Архитектура ПО

Пилотная архитектура (architectural spike) – архитектурное решение, которое частично реализуется для проверки его применимости. Если пилотная архитектура доказывается свою эффективность, то пилотную архитектуру принимают в качестве **базовой архитектуры**.

Модуль (unit of implementation) – часть реализации программного обеспечения. Обычно модуль состоит из файлов с исходным кодом, но также может включать xml-файлы, текстовые конфиграционные файлы и другие элементы реализации.

Архитектура ПО

Компонент (unit of runtime and deployment) тоже является частью программного обеспечения. В отличии от модуля компонент *является единицей развертывания и исполнения*. Компоненты распространяются в виде файлов, пригодных для исполнения. Компоненты в дальнейшем могут организовываться в более сложную программную систему сторонними пользователями. Не существует ограничений на способ распространения компонентов.

Сложные программные компоненты состоят из компонентных частей (component part). Каждая компонентная часть представлена одним или несколькими файлами (чаще всего динамическими подключаемыми библиотеками или исполняемым файлом).

Архитектура ПО

При создании программного обеспечения один компонент может реализован с использованием одного или нескольких модулей.

Фреймворк (framework) – готовая инфраструктура для создания программных приложений определенного типа. Фреймворк обычно включает готовые архитектурные решения и методы разработки, которых следует придерживаться разработчику, программные утилиты для автоматического создания программного кода, готовые компоненты, реализующие базовую архитектуру.

Архитектура ПО

Слабая связность (loose coupling) – принцип, при котором компоненты программного обеспечения используют минимальное количество связей с друг другом.

Сквозная функциональность (crosscutting concern) – логика программного приложения, обращение к которой многократно в исходном коде. К сквозной функциональности относят механизмы протоколирования, авторизации и аутентификации, кэширование, связь, обработка исключений, проверка корректности данных, введенных пользователем.

Архитектура ПО

Основные принципы качественно спроектированной программной архитектуры:

- *Разделение функций*. По возможности программное обеспечение разделяется на отдельные составные части (модули и компоненты) с минимальным перекрытием функциональности. Важным фактором является предельное уменьшение количества точек соприкосновения, что обеспечит высокую связность (high cohesion) и слабую связанность (low coupling) (т.е. система включает слабо зависимые части связанные только бизнес-логикой). Неверное разграничение функциональности может привести к высокой связанности и сложностям взаимодействия, даже несмотря на слабое перекрытие функциональности отдельных компонентов.

Архитектура ПО

- *Принцип основной ответственности*. Каждый отдельно взятый компонент или модуль должен отвечать за реализацию одного свойства или функциональной возможности (группы связанных функций).
- *Принцип минимального знания.* Составные элементы программного обеспечения (объекты, компоненты и др.) не осведомлены о внутренней реализации других элементов.
- *Отсутствие повторений*. Функциональность должна быть реализована только в одном модуле или компоненте. Дублирование реализации не допускается.

Архитектура ПО

- Проектирование наперед осуществляется только при необходимости. Если требования к программному обеспечению могут быть изменены или не определены однозначно, то следует проектировать только то, что необходимо в ближайшей перспективе. Обычно масштабное предварительное проектирование и тестирование архитектурных решений осуществляется в тех случаях, если неверная архитектура несет значительные издержки.

Проектирование в процессе разработки ПО

Проектированию программного продукта предшествует процесс выявления требований к программному продукту и их анализа.

После завершения фазы проектирования происходит реализация проекта – кодирование и тестирование.

Проектирование является одной из этапов создания программного приложения или системы программных приложений, которая присутствует в большинстве методологий разработки программного обеспечения. Длительность и значимость процесса проектирования варьируется в зависимости от методологий.

Проектирование в процессе разработки ПО

В легковесных процессах разработки ПО степень формализации стадии проектирования низка, как и количество создаваемой документации. Некоторые проекты предполагают больший объем работ по проектированию именно на стадии конструирования; другие проекты явно выделяют проектную деятельность в форме фазы проектирования.

