имеет интерфейс в виде набора методов для взаимодействия друг с другом (посылки сообщений).

1.2.2 Объектно-ориентированные языки

Насколько появление объектных языков и объектной модели явилось прямым развитием модульного подхода при написании программных систем, настолько и появление объектно-ориентированных механизмов явилось следующим шагом в развитии объектной модели.

Прежде, чем мы собственно обозначим, что же такое объектно ориентированный язык, давайте обратимся к известной всем со школьной скамьи системе классификации животного мира. Система эта иерархична. Рассмотрим такое существо как домашняя любимица Мурка. Прежде всего следует отметить, что Мурка является кошкой, причем, скорее всего, кошкой определенной породы, например сиамской. Таким образом, наша Мурка является объектом класса сиамских кошек.

Если полностью выписать классификационную принадлежность Мурки, то мы получим, что она принадлежит к сиамской породе домашних кошек, роду кошек, семейству кошачьих, отряду хищных, классу млекопитающих, подтипу позвоночных, типу хордовых, царству животных. Каждая из ступеней определяет ряд характеристик и особенностей, которыми обладает объект Мурка.

Тот факт, что данный объект принадлежит к классу сиамских кошек говорит нам о том, что он имеет определенный окрас, длину шерсти и особенности поведения. Кроме того, мы знаем, что при этом наша кошка Мурка является представителем вида домашних кошек, что определяет ее размер, сильно отличающийся от размера, к примеру, тигра. Как представитель семейства кошачьих, Мурка обладает опреде-

ленными особенностями анатомического строения; будучи хищником, она способна употреблять в пищу других животных; от класса млекопитающих она унаследовала способность выкармливать детенышей молоком и теплокровное строение организма и так далее. В терминах объектно-ориентированного подхода, мы говорим, что перечисленные классы образуют иерархию наследования, построенную на основе отношения «обобщение/специализация».

Классы не всегда образуют строго пирамидальную иерархию. Например, если мы определим класс домашних животных (представители которого могут жить в доме человека и взаимодействовать со своими хозяевами), то обнаружим, что домашние животные могут быть как теплокровными, так и хладнокровными. Свойство быть домашним животным не вписывается естественным образом в нашу иерархию живых организмов, а скорее играет роль признака (интерфейса), или «подмешивания». В дальнейшем это понятие будет рассмотрено более подробно.

Сложность окружающей действительности и многообразие объектов реального мира побуждает человека вырабатывать классификационные схемы, для того чтобы мыслить и оперировать объектами. Поэтому, совершенно естественным стало внедрение поддержки объектного подхода в языках программирования.

Наличие иерархических отношений наследования между абстракциями (классами и интерфейсами), экземплярами которых являются объекты в программе, является отличительной чертой объектно-ориентированного подхода в программировании.

Программа написанная на *объектно-ориентированном языке* представляет собой совокупность объектов, каждый из которых принадлежит к определенному абстрактному типу данных (классу), а классы образуют иерархию наследования.

1.2.3 Объектно-ориентированный анализ, дизайн и проектирование

Процесс программирования состоит в последовательной или итеративной реализации компонент программной системы средствами конкретного языка. Хорошо программировать — значит правильно и по назначению использовать средства, которые предоставляет язык.

Но для создания сложной программной системы требуется нечто большее, чем умение правильно программировать на том или ином языке. Основополагающей и определяющей составляющей процесса создания программной системы является этап проектирования. Проектирование, в отличие от программирования, основное внимание уделяет правильному и эффективному структурированию сложных систем.

Если объектно-ориентированное программирование основано на технологии представления программной системы в виде объектов, то совершенно естественным является определение объектно-ориентированного проектирования как проектирования на основе объектной модели. Буч дает следующее определение объектно-ориентированного проектирования(дизайна) — ООД:

Объектно-ориентированное проектирование — это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором — алгоритмами.

По аналогии с проектированием можно выделить отличительные особенности объектно-ориентированного анализа (ООА) по сравнению со структурным анализом, основанным на потоках данных в системе. Буч определяет ООА следующим образом:

Объектно-ориентированный анализ — это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области

Можно дать современные пояснения роли и сути ООА в процессе разработке программного обеспечения основываясь на методологии RUP(Rational Unified Process). Основными объектами, с которыми оперирует объектно-ориентированный анализ являются прецеденты взаимодействия (Use-Cases) и актеры (Actors). Причем, прецеденты взаимодействий рассматриваются в сфере отношений, схожих с отношениями возникающими между классами (расширения (extention), включения(inclusion), генерализации(generalization)).

1.3 Парадигмы программирования

Подводя итог рассмотрению истории развития языков и методологий программирования, следует упомянуть парадигмы различных методов построения программных систем, сформулированные Страуструпом [2, §2].

Процедурное программирование: *Реши какие требуются процедуры; используй наилучшие доступные алгоритмы.*

Модульное программирование: *Реши, какие требуются модули; разбей программу так, чтобы скрыть данные в модулях.(Принцип сокрытия данных)*

Объектное программирование: *Реши, какие требуются типы; обеспечь полный набор операция для каждого типа.*

Объектно-ориентированное программирование: *Реши, какие требуются классы; обеспечь полный набор операций для каждого класса; явно вырази общность через наследование.*

Обобщенное программирование: *Реши, какие требуются алгоритмы; параметризуй их так, чтобы они могли работать со множеством подходящих типов и структур данных.*

На этом заканчивается наш краткий экскурс в историю развития программных технологий. Теперь, после того как мы проследили весь путь становления объектно-ориентированного подхода, приступим к непосредственному изучению его отличительных особенностей и более детальному рассмотрению основополагающих принципов.

Глава 2

Составные части объектного подхода

Можно встретить различные понятия стиля программирования, здесь мы остановимся на следующем определении по Боборову и Стефику, которое приводит Буч[1, §2]. Стиль программирования — «это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле». Они выделили пять основных разновидностей стилей программирования, которые приведены ниже вместе с используемыми в них видами абстракций:

- процедурно-ориентированный — алгоритмы
- объектно-ориентированный — классы и объекты
- логико-ориентированный — цели, исчисление предикатов
- ориентированный на правила — правила «если-то»
- ориентированный на ограничения — инвариантные соотношения

Таким образом, для решения тех или иных задач могут подходить разные стили, и, соответственно, языки их поддерживающие, в зависимости от класса поставленной задачи. Для проектирования баз знаний более пригоден стиль ориентированный на правила (язык PROLOG), а для решения вычислительных задач — ориентированный на алгоритмы (например язык FORTRAN). Тем не менее, объектно-ориентированный стиль является наиболее широко применимым стилем программирования для основных классов задач в силу его близости к образу мыслей человека.

Концептуальной базой объектно-ориентированного стиля программирования является *объектная модель*, основывающаяся на четырех главных принципах:

- абстрагирование
- инкапсуляция
- модульность
- иерархия

Без следования любому из этих принципов модель не будет объектно-ориентированной. Кроме того, имеются еще три дополнительных принципа, которые не являются обязательными, но полезны.

- типизация
- парллелизм
- сохраняемость

Однако, принимая во внимание то, что в задачи курса входит изучение объектно-ориентированного программирования на языках C++ и Java, а Smalltalk в настоящее время выходит из употребления, данное классическое разделение требует корректировки, а именно, типизацию следует отнести к главным принципам. Типизация имеет очень

большое значение как в языке Java (сильная типизация), так и в C++, (типизация также является сильной, но существуют правила неявного преобразования типов, определяемых пользователем).

Следует также отметить, что в отличие от C++, в Java присутствует непосредственная языковая поддержка принципов параллелизма и сохраняемости.

Без следования этим принципам вы можете по прежнему программировать на объектно-ориентированных языках C++, Java, Smalltalk, Object Pascal, но под внешней красотой программы будет угадываться стиль Pascal, C или FORTRAN. Выразительные способности объектно-ориентированных языков будут при этом потеряны или искажены, и их использование не по назначению приведет к усложнению, а не облегчению решения поставленной задачи.

Рассмотрим в подробностях каждый из принципов объектной модели.

2.1 Абстрагирование

Принцип абстрагирования реализуется в ряде методов при решения задач с использованием объектной модели. В литературе можно встретить разные определения и расшифровки того, что понимается под термином *абстрагирование*.

Хоар дает следующее определение:«абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий».

Абстракция по Шоу — это «упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны».

Берзинс, Грей и Науман полагают: «идея квалифицируется как абстракция только, если она может быть изложена, понята и проанализи-

рована независимо от механизма, который будет в дальнейшем принят для ее реализации».

Если объединить эти точки зрения, получим определение абстракции по Бучу:

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Абстрагирование концентрирует внимание на внешних характеристиках объекта и позволяет отделить наиболее существенные особенности его поведения от менее существенных. Граница между существенными и несущественными с точки зрения разрабатываемой программной системы особенностями поведения объекта называется *барьером абстракции*. Последний определяется исходя из принципа минимизации связей, согласно которому интерфейс объекта должен описывать только существенные аспекты поведения. Так же следует соблюдать *принцип наименьшего удивления*. Следуя ему абстракция должна охватывать только поведение описываемого ей объекта, и, соответственно, не привносит сюрпризов и побочных эффектов, лежащих вне сферы ее применимости.

Выделение полного и достаточного набора абстракций при решении задачи с применением объектного подхода представляет собой главную задачу объектно-ориентированного проектирования.

Во время разработки программной системы могут появляться абстракции разных категорий, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, целесообразность использования которых сомнительна. Седвиц и Старк выделили следующий спектр абстракций:

- Абстракция сущности — Объект представляет собой полезную модель некой сущности в предметной области
- Абстракция поведения — Объект состоит из обобщенного множества операций

- Абстракция виртуальной машины — Объект группирует операции, которые вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
- Произвольная абстракция — Объект включает в себя набор операций, не имеющих друг с другом ничего общего

В качестве примера, рассмотрим автомобиль. Каждый человек имеет набор представлений и ассоциаций, связанных с этим понятием. Возьмем, например, двигатель. Это объект, который во взаимодействии с другими объектами (системой зажигания, карбюратором, коробкой передач) обеспечивает определенную функциональность. Эта функциональность заключается в способности создания крутящего момента вала, который передается коробке передач. При этом, двигатель использует при работе другие системы: систему зажигания и систему подачи топлива. В нашем случае двигатель является абстракцией, выделенной на основе своей наиболее существенной характеристики, а именно, способности превращать химическую энергию топлива в механическую энергию вращения распределительного вала. Продолжая анализ и декомпозицию автомобиля, можно выделить набор абстракций, элементами которого будут двигатель, коробка передач, система зажигания и т.д. Каждая из этих абстракций выделяется на основе своего интерфейса и функциональности.

2.2 Инкапсуляция

Следующим важным принципом, который непосредственно призван осуществлять поддержку абстрагирования, является *инкапсуляция*.

Процесс выделения ключевых абстракций при решении поставленной задачи в большей степени относится к объектно-ориентированному проектированию. Инкапсуляция же, напротив, является главенствующим принципом объектно-ориентированного программирования.

Абстрагирование направлено на наблюдаемое поведение (контракт) объекта, а инкапсуляция обеспечивает сокрытие его реализации. Барбара Лисков прямо утверждает, что «абстракция будет работать только вместе с инкапсуляцией». Практически же, это означает, что в любом классе присутствуют две части: интерфейс и реализация.

Интерфейс отражает внешнее поведение абстракции, специфицируя поведение всех объектов данного класса. Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта.

Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с другими объектами, а реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. Бритон и Парнас назвали такие детали «тайнами абстракции».

Буч определяет инкапсуляцию следующим образом:

Инкапсуляция — это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Современные объектно-ориентированные языки, такие как C++ и Java, имеют развитые средства поддержки принципа инкапсуляции. Эти средства выражаются в наличии механизмов управления доступом к методам и данным объекта [1].

Разумная инкапсуляция позволяет локализовать части реализации программной системы, которые могут подвергнуться изменениям. Например, по мере развития программного продукта, разработчики могут принять решение изменить внутреннее устройство тех или иных объектов с целью улучшения производительности или экономии памяти. При этом, важным преимуществом наличия механизмов разграничения доступа (механизмов инкапсуляции), является возможность вне-

[1] Широко известными являются применяемые для этих целей ключевые слова public, protected и private.

сения изменений в реализацию объекта (класса) без изменения других объектов (классов).

Если вернуться к рассмотрению автомобиля, то можно продемонстрировать принцип инкапсуляции на примере реализации акселератора. В одном случае механизм связи между педалью газа и заслонкой карбюратора может быть выполнен с помощью сложной системы тросов и рычагов, а в другом — с помощью датчика, сигнал с которого подается на электронное устройство управления карбюратором. В любом случае мы имеем дело с инкапсулированной реализацией этого механизма, при которой водителю нет дела до того как все устроено — он управляет подачей топлива через интерфейс, заключающийся в возможности нажатия на педаль акселератора.

2.3 Модульность

При разработке большой объектно-ориентированной программной системы одной из основных проблем становится наличие очень большого количества абстракций (классов и интерфейсов). Так как абстракции не существуют независимо, а взаимодействуют друг с другом, то ситуация усугубляется сложностью графа зависимостей между ними. Как одно из средств преодоления сложностей такого рода выступает принцип *модульности*.

Разбиение программы на модули не только позволяет бороться со сложностью, но и заставляет определять и хорошо документировать интерфейсы (или интерфейсные классы) между модулями, что в свою очередь, облегчает процесс объектной декомпозиции системы. Наличие четко определенных и хорошо задокументированных интерфейсов во многом способствует формированию цельного представления о программной системе и ее составляющих частях (подсистемах).

Использование модулей характерно не только для объектно-ориентированного программирования. В традиционном структурном программировании модули играют роль контейнеров в которые сгруппированы процедуры и данные, которыми они оперируют. Основная за-

дача при этом состоит в том, чтобы в один модуль попадали подпрограммы, использующие друг друга или изменяемые вместе.

В объектно-ориентированном программировании модули выполняют роль физических контейнеров и областей определения типов. Модули могут содержать определения классов, интерфейсов (Java), а также глобальные объекты и данные (C++).

Разные языки имеют разную поддержку модульности. Согласно Барбаре Лисков «модульность — это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями». В языке C++ это определение по прежнему может быть признано актуальным. В языке C и раннем C++, препроцессор и раздельная компиляция служили основными средствами поддержки рассматриваемого принципа. В процессе эволюции и стандартизации в C++ были введены пространства имен, которые являются средством логического разделения области видимости включаемых в них классов и данных.

Программы на Java, как правило, обладают более высокой степенью модульной гранулированности нежели программы на C++. Это связано с тем, что в Java принцип модульности с самого начала был положен в основу модели стандартной библиотеки и является основополагающим принципом при формировании физической модели размещения компонент Java программ. Этот язык имеет развитые средства поддержки принципа модульности и поощряет разработчиков на активное следование этому принципу.

Правильное разбиение программы на модули зачастую является почти столь же важной задачей, что и выбор правильного набора абстракций. Как и выделение ключевых абстракций, выделение модулей и определение зависимостей и взаимосвязей между ними — основная задач объектно-ориентированного проектирования. Ситуация зачастую осложняется тем, что уже на начальном этапе проектирования, когда выделены далеко не все абстракции, встает задача разбиения системы на модули.

Вероятность возникновения потребности во внесении изменений

в интерфейсные элементы модулей должна быть сведена к минимуму. Так как такие изменения могут существенным образом затрагивать части системы, использующие данный модуль.

Всегда нужно стремиться к минимизации интерфейсных частей модулей, естественно в разумных пределах, с целью уменьшения сцепления между разными частями системы. Модули должны объединять логически связанные абстракции. Следует помнить, что модуль является минимальной единицей переиспользования и размещения программной системы. Использование даже одного класса как правило означает зависимость от всего модуля.

