|  |
| --- |
| **Парадигмы программирования** |
| * [Императивная](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Процедурная](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Структурная](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Аспектно-ориентированная](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D0%BF%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Объектно-ориентированная](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [Агентно-ориентированная](https://ru.wikipedia.org/wiki/%D0%90%D0%B3%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4)     - [Компонентно-ориентированная](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [Прототипно-ориентированная](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Обобщённое программирование](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BE%D0%B1%D1%89%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) * [Декларативная](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Чистота языка](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)     - [Чистота функции](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%B8)   + [Функциональная](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [В терминах Рефал-машины](https://ru.wikipedia.org/wiki/%D0%A0%D0%95%D0%A4%D0%90%D0%9B)     - [Аппликативная](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%BB%D0%B8%D0%BA%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [Комбинаторная](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%B1%D0%B8%D0%BD%D0%B0%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [Бесточечная](https://ru.wikipedia.org/wiki/%D0%91%D0%B5%D1%81%D1%82%D0%BE%D1%87%D0%B5%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)       * ([чистая](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D1%82%D0%BE%D1%82%D0%B0_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) [конкатенативная](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%BA%D0%B0%D1%82%D0%B5%D0%BD%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F" \o "Конкатенативный язык программирования))   + [Логическая](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [Ограничениями](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2_%D0%BE%D0%B3%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%B5%D0%BD%D0%B8%D1%8F%D1%85) * [Конкатенативная](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%BA%D0%B0%D1%82%D0%B5%D0%BD%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) * [Векторная](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1)[[en]](https://en.wikipedia.org/wiki/Array_programming) * [Метапрограммирование](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%B0%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Языково-ориентированная](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA%D0%BE%D0%B2%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)     - [Предметно-ориентированная](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%BC%D0%B5%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA)     - [Пользователями](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1)[[en]](https://en.wikipedia.org/wiki/End-user_development)   + [Автоматизация процесса программирования](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)   [Рефлексивность](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))  [Гомоиконность](https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D0%BC%D0%BE%D0%B8%D0%BA%D0%BE%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C)   * Связанные темы   + [Программирование в крупном и мелком масштабе](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2_%D0%BA%D1%80%D1%83%D0%BF%D0%BD%D0%BE%D0%BC_%D0%B8_%D0%BC%D0%B5%D0%BB%D0%BA%D0%BE%D0%BC_%D0%BC%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B5&action=edit&redlink=1)[[en]](https://en.wikipedia.org/wiki/Programming_in_the_large_and_programming_in_the_small)   + [Модульность](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))   + [Полиморфизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))   + [Продолжения](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B4%D0%BE%D0%BB%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и [CPS](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%B2_%D1%81%D1%82%D0%B8%D0%BB%D0%B5_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%BF%D1%80%D0%BE%D0%B4%D0%BE%D0%BB%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9)   + [Параллелизм](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) и [конкурентность](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D0%B8%D0%B7%D0%BC_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)" \o "Параллелизм (информатика)) * Методы и алгоритмы   + [Автоматное](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Потоков данных](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%B2_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)   + [Событийно-ориентированное](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Реактивное](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B0%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)   + [Сервис-ориентированное](https://ru.wikipedia.org/w/index.php?title=%D0%A1%D0%B5%D1%80%D0%B2%D0%B8%D1%81-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) |

## Основные модели программирования

* [Императивное программирование](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Декларативное программирование](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Структурное программирование](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Функциональное программирование](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Логическое программирование](https://ru.wikipedia.org/wiki/%D0%9B%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Объектно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
  + [Программирование, основанное на классах](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#%D0%A2%D1%80%D0%B0%D0%B4%D0%B8%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4)
  + [Программирование, основанное на прототипах](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
  + [Субъектно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

Подходы и приёмы

* [Структурное программирование](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Процедурное программирование](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Аппликативное программирование](https://ru.wikipedia.org/wiki/%D0%90%D0%BF%D0%BF%D0%BB%D0%B8%D0%BA%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B4%D1%85%D0%BE%D0%B4_%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8E)
* [Обобщённое программирование](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BE%D0%B1%D1%89%D1%91%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Доказательное программирование](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D0%B0%D0%B7%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Порождающее программирование](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B5%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Аспектно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D0%BF%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Агентно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%90%D0%B3%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Рекурсия](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BA%D1%83%D1%80%D1%81%D0%B8%D1%8F)
* [Автоматное программирование](https://ru.wikipedia.org/wiki/%D0%90%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Событийно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%B1%D1%8B%D1%82%D0%B8%D0%B9%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Компонентно-ориентированное программирование](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)
* [Грамотное программирование](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%BC%D0%BE%D1%82%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

Шаблоны проектирования

«Банда четырех» описывает следующие **ЧЕТЫРЕ ЭЛЕМЕНТА ШАБЛОНА**

Название шаблона — идентификатор, который можно использовать для описания про­блемы проектирования, ее решений и последствий одним или двумя словами. Подбор удачных названий — один из самых сложных этапов процесса пополнения нашего каталога.

