Парадигмы программирования

Альжанов Леонид

02.03.2025

Содержание

Объектно-ориентированное программирование

Программирование по контракту

Рассмотрим данную функцию:

```
int power(int a, int n) {
    int r = 1;
    while (n != 0) {
        if (n % 2 == 1) {
            r *= a;
        }
        n /= 2;
        a *= a;
    }
    return r;
}
```

У нас есть предположение, что эта функция делает бинарное возведение в степень. Но мы в этом не уверены. Мы внимательно посмотрели на код и поняли, что вот это и правда возведение в степень. Но в реальности такое понимать довольно сложно. Давайте научимся **доказывать**, что наша программа делает именно то, что она должна на понятном математическом языке. Мы будем смотреть на программу, как на математический объект. Мы воспользуемся **тройками Хоара**.

В терминологии Хоара у нас есть:

- Блок кода C.
- Предусловие P — то, что должно выполняться, чтобы исполнить.
- Постусловие Q — то, что гарантирует блок C в результате того, что он был исполнен.

Обозначается в коде вот так:

```
// Pred: P
C
// Post: Q
```

Теорема Бёма - Якопини

Она гласит, что любой управляющий граф (любой код программы) можно составить из:

- Присвоения.
- Последовательного исполнения.
- Ветвления.
- Циклов while.

Присвоение

Пусть есть какое-то выражение expr и предусловие $P[x \to \text{expr}]$ (условие от x, где оно заменено на x). Тогда постусловием к $x := \text{expr}$ будет P.

```
// Pred: P[x -> expr]
x = expr;
// Post: P
```

Примеры:

```
// Pred: b = 0
a = b;
// Post: a = 0
```

```
// Pred: a + a = b
a += a;
// Post: a = b
```

Последовательное исполнение

Из:

```
// Pred: P1
S1
// Post: Q1
```

```
// Pred: P2
S2
// Post: Q2
```

$Q1 \Rightarrow P2$

Следует:

```
// Pred: P1
S1
S2
// Post: Q2
```

Ветвление

Из:

```
// Pred: P && cond
S1
// Post: Q
```

```
// Pred: P && !
cond
```

```
S2
// Post: Q
```

Следует:

```
// Pred: P
if (cond) {
    S1
} else {
    S2
}
// Post: Q
```

Цикл

Из:

```
// Pred: P && cond
S
// Post: P
```

Следует:

```
// Pred: P
while (cond) {
    S
}
// Post: P && !cond
```

В данном случае условие P называется инвариантом цикла. Но нужно доказать, что цикл завершится.

Давайте докажем работу функции из начала.

```
// Pred: n >= 0
// Post: R = a ^ n
int power(int a, int n) {
    int r = 1;
    // r' * a' ^ n' = a ^ n
    while (n != 0) {
        // I && n' != 0
        if (n % 2 == 1) {
            // I && n' % 2 = 1
            r *= a; n--;
            // I && n' % 2 = 0
        } else {
            // I && n' % 2 = 0
        }
        // I && n' % 2 = 0
        n /= 2; a *= a;
        // I
    }
    // r' * a' ^ n' = a ^ n && n' = 0
    // => r' = a ^ n
    // => R = r'
    return r;
}
```

Функции

Чистые функции

- Результат зависит только от аргументов
- Не имеет побочных эффектов (не меняют ничего снаружи самой себя)

Предусловие — условие, которое должно быть верно на момент вызова. Результат вызова с неверным предусловием не определен.

Постусловие — условие, которое верно на момент возврата. Если постусловие не выполнено, то в программе есть ошибка.

Например:

```
// Pred: x > 0
// Post: R * R = x ^ R >= 0
double sqrt(double x) {
    ...
}
```

Функции с состоянием

```
// Состояние
int value = 0;

// Pred: v >= 0
// Post: R = value + v && value' = value + v
int add(int v) {
    return value += v;
}
```

Добавим пред и постусловия на неотрицательность `value`:

```
// Состояние
int value = 0;

// Pred: v >= 0 && value >= 0
// Post: R = value + v && value' = value + v && value >= 0
int add(int v) {
    return value += v;
}
```

Инвариант — общая часть пред и постусловия. Выполняется всегда. Обозначается как `Inv`.

```
// Inv: value >= 0
int value = 0;

// Pred: v >= 0
// Post: R = value + v && value' = value + v
int add(int v) {
    return value += v;
}
```

Инвариант + предусловие + постусловие функции с состоянием называется **контрактом**.