Формализуя все выше сказанное, можно дать следующее определение модульности (по Бучу): *Модульность — это свойство системы, которая была разложена на внутренне связные, но слабо связные между собой модули.*

Возвращаясь к нашему автомобилю, можно выделить такие примеры модулей, как электрическая система (электропитание, освещение и т.п.), силовая установка (двигатель, карбюратор и др.) ходовая часть (трансмиссия, распределительный вал, колеса, тормозная система) и так далее.

2.4 Иерархия

Инкапсуляция и модульность позволяют в значительной степени упростить описание и процесс разработки системы абстракций, но зачастую, еще лучше позволяет бороться со сложностью принцип *иерархии*.

Инкапсуляция убирает из поля зрения внутренние содержание абстракций, модульность объединяет логически связанные абстракции в группы, а иерархия позволяет разделить абстракций на уровни, т.е. образует из абстракций иерархическую структуру. Буч определяет иерархию следующим образом:

Иерархия — это упорядочение абстракций путем расположения их по уровням.

В объектно-ориентированных системах используются два вида иерархических структур: структуры классов (иерархические отношения «is a») и структуры объектов (отношения вида «part of»).

Отношения вида «is a» реализуются в объектно-ориентированных языках с помощью *наследования или генерализации*.

Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует, а также расширяет и/или специализирует (уточняет) структуру и функциональный контракт одного или нескольких родительских классов[2]. Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение и функциональность от одного или нескольких суперклассов.

Часто подкласс достраивает или переписывает компоненты вышестоящего класса. В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии обобщение-специализация. Суперклассы, при этом, отражают наиболее общие, а подклассы — более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты.

При *одиночном* наследовании класс может иметь только одного родителя, но реализовывать несколько интерфейсов. При этом интерфейсы могут наследовать от нескольких родительских интерфейсов.

При *множественном* наследовании у класса может быть несколько суперклассов.

Структурно-агрегационные иерархические отношения («part of») мы рассмотрим, когда будем вести разговор об объектах и отношениях между ними.

---

[2]При этом класс-родитель является *суперклассом* для класса-потомка (*подкласса*)

2.5 Типизация

Понятие *типа* пришло в ООП из теории абстрактных типов данных. Дойч определяет тип, как «точную характеристику свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов»

Несмотря на то, что в некоторых языках существуют отличия между типом и классом, в современных объектно-ориентированных языках (в нашем случае C++ и Java) эти понятия неразделимы. Мы будем подразумевать под типами как примитивные типы (char, int, float), так и языковые средства абстракций, определяемых пользователем (классы, структуры и интерфейсы).

Типизация, как и инкапсуляция, больше относится к области объектно-ориентированного программирования, нежели к области объектно-ориентированного проектирования. Тем не менее, следует учитывать особенности языка при проектировании системы абстракций с тем, чтобы язык программирования обеспечивал соблюдение выработанных проектных решений.

Ниже приведено определение типизации по Бучу:

Типизация — это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.

Центральное место в типизации занимают механизмы согласования типов. Конкретный язык программирования может иметь сильный или слабый механизм типизации, или даже не иметь его вовсе. Сильно типизированные языки непреклонно следуют правилам использования типов. Так, в языках C++ и Java нельзя вызвать метод у объекта, если он не зарегистрирован в его классе, суперклассе или интерфейсе. Причем нарушение такого рода будет обнаружено уже на стадии компиляции. В Smalltalk, напротив, во время исполнения любое сообщение может быть послано любому объекту, при этом возникнет ошибочная ситуация, если объект не в состоянии обработать это сообщение (т.е. в его классе или суперклассе нет соответствующего метода).

Не смотря на то, что и C++ и Java являются сильно типизированными языками, механизмы согласования типов у них различны.

В Java определяемые пользователем типы (классы и интерфейсы) могут приводиться друг к другу только согласно иерархии наследования (ссылка на объект определенного класса может быть приведена к ссылке на совместный тип — суперкласс или интерфейс, реализуемый классом или суперклассом)[3].

В языке C++ существует механизм неявного приведения типов, которые могут быть не связаны друг с другом иерархией наследования (конструкторы классов от одного аргумента приводимого типа, не объявленные explicit). Также возможно приведение экземпляра класса к примитивным типам (например, можно определить для класса operator bool() с целью использования объекта этого класса в логических условиях).

И в C++, и в Java есть средства явного преобразования и проверки типов во время исполнения[4].

Сильная типизация заставляет разработчика соблюдать правила использования абстракций, поэтому она приносит неоценимую помощь в больших проектах. Однако у нее есть теневая сторона, заключающаяся в необходимости перекомпиляции всех подклассов, а также классов, использующих данный класс при внесении изменений в его интерфейс.

Вторым важным аспектом, который следует упомянуть в связи с сильной типизацией, является задача реализации контейнеров, и, в особенности, контейнеров, способных содержать разнотипные элементы. В Java контейнеры стандартной библиотеки хранят ссылки на хранимые объекты как на экземпляры класса Object, являющегося суперклас-

[3] В этом правиле есть одно исключение, а именно, возможность приведения любого объекта к типу String посредством неявного вызова метода toString() объявленного в корневом классе Object

[4] Следует заметить, однако, что если Java всегда гарантирует совместимость переменной ссылочного типа и объекта, на который эта переменная ссылается, то в C++ существуют средства полностью обойти или подавить правила типизации с помощью явных преобразований типа, таких как reinterpret_cast

сом для всех классов. Это заставляет разработчика заботиться об обратном приведении типов при извлечении объектов из контейнеров. В C++ задача решается с помощью шаблонов, но в этом случае в контейнере могут храниться только объекты типов, приводимых к типу параметра, заданного при инстанцировании шаблона.

Не следует путать понятия сильной (строгой) и статической типизации. Строгая типизация призвана обеспечить соответствие типов, а статическая типизация (так же называемая *статическим* или *ранним связыванием*), определяет время, когда имена (переменные ссылочного типа или указатели) связываются с типами адресуемых ими объектов. При статическом связывании тип адресуемого объекта, ровно как и тип результата любого выражения, определяется уже на стадии компиляции.

При *динамическом* (или *позднем*) *связывании* тип результата выражения или объекта, на который ссылается указатель или переменная ссылочного типа определяется во время исполнения программы. При этом указатель (или ссылка) могут ссылаться на объект любого типа совместимого по иерархии наследования с типом указателя или ссылки, соответственно. Данная особенность называется *полиморфизмом*: одно и то же имя может означать объекты разных типов, но имея общего предка, все они имеют и общее подмножество операций (контракт), которые можно над ними выполнить. Противоположность полиморфизму называется *мономорфизмом*, который характерен для языков с сильной типизацией и статическим связыванием.

Формулировка, приведенная выше, определяет «виртуальный полиморфизм» (результат взаимодействия наследования и динамического связывания), наряду с которым различают еще и «параметрический полиморфизм» — свойство языков позволяющее объявлять и использовать функции с одинаковыми именами, но отличающиеся типами и/или количеством своих аргументов.

Процесс выбора конкретной функции(метода) во время исполнения, в зависимости от типа объекта или аргументов, называется *разрешением* полиморфизма.

Виртуальный полиморфизм — одно из самых мощных и эффективных средств объектно-ориентированного программирования, отличающее его от программирования на основе абстрактных типов данных.

Таким образом, C++ и Java являются сильно типизированными языками с динамическим связыванием.

2.6 Параллелизм

Параллелизм в объектно-ориентированном программировании, как и другие принципы, возник не на пустом месте, а явился результатом привнесения объектной идеи в теорию параллельных вычислений.

Необходимость в разработке теории параллельных вычислений возникла давно. Задачи автоматизации очень часто требуют одновременной обработки нескольких событий. При решении задач, связанных с большой вычислительной трудоемкостью, очень часто бывает недостаточно мощности одного процессора и приходится искать решение основанное на распараллеливании вычислений на многопроцессорных системах.

Существует очень много классов задач, где возможность использования параллелизма может сильно улучшить характеристики разрабатываемой информационной системы. Фундаментальным понятием и единицей действия в теории параллельных вычислений является *поток управления*.

Традиционно, многопоточность бывает двух видов — *тяжелая* (основанная на процессах в операционной системе) и *легкая* (основанная на потоках в рамках одного процесса).

Потоки управления при тяжелой многопоточности существуют каждый в своем отдельном адресном пространстве в рамках операционной системы (с каждым процессом ассоциируется ровно один поток управления). Переключение контекстов исполняемых потоков при операциях межпроцессного взаимодействия в таких системах, как правило, сопряжено с большими накладными расходами.

При легкой многопоточности потоки внутри процесса разделяют общее адресное пространство. При этом возникает проблема обеспечения конкурентного доступа к данным из разных потоков. Программа, работающая в системе с легкой многопоточностью, представляет из себя совокупность из нескольких потоков управления и точек синхронизации. Точки синхронизации обеспечивают целостность совместно используемых данных и взаимодействие потоков между собой.

В объектно-ориентированной системе, использующей принципы и средства параллелизма, потоки управления представляются активными объектами, которые являются инициаторами всех происходящих в системе действий. Таким образом, объекты делятся на активные (являющиеся своего рода отдельными вычислительными центрами) и пассивные, на которые направленно воздействие активных объектов.

Параллелизм дает возможность объектам действовать одновременно. На основе этой идеи, Буч дает следующее определение параллелизма:

Параллелизм — это свойство, отличающее активные объекты от пассивных.

Как только в систему введен параллелизм, сразу возникает вопрос о синхронизации активных объектов друг с другом и последовательными пассивными объектами. Например, если два объекта посылают сообщения третьему, то должен существовать какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет «растерзан на части» несколькими независимыми процессами.

Параллелизм может обеспечиваться как средствами языка (Java, Ada, Smalltalk), так и специально написанными библиотеками, которые используются при написании параллельной системы с использованием языков, не имеющих встроенной поддержки этого принципа (C++).

Следует, однако, отметить, что даже если язык имеет встроенную

поддержку параллелизма, все равно, необходимо учитывать устройство и особенности организации многопоточности в конкретных операционных системах, под управлением которых планируется работа разрабатываемой программы.

2.7 Сохраняемость

Любой объект или данные в программной системе существуют во времени и пространстве (памяти ЭВМ). Одни объекты существуют только как промежуточные результаты вычисления выражения, другие могут вообще пережить программу, оставаясь сохраненными в базе данных. Этот спектр подразделяется на следующие уровни:

- Промежуточные результаты вычислений выражений
- Локальные переменные и объекты в блоках, а также, при вызове процедур и функций (как правило эти данные существуют на стеке)
- Статические переменные классов, а также, глобальные переменные и объекты в динамической памяти
- Данные, сохраняемые между сеансами выполнения программы
- Данные, сохраняемые при переходе на другую версию программы.
- Данные, переживающие программу

В языках программирования традиционно можно встретить поддержку верхних трех уровней этого спектра. Три нижних уровня, как правило, входят в компетенцию баз данных. Из этого правила существуют интересные исключения.

Бывают случаи, когда зоны влияния баз данных и языков расширяются за пределы их традиционных сфер ответственности. Такими примерами служат объектные базы данных, которые появились в результате союза объектных концепций и традиционных баз данных. Следующим примером могут служить языки, имеющие встроенную поддержку сохраняемости (Java[5]).

С проблемой сохраняемости связана не только проблема сохранения данных (состояния объектов), но и проблема сохранения информации о структуре этих данных, что при сохранении объектов приводит к сохранению классов в объектно-ориентированных базах данных. Серьезной проблемой становится задача миграции данных в ООБД при изменениях в структурах их классов.

Не смотря на развитие теории объектно-ориентированных баз данных, традиционные реляционные базы данных продолжают занимать лидирующее положение на рынке систем хранения данных. Этому служат множество причин:

- Реляционные базы данных обладают значительной универсальностью, хранящаяся в них информация может быть использована различными по своей идеалогии программными системами: как объектно-ориентированными, так и написанными с использованием процедурного или другого подхода.

- Реляционные базы данных обладают хорошими характеристиками в плане производительности, надежности и транзакционности.

- В мире уже накоплено огромное количество информации в реляционных базах данных, и, зачастую, крупные базы переживают не одно поколение использующих их программных систем.

Для обеспечения взаимодействия объектно-ориентированных систем и реляционных баз данных разработано множество средств ото-

[5] В Java сохраняемость объектов и контроль версий классов реализованы на уровне языковой библиотеки.

бражения объектно-ориентированных структур на реляционные базы данных для обеспечения сохраняемости (так называемые OR-mappers). Существуют библиотеки, предоставляющие объектно-ориентированный интерфейс для манипуляций с данными в RDBMS.

Кроме сохранения объектов во времени, актуальным является вопрос возможности транспортировки (переноса) объектов из одной среды исполнения в другую, возможно находящуюся на удаленном компьютере. Речь идет, прежде всего, о построении распределенных систем. Для решения этой задачи могут применяться специальные технологии на подобие CORBA, Microsoft COM+, SOAP, Java RMI.

Итак, можно привести следующее определение сохраняемости (по Бучу):

Сохраняемость — это способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.

На этом мы закончим рассмотрение основных принципов объектно-ориентированного программирования и перейдем к рассмотрению природы и основных свойств объектов, как кирпичиков для построения информационных систем.

Глава 3

Объекты

3.1 Что такое объект с точки зрения ООП

Что есть объект? Как объекты реального мира связаны с объектами в объектно-ориентированной системе? Какими свойствами обладают объекты в программировании? И, наконец, как объекты взаимодействуют между собой? Что такое роль и как это понятие связано с объектом? Все эти вопросы могут возникнуть у человека, начинающего изучать объектный подход в программировании. В данной части нашего повествования мы попытаемся дать ответы на эти вопросы и выработать понимание того, что такое объект в объектно-ориентированной системе.

Понятие объекта выделено человеком уже довольно давно. Еще древние философы пытались дать ответ на вопрос, что же такое объект.

Есть замечательный пример про ребенка и мяч, который приведен в книге Буча [1, §3]. Этот пример показывает, как эволюционирует понятие объекта в сознании человека.

Способностью к распознаванию объектов физического мира человек обладает с самого раннего возраста. Ярко окрашенный мяч привлекает внимание младенца, но, если спрятать мяч, младенец, как правило, не пытается его искать: как только предмет покидает поле зре-

ния, он перестает существовать для младенца. Только в возрасте около года у ребенка появляется представление о предмете: навык, который незаменим для распознавания. Покажите мяч годовалому ребенку и спрячьте его: скорее всего, ребенок начнет искать спрятанный предмет. Ребенок связывает понятие предмета с постоянством и индивидуальностью формы независимо от действий, выполняемых над этим предметом.

По мере своего развития и обучения, человек учится мыслить не только в терминах реальных и осязаемых объектов, но и в терминах абстрактных сущностей, которых может и не существовать в реальном мире. С точки зрения человека объектом может быть:

- осязаемый и (или) видимый предмет;
- нечто, воспринимаемое мышлением;
- нечто, на что направлена мысль или действие.

Термин *объект* в контексте программирования информационных систем впервые появился в языке Simula, разработанном для моделирования окружающей действительности. В простейшем случае объекты выделяются на основе реальных сущностей в предметной области, к которой относится разрабатываемая программная система.

Для более подробного рассмотрения понятия объекта читателю лучше обратиться к книге Буча[1]. Здесь же мы сформулируем и рассмотрим только основные ключевые идеи.