Проблема — описывает, когда следует применять шаблон. В данном случае объясня­ется проблема и ее суть. Иногда проблема будет включать список условий, которые должны быть предварительно соблюдены, чтобы применение шаблона имело смысл.

Решение — описывает элементы, образующие конструкцию, их отношения, функции и взаимодействия

Последствия — результаты и компромиссные решения в случае применения шаблона.  
Перечисление последствий явным образом поможет вам понять и оценить их.

Исторической справка: парадигма «Модель/Вид/ Контроллер» (Model/View/Controller — MVC).

**Порождающие шаблоны**

 Abstract factory (Абстрактная фабрика);

 Builder (Строитель);

 Factory method (Фабричный метод);

 Prototype (Прототип);

 Singleton (Одиночка).

**Структурные шаблоны**

Структурные шаблоны используются для создания более крупных конструкций из групп объектов.

Adapter (Адаптер);

Bridge (Мост);

Composite (Компоновщик);

Decorator (Декоратор);

Facade (Фасад);

Flyweight (Приспособленец);

Proxy (Заместитель).

**Поведенческие шаблоны**

К числу поведенческих шаблонов относятся следующие:

Chain of response (Цепочка ответственности);

Command (Команда);

Interpreter (Интерпретатор);

Iterator (Итератор);

Mediator (Посредник);

Memento (Хранитель);

Observer (Наблюдатель);

State (Состояние);

Strategy (Стратегия);

Template method (Шаблонный метод);

Visitor (Посетитель).

шаблон Iterator (Итератор) - широко используемых шаблонов, кото­рый в то же время реализован несколькими языками программирования.

**Антишаблоны**

В то время как шаблоны проектирования развиваются из успешных практик, *анти­шаблоны* можно представлять себе как наборы неудавшихся методик. Существуют убедительные документальные доказательства того, что большинство программных проектов в конечном счете признаются неудачными - по меньшей мере одна треть всех проектов полностью отменяется. По-видимому, многие из этих неудач обусловлены плохими проектными реше­ниями.

Два варианта антишаблонов:

описывающие плохое решение проблемы, которое приводит к возникновению скверной ситуации;

объясняющие, как выйти из скверной ситуации, а затем выработать хорошее решение.

Таким образом, антишаблоны приводят к пересмотру существующих конструк­ций и их непрерывному рефакторингу до тех пор, пока не будет найдено приемле­мое решение.

Шаблоны проектирования

Одна из любопытных особенностей разработки программного обеспечения заклю­чается в том, что при создании программной системы вы фактически моделируете реальную систему. Например, в случае с индустрией информационных технологий можно с уверенностью сказать, что эти технологии *являются* бизнесом — или, по крайней мере, они *претворяют в жизнь* бизнес. Чтобы создавать программные бизнес-системы, разработчики должны хорошо понимать бизнес-модели. Вслед­ствие этого они зачастую гораздо лучше знают бизнес-процессы компаний.

Мы уже сталкивались к этой концепцией на всем протяжении книги, поскольку она связана с нашими образовательными исследованиями. Например, когда мы го­ворили об использовании наследования для абстрагирования поведений и атрибутов классов млекопитающих, соответствующая модель базировалась на действительной, реальной модели, а не вымышленной, которую мы создали для собственных целей.

Таким образом, создав класс Mammal, мы можем применять его для генерирования бесчисленного множества других классов, например Dog, Cat и т. д., поскольку все классы млекопитающих будут совместно использовать определенные поведения и атрибуты. Такой подход работает, если речь идет о Dog, Cat, Squirrel и других классах млекопитающих, поскольку мы видим шаблоны. Эти шаблоны позволяют нам про­инспектировать тот или иной класс, представляющий животных, и решить, действи­тельно ли он является классом млекопитающих или, возможно, классом рептилий, для которых потребуются другие шаблоны, касающиеся поведений и атрибутов.

На всем протяжении истории люди использовали шаблоны во многих областях своей деятельности, включая проектирование. Эти шаблоны идут рука об руку со «Святым Граалем» разработки программного обеспечения, которым является повтор­ное использование программного кода. В этой главе мы рассмотрим шаблоны проек­тирования — относительно новую область в разработке программного обеспечения (первая книга, посвященная шаблонам проектирования, была издана в 1995 году).