Независимо от задачи проектирования, в этом процессе возможно выделить следующие составляющие:

- изучение проблемы;
- предварительное определение нескольких или как минимум одного решения;

Проектирование в процессе разработки ПО

- выявление и описание составных элементов каждого решения.

  В процессе проектирования проект разрабатывается постепенно: от неформального к более формальному.

  Обычно исходными данными являются варианты использования и сценарии поведения пользователя, функциональные требования, нефункциональные требования (включая параметры качества, такие как производительность, безопасность, надежность и другие), технологические требования, целевая среда развертывания и другие ограничения.

Проектирование и качество ПО

Высокие характеристики *сопровождаемости* являются одной из целей проектирования.

С увеличением сложности программных систем увеличивается продолжительность жизненного цикла. Часто участники коллектива, создавшего программный продукт, не участвуют в процессе сопровождения или не могут предоставить и вспомнить некоторые делали реализации. Документация может быть устаревшей или утерянной. Сопровождаемость определяется тем как легко и быстро можно обновить программное обеспечение.

Гибкость и способность к расширению также являются желанными характеристиками проектируемой системы.

Качество исходного кода и парадигмы программирования

Структурное программирование

Первой, получившей всеобщее признание (но не первой из придуманных), была парадигма структурного программирования, предложенная Эдсгером Вибе Дейкстрой в 1968 году.

Дейкстра показал, что безудержное использование безусловных переходов (инструкций goto) вредно для структуры программы. Он предложил заменить безусловные переходы более понятными конструкциями условных переходов и циклов.

Как итог, можно сказать, что: структурное программирование накладывает ограничение на прямую передачу управления.

Структурное программирование

В настоящее время все программисты используют парадигму структурного программирования, хотя и не всегда осознанно. Просто современные языки не дают возможности неограниченной прямой передачи управления.

Некоторые отмечают сходство инструкции break с меткой и исключений в Java с инструкцией goto. В действительности эти структуры не являются средствами неограниченной прямой передачи управления, имевшимися в старых языках, таких как Fortran или COBOL. Кроме того, даже языки, сохранившие ключевое слово goto, часто ограничивают возможность переходов границами текущей функции.

Структурное программирование

Структурное программирование дает возможность рекурсивного разложения модулей на доказуемые единицы, что, в свою очередь, означает возможность функциональной декомпозиции.

То есть решение большой задачи можно разложить на ряд функций верхнего уровня. Каждую из этих функций в свою очередь можно разложить на ряд функций более низкого уровня, и так до бесконечности.

Кроме того, каждую из таких функций можно представить с применением ограниченного набора управляющих структур, предлагаемых парадигмой структурного программирования.

Структурное программирование

Опираясь на этот фундамент, в конце 1970-х годов и на протяжении 1980-х годов приобрели популярность такие дисциплины, как структурный анализ и структурное проектирование.

Парадигма структурного программирования заставляет нас рекурсивно разбивать программы на множество мелких и доказуемых функций. В результате мы получаем возможность использовать тесты, чтобы попытаться доказать их неправильность. Если такие тесты терпят неудачу, тогда мы считаем функции достаточно правильными.

Возможность создавать программные единицы, неправильность которых можно доказать, является главной ценностью структурного программирования.

Объектно-ориентированное программирование

Чтобы объяснить природу ООП часто прибегают к трем волшебным словам: инкапсуляция, наследование и полиморфизм. Они подразумевают, что ООП является комплексом из этих трех понятий или, по крайней мере, что объектно-ориентированный язык должен их поддерживать.

Инкапсуляция упоминается как часть определения ООП потому, что языки ООП поддерживают простой и эффективный способ инкапсуляции данных и функций. Как результат, есть возможность очертить круг связанных данных и функций. За пределами круга эти данные невидимы и доступны только некоторые функции.