При проектировании информационной системы с использованием объектного подхода разработчик выделяет объекты на основе их наиболее существенных характеристик в предметной области. В общем случае, программные объекты обладают далеко не полным набором свойств и характеристик в сравнении с моделируемыми ими объектами реального мира.

Далеко не все объекты, существующие в информационной системе, имеют своих аналогов в реальном мире. Зачастую, в процессе проектирования и разработки появляется большое количество вспомогательных объектов, призванных обеспечить среду существования для

бизнес-объектов. Понятия *бизнес-логики* и *бизнес-объекта* служат для обозначения той части программной системы, которая непосредственно моделирует процессы реальной предметной области(задачи), для решения или автоматизации которой разрабатывается информационная система[1].

Важным отличием объекта от экземпляра абстрактного типа данных является то, что объект обладает *состоянием*, *поведением* и *уникальностью (идентичностью)*. Ниже приводится определение объекта по Бучу:

Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

3.2 Состояние

Рассмотрим хрестоматийный пример из книги Буча [1, §3] иллюстрирующий работу автомата по продаже газированной воды.

Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки, автомат выдает выбранный напиток. Что произойдет, если сначала будет нажата кнопка выбора напитка, а потом уже опущена монета? Большинство автоматов при этом просто ничего не сделают, так как пользователь нарушил их основные правила.

Другими словами, автомат играл роль (ожидание монеты), которую пользователь игнорировал, нажав сначала кнопку. Или предположим, что пользователь автомата не обратил внимание на предупреждающий сигнал «Бросьте столько мелочи, сколько стоит напиток» и опустил в автомат лишнюю монету. В большинстве случаев автоматы не дружественны к пользователю и радостно заглатывают все деньги.

В каждой из таких ситуаций мы видим, что поведение объекта определяется его историей: важна последовательность совершаемых над

[1] Наряду с бизнес-объектами существуют объекты, формирующие пользовательский интерфейс, систему ввода-вывода и так далее

объектом действий. Такая зависимость поведения от событий и от времени объясняется тем, что у объекта есть внутреннее состояние. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация - это набор воспринимаемых монет и запас напитков. На основании этого Буч дает следующее определение:

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

Одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно — существенная характеристика торгового автомата. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет. Сумма увеличивается по мере опускания монет в автомат и уменьшается, когда продавец забирает деньги из автомата. В некоторых случаях значения свойств объекта могут быть статическими (например, заводской номер автомата), поэтому в данном определении использован термин «обычно динамическими».

Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Мы говорим «как правило», потому что в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира.

Все свойства имеют некоторые значения, причем, существенным является то, что эти свойства могут быть как простыми количественными характеристиками (иметь числа в качестве значений), так и ссылками на другие объекты. Если говорить об автомате, то количественной характеристикой является сумма опущенной в автомат мелочи. С другой стороны состояние автомата определяется также наличием на-

питков, которые являются самостоятельными объектами, отличными от торгового автомата (их можно пить, а автомат нет, и совершать с ними иные действия).

В информационной системе тот факт, что объект имеет состояние, означает, что значения (количественные или ссылочные), образующие это состояние, должны где-то храниться. Как правило, эти значения хранятся в оперативной памяти, и, таким образом, объект занимает место в пространстве (памяти компьютера).

3.3 Поведение

Объекты не существуют изолированно, а взаимодействуют друг с другом, реализуя поведение. Здесь уместно привести известную аллегорию про самолет: *«Самолет представляет собой совокупность вещей, каждая из которых по отдельности стремиться упасть на землю, но вместе, во взаимодействии, они преодолевают эту тенденцию»*.

Буч определяет поведение следующим образом: *Поведение — это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений*. Иными словами, поведение объекта — это его наблюдаемая и проверяемая извне деятельность.

Взаимодействие объектов может быть описано в терминах операций. Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующие действия или реакцию. Например, в контракте нашей объектной реализации стека существуют операции **push** и **pop**, которые служат для управления стеком.

Вместо концепции операций (уходящей своими корнями в процедурное прошлое), можно использовать объектно-ориентированную концепцию передачи и обработки сообщений. При этом мы говорим, что какой-нибудь объект может передать нашему стеку сообщение **push** или **pop**, а наш стек воспримет и обработает полученное сообщение. Можно сказать что операция — суть реакция на сообщение.

В языках C++ и Java концепции передачи сообщений и операций реализуются через один и тот же механизм (вызов методов или функций-членов классов), поэтому, по своей сути, являются эквивалентными.

Мы будем отдавать предпочтение концепции передачи сообщений. В самом деле, коль скоро мы говорим об инкапсуляции, как основном средстве объектно-ориентированного программирования, то совершенно естественным будет подход к дизайну классов на основе сообщений. При этом функциональный контракт класса (и его предков) определяет набор сообщений, которые объект способен воспринять, а реализация реакции объекта на полученное сообщение остается инкапсулированной внутри реализации иерархии классов к которой принадлежит объект.

Реакция объекта на полученное сообщение (т.е. поведение объекта) зачастую зависит не только от самого сообщения (вызванного метода), но и от текущего состояния объекта. Например, если в автомат по продаже напитков опустить недостаточную сумму денег, то после выбора напитка, скорее всего, ничего не произойдет. Если же денег достаточно, то автомат выдаст требуемый напиток. Таким образом, реакция автомата на требование (сообщение) о выдачи напитка различна в зависимости от состояния параметра, определяющего текущую сумму мелочи, опущенной в автомат.

Итак, поведение объекта определяется выполняемыми над ним операциями (переданными ему сообщениями) и его текущим состоянием. Причем, некоторые операции могут изменить внутреннее состояние объекта (или в терминах концепции сообщений, реакция на некоторые сообщения ведет к изменению состояния объекта). В результате мы подошли к следующему важному утверждению:

Состояние объекта представляет суммарный результат его поведения.

3.3.1 Классификация методов объектов

Как уже было сказано, поведение объекта реализуется через методы его класса (или классов). Всего можно выделить пять видов методов (операций):

- Конструкторы — Методы создания объекта и/или его инициализации
- Деструкторы — Методы, освобождающие состояние и ресурсы объекта и/или разрушающие сам объект
- Селекторы — Методы, считывающие но не меняющие состояние объекта
- Модификаторы — Методы, способные изменить состояние объекта
- Итераторы — Методы, позволяющие организовать доступ к частям объекта-контейнера в строго определенной последовательности.

Все эти виды имеют разную поддержку в объектно-ориентированных языках. Конструкторы и деструкторы дают контроль над процессами создания/уничтожения объектов, и, в большей степени являются «операциями». Селекторы и модификаторы являются наиболее часто вызываемыми методами объекта и могут использоваться как обработчики сообщений. Итераторы обычно стоят несколько обособленно, причем, зачастую для реализации итеративного доступа к контейнеру используются специальные объекты. Мы рассмотрим данный механизм когда будем углубленно изучать конкретные языки.

В языке C++ конструкторы и деструкторы играют очень важную роль, так как в них, как правило, происходит выделение и высвобождение памяти под составные части объекта. Небрежное написание кода этих методов приводит к всевозможным проблемам связанным с

падением производительности и утечкой ресурсов. Всегда нужно помнить об особенностях, связанных с конструкторами и деструкторами по умолчанию, конструктором копии и оператором присваивания (**operator** =). Не смотря на то, что средства поддержки механизмов создания и удаления объектов могут показаться относительно сложными для начинающих программистов на C++, эти средства предоставляют полный контроль за управлением памятью в информационной системе. Всегда можно предсказать когда объект будет создан, сколько ресурсов он займет и когда будет уничтожен[2]. С этой точки зрения C++ позволяет писать максимально высокопроизводительные и надежные системы. Необходимо так же помнить, что конструкторы от одного аргумента в C++ используются для приведения типов.

Кроме этого, в C++ присутствует языковой механизм защиты данных объекта в методах-селекторах. Так, функции-члены, которые являются селекторами, должны быть объявлены со спецификаторами доступа **const**. Это обеспечивает защиту данных (состояния) объекта от изменений внутри этих методов и позволяет вызывать эти функции на константных объектах. Для объявления встраиваемых функций используется спецификатор **inline**. Использование данного механизма может существенно увеличить производительность[3].

Для C++ будет справедливым утверждение, что все методы — операции, но не все операции — методы: некоторые из них могут представляют собой свободные подпрограммы.

Язык Java, в отличие от C++, будучи платформенно-независимым и имеющий развитую систему безопасности, не предоставляет программисту средств управления памятью, выделяемой при размещения объектов. Этот язык имеет строгую систему типов и не имеет переопределяемых операторов, что существенно минимизирует риск связанный с небрежным написанием кода программ. Наличие сборщика

[2]Мы не будем сейчас здесь заострять внимание на проблеме фрагментации памяти

[3]Следует сказать, что современные компиляторы как C++, так и Java, во многом берут задачу оптимизации на себя и могут исправлять наиболее распространенные недостатки небрежно написанного кода программы

мусора, с одной стороны, помогает справляться с проблемой утечки ресурсов; с другой стороны, не позволяет точно предсказать производительность Java приложений[4]. В Java есть конструкторы, в том числе и конструкторы по умолчанию, но нет явных деструкторов. Для обеспечения операций клонирования в языке и языковой библиотеке есть специализированный механизм, а для освобождения ресурсов объекта перед удалением используется специализированный метод finalize(). Для высвобождения ресурсов при работе распределенных Java приложений, в частности основанных на использовании Java RMI, существует механизм удаленной сборки мусора.

Следует упомянуть, что в Java нет свободных подпрограмм (функций). Для реализации эквивалентной функциональности можно (и нужно) пользоваться статическими методами классов-утилит[5]. Желательно использовать подобный подход и в C++, так как он способствует более легкой расширяемости программного продукта.

К сожалению, в Java нет средств языковой поддержки защиты данных при написании функций-селекторов[6]. Внимательный читатель может обнаружить, что в таких классах стандартной библиотеки Java, как String, Integer, Float и подобных, попросту отсутствуют методы позволяющие изменить содержимое объекта.

3.3.2 Роли объектов

Совокупность всех методов и свободных процедур, относящихся к конкретному объекту, образуют *протокол* этого объекта. Протокол, таким образом, определяет поведение объекта, охватывающее все его ста-

[4]Современные реализации Java машины, такие например как Java HotSpot версии 1.3.1 от фирмы Sun, имеют достаточно мощный адаптивный алгоритм реализации сборщика мусора

[5]Класс-утилита это класс, в котором присутствуют только статические методы выполняющие роль обычных алгоритмов или подпрограмм

[6]Это связано с особенностями платформы, и, в частности, с тем, что в Java нет объектов, создаваемых в автоматической памяти (на стеке). Переменные могут быть только встроенных или ссылочных типов.

тические и динамические аспекты. Коль скоро мы стремимся не использовать свободных функций, то будем дальше считать, что протокол объекта образуется только совокупностью методов классов, к которым объект принадлежит.

Для сложных объектов очень полезным бывает разделение контракта на несколько отдельных абстракций (интерфейсов) с целью выделения различных, независимых с точки зрения групп внешних объектов, областей ответственности[7]. Обычно такая ситуация идентифицируется следующим образом: если две разные абстракции используют непересекающиеся подмножества методов объекта, то следует эти подмножества выделить в отдельные интерфейсы, которые реализует класс этого объекта. В этом случае мы будем говорить, что объект играет разные роли по отношению к этим абстракциям, его использующим[8]. Можно сказать, что состояние и поведение объекта определяют исполняемые им роли, а те, в свою очередь, необходимы для выполнения ответственности данной абстракции.

Как правило, проектирование системы абстракций начинается с идентификации ролей, которые может играть тот или иной объект в системе. В последующем, роли всех объектов в системе сводятся в иерархические структуры и группируются по пакетам и подсистемам.

3.3.3 Связь объектов и автоматов, активные и пассивные объекты

Так как большинство интересных объектов обладает состоянием и их поведение зависит от предистории, то многие объекты естественным образом могут быть представлены в виде конечных автоматов.

---

[7] В объектно-ориентированном дизайне данный принцип носит название *принципа сегрегации (разделения) интерфейсов*

[8] Еще раз разъясним ситуацию: классы и интерфейсы являются абстракциями; объект является экземпляром класса, который принадлежит к иерархии классов и может реализовывать несколько интерфейсов. При этом, в итоге, объект будет принадлежать ко множеству абстракций, образованных его классом, его суперклассами, а также интерфейсами, которые реализуют его класс и суперклассы.

Кроме того, вспоминая принцип параллелизма, мы приходим к пониманию сути активных и пассивных объектов. Активный объект имеет свой поток управления, а пассивный — нет. Активный объект в общем случае автономен, то есть он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы - источники управляющих воздействий. Если система имеет несколько потоков управления, то и активных объектов может быть несколько. В последовательных системах обычно в каждый момент времени существует только один активный объект, например, главное окно, диспетчер которого ловит и обрабатывает все сообщения. В таком случае остальные объекты пассивны: их поведение проявляется, когда к ним обращается активный объект. В других видах последовательных архитектур (системы обработки транзакций) нет явного центра активности, и управление распределено среди пассивных объектов системы.

3.4 Идентичность

Буч приводит следующее определение по Хошафяну и Коуплэнду:

Идентичность - это такое свойство объекта, которое отличает его от всех других объектов

Они отмечают, что «в большинстве языков программирования и управления базами данных для различения временных объектов их именуют, тем самым путая адресуемость и идентичность. Большинство баз данных различают постоянные объекты по ключевому атрибуту, тем самым смешивая идентичность и значение данных». Источником множества ошибок в объектно-ориентированном программировании является неумение отличать имя объекта от самого объекта.

Идентичность (уникальность) объекта является очень важной его характеристикой. Практически всегда объекты можно различить хотябы потому, что они занимают отдельное место в памяти ЭВМ. При этом мы имеем дело с уникальностью, основанной на адресуемости.

Разные языки могут предоставлять различные средства для поддержки идентичности объектов. Например, в C++ можно принимать решение об идентичности как на основе конкретного адреса размещения объекта, так и на основе результата определяемого пользователем оператора **operator==**. В Java мы можем воспользоваться значением переменной ссылочного типа, которая ссылается на объект, а можем использовать метод **equals()** определенный в классе Object[9].

Иногда, для идентификации объекта заводят специальное поле, в котором содержится уникальный ключ. Особенно часто такой метод встречается в приложениях, работающих с базами данных. В подобной ситуации всегда нужно отдавать себе отчет в том, насколько оправдано включение такого ключевого поля в класс объекта. Например, если для идентификации сотрудника используется табельный номер, то такой подход вполне оправдан, так как подобный ключ существует в реальной жизни и является существенной характеристикой объекта. В противном случае, данное решение выглядит «некрасиво» с точки зрения объектного подхода, так как порождает неявные связи между абстракциями. Для идентификации объекта во время исполнения программы, в большинстве случаев, достаточно использовать ссылку или указатель на объект. Потребность в ключевых полях возникает при решении задачи сохранения иерархии объектов в неструктурированном хранилище данных, когда, впоследствии, необходимо восстанавливать взаимосвязи между объектами.

В любом случае, всегда лучше возложить обязанности по сохранению/восстановлению взаимосвязей на специально выделенный уровень сохранения, нежели вносить поля-ключи в объекты бизнес-логики.

[9]Здесь следует сказать, что по умолчанию решение об эквивалентности объектов в методе equals() класса Object принимается на основе адреса размещения объекта. Этот метод можно перегрузить, взяв контроль за уникальностью на себя. Так, например, классы String, Float, Integer, Double и им подобные перегружают этот метод и принимают решение об идентичности объектов на основе их содержимого, а не адреса.