Шаблоны проектирования, пожалуй, являются самыми важными достижения­ми участников объектно-ориентированного движения за несколько последних лет. Они идеально сочетаются с концепцией разработки программного обеспечения, пригодного для повторного использования. Поскольку объектно-ориентированная разработка всецело связана с повторным использованием, шаблоны прекрасно в нее вписываются.

Базовая концепция шаблонов проектирования «вращается» вокруг принципа наиболее оптимальных методик. Под использованием *наиболее оптимальных ме­тодик* мы подразумеваем, что при создании хороших и эффективных решений они документируются таким образом, что другие разработчики смогут извлечь пользу из того, чего кто-то успешно добился ранее.

Одна из важнейших книг, посвященных объектно-ориентированной разработ­ке программного обеспечения, называется «Приемы объектно-ориентированного проектирования. Паттерны проектирования» *(Design Patterns: Elements of Reusable Object-Oriented Software)*, авторами которой выступили Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides). Появление этой книги стало важным событием в ин­дустрии ПО, при этом термины из нее настолько укоренились в лексиконе, ис­пользуемом в сфере информатики, что авторы приобрели известность как «Бан­да четырех» (GoF — Gang of Four).

Цель этой главы — объяснить, что такое шаблоны проектирования (подробное описание каждого шаблона проектирования выходит далеко за рамки этой книги и заняло бы не один том). Для этого мы исследуем каждую из трех категорий ша­блонов проектирования (порождающие, структурные и поведенческие), как опре­делено «Бандой четырех», и рассмотрим конкретный пример одного шаблона из каждой категории.

**Зачем нужны шаблоны проектирования**

Нельзя сказать, что концепция шаблонов проектирования появилась от необходи­мости в программном обеспечении, пригодном для повторного использования. Фактически основополагающий труд по шаблонам проектирования посвящен воз­ведению зданий и городов. Как отмечает Кристофер Александер (Christopher Alexander) в книге «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)*, «каждый шаблон описывает проблему, которая снова и снова возникает в нашей среде, а также ядро решения этой про­блемы таким путем, что вы сможете использовать это решение миллион раз, ни­когда не делая это дважды одинаковым образом».

**ЧЕТЫРЕ ЭЛЕМЕНТА ШАБЛОНА**

«Банда четырех» описывает следующие четыре важные характеристики шаблона.

Название шаблона — идентификатор, который можно использовать для описания про­блемы проектирования, ее решений и последствий одним или двумя словами. Неза­медлительное присвоение названия шаблону расширяет наш словарь проектировщи­ка. Все это позволяет нам проектировать на более высоком уровне абстрагирования. Располагая словарем шаблонов, мы можем говорить о них с коллегами, упоминать их в своей документации и даже дискутировать на эту тему с самими собой. Так легче обдумывать конструкции и разъяснять их другим людям. Подбор удачных названий — один из самых сложных этапов процесса пополнения нашего каталога.

Проблема — описывает, когда следует применять шаблон. В данном случае объясня­ется проблема и ее суть. Могут очерчиваться определенные проблемы проектирова­ния, например представление алгоритмов как объектов. Могут характеризоваться классовые или объектные структуры, которые являются признаком негибкой кон­струкции. Иногда проблема будет включать список условий, которые должны быть предварительно соблюдены, чтобы применение шаблона имело смысл.

Решение — описывает элементы, образующие конструкцию, их отношения, функции и взаимодействия. Решение не описывает специфическую, конкретную конструкцию или реализацию, поскольку шаблон — это образец, который может применяться во многих ситуациях. Вместо этого шаблон обеспечивает абстрактное описание пробле­мы проектирования, а также того, как общее расположение элементов (классов и объ­ектов в нашем случае) позволяет решить ее.

Последствия — результаты и компромиссные решения в случае применения шаблона. Несмотря на то что последствия часто не озвучиваются, при описании проектных решений они имеют критическое значение для оценки вариантов этих решений, по­зволяя понять, какие издержки и выгоды повлечет применение шаблона. Когда речь идет о программном обеспечении, то последствия часто связаны с компромиссными решениями в плане места и времени. Они также могут касаться вопросов языка и реализации. Поскольку повторное использование часто свойственно объектно-ориентированному проектированию, последствия применения шаблона включают влияние на гибкость системы, ее расширяемость и переносимость. Перечисление последствий явным образом поможет вам понять и оценить их.