Объектно-ориентированное программирование

Воплощение этого понятия можно наблюдать в виде приватных членов данных и общедоступных членов-функций класса.

Эта идея определенно не уникальная для ОО. Например, в языке С имеется превосходная поддержка инкапсуляции. Рассмотрим простую программу на С:

```
/* point.h */
#ifndef POINT_H_
#define POINT_H_
struct Point;
struct Point* makePoint(double x, double y);
double distance (struct Point *p1, struct Point *p2);
#endif /* POINT_H_ */
```

Объектно-ориентированное программирование

```c
/* point.c */
#include "point.h"
#include <stdlib.h>
#include <math.h>

struct Point {
   double x, y;
};

struct Point* makePoint(double x, double y) {
   struct Point* p = malloc(sizeof(struct Point));
   p->x = x;
   p->y = y;
   return p;
}
```

Объектно-ориентированное программирование

```
double distance(struct Point* p1, struct Point* p2) {
   double dx = p1->x - p2->x;
   double dy = p1->y - p2->y;
   return sqrt(dx * dx + dy * dy);
}
```

Пользователи `point.h` не имеют доступа к членам структуры `Point`. Они могут вызывать функции `makePoint()` и `distance()`, но не имеют никакого представления о реализации структуры `Point` и функций для работы с ней.

Это отличный пример поддержки инкапсуляции не в объектно-ориентированном языке.

Объектно-ориентированное программирование

Но затем пришел объектно-ориентированный C++ и превосходная инкапсуляция в C оказалась разрушенной.

Чтобы иметь возможность определить размер экземпляра каждого класса компилятор C++ требует определять переменные-члены класса в заголовочном файле. Теперь пользователи заголовочного файла `point.h` знают о переменных-членах `x` и `y`! Компилятор не позволит обратиться к ним непосредственно, но клиент все равно знает об их существовании. Введением в язык ключевых слов `public, private` и `protected` инкапсуляция была частично восстановлена.

Объектно-ориентированное программирование

Языки Java и C# полностью отменили деление на заголовок/реализацию, ослабив инкапсуляцию еще больше. В этих языках невозможно разделить объявление и определение класса.

По описанным причинам трудно согласиться, что ООП зависит от строгой инкапсуляции. В действительности многие языки ООП практически не имеют принудительной инкапсуляции. ООП безусловно полагается на поведение программистов — что они не станут использовать обходные приемы для работы с инкапсулированными данными. То есть языки, заявляющие о поддержке ООП, фактически ослабили превосходную инкапсуляцию, некогда существующую в C.

Объектно-ориентированное программирование

Языки ООП не улучшили инкапсуляцию, зато они дали нам наследование.

Точнее — ее разновидность. По сути, наследование — это всего лишь повторное объявление группы переменных и функций в ограниченной области видимости. Нечто похожее программисты на С проделывали вручную задолго до появления языков ООП, хотя это и был трюк, не настолько удобный, как настоящее наследование.

В итоге - языки ООП действительно сделали маскировку структур данных более удобной, хотя это и не совсем новая особенность.

Объектно-ориентированное программирование

Но у нас есть еще одно понятие - полиморфизм.

Была ли возможность реализовать полиморфное поведение до появления языков ООП – ответ да.

Суть полиморфизма заключается в применении указателей на функции. Программисты использовали указатели на функции для достижения полиморфного поведения еще со времен появления архитектуры фон Неймана в конце 1940-х годов.

Но, языки ООП сделали полиморфизм намного надежнее и удобнее. Проблема явного использования указателей на функции для создания полиморфного поведения в том, что указатели на функции по своей природе опасны.

Объектно-ориентированное программирование

Такое их применение оговаривается множеством соглашений. Вы должны помнить об этих соглашениях и инициализировать указатели. Вы должны помнить об этих соглашениях и вызывать функции посредством указателей.