3.5 Жизненный цикл объекта

Как мы уже говорили, объект существует в пространстве уже хотя-бы потому, что для сохранения его состояния требуется оперативная память. Мы не будем сейчас рассматривать ситуацию, когда объект существует в хранилище данных в некотором «сериализованном» виде, так как в конечном счете объект должен оказаться в оперативной памяти для того, чтобы с ним можно было работать. Поэтому, будем считать, что с точки зрения программы на этапе исполнения объект рождается тогда, когда под него выделяется память и происходит инициализация его состояния. Объект заканчивает свой жизненный путь тогда, когда высвобождаются занятые им ресурсы, и память возвращается в систему для дальнейшего использования.

В разных объектно-ориентированных языках существуют разные механизмы управляющие рождением и уничтожением объектов. Вкратце перечислим основные моменты, присущие C++ и Java.

Как уже было сказано, в C++ для управления процессом создания и уничтожения объектов используются конструкторы, деструкторы, а также операторы **operator new** и **operator delete**.

Объект в программе на C++ может быть создан, как:

- Именованный объект в автоматической памяти (на стеке), который создается каждый раз, когда поток исполнения проходит через его декларацию и уничтожается, когда поток покидает блок, в котором данный объект объявлен. Этот случай включает в себя ситуацию, когда создается копия объекта-параметра функции (метода) при передаче аргумента по значению.

- Объект в свободной памяти (куче), который создается с помощью оператора **new** и уничтожается с помощью оператора **delete**.

- Нестатическое часть-поле другого объекта, которое создается и уничтожается одновременно с объектом, его содержащим.

- Элемент массива, который создается и уничтожается одновременно с самим массивом.
- Локальный статический объект, который создается один раз, когда поток исполнения проходит через его декларацию, и уничтожается, когда программа завершает свою работу.
- Глобальный объект, член namespace, или статический член класса. При этом объект создается в момент загрузки и запуска программы и уничтожается, когда программа завершает свою работу.
- Промежуточный результат при вычислении выражений, который создается в процессе вычисления выражения и уничтожается по завершении выражения. Этот случай включает в себя ситуацию, когда объект является возвращаемым результатом функции (метода) при возврате по значению.
- Объект, размещенный в памяти, выделяемой с помощью определенной пользователем функцией-аллокатором.
- Член объединения[10].

Как видно из приведенного списка, C++ является довольно сложным языком в вопросах управления жизненным циклом объектов.

В отличие от C++, в Java меньше ситуаций, когда возможно создание объекта:

- Объект может быть создан при помощи оператора **new**.
- Объект может быть создан как копия другого объекта с помощью метода clone() для классов, реализующих интерфейс Cloneable.

[10]Объединения обычно используются при низкоуровневом программировании, и, поэтому, не следует использовать объекты-члены объединения в объектно-ориентированных программах. Компилятор не может обеспечить корректный жизненный цикл объекта, являющегося членом объединения

- Объект класса String может быть создан как промежуточный результат выполнения операции конкатенации строк с использованием оператора +.
- «Immediately enclosing instance of»[11] объект создается во время вычисления «primary» конструкции при создании экземпляра класса, родитель которого является внутренним (inner) классом другого класса.
- Объект может быть создан с помощью вызова метода newInstance() объекта класса Class.

Во всех случаях объект создается в свободной памяти, и его время жизни регулируется сборщиком мусора, который уничтожает объекты, на которые больше нет ссылок. Перед уничтожением объекта у него вызывается метод finalize().

3.6 Отношения между объектами

Так как только в процессе взаимодействия объектов реализуется система, то следующим нашим шагом будет рассмотрение вопросов, касающихся взаимоотношений между объектами.

Все отношения между объектами могут быть сведены к двум типам:

- *Ассоциация (связь)* — отношение, позволяющее реализовать взаимодействие клиент-сервер.
- *Агрегация (композиция)* — отношение, служащее для определения понятия целое-часть и организации иерархий объектов.

Рассмотрим подробно каждый из этих видов взаимодействий.

[11] Для данного термина не известно удачного русского перевода.

3.6.1 Взаимодействие клиент-сервер

По Румбаху, *ассоциация или связь — есть физическое или концептуальное соединение между объектами*. Практически это означает, что для того, чтобы объекты могли взаимодействовать друг с другом, между ними должна существовать прямая или косвенная связь. Используя эту связь, объект, являющийся *клиентом* или инициатором взаимодействия, может вызвать метод (послать сообщение) у *сервера* — объекта являющегося адресатом взаимодействия, т.е. объект-клиент запрашивает некоторый сервис у объекта-сервера.

Семантика

С точки зрения языка программирования, прямая связь означает, что сервер достижим (адресуем) со стороны клиента напрямую с использованием переменной, ссылки или указателя, а так же с использованием «глобальной» видимости. Косвенная связь может быть реализована с помощью ассоциативного контейнера, ключа или специального механизма, который может обеспечить в конечном счете доступ от клиента к серверу.

Несмотря на то, что взаимодействие является однонаправленным, данные могут передаваться в обоих направлениях: от клиента данные передаются через аргументы при вызове метода, а от сервера данные передаются в возвращаемом методом значении. Даже если между клиентом и сервером существует двусторонняя связь (клиент и сервер могут иметь ссылочные переменные или указатели, адресующие друг-друга), эта связь все равно, в конечном счете, представляется двумя отдельными связями.

При глобальном рассмотрении взаимодействий, можно выделить три основные категории объектов:

- *Объекты-актеры* — объекты, которые воздействуют на другие объекты, но сами никогда не подвергаются воздействию; это, своего рода, источники всех взаимодействий в системе.

- *Объекты-серверы* — объекты, которые могут только подвергаться воздействию со стороны других объектов, но никогда не выступающие в роли взаимодействующих объектов; это, своего рода, конечные точки взаимодействий в системе.

- *Объекты-агенты* — объекты, которые выступают как в активной, так и в пассивной роли; в конечном счете, они являются переносчиками взаимодействий в системе[12].

При локальном рассмотрении взаимодействий, «актеры» всегда выступают в роли клиентов, «серверы» всегда выступают в роли серверов, а «агенты» при взаимодействии для одних объектов являются серверами, а для других — клиентами. При этом, как правило, агент выполняет операции в интересах какого-либо актера или агента.

При рассмотрении взаимодействий клиент-сервер нужно упомянуть еще два важных понятия — *видимость* и *синхронизацию*

Видимость

Для того, чтобы объект-клиент мог вызвать метод объекта-сервера, во-первых, необходимо, чтобы сервер был «видим» для клиента, и во-вторых, клиент должен знать о контракте, предоставляемом сервером. Информация о контракте задается типом (классом или интерфейсом, т.е. абстракцией) переменной (ссылки, указателя) через которую сервер доступен клиенту. В этом смысле, как мы увидим, при глубоком рассмотрении классов, эта зависимость выражается в семантике абстракций (классов) клиента и сервера.

Всего существует четыре способа обеспечения видимости:

- Сервер имеет глобальную видимость по отношению к клиенту.

- Сервер передан клиенту в качестве параметра операции (метода).

[12] В англоязычной литературе используются термины Actor, Server и Peer соответственно.

- Сервер является частью клиента.
- Сервер локально порождается клиентом в ходе выполнения какой-либо операции.

В C++ видимость может быть обеспечена с помощью именованных типизированных переменных, ссылок или указателей, которые могут быть глобальными, локальными, статическими, или являться параметрами методов; либо с помощью типизированных вычислимых значений lvalue. При этом, компилятор гарантирует семантическое соответствие вызываемого метода типу адресуемого объекта (строгая типизация).

В Java видимость обеспечивается с помощью ссылки на объект-сервер. При этом, ссылка на объект может храниться в переменной ссылочного типа, может быть получена как результат вызова метода (в том числе и конструктора), может является результатом вычисления выражения[13], либо используется неявно при вызове метода сервера, который является «immediately enclosing instance of» объекта-клиента.

Для пояснения последнего случая, рассмотрим пример:

```
class Outer {
   void doIt() {
      doAnother();
   }
   void doAnother() {
   }
   class Inner {
      void doInner()
      {
         doIt();
      }
   }

   private Inner theInner = new Inner();
```

[13]Это касается только ссылок на объекты класса String при вычислении выражений конкатенации строк с использованием оператора +

```
    public static void main(String args[]) {
        Outer theOuter = new Outer();
        theOuter.theInner.doInner();
    }
}
```

Объект theOuter является «immediately enclosing instance of» объекта theOuter.theInner ранга (степени) 1. При этом происходит прямой вызов метода doIt() объекта theOuter во время выполнения метода doInner() объекта theOuter.theInner. Любой объект является «immediately enclosing instance of» ранга 0 по отношению к самому себе (обратите внимание на вызов метода doAnother() из метода doIt()). Как и в C++, компилятор гарантирует семантическую совместимость вызываемого метода и типа ссылки на объект-сервер.

Синхронизация

Когда клиент посылает сообщение серверу, происходит их *синхронизация*. В однопоточном приложении синхронизация состоит в запуске метода. Фактически это означает, что в системе существует только один активный объект, поэтому задача разделения доступа к объектам-серверам не возникает.

В многопоточной системе ситуация существенным образом усложняется. Возникает задача обеспечения конкурентного доступа к объекту-серверу со стороны нескольких активных объектов. Действительно, если на одном сервере одновременно выполняются несколько методов, использующих одни и те же данные, то необходимо обеспечить консистентность этих данных для каждого потока.

Существует три варианта организации доступа к совместно используемым серверам и агентам:

- Последовательный — семантика пассивного объекта обеспечивается в присутствии только одного активного потока.

- Защищенный — семантика пассивного объекта объекта обеспечивается в присутствии многих потоков управления, но взаимное исключение обеспечивается самими активными объектами либо внешними по отношению к серверу механизмами.

- Синхронный — семантика пассивного объекта обеспечивается в присутствии многих потоков управления; взаимное исключение обеспечивает сам сервер.

В языке C++ нет непосредственной поддержки многопоточности, поэтому возникает необходимость в использовании дополнительных библиотек при разработке параллельных систем.

В Java поддержка многопоточности встроена в язык и платформенную библиотеку: существует специальный класс Thread, реализующий поток исполнения; volatile переменные; synchronized блоки (ими можно пользоваться для реализации защищенного доступа); а также имеется возможность объявления synchronized методов для организации синхронного сервера.

3.6.2 Иерархии объектов

В отличие от классов, образующих иерархии наследования (отношение «is a»), объекты образуют агрегационные иерархии (отношение «part of»).

Не следует путать агрегацию классов и агрегацию объектов. Как будет видно в следующей главе, отношение агрегации, определенное на уровне абстракций (классов), может приводить как к композиционным, так и к ассоциативным (фактически) связям между объектами.

Концептуально, агрегация может означать, а может не означать включение одного объекта в другой. При рассмотрении отношения агрегации применительно к объектам, Буч [1] приводит примеры про самолет, состоящий из фюзеляжа, двигателей, крыльев, шасси и прочих частей, и про акционера и акции которыми он владеет. В обоих случаях он опре-

деляет эти отношения как агрегацию, в первом случае предусматривающую физическое включение, а во втором — нет.

Логически, агрегация определяет отношение вида целое-часть. При этом, физически, она может быть реализована как с помощью *композиции*, так и с помощью *ассоциации (связи)*.

При композиционном отношении между объектами существует зависимость по времени жизни. Так, *агрегант (часть)* не может существовать без *агрегата (целого)*. При уничтожении объекта-целого, уничтожаются и объекты-части. При ассоциативном отношении объекты могут существовать независимо друг от друга по времени жизни.

Варианты реализации композиционных и ассоциативных отношений на C++ и Java мы будем рассматривать при подробном изучении этих языков. Сейчас же, мы только укажем на средства обеспечения безусловной композиции в этих языках.

Если в программе, написанной на C++, один объект объявлен как обычная (не ссылка и не указатель) переменная (поле) другого объекта, то эти объекты будут находиться в отношении композиции. Связь по времени жизни, в этом случае, обеспечивается самим компилятором (при создание объекта-агрегата будет создан и объект-агрегант; при уничтожении агрегата, агрегант будет уничтожен). В остальных случаях композиция определяется логикой программы.

В программах, написанных на Java безусловная композиция присутствует только в отношениях между объектом и его «immediately enclosing instance» ранга большего или равного 1. Экземпляр внутреннего (inner) класса не может существовать вне контекста экземпляра внешнего (outer) класса. В остальных случаях, физически, отношения всегда являются ассоциативными (по ссылке), гарантированной связи по времени жизни нет, и время уничтожения объектов определяется логикой работы сборщика мусора.

Глава 4

Классы

Понятие *класса* является фундаментальным для объектно-ориентированного подхода в программировании. Трудно представить современного программиста, который бы не слышал про ООП и не был знаком с этим понятием. Тем не менее, для того, чтобы наша картина основ теории объектно-ориентированного программирования выглядела полной, мы должны рассмотреть все основные вопросы, касающиеся классов и отношений между ними.

Как мы уже говорили, в основе объектного подхода лежит принцип абстрагирования. Так вот, именно класс является самой важной формой абстракции. С точки зрения программирования, класс является основным кирпичиком при построении объектно-ориентированных программ, позволяя выразить все аспекты связанные как с интерфейсной частью абстракции, так и ее реализацией. Иерархии классов и интерфейсов (забегая вперед скажем, что фактически, интерфейс является вырожденным абстрактным классом) задают семантику и строение всех существующих в системе объектов.

Буч дает следующее определение класса:

Класс — это некое множество объектов, имеющих общую структуру и общее поведение.

4.1 Природа классов

Современным системам присущи как функциональная, так и композиционная сложность. Основными способами преодоления сложности служат декомпозиция и моделирование.

Для формализации различных уровней моделирования системы, мы обратимся к многоуровневой модели основанной на модели OMG MOF[1] [3], приведенной на Рис. 4.1. На самом верху этой модели находится собственно система из реального мира, которую необходимо автоматизировать.

На следующем уровне (уровень m3) находятся конкретные объекты, расположенные в памяти ЭВМ во время исполнения программы. Таким образом, мы моделируем нашу систему в терминах объектов.

Объекты могут быть выражены в терминах классов (уровень m2), которые описываются на объектно-ориентированных языках программирования; диаграммы классов могут быть выражены на унифицированном языке моделирования UML[4]. Обычно разработка программной системы включает в себя работу на уровнях m3 и m2 (архитектурное моделирование, программирование и отладка).

Нижележащий уровень (уровень метамодели или m1 уровень) используется при разработке систем автоматической генерации кода по моделям, реализованным на UML или его диалектах. На данном уровне описывается сам язык UML в терминах языка метамоделирования MOF. В частности, на этом уровне определяются такие понятия как класс, операция, атрибут (поле), ассоциация и другие. Данные понятия выражаются в терминах метаклассов, метаатрибутов, метаопераций и так далее.

Самый низкий уровень m0 позволяет выразить язык MOF на самом себе. Этот уровень используется при написании систем автоматической генерации кода для «генераторов» используемых на уровне m1.

[1] Object Management Group Meta Object Facility

Таким образом, классы являются единицами представления системы на уровне m2.

