УДК 004.43
ББК 32.973.26-018.1
В73

Вонг, Стивен.

В73 Принципы объектно-ориентированного программирования / Стивен Вонг, Дунг Нгуен. — Москва : Издательство АСТ, 2024. — 192 с. : ил. — (Программирование от экспертов).

ISBN 978-5-17-160272-7.

В данном издании подробно рассматриваются самые важные вопросы, связанные с объектно-ориентированным программированием (ООП), которое предполагает подход к созданию кода как к моделированию информационных объектов. На более высоком абстрактном уровне основная задача ООП — структурирование информации с точки зрения управляемости, что позволяет успешно реализовывать крупные программные проекты. В этой книге пристальное внимание уделяется таким концепциям методологии объектно-ориентированного программирования как абстракция, инкапсуляция, наследование и полиморфизм. Освоив неизменно эффективные принципы ООП, начинающие и опытные программисты получат в свое распоряжение отлично структурированный способ управления сложным кодом, а также смогут более продуктивно использовать и поддерживать его.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-17-160272-7

Перевод на русский язык: ООО «Интеджер»
Издание на русском языке: ООО «Издательство АСТ»

Содержание

Глава 1. Введение 9
1.1. Абстракция 9
1.1.1. Введение 9
1.1.2. Абстракция данных 9
1.1.2.1. Абстракция типа 10
1.1.2.2. Моделирование человека 14
1.2. Объекты и классы 17
1.2.1. Объекты 17
1.2.2. Классы 19
1.2.2.1. Реализация на Java 20
1.2.3. Загрузка кода 23
1.3. Объектные отношения 24
1.3.1. Отношения is-a, или наследование 24
1.3.2. Отношения has-a, или компоновка 27
1.4. Диаграммы UML 29
1.4.1. Диаграммы классов 30

Глава 2. Полиморфизм в действии 33
2.1. Объединительный шаблон проектирования: наследование и полиморфизм 33
2.1.1. Объединительный шаблон проектирования 33
2.1.1.1. Абстракция против общности 35
2.1.2. Наследование и полиморфизм 40
2.1.3. Изучение полиморфизма 41
Упражнения 41
2.2. Ballworld, основанный на наследовании 43
2.2.1. Абстрактные классы 46
2.2.1.1. Абстрактные методы 47

2.2.2. Абстрактные классы против интерфейсов....49
2.2.3. Вариативное и инвариантное поведение....49
2.2.4. Проблемы синтаксиса Java....50
2.2.4.1. Пакеты....50
2.2.4.2. Статические поля и методы....51
2.2.4.3. Вызов методов суперкласса....52
2.3. Ballworld, основанный на компоновке....53
2.3.1. Проблема наследования....53
2.3.2. Пиццы и фигуры....54
2.3.3. От наследования к компоновке....55
2.3.3.1. Составление моделей поведения....58
2.3.4. Решение упражнений главы 2....64

Глава 3. Структура неизменяемого списка....67
3.1. Структура списка и шаблон проектирования компоновщика....67
3.1.1. Отправляясь за покупками....67
3.1.2. Что такое список?....68
3.1.3. Проектирование списков и шаблон проектирования компоновщика....69
3.1.4. Создание списка....70
3.1.5. Обработка списков....71
3.2. Структура списка и шаблон проектирования интерпретатора....72
3.2.1. Обработка списков....72
3.2.1.1. Что может делать список?....72
3.2.1.2. Пример кода....74
3.3. Рекурсия....75
3.3.1. Рекурсивные структуры данных....75
3.3.2. Рекурсивные алгоритмы....76
3.3.3. Хвостовая рекурсия....78
3.4. Шаблон проектирования посетителя....81
3.4.1. Отделение алгоритмов от структур данных....81

3.4.2. Готовить или не готовить 83
3.4.2.1. Еда 83
3.4.2.2. Потребители продуктов питания 83
3.4.2.3. Шеф-повар 84
3.4.2.4. Заказ еды у шеф-повара 85
3.4.3. Шаблон посетителя 87
3.4.4. Фундаментальная методология объектно-ориентированного проектирования 88
3.4.5. Примеры посетителей списков 90
3.5. Абстрактный фабричный шаблон проектирования 93
3.5.1. Сокрытие информации 93
3.5.2. Абстрактное поведение списка 95
3.5.3. Абстрактная фабрика списков 96
3.5.4. Фреймворки 102
3.5.5. Перезагрузка 104
3.6. Внутренние классы 105
3.6.1. Помощники — это варианты 105
3.6.2. Скрытие помощников 108
3.6.3. Анонимные помощники 110
3.6.4. Фабрика с анонимными внутренними классами 112
3.6.5. Классы, определенные внутри другого класса 114
3.6.5.1. Определитель области видимости 114
3.6.5.2. Спецификатор доступа 115
3.6.5.3. Спецификатор расширяемости 115
3.6.5.4. Абстрактный спецификатор 115
3.6.5.5. Спецификатор наследования 115
3.6.5.6. Использование 115
3.6.6. Закрытие 116
3.6.6.1. Примеры 118
3.6.7. Смена «состояний» — переход к нефункциональному программированию 120