**Парадигма «Модель/Вид/Контроллер» в Smalltalk**

Ради исторической справки нам нужно рассмотреть парадигму «Модель/Вид/ Контроллер» (Model/View/Controller — MVC), представленную в Smalltalk (а так­же используемую в других объектно-ориентированных языках). Парадигма «Мо­дель/Вид/Контроллер» часто применяется для иллюстрирования истоков шабло­нов проектирования. Именно она задействуется при создании интерфейсов пользователя на Smalltalk. Пожалуй, Smalltalk стал первым *популярным* объектно- ориентированным языком.

**SMALLTALK**

Smalltalk — это результат применения ряда отличных идей от Xerox PARC. Он оказался замечательным языком, который обеспечил фундамент для всех последующих объектно- ориентированных языков. Одна из жалоб на C++ заключается в том, что он, в отличие от Smalltalk, не является подлинным объектно-ориентированным языком. Хотя у C++ было больше последователей в годы зарождения объектно-ориентированного подхода, у Smalltalk всегда была базовая группа очень преданных сторонников.

В книге «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)* компоненты парадигмы «Модель/Вид/Контроллер» определяются следующим образом:

«Модель является объектом приложения, вид — это экранное представление, а контрол­лер определяет, как пользовательский интерфейс будет реагировать на вводимые поль­зователем данные».

Проблема с предшествующими парадигмами заключалась в том, что модель, вид и контроллер смешивались в общую массу, в единое целое. Например, один объект включал бы все три этих компонента. В парадигме «Модель/Вид/Контрол­лер» эти три компонента обладают раздельными, индивидуальными интерфейсами. Поэтому, если вы захотите изменить пользовательский интерфейс приложения, вам потребуется лишь модифицировать вид. На рис. 15.1 показано, как выглядит парадигма «Модель/Вид/Контроллер».

Помните, что в объектно-ориентированной разработке многое имеет отношение к интерфейсам и реализации. Нам необходимо максимально обособить интерфейс от реализации. Нам также нужно максимально обособить интерфейсы друг от друга. Например, мы не хотим смешивать множественные интерфейсы, которые никак не связаны между собой (или с имеющимся решением определенной про­блемы). Парадигма «Модель/Вид/Контроллер» стала одной из первых, позволяющих разделять интерфейсы. Она явным образом определяет интерфейсы между специфическими компонентами, которые связаны с широко распространенной и основной проблемой программирования — созданием интерфейсов пользователя и их связыванием с бизнес-логикой и данными, стоящими за ними.

Если вы будете придерживаться парадигмы «Модель/Вид/Контроллер» и ста­нете разделять интерфейс пользователя, бизнес-логику и данные, то ваша система получится гибкой и надежной. Предположим, что интерфейс пользователя находит­ся на клиентском компьютере, бизнес-логика — на сервере приложений, а информа­ция — на сервере данных. Разработка вашего приложения таким путем позволила бы вам изменить внешний вид графического интерфейса пользователя, не затронув бизнес-логику или данные. Аналогичным образом, если вам потребуется изменить свою бизнес-логику и вычислять значение определенного поля по-другому, вы смо­жете модифицировать ее без необходимости изменять графический интерфейс пользователя. И наконец, если вы захотите заменить базы данных и хранить свою информацию по-другому, то у вас будет возможность изменить подход к хранению информации на сервере данных, не затронув при этом ни графический интерфейс пользователя, ни бизнес-логику. В данном случае, конечно же, предполагается, что интерфейсы между этими тремя компонентами не изменятся.

**ПРИМЕР С ИСПОЛЬЗОВАНИЕМ ПАРАДИГМЫ «МОДЕЛЬ/ВИД/КОНТРОЛЛЕР»**

В качестве дополнительного примера, скажем поля списка, представьте себе графический интерфейс пользователя, включающий список телефонных номеров. Видом будет поле списка, моделью — список телефонных номеров, а контроллером — логика, связы­вающая поле списка со списком телефонных номеров.

**НЕДОСТАТКИ ПАРАДИГМЫ «МОДЕЛЬ/ВИД/КОНТРОЛЛЕР»**

Хотя парадигма «Модель/Вид/Контроллер» представляет собой отличный подход при проектировании, он может оказаться сложным в том смысле, что приходится заблаговременно уделять много внимания различным вещам. В целом это проблема, которая имеет место в случае с объектно-ориентированным проектированием, — существует тон­кая грань между грамотным подходом к проектированию и таким, который приводит к созданию громоздкой конструкции. Остается вопрос: насколько сложной вам следует сделать систему, полностью спроектировав ее?