При дальнейшем изложении мы будем пользоваться языком моделирования UML для наглядной иллюстрации структуры и взаимоотношений между классами. Для ознакомления с этим языком, читателю следует обратиться к книге Буча, Рамбо и Джекобсона [4]

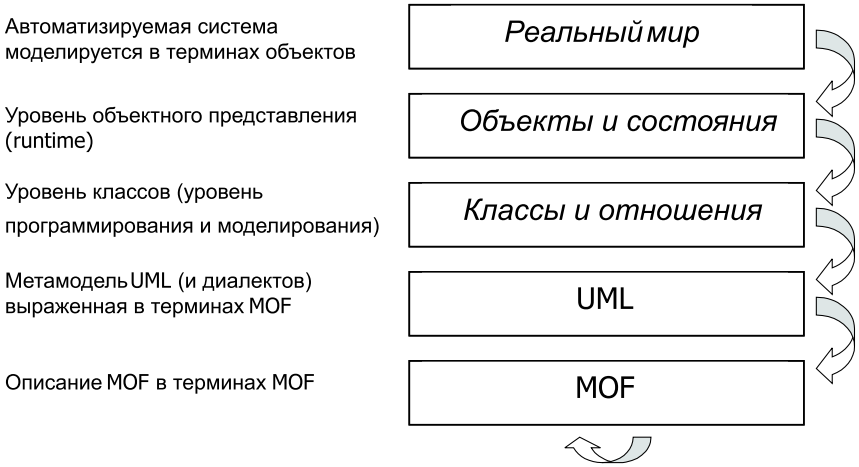

Рис. 4.1: Многоуровневая модель представления

4.1.1 Структура класса

По своей природе, класс — это генеральный контракт между абстракцией и всеми ее клиентами. Выражением обязательств класса служит его интерфейс.

Класс задается своим типом (именем класса), информацией о суперклассах и реализуемых интерфейсах, а также своими членами (полями, методами и внутренними абстракциями). Члены класса могут реализовывать свойства объекта (поля и методы экземпляра), а могут

реализовывать свойства собственно класса (статические поля и методы).

Для поддержки принципа инкапсуляции, существуют четыре основных уровня доступа к членам класса. Разные языки могут поддерживать разное из этих уровней. Приведем их в порядке открытости для внешних абстракций:

- Открытый (public) доступ — члены с этим уровнем доступа видимы всем клиентам класса.
- Защищенный (protected) доступ — члены этого уровня видимы самому классу, его подклассам, и абстракциям, находящимся с ним в одном пакете.
- Пакетный доступ — члены этого уровня доступны только самому классу и абстракциям, находящимся с ним в одном пакете.
- Закрытый (private) доступ — члены этого уровня видимы только изнутри самого класса.

Некоторые языки (например C++) имеют средства обойти эти уровни и обеспечить доступ к закрытой части класса[2]. Использовать такие средства следует лишь в исключительных случаях.

Для реализации свойства объекта иметь состояние, в классе существуют атрибуты (поля). В языках программирования Java и C++ атрибуты являются строго типизированными переменными. В этих переменных могут храниться как простые величины (для атрибутов примитивного типа), так и ссылки (указатели) на объекты других классов для ссылочных переменных типов, определяемых пользователем. Переменные являются частью реализации класса и должны быть закрыты от модификации извне функций-членов (методов) класса. Поэтому переменные экземпляра и неконстантные статические переменные класса **всегда** должны быть объявлены с уровнем доступа private.

[2] Речь идет о механизме дружественности в языке C++/

Для реализации поведения объекта используются функции-члены класса. При этом, функции-члены экземпляра работают в контексте состояния конкретного объекта и имеют доступ как к членам экземпляра, так и к статическим членам класса. Статические функции-члены реализуют поведение собственно класса и имеют доступ только к статическим членам класса.

Открытые (public) методы класса составляют внешний контракт (интерфейс) класса по отношению к его клиентам, и, наряду с именем класса формируют смысл абстракции.

Внутренние классы и интерфейсы используются для ограничения области видимости определяемых ими абстракций рамками одного класса. Как правило, эти абстракции не используются или не могут быть использованы извне класса, в котором они объявлены.

Особенности программирования классов на конкретных языках мы рассмотрим когда будем изучать в деталях языки C++ и Java.

4.1.2 Абстрактные классы, интерфейсы и классы-утилиты.

Абстрактные классы — это классы, которые не могут иметь экземпляров. Практически это означает, что в них присутствуют функции-члены, которые объявлены, но не определены. Эти функции определяются в наследниках, которые уточняют данную абстракцию.

Интерфейс — это абстрактный класс, который содержит только объявления методов и статические константные поля. Таким образом, интерфейс определяет чистую абстракцию поведения. Для интерфейса практический смысл имеет только открытый уровень доступа к его членам. Язык C++ не имеет специальной языковой конструкции для декларации интерфейса, но для получения эквивалентной абстракции можно объявить класс, содержащего только чистые виртуальные функции и статические константы.

Класс-утилита — это класс, в котором присутствуют только статические члены. Такие классы используются для группировки наиболее часто используемых общих алгоритмов, работающих с другими аб-

стракциями или примитивными данными. Примером типичного класса-утилиты может являться класс java.lang.Math.

4.2 Отношения между классами

Отношения между классами определяют формы отношений и семантику объектов. Иерархические отношения (отношения вида «is a») формируют внутреннюю структуру и контрактные обязательства объектов. Ассоциативные отношения между классами, в свою очередь, определяют отношения между объектами.

Для иллюстрации многообразия отношений между классами приведем пример из книги Буча[1, §3].

Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:

- Маргаритка - цветок.
- Роза - (другой) цветок.
- Красная и желтая розы - розы.
- Лепесток является частью обоих видов цветов.
- Божьи коровки питаются вредителями, поражающими некоторые цветы.

Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области ключевые абстракции взаимодействуют многими интересными способами. Отношения между классами могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и маргаритки, и розы - это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и так далее. Во-вторых, может быть какая-то семантическая связь. Например, красные розы больше похожи на желтые

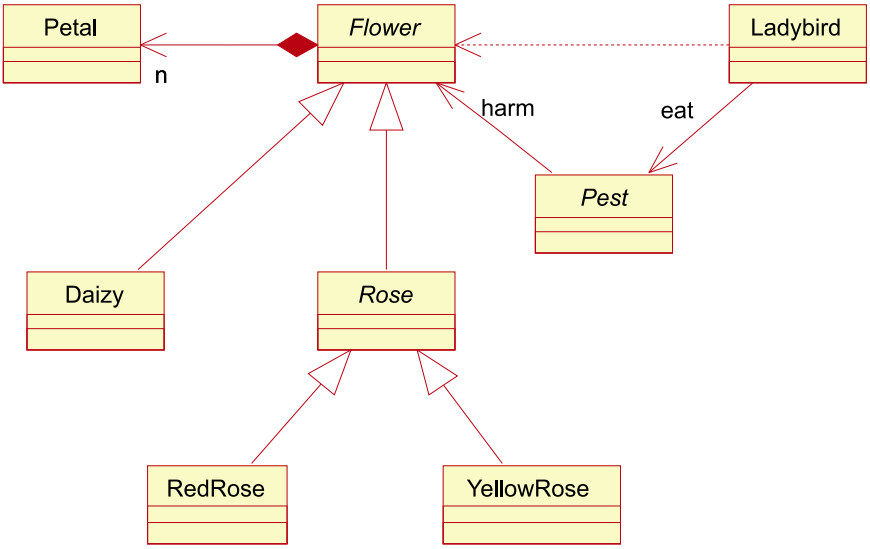

Рис. 4.2: Пример диаграммы отношений между классами

розы, чем на маргаритки. Но между розами и маргаритками больше общего, чем между цветами и лепестками. Также существует симбиотическая связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам. UML диаграмма этих классов и отношений между ними представлена на Рис 4.2.

Известны три основных типа отношений между классами. Во-первых, это отношение «обобщение/специализация» (общее и частное), известное как «is-a». Розы — суть цветы, что значит: розы являются специализированным частным случаем, подклассом более общего класса «цветы». Во-вторых, это отношение «целое/часть», известное как «part of». Так, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, божьи коровки ассоциируются с цветами - хотя, казалось бы, что у них общего. Или вот: розы и свечи — и то, и другое можно использовать для

украшения стола.

4.2.1 Ассоциация

Давайте рассмотрим часть задачи автоматизации процесса продажи какого-нибудь товара. Естественным образом мы можем выделить следующие абстракции: продавец, покупатель, товар, сделка, платежная квитанция. Эти абстракции будут естественным образом связаны друг с другом. В данном случае у нас классы будут связаны ассоциативными связями (См. Рис 4.3).

Ассоциация обозначает наличие логической связи между классами. Она может быть однонаправленной, двунаправленной или не иметь специфицированного направления. Концы ассоциации могут иметь имена, обозначающие роли, которые играют классы (объекты) друг для друга. Кроме направления и имени, с ассоциацией связанно понятие *мощности*[3]. Так, можно выделить следующие разновидности мощности ассоциаций:

- «один-к-одному»
- «один-ко-многим»
- «многие-ко-многим»

В нашем примере (см Рис 4.3) некоторые ассоциации имеют имена ролей. С покупателем могут быть связаны несколько покупок (отношение «один-ко-многим»), с продавцом также могут быть связаны несколько сделок. С покупкой связан список приобретенных товаров. А вот сделка и платежная транзакция состоят в отношении «один-к-одному».

[3] В англоязычной литературе — multiplicity или cardinality.

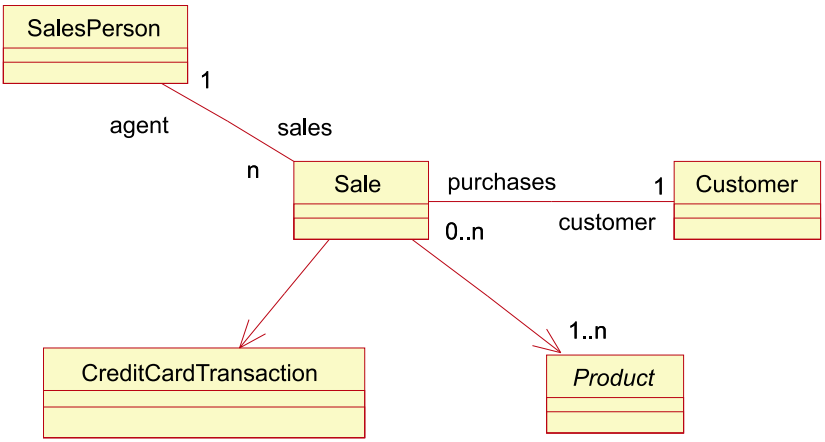

Рис. 4.3: Диаграмма классов торговой сделки.

4.2.2 Агрегация и композиция

Мы уже говорили об агрегации в главе, посвященной объектам. Агрегация является частным случаем ассоциации. Если экземпляр класса-целого связан с экземпляром класса-части по времени жизни (имеет место физическое включение одного объекта в другой), то мы говорим о композиции. Пример агрегации и композиции приведен на Рис 4.4. В этом примере автомобиль состоит в отношениях композиции со своими частями, но в отношении простой агрегации со своими пассажирами.

4.2.3 Использование

Отношение использования означает, что одна абстракция зависит от функционального интерфейса (контракта) другой абстракции. Т. е. при изменении контракта одного класса или интерфейса необходимо вносить изменения в код класса, который от него зависит. Например, перед нами стоит задача разработки системы классов для графического ре-

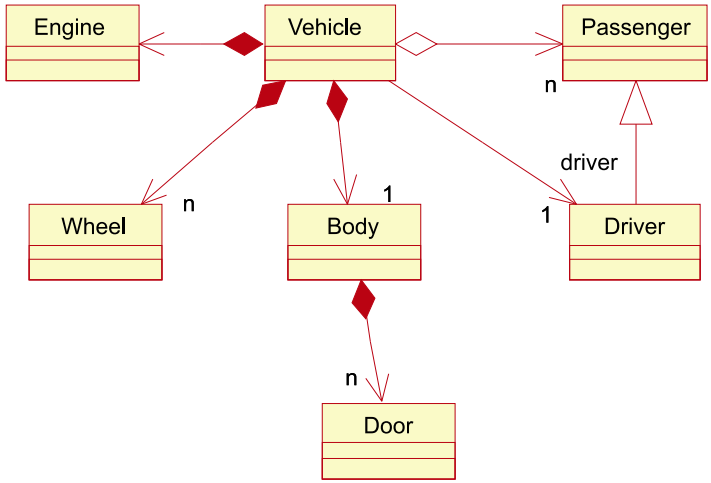

Рис. 4.4: Диаграмма классов представляющая средство передвижения.

дактора. В процессе работы мы выделяем такие абстракции как Shape и Canvas (см Рис 4.5). При этом фигура использует экранный холст для того чтобы отображать себя. Если мы в какой-то момент в интерфейсе Canvas изменим функциональный контракт, то скорее всего, потребуется вносить изменения в механизм отрисовки фигуры. Причем, фигура может отрисовать себя на любом устройстве, которое реализует интерфейс холста (в данном случае отрисовка возможна в окне и на графопостроителе).

4.2.4 Наследование

Отношение наследования, или генерализации, означает зависимость вида «обобщение/специализация». Для иллюстрации этого типа взаимоотношений продолжим рассмотрение графического редактора. Посмотрите на диаграмму классов, приведенную на Рис 4.6. Класс Shape является абстрактным. В нем не определены методы resize и draw, что

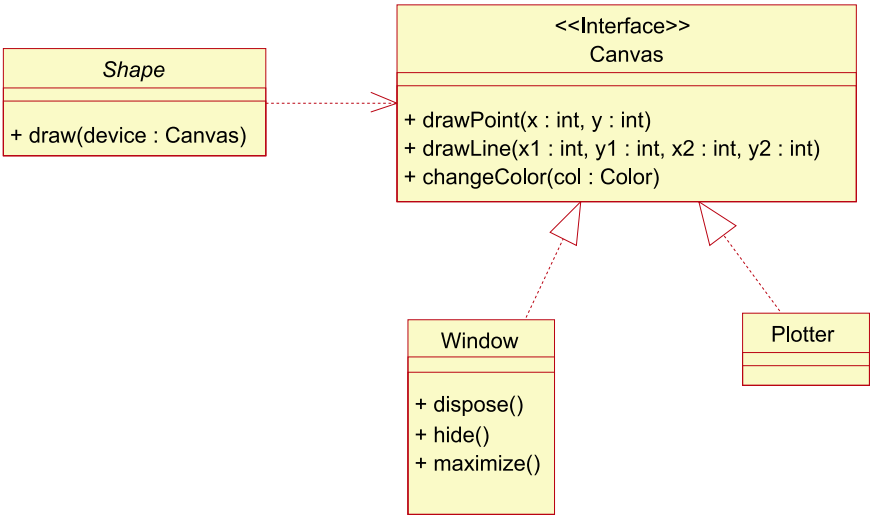

Рис. 4.5: Пример зависимости между абстракциями.

вполне естественно, так как на данном уровне абстракции мы не знаем как отрисовывается фигура. Эти методы определяются в конкретных классах Circle, Rectangle и Polygon. При этом, класс Picture не зависит от этих конкретных классов и может работать с окружностями, прямоугольниками и полигонами как с обобщенными фигурами (используя интерфейс класса Shape).

Заполненный прямоугольник (объект) будет экземпляром классов SolidRectangle, Rectangle и Shape. Разные клиенты к нему смогут обращаться через контракт любого из этих классов. Класс SolidRectangle расширяет абстракцию прямоугольника (являясь абстракцией закрашенного прямоугольника) и должен переопределить метод draw. Для вычисления размера прямоугольника он может воспользоваться защищенным методом getCorner.