Глава 4. Изменяемые структуры данных 124
4.1. Шаблон проектирования состояний 124
4.1.1. Вперед! 127
4.2. Изменяемая линейная рекурсивная структура 128
4.2.1. Шаблон состояния и динамическая реклассификация 128
4.2.2. Фреймворк изменяемой линейной рекурсивной структуры 130
4.2.3. Пример 132
4.3. Структура двоичного дерева 134
4.3.1. Объектная модель двоичного дерева 134
4.3.2. Детали реализации 136
4.4. Массивы и обработка массивов 138
4.4.1. Массивы в Java 138
4.4.1.1. Типы массивов 139
4.4.1.2. Переменные массива 140
4.4.1.3. Создание массива 140
4.4.1.4. Доступ к массивам 141
4.4.2. Обработка массивов при помощи циклов 141
4.4.2.1. Циклы while 145
4.4.2.2. Циклы for-each 147
4.4.3. Массивы против списков 147

Глава 5. Контейнеры с ограниченным доступом 150
5.1. Использование контейнеров с ограниченным доступом 150
5.1.1. Введение 150
5.1.2. Еще о контейнерах с ограниченным доступом 151
5.1.2.1. IRAContainer.java 151
5.1.2.2. IRACFactory.java 153
5.1.3. Примеры 153
5.1.3.1. ALRSRACFactory.java 155
5.1.3.2. LRSStackFactory.java 158

5.1.3.3. LRSQueueFactory.java ... 158
5.1.3.4. RandomRACFactory.java ... 159

Глава 6. Программирование графического интерфейса пользователя ... 160
6.1. Графические пользовательские интерфейсы в Java ... 160
6.1.1. Графические пользовательские интерфейсы в Java ... 160
6.1.2. Абстрактный фрейм (AFrame.java) ... 160
6.1.2.1. Событие (Event) ... 161
6.1.2.2. Шаблонный метод: выражение инвариантного поведения в терминах вариативного поведения ... 162
6.1.3. Простой JFrame (Frameo.java) ... 164
6.1.4. JFrame с кнопками JButton, но без обработчиков событий (Frame1.java) ... 164
6.1.4.1. Шаблон стратегии ... 165
6.2. Еще о программировании графических интерфейсов на Java ... 166
6.2.1. JFrame с кнопками JButton и обработчиками событий (Frame2.java) ... 166
6.2.1.1. Шаблон проектирования команд ... 166
6.2.2. JFrame с кнопками JButton и адаптерами (Frame3.java) ... 166
6.2.2.1. Шаблон нулевого объекта ... 167

Глава 7. Лабораторные работы ... 171
7.1. DrJava ... 171
7.1.1. Редактирование ... 171
7.1.2. Компиляция ... 171
7.1.3. Запуск ... 172
7.1.4. Тестирование ... 172
7.1.5. Отладка ... 172
7.2. Модульное тестирование с помощью JUnit в DrJava ... 176

7.2.1. Использование JUnit в DrJava ... 177
7.2.1.1. assertEquals(...) ... 178
7.2.1.2. assertTrue(...) ... 181
7.2.1.3. fail(...) ... 182
7.2.1.4. Дополнительные возможности ... 182

Глава 8. Ресурсы ... 184
8.1. Синтаксис языка Java ... 184
8.1.1. Конструкторы ... 184

Авторство ... 186

Глава 1.
Введение

1.1. Абстракция

1.1.1. Введение

Абстракция (абстрагирование) — это процесс сокрытия деталей и раскрытия только основных характеристик конкретного понятия или объекта. Ученые-компьютерщики используют абстракцию для понимания и решения проблем и передачи своих решений компьютеру на определенном компьютерном языке. Мы проиллюстрируем этот процесс на примере попытки решить следующую задачу с помощью компьютерного языка Java.