Если какой-то программист забудет о соглашениях, возникшую в результате ошибку будет очень трудно отыскать и устранить.

Поддержка полиморфизма на уровне языка делает его использование тривиально простым. Это обстоятельство открывает новые возможности, о которых программисты на C могли только мечтать.

Объектно-ориентированное программирование

ООП дает, посредством поддержки полиморфизма, абсолютный контроль над всеми зависимостями в исходном коде.

Это позволяет создавать архитектуру со сменными модулями (плагинами), в которой модули верхнего уровня не зависят от модулей нижнего уровня. Низкоуровневые детали не выходят за рамки модулей плагинов, которые можно развертывать и разрабатывать независимо от модулей верхнего уровня.

Функциональное программирование

Многие идеи функционального программирования появились задолго до появления самого программирования.

Эта парадигма в значительной мере основана на λ-исчислении, изобретенном Алонзо Чёрчем в 1930-х годах.

Суть функционального программирования проще объяснить на примерах. С этой целью исследуем решение простой задачи: вывод квадратов первых 25 целых чисел (то есть от 0 до 24).

В языках, подобных Java, эту задачу можно решить так:

Функциональное программирование

```
public class Squint {
  public static void main(String args[]) {
    for (int i=0; i<25; i++)
      System.out.println(i*i);
  }
}
```

В Clojure, функциональном языке и производном от языка Lisp, аналогичную программу можно записать так:

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

Этот код может показаться немного странным, если вы не знакомы с Lisp.

Функциональное программирование

Поэтому, немного переоформим его и добавим несколько комментариев:

```
(println ;_____________________ Вывести
(take 25 ;_____________________ первые 25
(map (fn [x] (* x x)) ;__ квадратов
(range)))) ;____________ целых чисел
```

Обращаем внимание, что `println, take, map` и `range` — это функции. В языке Lisp вызов функции производится помещением ее имени в круглые скобки. Например, (`range`) — это вызов функции `range`.

Функциональное программирование

Выражение `(fn [x] (* x x))` — это анонимная функция, которая, в свою очередь, вызывает функцию умножения и передает ей две копии входного аргумента. Иными словами, она вычисляет квадрат заданного числа.

Взглянем на эту программу еще раз, начав с самого внутреннего вызова функции:

- функция `range` возвращает бесконечный список целых чисел, начиная с `0`;
- этот список передается функции `map`, которая вызывает анонимную функцию для вычисления квадрата каждого элемента и производит бесконечный список квадратов;

Функциональное программирование

- список квадратов передается функции `take`, которая возвращает новый список, содержащий только первые 25 элементов;
- функция `println` выводит этот самый список квадратов первых 25 целых чисел.

Упоминание бесконечных списков в действительности работает следующим образом: программа создаст только первые 25 элементов этих бесконечных списков, т.к. новые элементы бесконечных списков не создаются, пока программа не обратится к ним.

Функциональное программирование

Важнейшее отличие между программами на Clojure и Java в том, что в программе на Java используется изменяемая переменная — переменная, состояние которой изменяется в ходе выполнения программы. Это переменная `i` — переменная цикла. В программе на Clojure, напротив, нет изменяемых переменных. В ней присутствуют инициализируемые переменные, такие как x, но они никогда не изменяются.

В результате мы пришли к утверждению: *переменные в функциональных языках не изменяются*.

Функциональное программирование

Почему этот аспект важен с архитектурной точки зрения?

- Все состояния гонки (race condition), взаимоблокировки (deadlocks) и проблемы параллельного обновления обусловлены изменяемостью переменных. Если в программе нет изменяемых переменных, она никогда не окажется в состоянии гонки и никогда не столкнется с проблемами одновременного изменения.

В отсутствие изменяемых блокировок программа не может попасть в состояние взаимоблокировки.