Наследование подразумевает гораздо большую сцепленность и зависимость абстракций друг от друга, нежели ассоциация или использование. Иногда вместо наследования лучше воспользоваться агрега-

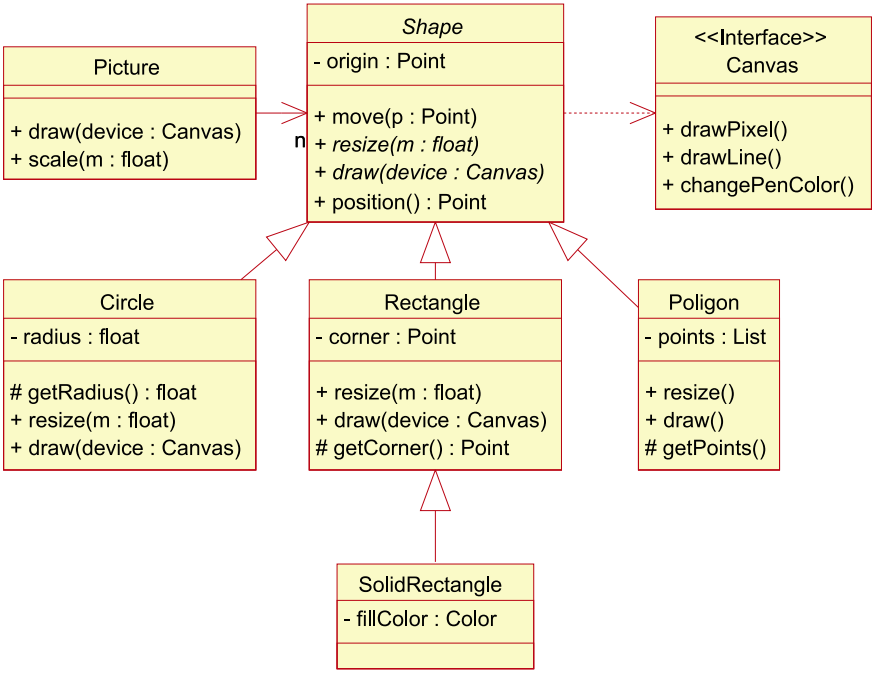

Рис. 4.6: Элемент диаграммы классов графического редактора.

цией совместно с реализацией требуемого интерфейса.

До сих пор остается открытым вопрос о преимуществах, предпочтении, а также пользе и вреде множественного наследования. Как показывает опыт, в хорошо спроектированной объектно-ориентированной системе, как правило, не возникает необходимости в использовании множественного наследования. В большинстве случаев вместо множественного наследования лучше воспользоваться агрегацией совместно с реализацией требуемого интерфейса.

При хорошо организованной системе классов и интерфейсов критические и ключевые абстракции обычно являются интерфейсами, что позволяет легко встраивать в систему альтернативные реализации этих

абстракций.

Разные языки могут иметь разную поддержку наследования. В C++ возможно *множественное* наследование от нескольких классов, а в Java возможно только *одиночное* наследование для классов и множественное для интерфейсов. При этом классы могут реализовать несколько интерфейсов.

Программируя на C++ следует, по возможности, избегать множественного наследования конкретных и абстрактных классов, так как оно, как правило, порождает проблем больше, чем решает. В качестве конструкций, эквивалентных интерфейсам можно воспользоваться чисто виртуальными классами[4]. Использование наследования от нескольких чисто виртуальных классов вполне допустимо.

Основные аспекты и проблемы, связанные с множественным наследованием, мы будем обсуждать во время подробного рассмотрения языка C++.

4.3 Инстанцирование

Инстанцирование, в общем случае, есть операция создания элемента модели уровня mN на основе элемента (или группы элементов) модели уровня mN-1.

Инстанцирование объектов происходит во время исполнения программы путем выделения памяти и вызова конструкторов классов.

Инстанцирование классов из шаблонов классов происходит во время компиляции программы. Инстанцирование классов на основе элементов метамодели происходит во время работы генераторов кода.

[4] чисто виртуальные классы (pure virtual classes) — это классы, которые не имеют членов-данных, и все методы которых являются чисто виртуальными функциями.

Часть II

Основы объектно-ориентированного дизайна

Во второй части пособия мы пытаемся ответить на вопрос, мало проработанный в литературе на русском языке, а именно: *Что же такое хороший ОО дизайн?* Слово «design» в контексте ОО подхода переводится с английского как «проектирование», но мы будем здесь использовать именно «дизайн» по причине широкой распространенности этого англицизма и аббревиатуры OOAD[5] в профессиональной среде.

Эта часть посвящена принципам объектно-ориентированного дизайна и предназначена для студентов, знакомых с основными концепциями ООП и хотя бы с одним ОО языком программирования. Мы не будем здесь затрагивать вопросы моделирования на языке UML, достаточно хорошо изложенные в известном пособии «трех друзей» [4], и рекомендуем обратиться к этой книге, а за подробным изложением техники ОО анализа мы отсылаем читателя к книге Г. Буча «ОО анализ и проектирование» [1].

Автор популярного ОО языка C++ Б.Страуструп как-то заметил, что вопрос о том, как пишут хорошие ОО программы похож на вопрос о том, как пишут хорошую английскую прозу. Не претендуя на умение научить читателя писать «хорошую прозу», мы, однако, попробуем рассмотреть здесь некоторые правила, которые позволяют, как минимум, идентифицировать проблемы в дизайне системы[6]. Этот набор принципов не является ортогональным, некоторые их них взаимно дополняют и повторяют друг друга, и во многих случаях неправильного дизайна мы имеем дело с нарушением сразу нескольких из них. Тем не менее, все они являются широко используемыми и входят в повседневный словарь архитектора ПО и потому приводятся здесь полностью.

Во избежание искажений смысла, нередких при переводе английских терминов на русский язык, и учитывая то, что современный архитектор ПО вынужден ежедневно работать с англоязычной докумен-

[5] Object-Oriented Analysis and Design

[6] То, что в ООП не существует единственно правильных ответов, а только способы идентификации неправильных, лишний раз доказывает, что программирование является искусством

тацией, мы везде, где считаем необходимым, приводим оригинальные формулировки наряду с переводом.

Сначала попытаемся понять, какой дизайн нам нужен. Какую программу мы назовем спроектированной хорошо, а какую — плохо? Для ответа на эти вопросы необходимо вспомнить, что развитие программы не заканчивается с ее созданием, и объем усилий по сопровождению нередко превышает затраты на написание программного кода. Действительно полезные программы живут многие годы, коллективы разработчиков постоянно обновляются, а сама программа модифицируется под воздействием изменяющихся условий. Имея в виду эти факты, можно заключить, что хорошим, то есть наиболее приемлемым как для производителя, так и для потребителя программного продукта является тот дизайн, который обеспечит:

- Наибольшую гибкость структуры ПО
- Низкую стоимость сопровождения
- Возможность повторного использования кода программ

Гибкость — это возможность внесения изменений в код без существенной его переработки. Локальные изменения требований должны приводить к локальным же изменениям в коде; желание пользователя изменить, например, формат представления даты не должно заставлять нас изменять каждый модуль программы, где используется ввод-вывод дат. Другими словами, гибкость как свойство отображения требований пользователя на код программы есть мера устойчивости кода к изменениям требований.

Стоимость сопровождения — стоимость внесения изменений в фазе сопровождения ПО. Поскольку, как это уже упоминалось выше, изменения в коде программ могут происходить и происходят уже после окончания разработки, чрезвычайно важно, чтобы их стоимость была невысока.

Повторное использование кода — это возможность использовать единожды определенные абстракции (классы, модули) в разных ком-

понентах одного и того же или различного ПО. Учитывая все возрастающую сложность программ, и, как следствие, немалую стоимость работ по анализу и проектированию, это свойство становится все более актуальным.

Улучшение вышеуказанных характеристик программного продукта реализуется в ОО подходе путем

- Дизайна иерархий классов (как собственно классов, так и отношений между ними)
- Дизайна пакетов (содержимого пакетов и связей между ними)
- Поиска и повторного использования проверенных решений конкретных проблем. Обобщенные решения таких проблем носят название образцов дизайна.

Что касается применения образцов[7], то эта тема замечательно проработана в книге Э.Гамма и др. «Приемы объектно-ориентированного проектирования»[5], которой мы рекомендуем воспользоваться.

Принципы дизайна рассматриваются в том порядке, в котором они обычно применяются при создании программных систем: принципы проектирования основных классов, способы оптимизации графа зависимостей и применение формальных метрических методов для оценки качества дизайна.

[7] английский термин - design pattern

Глава 5

Принципы ОО дизайна

ОО программы представляют собой набор взаимодействующих объектов некоторых классов, большей частью являющихся абстрактами понятий, обнаруженных в процессе анализа предметной области, для которой создается программа. Правильно найденные и реализованные абстракции — ключевой фактор для достижения необходимой при создании сложного приложения гибкости кода и снижения затрат на сопровождение. В этой главе мы рассмотрим принципы, которым необходимо следовать при проведении объектной декомпозиции, т.е. создании иерархии классов.

5.1 Единственность абстракции

Правило единственности абстракции было сформулировано Робертом Мартином (R. Martin) [6]:

ORR[1] — *Правило единственности абстракции*

Класс должен обладать единственной ответственностью, реализуя ее полностью, реализуя ее хорошо и реализуя только ее[2].

[1] One responsibility Rule

[2] A class has a single responsibility: it does it all, it does it well, it does it only.

Класс должен реализовывать *одну и только одну абстракцию*, и реализовывать ее *полностью*, чтобы программист, ее использующий, с одной стороны, не попадал в положение Кэрролловской Алисы: «Кот с улыбкой —и то редкость, но уж улыбка без Кота — это я прямо не знаю что такое!», а с другой — не был бы поставлен перед необходимостью иметь дело с монстрообразными классами, реализующими несколько абстракций одновременно. Очевидно, что реализация в одном классе нескольких абстракций снижает *переиспользуемость*, однако, этот принцип является одним из наиболее часто нарушаемых. Признаками нарушения принципа единственности абстракции являются:

- Затруднения с выбором подходящего имени класса
- Большое количество методов в классе
- Большое количество входящих и/или исходящих зависимостей
- Систематическое нарушение других принципов дизайна

Действительно, если вы затрудняетесь дать абстракции имя, ясно говорящее о том, что она делает или чем является, то стоит задуматься - а существует ли такой класс вообще? Большое количество методов в классе чаще всего говорит о том, что разные их подмножества предназначены для совершенно разных целей, и, возможно, описывают поведение разных абстракций, собранных по ошибке в один класс. Большое количество зависимостей также должно наводить на мысль о том, что класс использует или используется разными группами других классов для совершенно различных целей, а стало быть, может быть расщеплен на несколько составляющих. Число методов или зависимостей, превышающее 5-7 должно, как минимум, заставить поискать обоснование.

5.2 Принцип инверсии зависимости

Принцип инверсии зависимости был также сформулирован Робертом Мартином (R. Martin) [6]:

DIP[3] — *Принцип инверсии зависимости* Модули высокого уровня не должны зависеть от модулей низкого.
И те, и другие, должны зависеть от абстракций.
Абстракции не должны зависеть от деталей реализации.

Проиллюстрируем сказанное примером на рис. 5.1.

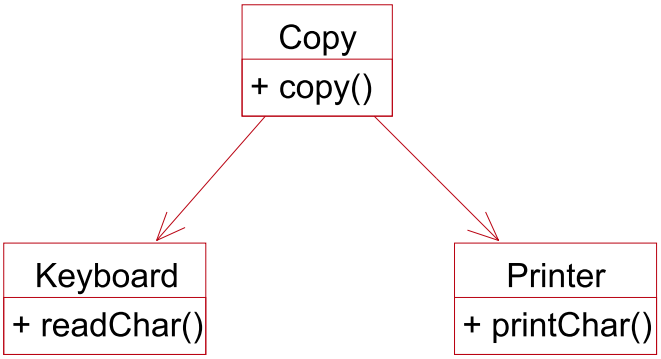

Рис. 5.1: Нарушение принципа инверсии

На этой диаграмме классов абстракция высокого уровня (алгоритм копирования в одного устройства на другое), зависит от таких низкоуровневых понятий, как клавиатура и принтер. Проблемы такого дизайна очевидны: мы не можем легко добавить поддержку ни новых устройств ввода, ни новых устройств вывода информации, хотя сам алгоритм копирования с одного символьного устройства на другое для всех таких устройств совершенно одинаков. Правильное решение этой задачи могло бы выглядеть так, как показано на диаграмме классов 5.2.

Здесь, как мы видим, абстракции высокого (класс Copy) и низкого (Printer и Keyboard) уровней зависят от еще более высокоуровневых абстракций Reader Writer, что повышает гибкость нашего кода, так

[3]Dependency Inversion Principle

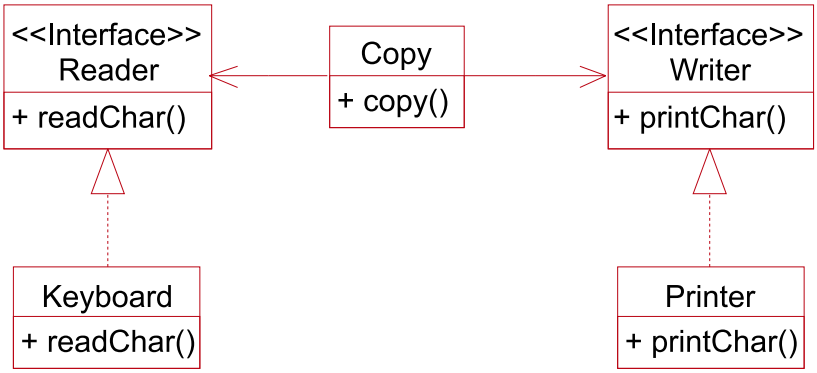

Рис. 5.2: Соблюдение принципа инверсии

как теперь мы можем легко расширить программу добавлением новых устройств ввода-вывода, не модифицируя уже реализованные классы.

5.3 Принцип Деметера

Одной из основных проблем, снижающих переиспользуемость программного кода, является сильное *сцепление*[4], определяемое как мера связанности классов между собой. Чем больше у класса связей (зависимости, наследование, агрегация) с другими классами, тем сильнее его сцепление, и тем сложнее его модифицировать и переиспользовать в другой программе. Рассматриваемый далее принцип Деметера позволяет, при его соблюдении, уменьшить сцепление и, соответственно, повысить гибкость и переиспользуемость. Оригинальная формулировка этого принципа, возникшая в процессе работы над одноименным проектом (Demeter) в Northeastern University осенью 1987 года, принадле-

[4] Английский термин: coupling. Сцепление может быть выражено числом, но здесь мы не будем останавливаться на метриках сцепления, заинтересованный читатель легко найдет информацию на Internet.

жит Яну Холланду (Ian Holland):

Принцип Деметера[5]

Каждый элемент программы должен обладать ограниченным знанием о других элементах и использовать только тесно связанные с ним элементы.[6]

Как видно из формулировки, принцип Деметера не ограничивается классами, но претендует на правильность в применении ко всем элементам ПО, таким как пакеты, модули, процедуры. Здесь мы рассмотрим применение принципа Деметера к иерархиям классов, а поиск аналогий оставим читателю. В контексте классов под элементом Холланд понимает метод класса. Методами, тесно связанными (closely related) с методом f класса C, называются

- методы класса C
- методы классов, являющихся параметрами f
- методы полей класса C
- методы классов объектов, созданных в f

Рассмотрим следующий пример (диаграмма на рис. 5.3):

```
class Application {
  // ....
  public static void main( String[] parameters ) {
    viewer.getDocument().getDate();
  }
}
```

Обратим внимание на то, что в этой, характерной для языка JAVA[7] программе, код класса Application неявно зависит от методов класса

[5] Demeter principle

[6] Each unit should have only limited knowledge about other units: only units «closely» related to the current unit. Одним из мотивов для ввода данного принципа послужило соображение об ограниченности возможностей человеческого мозга (известное правило 7 ± 2).

[7] Стандартные библиотеки языка JAVA содержат массу примеров такого нарушения принципа Деметера и просто провоцируют программиста продолжать в том же духе.