Задача: Дан прямоугольник шириной 4,5 метра и высотой 7,2 метра, вычислите его площадь.

Мы знаем, что площадь прямоугольника равна его ширине, умноженной на высоту. Поэтому все, что нам нужно сделать для решения приведенной выше задачи, — это умножить 4,5 на 7,2 и получить ответ. Вопрос в том, как выразить вышеприведенное решение на языке Java, чтобы компьютер мог выполнить вычисления.

1.1.2. Абстракция данных

Произведение 4,5 на 7,2 выражается на языке Java как: `4.5 * 7.2`. В этом выражении символ `*` обозначает операцию умножения. 4.5 и 7.2 называются *литералами* чисел. Используя среду разработки DrJava, мы можем ввести выражение `4.5 * 7.2` прямо в окно взаимодействия и увидеть ответ.

Теперь предположим, что мы изменим задачу и вычислим площадь прямоугольника шириной 3,6 и высотой 9,3. Изменилась

ли при этом исходная задача? Другими словами, изменилась ли *суть* исходной задачи? Ведь формула для вычисления ответа осталась прежней. Все, что нам нужно сделать, это ввести `3,6 * 9,3`. Что *не* изменилось (*инвариант*)? А что изменилось (*вариант*)?

1.1.2.1. Абстракция типа

Задача не изменилась, поскольку в ней по-прежнему рассматривается одна и та же геометрическая фигура — прямоугольник, описанный в терминах одних и тех же размеров, его ширины и высоты. Что изменилось, так это просто значения ширины и высоты. Формула для вычисления площади прямоугольника с учетом его ширины (width) и высоты (height) не меняется:

```
width * height
```

Здесь неважно, каковы реальные конкретные значения ширины и высоты. Важно только то, что значения ширины и высоты должны быть такими, чтобы операция умножения имела смысл. Как выразить вышеуказанные инварианты на языке Java?

Мы просто хотим считать ширину и высоту заданного прямоугольника элементами множества вещественных чисел. В вычислительной технике мы объединяем значения с общими характеристиками в набор и называем его *типом* (`type`). В Java тип `double` — это набор вещественных чисел, которые реализуются внутри компьютера определенным образом. Детали этого внутреннего представления несущественны для нашей цели и поэтому могут быть проигнорированы. Помимо типа `double` Java предоставляет множество других готовых типов, таких как `int` для представления набора целых чисел и `char` для представления набора символов. Мы рассмотрим и используем их по мере необходимости в будущих примерах. Что касается нашей задачи, то нам нужно ограничиться типом `double`.

Мы можем определить ширину и высоту прямоугольника как `double` в Java следующим образом.

```
double width;
double height;
```

Два приведенных выше утверждения называются определениями *переменных*, где `width` и `height` — это имена переменных. В Java

переменная представляет собой ячейку памяти внутри компьютера. Мы определяем переменную, сначала объявляя ее тип, затем за типом следует имя переменной, и завершаем определение точкой с запятой. Это правило синтаксиса Java. Нарушение синтаксического правила является ошибкой. Когда мы определяем переменную таким образом, связанное с ней содержимое памяти инициализируется значением по умолчанию, заданным языком Java. Для переменных типа `double` значение по умолчанию равно 0.

> **Примечание:** Используйте панель взаимодействия DrJava, чтобы узнать значения ширины и высоты и убедиться, что они равны 0.

После того как мы определили переменные ширины и высоты, мы можем решить нашу задачу, записав выражение, которое вычисляет площадь соответствующего прямоугольника по ширине и высоте следующим образом.

```
width * height
```

Обратите внимание, что два определения переменных вместе с выражением для вычисления площади, представленные выше, напрямую приводят к описанию задачи — два вещественных числа, представляющих ширину и высоту прямоугольника, и высокоуровневое представление о том, каким должно быть решение задачи — площадь равна ширине, умноженной на высоту. Мы только что выразили инварианты задачи и ее решения. Теперь как мы будем изменять ширину и высоту в Java? Мы будем использовать так называемую операцию *присваивания*. Чтобы присвоить переменной `width` значение 4.5, а переменной `height` — значение 7.2, мы напишем следующие операторы присваивания на языке Java:

```
width = 4.5;
height = 7.2;
```

Синтаксическое правило для оператора присваивания в Java таково: сначала пишется имя переменной, за ним следует знак равенства, после знака равенства следует выражение Java, и завершается оно точкой с запятой.