Реализация ООП

Давайте напишем структуру данных стек.

Описание

- Переменные
  - `size` — число элементов
  - `elements` — массив элементов
- Методы:
  - `push(element)` — добавить элемент
  - `pop()` — удалить элемент
  - `peek()` — получить элемент на вершине
  - `size()` — число элементов
  - `isEmpty()` — проверка на пустоту

Модель

- Последовательность чисел $a_1, a_2, ..., a_n$. Операции выше проводятся с последним элементом.
- Инвариант:
  - $n \geq 0$
  - $\forall i = 1, ..., n : a_i \neq \text{null}$
- Вспомогательные определения:
  - $\text{immutable}(k) = \forall i = 1, ..., k : a_i{}' = a_i$

Контракт

- `push(element)`:

```
// Pred: element != null
// Post: n' = n + 1 && immutable(n) && a'[n'] = element
void push(Object element)
```

- `pop()`:

```
// Pred: n > 0
// Post: R = a[n] && n = n' - 1 && immutable(n')
Object pop()
```

- `peek()`:

```
// Pred: n > 0
// Post: R = a[n] && n = n' && immutable(n)
Object pop()
```

- `size()`:

```
// Pred: true
// Post: R = n && n = n' && immutable(n)
Object pop()
```

- `isEmpty()`:

```
// Pred: true
// Post: R = n > 0 && n = n'&& immutable(n)
Object pop()
```

Процедурная реализация

Мы показывем, как работает структура стека, но только на одном экземпляре. Процедурная реализация создана для простых задач, но ее трудно поддерживать и расширять для сложных проектов.

```
public class ArrayStackModule {
    private static int size;
    private static Object[] elements = new Object[1];

    private static void ensureCapacity(int capacity) {
        if (capacity > elements.length) {
            elements = Arrays.copyOf(elements, 2 * capacity);
        }
    }

    public static void push(Object element) {
        Objects.requireNonNull(element);
        ensureCapacity(size + 1);
        elements[size++] = element;
    }

    public static Object pop() {
        assert size > 0;
        size--;
        Object result = elements[size];
        elements[size] = null;
        return result;
        // Если мы собрались управлять памятью руками,
        // то освобождать её мы тоже должны руками
    }

    public static Object peek() {
```

```
        assert size > 0;
        return elements[size - 1];
    }

    public static int size() {
        return size;
    }

    public static boolean isEmpty() {
        return size == 0;
    }
}
```

Пример использования:

```
for (int i = 0; i < 10; i++) {
    ArrayStackModule.push(i);
}
```

Реализация на структурах

Будем передавать в каждую процедуру экземпляр, с которым происходит действие. Это позволит создать несколько отдельно существующих экземпляров. Это абстрактный класс с явной передачей ссылки на результат. Улучшает связь данных, но все еще не подходит, так как никто не мешает нам производить действия в данной структуре в другом файле(например).

```
public class ArrayStackADT {
    private static int size;
    private static Object[] elements = new Object[1];

    public static ArrayStackADT create() {
        ArrayStackADT stack = new ArrayStackADT();
        stack.elements = new Object[1];
        return stack;
    }

    private static void ensureCapacity(ArrayStackADT stack, int capacity) {
        if (stack.elements.length < capacity) {
            stack.elements = Arrays.copyOf(stack.elements, capacity * 2);
        }
    }

    public static void push(ArrayStackADT stack, Object element) {
        Objects.requireNonNull(element);
        ensureCapacity(stack, stack.size + 1);
        stack.elements[stack.size++] = element;
    }

    public static Object pop(ArrayStackADT stack) {
        assert stack.size > 0;
        stack.size--;
        Object result = stack.elements[stack.size];
        stack.elements[stack.size] = null;
        return result;
```

```
    }

    public static Object peek(ArrayStackADT stack) {
        assert stack.size > 0;
        return stack.elements[stack.size - 1];
    }

    public static int size(ArrayStackADT stack) {
        return stack.size;
    }

    public static boolean isEmpty(ArrayStackADT stack) {
        return stack.size == 0;
    }
}
```

Пример использования:

```
ArrayStackADT stack = ArrayStackADT.create();
for (int i = 0; i < 10; i++) {
    ArrayStackADT.push(stack, i);
}
```

Преобразование в класс (ООП)

По сути классы — синтаксический сахар поверх реализации выше. У каждого метода есть невидимый аргумент `this`, к которому обращаются при обращении к любому полю класса.