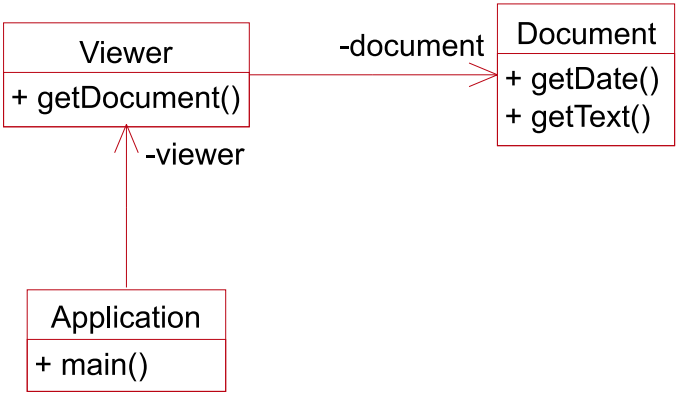

Рис. 5.3: Нарушение принципа Деметера

Document, тем самым порождая две проблемы. Первая состоит в том, что возможные изменения кода не локализованы: попытка изменить класс Document (а также любой из тех, от которых он зависит) может отразиться на классе Viewer, а через него и на Application. Вторая, самая неприятная для архитектора, проблема состоит в том, что связь между Application и Document может не попасть на диаграмму классов, так как появляется в процессе написания кода, а не во время построения модели. Формальная проверка программистом кода на соответствие принципу Деметера легко выявляет потенциальную проблему, поскольку метод main класса Application обращается к методу getDate класса Document, который

- Не является методом ни самого класса Application, ни классов-аргументов метода main
- Не является методом никакого из полей класса Application
- Не является методом класса объекта, созданного в main

Корень зла содержится, конечно же, не в методе main, а немножко рань-

ше — в классе Document, беззаботно открывающем секреты своей реализации незнакомцам[8]. Пример показывает, что нарушение принципа Деметера ведет к неконтролируемому распространению зависимостей между классами, что, в конечном счете, неизбежно приводит к снижению гибкости кода и увеличению стоимости сопровождения. Действительно, изменение, например, способа доступа к документу в классе Viewer становится затруднительным, так как может вызвать каскадное распространение изменений на классы, использующие метод getDocument. Неконтролируемые, а стало быть, потенциально слишком длинные, цепочки зависимостей между классами безусловно снизят переиспользуемость, поскольку слишком много классов придется переиспользовать «в нагрузку», а также могут стать причиной появления циклических зависимостей между классами и пакетами[9].

Гради Буч по этому поводу заметил: «Основной эффект от применения закона Деметера — создание слабо связанной иерархии классов, детали реализации которых скрыты друг от друга. Такие классы очень незагроможденны, чтобы понять один класс, вы не должны понимать детали многих других классов»[10]

5.4 Принцип подстановки Лисковой

Следующее правило, предложенное Барбарой Лисковой (Barbara Liskov) в 1988 году, фактически, задает определение подкласса:

LSP[11] — *Принцип подстановки Лисковой*

[8] Одна из формулировок принципа Деметера звучит так: «Only talk to your immediate friends. Never talk to strangers.» Разговаривайте только с близкими друзьями. Никогда не разговаривайте с незнакомцами.

[9] см. также Принцип ацикличности зависимостей

[10] «The basic effect of applying this Law is the creation of loosely coupled classes, whose implementation secrets are encapsulated. Such classes are fairly unencumbered, meaning that to understand the meaning of one class, you need not understand the details of many other classes.»

[11] Liskov Substitution Principle

Если для каждого объекта o1 типа S существует объект o2 типа T, такой что для всех программ P определенных в терминах T поведение P не изменяется при замене o2 на o1, то S является подклассом T[12].

Позднее, в 1996 г., Роберт Мартин дал не столь строгое, но более практичное определение: «Методы, принимающие в качестве параметра указатели и ссылки на объекты базового класса должны иметь возможность использовать эти объекты без необходимости знать, к какому классу (базовому или любому из производных) они принадлежат.»

Рассмотрим на примере диаграммы 5.4 и идущей ниже программы, к каким последствиям может привести нарушение этого принципа.

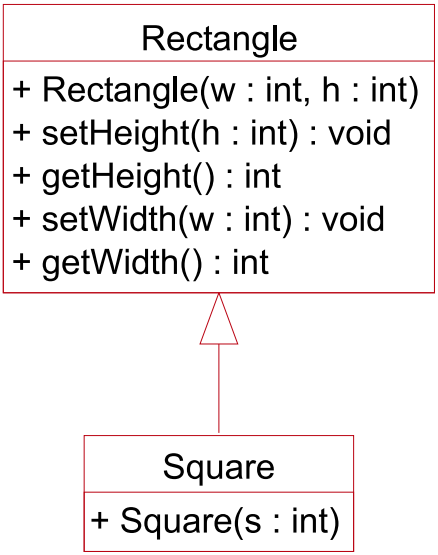

Рис. 5.4: Нарушение принципа Лисковой.

[12]If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T the behavior of P is unchanged when o1 is substituted by o2 then S is subtype of T.

```
class Rectange
{
  private int h;
  private int w;

  public Rectangle( int w, int h ) { this.h = h; this.w = w; }

  public void   setHeight( int h ) { this.h = h; }
  public int    getHeight()        { return h; }

  public void   setWidth( int w )  { this.w = w; }
  public int    getWidth ()        { return w; }
}

class Square extends Rectangle
{
  public Square( int s ) { super( s, s ); }
}
```

Применение наследования в этом случае достаточно типично для начинающих программистов, но не является правильным. Проблема такого дизайна заключается в потенциальной возможности для пользователя абстракции Square нарушить ее целостность самым легальным образом, через вызов метода, например:

```
class SomeClass
{
  void f()
  {
    Square s = new Square( 6 );
    s.setHeight( 5 ); // s уже не квадрат!
  }
}
```

Действительно, после вызова метода setHeight с аргументом 6, объект типа Square, первоначально инициализированный со значением стороны 5, попросту перестает быть квадратом. Напрашивающийся способ устранения указанной проблемы через переопределение методов

setHeight и setWidth в производном классе только на первый взгляд является хорошим решением. Рассмотрим модифицированный пример (диаграмма 38):

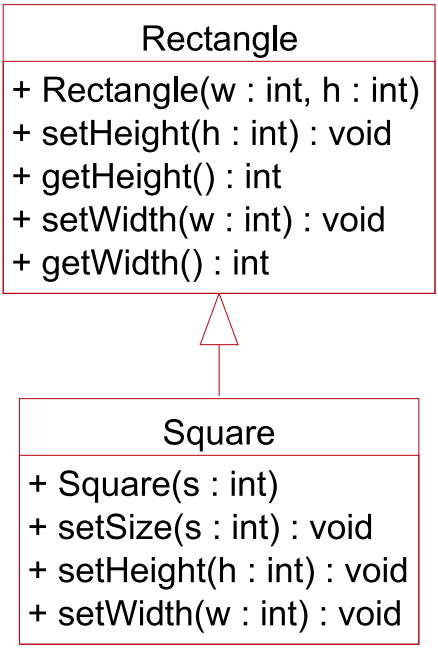

Рис. 5.5: Некорректное исправление нарушения принципа Лисковой.

```
class Rectange
{
  private int h;
  private int w;

  public Rectangle( int w, int h ) { this.h = h; this.w = w; }

  public void   setHeight( int h ) { this.h = h; }
  public int    getHeight()        { return h; }
```

```
  public void   setWidth ( int w ) { this.w = w; }
  public int    getWidth ()        { return w; }
}

class Square extends Rectangle
{
  public Square( int s )        { super( s, s ); }

  public void setSize  ( int s ){ super.setHeight(s); super.setWidth(s); }
  public void setHeight( int h ){ setSize(h); }
  public void setWidth ( int w ){ setSize(w); }
}

class SomeClass
{
  void f( Rectangle r ) throws Exception
  {
    r.setHeight(4);
    r.setWidth(5);
    if( r.getHight() * r.getWidth() != 20 ) throw new Exception( Surprise! );
  }
}
```

Предположим, что некто пытается отладить код, использующий вышеприведенный метод f, разработанный другим разработчиком. Иногда метод будет отрабатывать как предполагалось, а иногда выбрасывать исключение. Причем для того, чтобы понять, что же именно произойдет в каждой конкретной ситуации, разработчик должен знать, чем именно является переданный в метод аргумент — прямоугольником или квадратом. Но ведь метод f проектировался как работающий с прямоугольником и его разработчик мог даже не догадываться, что некогда другой член команды напишет производный класс! Обратно, разработчик класса Square может не знать, что где-то в программе существует метод, подобный f. На самом деле сама возможность написать такой метод показывает, что дизайн класса Square нарушает принцип подста-

новки Лисковой, так как поведение метода f существенным (и весьма неожиданным для программиста, его написавшего) образом, зависит от того, какому типу принадлежит аргумент — базовому или производному. Таким образом, задать Квадрат наследованием Прямоугольника, не нарушив принцип подстановки, нельзя. Читатель, надо полагать, уже недоумевает: общеизвестно, что квадрат — частный случай прямоугольника, а наследование — как раз способ задания отношения частное-общее, не так ли? Следовательно, либо принцип подстановки неверен, либо наследование — не есть способ задания отношений частное-общее. Кто прав? Правда, как это часто бывает, находится посередине: принцип верен и наследование прекрасно работает как способ выражения отношения частное-общее, но не в нашем случае. Дело в том, что принцип подстановки требует от нас при принятии решения о наследовании исходить из общности *поведения*; мы же, говоря, что Квадрат — частный случай Прямоугольника, неявно исходим из общности их структуры. При том наборе методов[13], который мы выбрали для Прямоугольника, Квадрат действительно *не является* его частным случаем в смысле поведения, так как, при таком свойстве (поведении) Прямоугольника, как способность изменять стороны, Квадрат и Прямоугольник —существенно различные абстракции[14]. **LSP**, таким образом, может использоваться для доказательства неправильности дизайна отношений типа класс-суперкласс. Доказать правильность в общем случае затруднительно.

[13]Набор открытых методов класса также часто называют *контрактом* класса.

[14]Если развивать эту тему дальше, то можно заметить, что вышеописанный Прямоугольник и Квадрат могут быть связаны отношением, не имеющим явного способа выражения в ОО языках, а именно: Квадрат может быть представлен как *пересечение* абстракций Прямоугольник и Правильный Многоугольник. Хорошим приближением может быть реализация в классе Квадрат интерфейса Правильный Многоугольник путем агрегации объекта типа Прямоугольник.

5.5 Принцип разделения интерфейсов

В следующем принципе речь пойдет о дизайне ассоциаций между классами.

ISP[15] *—Принцип разделения интерфейсов*

Классы не должны зависеть от контрактов, которых они не используют.[16]

Принцип достаточно очевиден, ведь никто из нас не захотел бы платить за ненужный ему сервис. Любой из нас как минимум удивится, если ему предложат переоформить договор страхования автомобиля на том основании, что страховая компания изменила политику страхования жилья от пожара. В программировании, однако, люди достаточно легко делают подобные предложения своим товарищам по работе, предлагая им иметь дело с избыточными контрактами классов. Рассмотрим пример сервиса по передаче сообщений (диаграмма 5.6). Проблема здесь в том, что класс Source (источник сообщений) никак не использует метод subscribe класса MessagingService, этот метод явно предназначен для совершенно других сущностей, а именно, получателей сообщений. Последние, в свою очередь, также вынуждены зависеть от совершенно ненужного им метода send. Изменения в классе MessagingService, относящиеся к подписке на сообщения, заставляют авторов источников почем зря перекомпилировать свои классы. Кроме того, такое решение затрудняет использование класса Source с другими реализациями сервиса доставки сообщений, то есть ведет к существенной потере гибкости. Дизайн, соблюдающий принцип разделения интерфейсов, приведен на рисунке 5.7.

Как нетрудно видеть, добавление интерфейсных классов DeliveryService и MessageDistributor полностью устраняет проблемы, описанные выше. Изменения в MessagingService теперь совершенно не влияют на источники и получателей, и, в качестве бонуса за соблюдение принци-

[15] Interface Segregation Principle

[16] В оригинальной формулировке Р.Мартина этот принцип звучит так: Clients should not be forced to depend upon services they do not use.

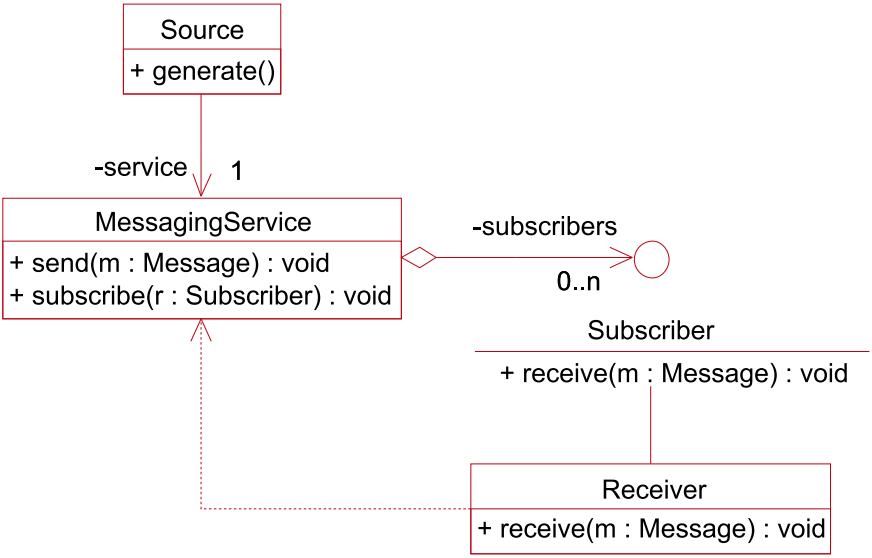

Рис. 5.6: Сервис по передаче сообщений

па, мы легко можем заменить MessagingService другим классом. Такая особенность дизайна системы будет очень полезной, например, при возникновении необходимости разнести код источников и получателей сообщений в разные программы и/или на разные машины.

5.6 Принцип ацикличности зависимостей

ADP[17] —*Принцип ацикличности зависимостей*

> Структура зависимостей между элементами (классами, пакетами, методами) должна представлять собой направлен-

[17] Acyclic Dependencies Principle

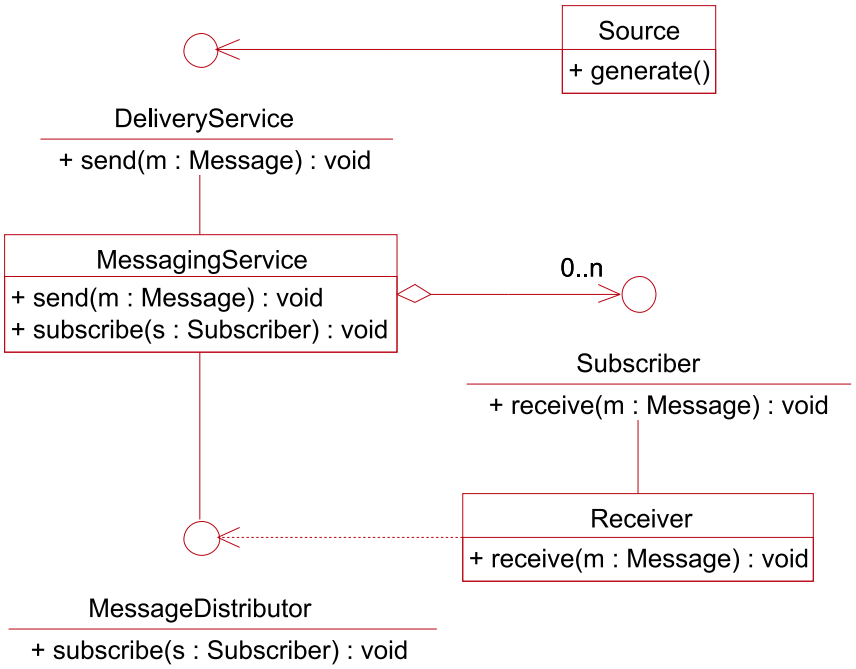

Рис. 5.7: Улучшенный сервис по передаче сообщений

ный ациклический граф[18]. R.Martin, 1996

Хотя Р. Мартин здесь говорит только о классах, можно легко заметить, что этот принцип в равной степени применим ко всем элементам ПО. Действительно, любые циклически связанные между собой элементы не могут быть использованы отдельно друг от друга, а, значит, нет никаких преимуществ в их разделении на отдельные составляющие, недостатки же очевидны: любое изменение элемента цикла влечет перекомпиляцию всех остальных, а минимально возможной едини-

[18]The dependency structure between entities (classes, packages, functions) must be a Directed Acyclic Graph (DAG).