**Типы шаблонов проектирования**

В книге «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)* представлено 23 шаблона, которые разделены на категории, перечисленные ниже. Большинство из этих примеров написано на C++, а некоторые созданы с применением Smalltalk. Время публикации этой книги говорит о том, что тогда уже использовались C++ и Smalltalk. В год издания — 1995-й — интернет- революция как раз достигла своего пика, а язык программирования Java обрел соответствующую популярность. После того как преимущества шаблонов проек­тирования стали очевидны, авторы поспешили написать множество других книг, стремясь насытить новый рынок. Многие из последующих книг были написаны с ориентиром на использование языка Java.

Так или иначе, неважно, каким фактически окажется используемый язык. Кни­га «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)*, по сути, посвящена проектированию, а шаблоны могут быть реализованы с применением любого количества языков. Авторы этой книги раз­делили шаблоны на три категории.

*Порождающие* — создают за вас объекты, чтобы вам не пришлось заниматься непосредственным созданием экземпляров объектов. Это позволит вашей про­грамме проявлять б*о*льшую гибкость при решении о том, какие объекты над­лежит создать в конкретном случае.

*Структурные* — помогают вам составлять из групп объектов более крупные структуры вроде комплексных интерфейсов пользователя или учетных данных.

*Поведенческие* — помогают вам определять взаимодействие между объектами в вашей системе, а также способ управления потоком в комплектной программе.

Далее мы рассмотрим по одному примеру из каждой названной категории, чтобы вы смогли узнать, что именно представляют собой шаблоны проектирования. Исчерпывающий перечень, а также описание отдельных шаблонов проектирования вы найдете в книгах, список которых приведен в конце этой главы.

**Порождающие шаблоны**

К числу порождающих шаблонов относятся следующие:

Abstract factory (Абстрактная фабрика);

Builder (Строитель);

Factory method (Фабричный метод);

Prototype (Прототип);

Singleton (Одиночка).

Как уже отмечалось ранее, в этой главе мы попробуем разобраться, что такое шаблоны проектирования, а не будем пытаться охарактеризовать все без исключе­ния шаблоны из книги «Банды четырех». Поэтому мы рассмотрим по одному ша­блону из каждой категории. Учитывая это, обратимся к примеру порождающего шаблона и взглянем на шаблон Singleton (Одиночка).

**Шаблон проектирования Singleton (Одиночка).** Этот шаблон (рис. 15.2) явля­ется порождающим и используется для регулирования создания объектов, начиная с класса и заканчивая одним объектом. Например, если у вас есть сайт, который включает объект Counter для подсчета количества посещений вашего сайта, то вам, конечно же, не понадобится, чтобы при каждом посещении вашей веб-страницы создавался новый экземпляр объекта Counter. Вы захотите, чтобы экземпляр этого объекта создавался только при первом посещении, а после этого использовался существующий объект для увеличения значения счетчика.

**Рис. 15.2.** Модель на основе Singleton (Одиночка)

Хотя могут быть и другие способы, позволяющие регулировать создание объектов, зачастую лучше всего позволить классу самому позаботиться о решении этой про­блемы. Однако во многих ситуациях использование внешней фабрики оказывается полезным или даже необходимым — в частности, там, где важны такие шаблоны, как Factory (Фабрика), Abstract Factory (Абстрактная фабрика) и Bridge (Мост).

**ПРИМЕЧАНИЕ**

Помните, что одно из важнейших объектно-ориентированных правил заключается в том, что объект должен отвечать за себя. Это означает, что вопросы, касающиеся жизненного цикла класса, должны решаться в рамках этого класса, а не делегироваться языковым конструкциям вроде static и т. д.

На рис. 15.3 показана UML-модель Singleton, взятая непосредственно из книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования» *(Design Patterns: Elements of Reusable Object-Oriented Software)*. Обратите внимание на свойство uniqueInstance, которое является статическим объектом Singleton, а так­же на метод Instance(). Другие свойства и методы располагаются там, указывая на то, что они потребуются для поддержки бизнес-логики соответствующего класса.

**Рис. 15.3.** UML-диаграмма Singleton

Любому другому классу, которому потребуется доступ к экземпляру Singleton, придется прибегнуть к методу Instance(). Создание объекта должно контролировать­ся конструктором точно так же, как при объектно-ориентированном проектировании чего-либо другого. Мы можем сделать так, что клиенту потребуется прибегнуть к методу Instance(), который он затем использует для вызова конструктора.