Семантика (то есть смысл) такого присваивания такова: вычислить выражение, стоящее справа от знака равенства, и присвоить полученное значение в ячейку памяти, представленную именем

переменной, стоящей слева от знака равенства. Ошибкой является, если тип выражения в правой части *не является подмножеством* типа переменной в левой части.

Теперь, если мы снова вычислим `width * height` (используя окно взаимодействий DrJava), мы должны получить желаемый ответ. Пока все хорошо, но есть небольшое неудобство: нам приходится вводить выражение `width * height` каждый раз, когда нас просят вычислить площадь прямоугольника с заданной шириной и заданной высотой. Это может быть нормально для такой простой формулы, но что если формула намного сложнее, например, для вычисления длины диагонали прямоугольника? Каждый раз набирать формулу заново — довольно опасный процесс, чреватый ошибками. Есть ли способ заставить компьютер запомнить формулу и выполнить вычисления «за сценой», чтобы нам не пришлось запоминать и переписывать ее самим? Ответ — да, но для достижения этой цели в Java требуется немного больше работы.

Мы хотим построить эквивалент черного ящика с кнопкой, который принимает на вход два вещественных числа (напомним, тип `double`). Когда мы вводим два числа и нажимаем на кнопку, «волшебным образом» черный ящик вычисляет произведение двух введенных чисел и выдает результат, который мы интерпретируем как площадь прямоугольника, ширина и высота которого заданы двумя введенными числами. Этот черный ящик, по сути, является специализированным калькулятором, который может вычислить только одно: площадь прямоугольника, заданного шириной и высотой. Чтобы создать этот ящик в Java, мы используем конструкцию под названием класс, которая выглядит следующим образом.

```
class AreaCalc {
    double rectangle(double width, double height) {
        return width * height;
    }
}
```

Этот Java-код означает примерно следующее: `AreaCalc` — это *проект* специализированной вычислительной машины, способной принимать на вход два значения `double`, одно из которых

обозначено шириной, а другое — высотой, вычислять их произведение и возвращать результат. Этому вычислению дано имя: `rectangle`. На языке Java это называется *методом* класса `AreaCalc`.

Вот пример того, как мы используем `AreaCalc` для вычисления площади прямоугольника шириной 4.5 и высотой 7.2. В панели взаимодействий DrJava введите следующие строки кода.

```
AreaCalc calc = new AreaCalc();
calc.rectangle(4.5, 7.2)
```

Первая строка кода определяет `calc` как переменную типа `AreaCalc` и присваивает ей *экземпляр* класса `AreaCalc`. Слово `new` — это ключевое слово, дескриптор в Java. Оно является примером того, что называется оператором класса. Оно работает с классом и создает экземпляр (также называемый *объектом*) данного класса.

Вторая строка кода — это вызов объекта `calc` для решения задачи о прямоугольнике (`rectangle`), где `width` присвоено значение 4,5, а `height` — 7,2. Чтобы получить площадь прямоугольника 5,6 на 8,4, мы просто снова используем тот же `calc`:

```
calc.rectangle(5.6, 8.4);
```

Поэтому вместо того, чтобы решать только одну задачу — дан прямоугольник шириной 4,5 метра и высотой 7,2 метра, вычислить его площадь, — мы построили «машину», которая может вычислить площадь *любого* заданного прямоугольника. Но как же вычислить площадь правильного треугольника с высотой 5 и основанием 4? Мы не можем просто использовать этот же калькулятор. Нам нужен другой специализированный калькулятор, который может производить другие вычисления.

Существует как минимум два варианта конструкции такого калькулятора.

- Создать новый класс `AreaCalc2` с одним методом под названием `rightTriangle` с двумя входными параметрами типа `double`. Это соответствует проектированию другого калькулятора площади с кнопкой с названием `rightTriangle` (правильный треугольник) и двумя входными слотами (полями ввода информации).
- Добавить в `AreaCalc` метод `rightTriangle` с двумя входными параметрами типа `double`. Это соответствует проектирова-

нию калькулятора площади с двумя кнопками: одной с надписью `rectangle` с двумя входными параметрами и другой с надписью `rightTriangle`, также с двумя входными параметрами.