цей переиспользования является весь цикл. Например, в приложении,

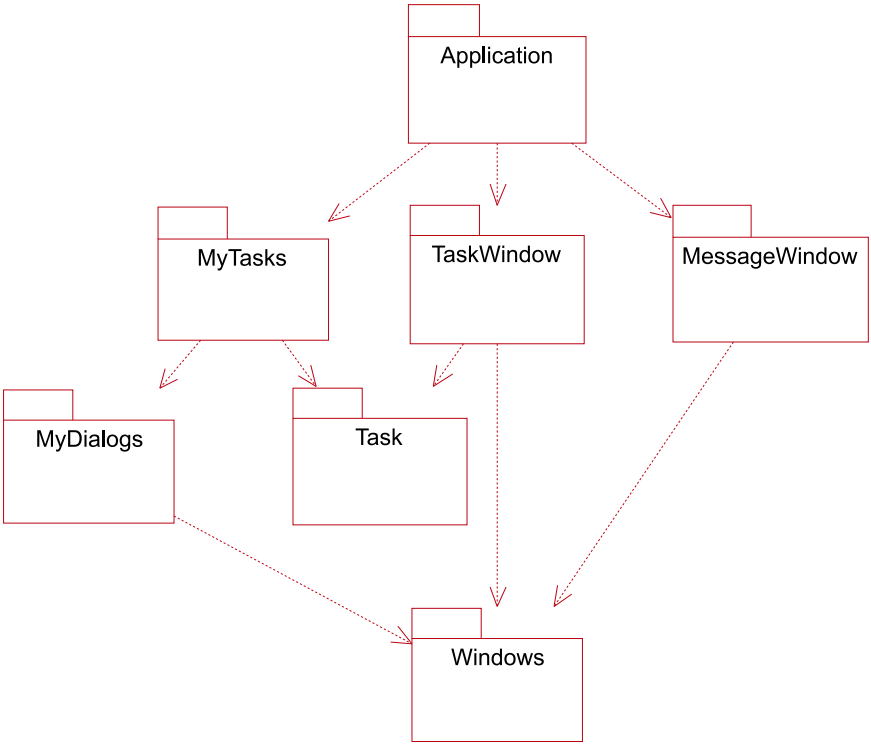

Рис. 5.8: Нарушение принципа ацикличности.

состоящем из пакетов, изображенных на диаграмме 5.8, может возникнуть ситуация, когда какому-либо классу из пакета MyDialogs потребуется доступ к глобальным данным, находящимся в пакете Application. В этом случае появится зависимость пакета MyDialogs от пакета Application и образуется цикл Application — MyTasks — MyDialogs. Вся цепочка пакетов при этом становится не переиспользуемой иначе, как целиком. Неприятным эффектом является то, что циклическую зависимость со-

здает пакет MyDialogs, а переиспользуемость теряет ничего не подозревающий MyTasks. Правильным решением в этом случае будет выделение глобальных данных в отдельный пакет, как это показано на диаграмме 5.9.

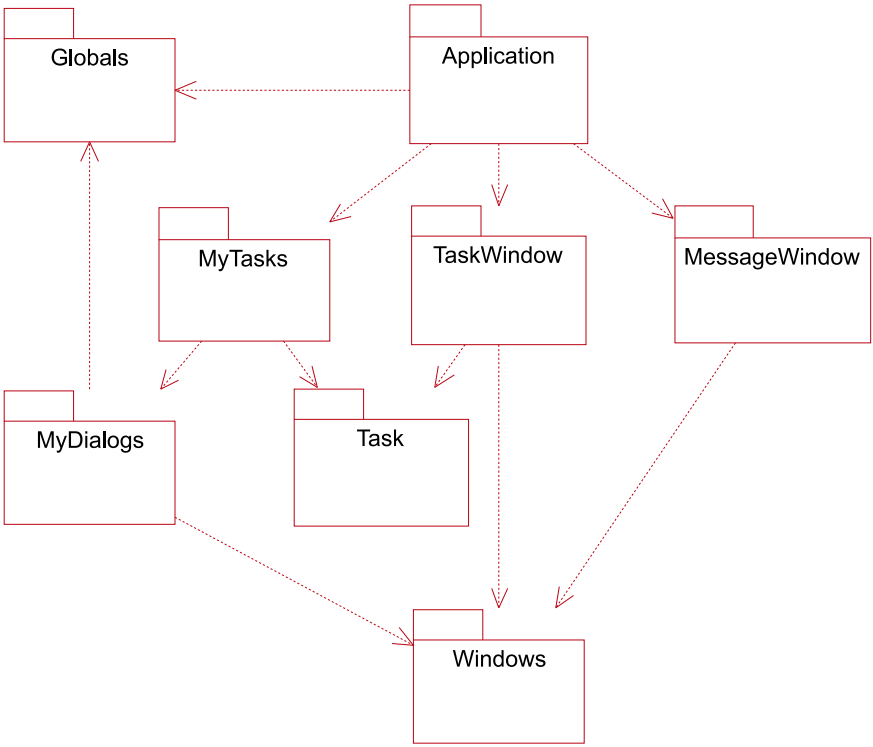

Рис. 5.9: Соблюдение принципа ацикличности.

Глава 6

Метрики

Описанные выше принципы дизайна носят, в основном, качественный характер. Мы много говорили о том, как важно повышать гибкость и переиспользуемость, но никак не пытались измерить их количественно. Конечно, способы метризации этих и других параметров существуют и используются в больших проектах для измерения качества дизайна программных систем, о чем читатель легко найдет массу полезной информации в Internet. Здесь же мы приведем, в качестве примера, метрики для двух важных принципов: *стабильности зависимостей* и *стабильности абстракций*.

6.1 Стабильность зависимостей

Как мы видели раньше, при рассмотрении принципа Деметера, изменения в одних классах системы могут приводить к изменениям в других вследствие изменения их контрактов. Очевидно, что некоторые классы системы изменяются чаще других, и было бы логично с точки зрения улучшения гибкости группировать классы в пакеты таким образом, чтобы частые изменения в одних классах не приводили, по возможности, к каскадному изменению других. Это простое рассуждение приво-

дит нас к понятию *нестабильности* пакета и принципу стабильности зависимостей.

> Нестабильность пакета определяется как
> **I = Ce/(Ca+Ce)**, где
> **Ce** — количество классов внутри пакета, зависящих от классов вовне пакета
> **Ca** — количество классов вовне пакета, зависящих от классов внутри пакета

Иными словами, пакет тем более нестабилен, чем выше количество исходящих зависимостей, а значит, и вероятность его изменения под под воздействием изменений в других пакетах. Если все пакеты системы будут максимально стабильны, то система будет попросту неизменяема, следовательно, часть пакетов должна очень низкую стабильность, а часть - высокую, но связи между ними необходимо выстраивать таким образом, чтобы пакеты с низкой стабильностью не влияли на таковые с высокой, то есть:

SDP[1] — *Принцип стабильности зависимостей*

> Пакет должен зависеть только от более стабильных пакетов. R.Martin, 1996

Действительно, было бы странно пытаться минимизировать количество исходящих связей пакета в надежде улучшить его стабильность, если эти связи ведут к еще более нестабильным пакетам.

6.2 Стабильность абстракций

Принцип стабильности зависимостей хорош для применения и сам по себе, но еще более полезен в сочетании со следующим:

SAP[2] — *Принцип стабильности абстракций*

[1] Stable Dependencies Principle
[2] Stable Abstractions Principle

Стабильность пакета должна быть пропорциональна его абстрактности, где абстрактность пакета определяется как
A = Na/N, где
Na — количество абстрактных классов пакета
N — общее количество классов пакета

Дистанция пакета от прямой (0,1)-(1,0) на графике IA определяется как
D = | A + I - 1 | R.Martin, 1996

Принцип утверждает, что наилучшим балансом между абстрактностью и стабильностью обладают пакеты с наименьшей дистанцией. Рассмотрим применение этого принципа на примере диаграммы 6.1: Диаграмма показывает типичную для архитектур клиент-сервер диаграмму пакетов и классов. Рядом с пакетами указаны их метрики: абстрактность, нестабильность и дистанция. Заметим, что такая диаграмма полностью соответствует принципу стабильности зависимостей, так как все пакеты зависят только от более стабильных. Представим теперь, что по каким-то причинам нам захотелось избавиться от класса ResultSet и переместить класс SomeResultSet на его место в пакет AbstractServer. Это очевидно неправильное решение, так как абстракция более высокого уровня Server начинает зависеть от обычного класса SomeResultSet[3], но посмотрим на диаграмме 6.2, что нам скажут метрики. Как мы видим, выбранные метрики чрезвычайно чуствительны к таким нарушениям, так как дистанция пакетов AbstractServer и MyServer стала равной 1/2 против 0 и 1/3 до внесения изменений.

Некоторые CASE-средства, такие как Together, поддерживают вычисление различных метрик по UML модели. Для вычисления описанных выше метрик стабильности и абстрактности в JAVA-программах можно воспользоваться утилитой jdepend, которую читатель может найти в сети Internet.

[3]Нарушая, таким образом, принцип инверсии зависимости

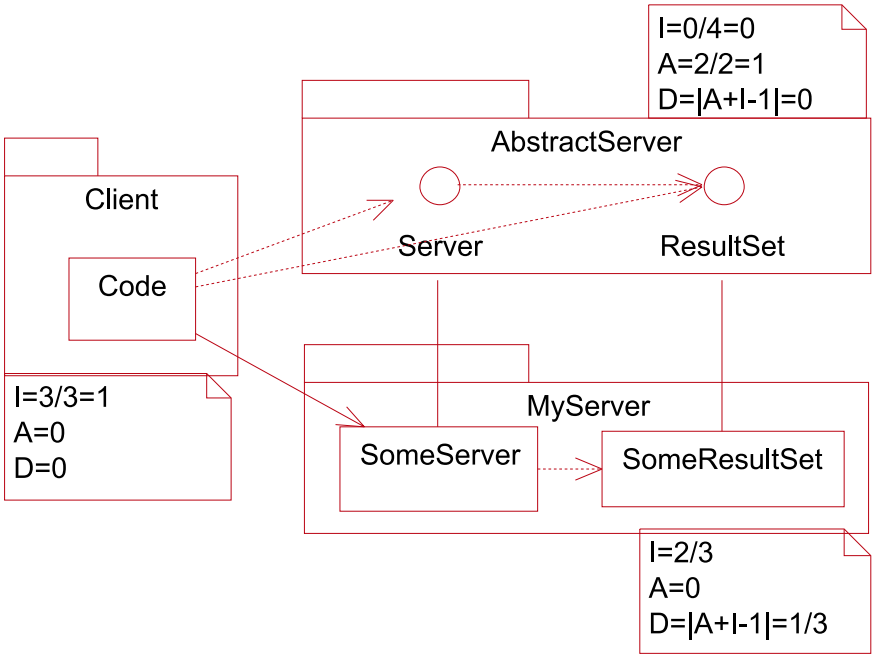

Рис. 6.1: Диаграмма пакетов и классов

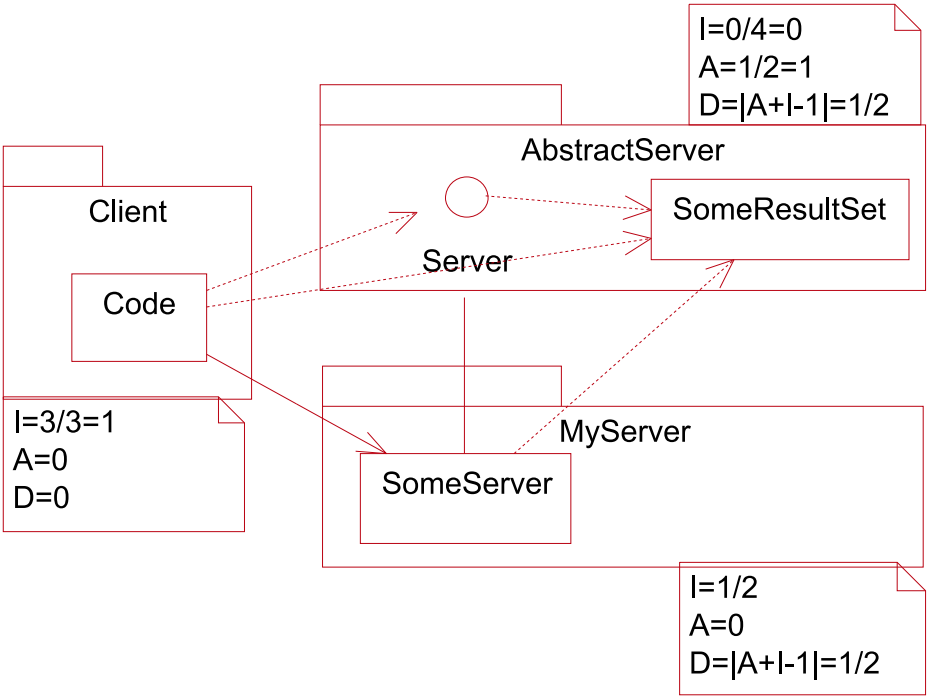

Рис. 6.2: Метрики

Литература

[1] *Г. Буч* Объектно ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд./Пер. с англ. — СПб.; М.: «Невский Диалект» — «Издательство БИНОМ», 1999 г.

[2] *Б. Страуструп* Язык программирования C++, 3-е изд./Пер. с англ. — СПб.; М.: «Невский Диалект» — «Издательство БИНОМ», 1999 г.

[3] OMG Unified Modeling Language Specification version 1.3, *Object Management Group*, 1999, http://www.omg.org

[4] *Буч Г., Рамбо Д., Джекобсон А.* Язык UML. Руководство пользователя: Пер. с англ. — М. ДМК, 2000.

[5] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования, СпБ.: Питер, 2001

[6] - *Martin R.* Granularity, C++ Report, 1996

Оглавление

В. В. Мухортов, В. Ю. Рылов

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ, АЕАЛИЗ И ДИЗАЙН

Методическое пособие

Подписано в печать 20.03.2002. Формат 60x84/16. Печать офсетная.
Уч.-изд. л. 2,0. Усл.-печ. л. 4,16. Тираж 50 экз. Заказ № 21. Бесплатно

Лицензия ПЛД № 57-43 от 22 апреля 1998 г.
Отпечатано на полиграфическом участке ИМ СО РАН
пр. акад. Коптюга, 4, 630090, Новосибирск, Россия