Функциональное программирование

Т.е. все проблемы, характерные для приложений с конкурирующими вычислениями, — с которыми нам приходится сталкиваться, когда требуется организовать многопоточное выполнение и задействовать вычислительную мощность нескольких процессоров, исчезают сами собой в отсутствие изменяемых переменных.

В каких случаях достижима неизменяемость на практике - если у вас есть неограниченный объем памяти и процессор с неограниченной скоростью вычислений. В отсутствие этих бесконечных ресурсов ответ выглядит не так однозначно, неизменяемость достижима, но при определенных компромиссах.

Функциональное программирование

Один из самых общих компромиссов, на которые приходится идти ради неизменяемости, — деление приложения или служб внутри приложения на изменяемые и неизменяемые компоненты.

Неизменяемые компоненты решают свои задачи исключительно функциональным способом, без использования изменяемых переменных. Они взаимодействуют с другими компонентами, не являющимися чисто функциональными и допускающими изменение состояний переменных.

Изменяемое состояние этих других компонентов открыто всем проблемам многопоточного выполнения, поэтому для защиты изменяемых переменных от конкурирующих обновлений

Функциональное программирование

и состояния гонки часто используется некоторая разновидность *транзакционной памяти*.

Транзакционная память интерпретирует переменные в оперативной памяти подобно тому, как база данных интерпретирует записи на диске. Она защищает переменные, применяя механизм транзакций с повторениями (рис. 2.1)

Суть в том, что правильно организованные приложения должны делиться на компоненты, имеющие и не имеющие изменяемых переменных. Такое деление обеспечивается использованием подходящих дисциплин для защиты изменяемых переменных.