Приведенный далее Java-код демонстрирует общий вариант Singleton:

public class ClassicSingleton {

private static ClassicSingleton instance = null;

protected ClassicSingleton() {

// Существует только для того, чтобы помешать созданию экземпляров

}

public static ClassicSingleton getInstance() {

if(instance == null) {

instance = new ClassicSingleton();

}

return instance;

}

}

Мы можем создать более конкретный пример с образчиком Counter для веб- страницы, который был использован нами ранее:

public class Counter

{

private int counter;

private static Counter instance = null;

protected Counter()

{

}

public static Counter getInstance() {

if(instance == null) {

instance = new Counter ();

System.out.println("Новый экземпляр создан\n");

}

return instance;

}

public void incrementCounter()

{

counter++;

}

public int getCounter()

{

return(counter);

}

}

Главное, что следует отметить в этом коде, — он регулирует создание объектов. Может быть создан только один объект Counter. Код для этого выглядит так:

public static Counter getInstance() {

if(instance == null) {

instance = new Counter ();

System.out.println("Новый экземпляр создан\n");

}

return instance;

}

Обратите внимание, что, если instance присвоено значение null, это подразумева­ет, что экземпляр объекта еще предстоит создать. В данной ситуации создается новый объект Counter. Если значением instance не является null, то это говорит о том, что был создан экземпляр объекта Counter, а новые объекты создавать не потребуется. В данном случае ссылка на единственный доступный объект возвращается приложению.

**БОЛЕЕ ОДНОЙ ССЫЛКИ**

Вполне допускается несколько ссылок на Singleton. Если вы создадите ссылки в при­ложении, каждая из которых будет вести к Singleton, то вам придется управлять множе­ственными ссылками.

Хотя этот код, несомненно, интересен, важно понимать, как происходит созда­ние экземпляра Singleton и каким образом им управляет приложение. Взгляните на такой код:

public class Singleton

{

public static void main(String[] args)

{

Counter counter1 = Counter.getInstance();

System.out.println("Счетчик : " + counter1.getCounter() );

Counter counter2 = Counter.getInstance();

System.out.println("Счетчик : " + counter2.getCounter() );

}

}

**ДВЕ ССЫЛКИ НА ОДИН COUNTER**

Знайте, что в этом примере к Counter ведут две отдельные ссылки. Таким образом, когда значение Counter изменится, обе эти ссылки будут отражать соответствующее обновление.

Этот код задействует Singleton с Counter. Посмотрите, как создаются объекты:

Counter counter1 = Counter.getInstance();

Здесь конструктор не используется. Создание экземпляра объекта контролиру­ется методом getInstance(). На рис. 15.4 показано, что произойдет при выполнении этого кода. Обратите внимание, что сообщение Новый экземпляр создан выводится только один раз. При создании counter2 он получает копию оригинального объек­та — так же, как counter1.

**Рис. 15.4.** Использование Singleton с Counter

Убедимся в том, что ссылки для counter1 и counter2 одинаковы. Мы можем об­новить программный код следующим образом:

public class Singleton

{

public static void main(String[] args)

{

Counter counter1 = Counter.getInstance();

counter1.incrementCounter();

counter1.incrementCounter();

System.out.println("Счетчик : " + counter1.getCounter() );

Counter counter2 = Counter.getInstance();

counter2.incrementCounter();

System.out.println("Счетчик : " + counter2.getCounter() );

}

}

На рис. 15.5 показан вывод приложения Singleton. Обратите внимание, что в данной ситуации мы увеличиваем значение counter1 вдвое, поэтому значение счетчика будет равно 2. После создания ссылки для counter2 она ведет к тому же объекту, что и ссылка для counter1, поэтому при увеличении значения счетчика оно теперь будет равно 3 (2 + 1).

**Рис. 15.5.** Использование обновленного Singleton с Counter

**ЕЩЕ РАЗ О ПАКЕТНЫХ ФАЙЛАХ**

Как уже демонстрировалось в предыдущих главах, для облегчения тестирования простых, но функциональных образовательных примеров такого рода мне нравится создавать ста­ромодные пакетные файлы, чтобы сделать ввод путей к Java-классам более точным. Вы мо­жете запустить приложение Singleton, поместив приведенный далее код в пакетный файл Singleton.bat, а затем введя Singleton в командной строке:

"C:\Program Files (x86)\Java\jdk1.6.0\_35\bin\java" Singleton

**Структурные шаблоны**

Структурные шаблоны используются для создания более крупных конструкций из групп объектов. Приведенные далее семь шаблонов проектирования относятся к категории структурных:

Adapter (Адаптер);

Bridge (Мост);

Composite (Компоновщик);

Decorator (Декоратор);

Facade (Фасад);

Flyweight (Приспособленец);

Proxy (Заместитель).