Наиболее мощный и гибкий подход для больших и сложных проектов, обеспечивающий инкапсуляцию, наследование и полиморфизм, но требует больше планирования и проектирования.

```
public class ArrayStack {
    private int size;
    private Object[] elements = new Object[1];

    private void ensureCapacity(int capacity) {
        if (elements.length < capacity) {
            elements =
                Arrays.copyOf(elements, capacity * 2);
        }
    }

    public void push(Object element) {
        Objects.requireNonNull(element);
        ensureCapacity(size + 1);
        elements[size++] = element;
    }

    public Object pop() {
        assert size > 0;
        size--;
        Object result = elements[size];
        elements[this] = null;
        return result;
    }
```

```
    public Object peek() {
        assert size > 0;
        return elements[size - 1];
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

Пример использования:

```
ArrayStack stack = new ArrayStack();
for (int i = 0; i < 10; i++) {
    stack.push(i);
}
```

Конструкторы

Давайте создадим новый тип стека на основе связного списка. Для этого нужна структура, хранящая информацию об элементе связного списка. Не забываем про инвариант, который хочется соблюдать.

```
// Inv: value != null
public class Node {
   Object value;
   Node next;
}
```

Заметим, что если не инициализировать наши значения, то инвариант будет нарушаться, так как по умолчанию `value = null`. Создадим метод для инициализации значений:

```
public init(Object value, Node next) {
    assert value != null;
    this.value = value;
    this.next = next;
}
```

Для инициализация объектов придумали специальный метод, который выполняется во время создания объекта, называющийся **конструктором**. В Java и в многих других языках он объявляется так:

```
public Node(Object value, Node next) {
    assert value != null;
    this.value = value;
    this.next = next;
}
```

Во время создания нового объекта конструктор вызывается так:

```
```

```
Node node = new Node(1, null);
```

Если других конструкторов нет, то создаётся конструктор по умолчанию. У него нет аргументов, и он оставляет все значения такими, как они объявлены.

Интерфейсы

Вернёмся к созданию стека на основе связного списка. У класса будет всего 2 поля: размер и указатель на вершину стека:

```
private int size;
private Node head;
```

Начнём реализовывать методы, которые есть в другом стеке:

```
// Pred: element != null
// Post: n = n' + 1 && forall i = 1, ..., n': a[i]' = a[i] && a[n] = element
public void push(Object element) {
    assert element != null;
    size++;
    head = new Node(element, head);
}
```

```
// Pred: n > 0
// Post: R = a[n + 1] && n = n'- 1 && forall i = 1, ..., n: a[i]' = a[i]
public Object pop() {
    assert size > 0;
    size--;
    Object result = head.value;
    head = head.next;
    return result;
}
```

```
// Pred: n > 0
// Post: R = a[n] && immutable
public Object peek() {
    assert size > 0;
    return head.value;
}
```

```
// Post: R = n && immutable
public int size() {
    return size;
}
```

```
// Post: R = n > 0 && immutable
public boolean isEmpty() {
    return size == 0;
}
```

Пример использования:

```
LinkedStack stack = new LinkedStack();
for (int i = 0; i < 10; i++) {
```

```
    stack.push(i);
}
```

Возникает проблема. В обоих реализациях стека одинаковая модель, одинаковый контракт и почти одинаковое использование. Самое главное, что во время использования, код, использующий стек, знает, с каким видом стека он работает. Нам хотелось бы, чтобы коду не было разницы, с чем он работает, лишь бы это выполняло контракт.

Решение есть, и оно называется **интерфейсом**. Он представляет собой набор методов и контракт, который диктует, что должна соблюдать любая его реализация.

На нашем примере со стеками, этот интерфейс выглядит так:

```
public interface Stack {
    void push(Object element);
    Object pop();
    Object peek();
    int size();
    boolean isEmpty();
}
```

Все методы, объявляющиеся в интерфейсе, являются публичными. Для того, чтобы объявить имплементацию интерфейса, в Java нужно написать:

```
public class ArrayStack implements Stack { ... }
public class LinkedStack implements Stack { ... }
```

В таком случае, реализация методов проверяется компилятором, а выполнение контракта гарантируется программистом. Теперь стеками можно пользоваться так:

```
public static void fill(Stack stack) {
    for (int i = 0; i < 10; i++) {
        stack.push(i);
    }
}
```

Заметим, что код действительно не знает, с каким стеком он работает.

Теперь нужно понять, как внутренне работают интерфейсы.

Реализация на основе `instanceof`

Поведение можно реализовать при помощи команды `a instanceof T`, которая выдаёт `true`, если объект `a` может быть приведён к типу `T`:

```
class StackImpl {
    public static void pop(Stack this) {
        if (this instanceof ArrayStack) {
            return ArrayStack.pop(this);
        } else if (this instanceof LinkedStack) {
            return LinkedStack.pop(this);
        } else {
            // ?
        }
    }
```

```
    ...
}
```

Плюс в данной реализации заключается в простоте реализации, однако подобный способ приводит к разбуханию кода, и работе каждого метода за линейное от количества реализующих классов время.

Реализация на основе таблицы виртуальных функций

Вместо линейной проверки принадлежанию конкретному классу, В каждой реализации создадим таблицу указателей на функции, реализующие конкретные методы. Если в Java можно было делать указатели на функции, то это бы выглядело вот так:

```
class StackImpl {
    public static void pop(Stack this) {
        this.vtbl["pop()"]();
    }
    ...
}
```

Плюс в этой реализации заключается в константном от количества реализующих классов времени, но возникают проблемы:

1. Как делать множественное наследование
2. Затраты памяти на хранение таблицы

Первую проблему можно решить, храня таблицу таблиц указателей на функции, где первым ключом будет название интерфейса, а вторым - название метода:

```
public static void pop(Stack this) {
    this.vtbl["Stack"]["pop()"]();
}
```

Вторую проблему можно решить путём замены имён на уникальные номера во время компиляции и использование обычного массива ссылок вместо таблицы строк.

Полиморфизм

Суть полиморфизма заключается в использовании одного и того же кода для разных типов. Есть два вида полиморфизма:

- Ad-hoc — для каждого типа свое поведение.
- Универсальный — одинаковое поведение для всех типов.

Пример полиморфизма для наших стеков:

```
void test(LinkedStack stack) { ... }
void test(ArrayStack stack) { ... }
...
test(new LinkedList());
test(new ArrayList());
```

Есть две разные функции для разных типов с одним и тем же названием — Ad-hoc полиморфизм.

Наследование

Заметим, что у двух реализаций стека есть много общего поведения:

- `size()` всегда просто возвращает значение поля `size`

- `isEmpty()` всегда возвращает равенство `size` и 0
- `push(element)` всегда увеличивает `size` на 1 и проверяет `element` на `null`
- `pop()` всегда уменьшает `size` на 1 и проверяет на стек на пустоту
- `peek()` всегда проверяет на стек на пустоту

Это общее поведение иногда полностью совпадает:

```
public boolean isEmpty() {
    return size == 0;
}
```

А иногда зависит от реализации:

```
public Object peek() {
    assert size > 0;
    // Получить элемент
    // Зависит от реализации
}
```

Решение есть — абстрактные методы. Они могут быть объявлены в абстрактном классе. Теперь наши классы будут выглядеть так:

```
public abstract class AbstractStack implements Stack {
    protected int size;
    ...
    protected abstract Object doPeek();

    public Object peek() {
        assert size > 0;
        return doPeek();
    }
    ...
}
```

```
public class ArrayStack extends AbstractStack {
    ...
    protected Object doPeek() {
        return elements[size - 1];
    }
    ...
}
```

```
public class LinkedStack extends AbstractStack {
    ...
    protected Object doPeek() {
        return head.value;
    }
    ...
}
```

Информация о курсе

Поток — y2024.
Группы M3138-M3139.
Преподаватель — Корнеев Георгий Александрович.