Функциональное программирование

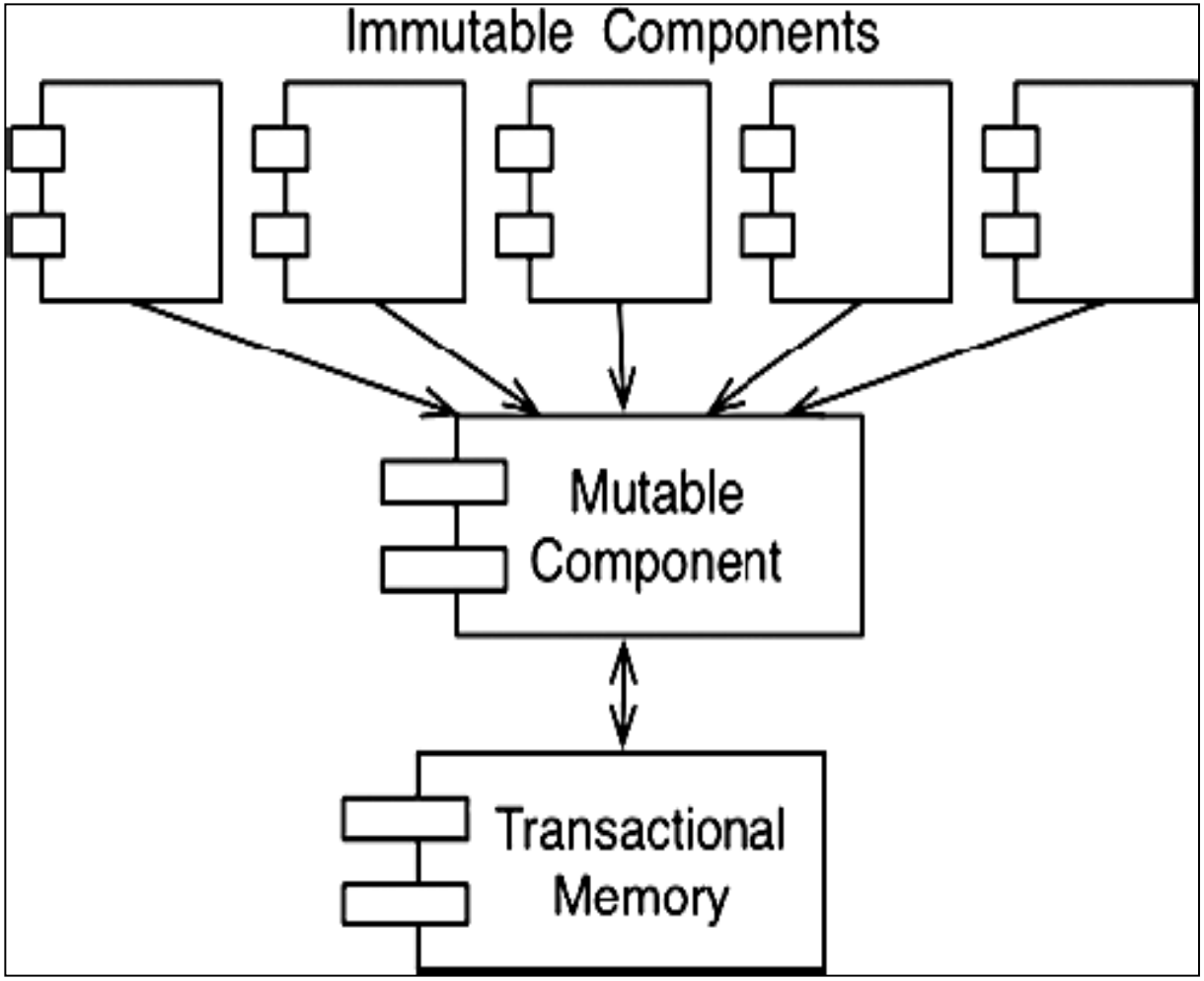

Рис. 2.1 Изменяемое состояние и транзакционная память

Функциональное программирование

Со стороны разработчиков было бы разумно как можно больше кода поместить в неизменяемые компоненты и как можно меньше — в компоненты, допускающие возможность изменения.

Итак:

- Структурное программирование накладывает ограничение на прямую передачу управления.
- Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления.
- Функциональное программирование накладывает ограничение на присваивание.

Функциональное программирование

Каждая из этих парадигм что-то отнимает у нас. Каждая ограничивает подходы к написанию исходного кода. Ни одна не добавляет новых возможностей.

Фактически последние 50 лет происходило обучение тому, как не надо делать, таким образом, разработка программного обеспечения не является быстро развивающейся индустрией. Правила остаются теми же, какими они были в 1946 году, когда Алан Тьюринг написал первый код, который мог выполнить электронный компьютер. Инструменты изменились, аппаратура изменилась, но суть программного обеспечения осталась прежней.

Принципы дизайна SOLID

SOLID - 5 основных принципов объектно-ориентированного программирования и проектирования.

Принципы SOLID определяют, как объединять функции и структуры данных в классы и как эти классы должны сочетаться друг с другом. Использование слова «класс» не означает, что эти принципы применимы только к объектно-ориентированному программному коду. В данном случае «класс» означает лишь инструмент объединения функций и данных в группы. Любая программная система имеет такие объединения, как бы они ни назывались, «класс» или как-то еще.

Принципы дизайна SOLID

Принципы SOLID применяются к этим объединениям. Цель принципов — создать программные структуры среднего уровня, которые:

- терпимы к изменениям;
- просты и понятны;
- образуют основу для компонентов, которые могут использоваться во многих программных системах.

Термин «средний уровень» отражает тот факт, что эти принципы применяются программистами на уровне модулей и помогают определять программные структуры, используемые в модулях и компонентах.

Принципы дизайна SOLID

SOLID получается как первые буквы следующих выражений:

- SRP - Single Responsibility Principle — принцип единственной ответственности.
- OCP - Open-Closed Principle — принцип открытости / закрытости.
- LSP - Liskov Substitution Principle — принцип подстановки Барбары Лисков.
- ISP - Interface Segregation Principle — принцип разделения интерфейсов.
- DIP - Dependency Inversion Principle — принцип инверсии зависимости.