В качестве примера из категории структурных шаблонов рассмотрим Adapter (Адаптер). Это также один из важнейших шаблонов проектирования. Он пред­ставляет собой хороший пример того, как разделяются реализация и интерфейс.

**Шаблон проектирования Adapter (Адаптер).** Данный шаблон — это инстру­мент, с помощью которого вы можете создать отличающийся интерфейс для уже существующего класса. Этот шаблон по сути предусматривает создание классовой обертки. Другими словами, вы создадите новый класс, который объединит (обернет собой) функциональность существующего класса с новым и — в идеале — лучшим интерфейсом. Простой пример обертки — Java-класс Integer, который обертывает собой одно целочисленное значение. Вы можете спросить, зачем так делать. Пом­ните, что в объектно-ориентированной системе все сущности являются объектами. В Java примитивы, например целые, дробные числа и т. д., не являются объектами. Если вам понадобится выполнить над ними какое-либо действие, скажем преоб­разование, то придется обращаться с ними как с объектами. Вы создадите объект- обертку и обернете им соответствующий примитив. Таким образом, вы сможете взять примитив вроде следующего:

int myInt = 10;

и обернуть его объектом Integer:

Integer myIntWrapper = new Integer (myInt);

Теперь можно выполнить преобразование, чтобы мы смогли обращаться с ним как со строкой:

String myString = myIntWrapper.toString();

Эта обертка дает нам возможность обращаться с исходным целочисленным значе­нием как с объектом, благодаря чему обеспечиваются все преимущества объектов.

В том, что касается шаблона Adapter (Адаптер) как такового, рассмотрим при­мер интерфейса MailTool. Предположим, что вы приобрели код, который обеспечи­вает всю функциональность, требуемую вам для реализации почтового клиента. Этот инструмент предоставляет все, что вам нужно от почтового клиента, с тем исключением, что вы хотели бы немного изменить интерфейс. Фактически все, что вы хотите сделать, — это изменить API-интерфейс для извлечения своей электрон­ной почты.

Приведенный далее класс — очень простой образец почтового клиента для это­го примера:

package MailTool;

public class MailTool {

public MailTool () {

}

public int retrieveMail() {

System.out.println ("Вам пришла почта");

return 0;

}

}

При вызове метода retrieveMail() ваша почта откликается с использованием очень оригинального приветствия Вам пришла почта. Теперь предположим, что вы захотели изменить интерфейс всех клиентов в вашей компании с retrieveMail() на getMail(). Вы можете создать интерфейс, чтобы претворить это в жизнь:

package MailTool;

interface MailInterface {

int getMail();

}

Теперь вы можете создать свой MailTool, который будет оберткой для ориги­нального MailTool, и предусмотреть собственный интерфейс:

package MailTool;

class MyMailTool implements MailInterface {

private MailTool yourMailTool;

public MyMailTool () {

yourMailTool= new MailTool();

setYourMailTool(yourMailTool);

}

public int getMail() {

return getYourMailTool().retrieveMail();

}

public MailTool getYourMailTool() {

return yourMailTool ;

}

public void setYourMailTool(MailTool newYourMailTool) {

yourMailTool = newYourMailTool;

}

}

Внутри этого класса вы создаете экземпляр оригинального MailTool, который хотите модифицировать. Данный класс реализует MailInterface, из-за чего вам приходится реализовать метод getMail(). Внутри этого метода вы буквально вы­зываете метод retrieveMail() оригинального MailTool.

Чтобы использовать свой новый класс, вам потребуется создать экземпляр собственного нового MailTool и вызвать метод getMail():

package MailTool;

public class Adapter

{

public static void main(String[] args)

{

MyMailTool myMailTool = new MyMailTool();

myMailTool.getMail();

}

}

При вызове метода getMail() вы будете использовать этот новый интерфейс для вызова метода retrieveMail() оригинального MailTool. Это очень простой пример. Вместе с тем, создав данную обертку, вы расширите интерфейс и добавите соб­ственную функциональность в оригинальный класс.

Концепция шаблона Adapter (Адаптер) довольно проста, однако вы сможете создавать новые и мощные интерфейсы, используя этот шаблон.

**Поведенческие шаблоны**

К числу поведенческих шаблонов относятся следующие:

Chain of response (Цепочка ответственности);

Command (Команда);

Interpreter (Интерпретатор);

Iterator (Итератор);

Mediator (Посредник);

Memento (Хранитель);

Observer (Наблюдатель);

State (Состояние);

Strategy (Стратегия);

Template method (Шаблонный метод);

Visitor (Посетитель).

В качестве примера из категории поведенческих шаблонов взглянем на шаблон Iterator (Итератор). Это один из наиболее широко используемых шаблонов, кото­рый в то же время реализован несколькими языками программирования.