В любом случае пользователь калькулятора обязан выбрать подходящий калькулятор или нажать соответствующую кнопку на калькуляторе, чтобы правильно рассчитать площадь заданной геометрической фигуры. Поскольку для этих двух вычислений требуется одинаковое количество входных параметров одного и того же типа, пользователь калькулятора должен быть внимателен, чтобы не перепутать их. Это может не доставить особых неудобств, если на выбор есть только два вида фигур: прямоугольник и правильный треугольник. Но что, если пользователю придется выбирать из сотен различных фигур? Или из бесконечного числа фигур? Как мы, программисты, можем создать калькулятор, способный работать с *бесконечным* числом фигур? Ответ кроется в *абстракции*. Чтобы понять, как концептуализировать проблему, давайте отвлечемся и рассмотрим поведение ребенка!

1.1.2.2. Моделирование человека

Первые несколько лет своей жизни Питер не знал, что такое день рождения, не говоря уже о дате своего рождения. Он был неспособен ответить на ваш вопрос о своем дне рождения. Именно его родители планировали тщательно продуманные вечеринки по случаю его дня рождения на месяцы вперед. Тогда мы могли считать Питера «не очень умным» человеком с очень низким уровнем интеллекта и способностей. Но теперь Питер — студент колледжа. В его мозгу есть кусочек памяти, который хранит дату его рождения: это 12 сентября 1985 года! Теперь Питер довольно умный человек. Он может вычислить, сколько месяцев осталось до следующего дня рождения, и отправить по электронной почте список желаний за два месяца до дня рождения. Как смоделировать такого «умного» человека, как Питер? Моделирование такого человека подразумевает:

- моделирование даты рождения;

- вычисление количества месяцев до следующего дня рождения с учетом текущего месяца.

Дата рождения состоит из месяца, дня и года. Каждое из этих данных может быть представлено целым числом, которое в Java называется числом типа `int`. Как и при вычислении площади прямоугольника, вычисление количества месяцев до следующего дня рождения с учетом текущего месяца может быть представлено в виде метода некоторого класса. В этом случае, в отличие от калькулятора площади, мы объединим в один класс и данные (то есть дату рождения), и вычисления, связанные с датой рождения. Группировка данных и вычислений над ними в одном классе называется *инкапсуляцией*. Ниже приведен Java-код, моделирующий умного человека, который знает, как вычислить количество месяцев до своего следующего дня рождения. Номера строк указаны для удобства и не являются частью кода.

```
1  | public class Person {
2  |         /**
3* |         Все поля данных являются закрытыми, чтобы
предотвратить коду за пределами
4  |         этого класса доступ к ним.
5  |         */
6  |         private int bDay; // день рождения
7  |         private int bMonth; // месяц рождения; например, 3
означает Март.
8  |         private int bYear; // год рождения
9  |
10 |         * Конструктор: специальный код, используемый для
инициализации полей класса.
11 |         * Единственный способ создать объект Person – это
вызвать в конструкторе new.
12 |         * Например: new Person(28, 2, 1945) создаст объект
Person
13 |         * с датой рождения 28 февраля 1945 года.
14 |         */
15 |         public Person(int day, int month, int year) {
16 |             _bDay = day;
17 |             _bMonth = month;
18 |             _bYear = year;
19 |         }
```

```
20 |         /**
21 |         * Использует арифметические вычисления "modulo"
для вычисления количества месяцев до следующего
22 |         * дня рождения с учетом текущего месяца.
23 |         * @param currentMonth – это int, представляющий
текущий месяц.
24 |         */
25 |         public int nMonthTillBD(int currentMonth) {
26 |             return (_bMonth - currentMonth + 12) % 12;
27 |         }
28 |  }
```

Загрузите приведенный выше код:

Person.java (http://www.opentextbooks.org.hk/system/files/resource/8/8163/8170/media/Person.java

Теперь мы объясним построчно, что означает приведенный выше Java-код.

- В строке 1 определяется класс `Person`. Открывающая фигурная скобка в конце строки и соответствующая закрывающая скобка в строке 28 разграничивают содержимое класса `Person`. Ключевое слово `public` называется *спецификатором доступа* и означает, что весь Java-код в системе может ссылаться на этот класс.
- Строки 2–5 — это комментарии. Все, что находится между `/*` и `*/`, компилятор игнорирует.
- Строки 6–8 определяют три целочисленные переменные. Эти переменные называются *полями* класса. Ключевое слово `private` — это еще один спецификатор доступа, который предотвращает доступ кода за пределами класса. Только код внутри класса может получить к ним доступ. За каждым полем следует комментарий, разделенный символами `//` и конца строки. Таким образом, в Java существует два способа комментирования кода: начинать с `/*` и заканчивать `*/` или начинать с `//` и заканчивать концом строки.
- Строки 9–14 — это комментарии.