**Шаблон проектирования Iterator (Итератор).** Итераторы представляют собой стандартный механизм обхода коллекций, например векторов. При этом должна обеспечиваться функциональность, благодаря которой будет возможен доступ к каждому элементу коллекции поодиночке. Шаблон Iterator (Итератор) обеспе­чивает скрытие информации, сохраняя защищенной внутреннюю структуру коллекции. Этот шаблон также предполагает, что можно создать несколько итера­торов и они не станут мешать друг другу. Java предусматривает собственную реа­лизацию шаблона Iterator (Итератор). Приведенный далее код создает вектор, после чего вставляет в него несколько строк:

package Iterator;

import java.util.\*;

public class Iterator {

public static void main(String args[]) {

// Создать экземпляр ArrayList

ArrayList<String> names = new ArrayList();

// Добавить значения в ArrayList

names.add(new String("Джо"));

names.add(new String("Мэри"));

names.add(new String("Боб"));

names.add(new String("Сью"));

// Теперь осуществить итерацию по именам

System.out.println("Имена:");

iterate(names );

}

private static void iterate(ArrayList<String> arl) {

for(String listItem : arl) {

System.out.println(listItem.toString());

}

}

}

Затем мы создаем перечисление, чтобы можно было осуществить по нему ите­рацию. Для обеспечения функциональности, связанной с выполнением итераций, предусмотрен метод iterate(). В нем мы используем Java-метод перечисления hasMoreElements(), который позволяет выполнить обход вектора и показать все имена.

**Антишаблоны**

В то время как шаблоны проектирования развиваются из успешных практик, *анти­шаблоны* можно представлять себе как наборы неудавшихся методик. Существуют убедительные документальные доказательства того, что большинство программных проектов в конечном счете признаются неудачными. Фактически, как отмечается в статье «Создавая хаос» *(Creating Chaos)*, написанной Джонни Джонсоном (Johnny Johnson), по меньшей мере одна треть всех проектов полностью отменяется. По-видимому, многие из этих неудач обусловлены плохими проектными реше­ниями.

Термин «антишаблон» стал следствием того факта, что шаблоны проектирова­ния создаются с целью заранее решать проблемы определенного типа. Антишаблон, с другой стороны, является реакцией на проблему и происходит из плохого опыта. Одним словом, в то время как шаблоны проектирования базируются на надежных методиках проектирования, антишаблоны можно рассматривать как методики, которых следует избегать.

В выпуске журнала *C++ Report* за ноябрь 1995 года Эндрю Кениг (Andrew Koenig) охарактеризовал два варианта антишаблонов:

описывающие плохое решение проблемы, которое приводит к возникновению скверной ситуации;

объясняющие, как выйти из скверной ситуации, а затем выработать хорошее решение.

Многие люди считают, что антишаблоны полезнее шаблонов проектирования, так как призваны решать проблемы, которые уже возникли. В данном случае все сводится к концепции анализа первопричин. Можно провести исследование с ис­пользованием данных, которое способно показать, почему первоначальная кон­струкция или, возможно, текущий шаблон проектирования не позволили добить­ся требуемой цели. Таким образом, антишаблоны обладают преимуществом в виде ретроспективного взгляда.

Например, в своей статье «Повторное использование шаблонов и антишабло­нов» *(Reuse Patterns and Antipatterns)* Скотт Амблер приводит шаблон Robust Artifact (Надежный артефакт) и определяет его следующим образом:

«Инструмент, который хорошо документирован, создан отвечать общим требованиям, а не специфичным для определенного проекта, как следует протестирован и предусматри­вает несколько примеров того, как с ним работать. Намного более вероятно, что повтор­но будут использоваться инструменты, обладающие такими качествами, а не те, которые их лишены. Robust Artifact (Надежный артефакт) — это инструмент, легкий для пони­мания и простой в эксплуатации».

Однако, несомненно, бывает много ситуаций, когда то или иное решение объ­является пригодным для повторного использования, однако затем никто так ни­когда и не прибегает к нему снова. Поэтому, чтобы пояснить антишаблон, Скотт Амблер пишет:

«Сторонний разработчик должен проанализировать Reuseless Artifact (Артефакт, никем не используемый повторно), чтобы выяснить, заинтересует ли он кого-нибудь. Если да, то его надлежит переработать так, чтобы он превратился в Robust Artifact (Надежный артефакт)».

Таким образом, антишаблоны приводят к пересмотру существующих конструк­ций и их непрерывному рефакторингу до тех пор, пока не будет найдено приемле­мое решение.