Принципы дизайна SOLID

Принцип единственной ответственности (SRP) можно описать так:

Для каждого класса должно быть определено единственное назначение. Все ресурсы, необходимые для его осуществления, должны быть инкапсулированы в этот класс и подчинены только этой задаче.

Чтобы лучше понять рассмотрим признаки нарушения этого принципа.

Признак 1: непреднамеренное дублирование

Класс `Employee` из приложения платежной ведомости - он имеет три метода: `calculatePay()`, `reportHours()` и `save()` (рис. 2.2).

Принципы дизайна SOLID

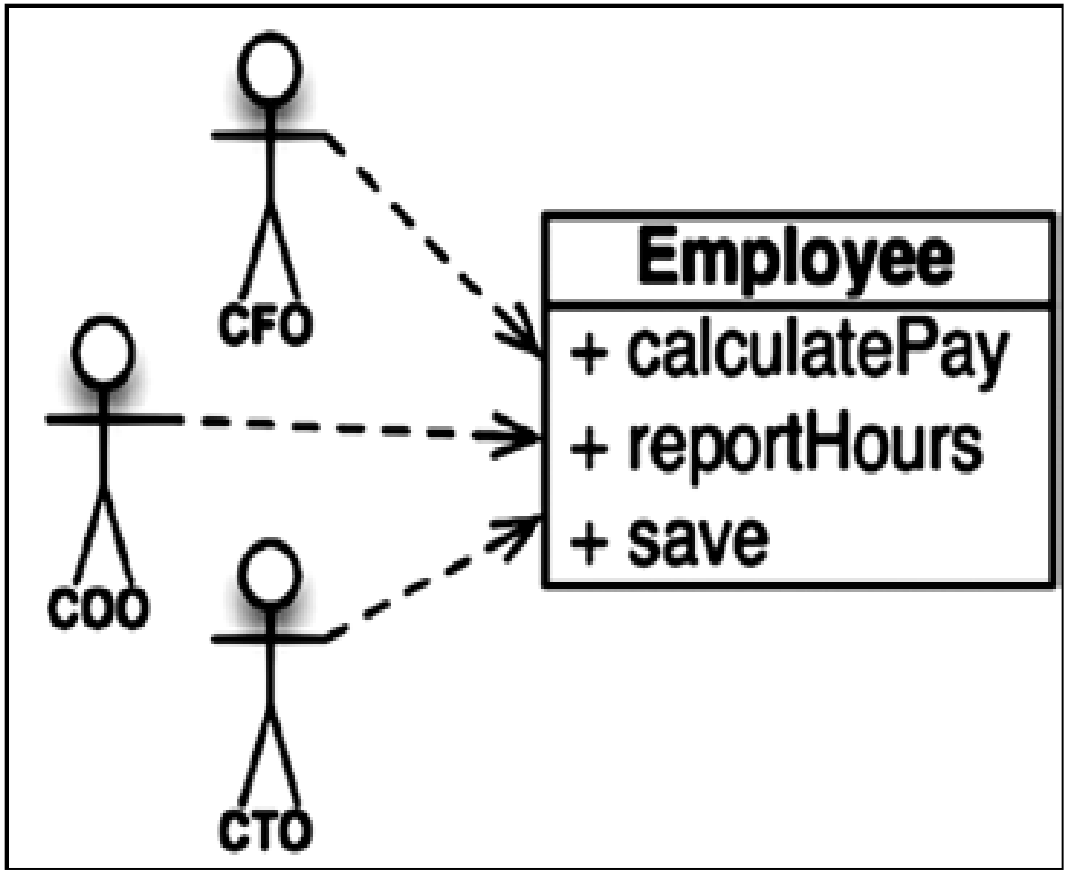

Рис. 2.2 - Класс Employee

Принципы дизайна SOLID

Этот класс нарушает принцип единственной ответственности, потому что три его метода отвечают за три разных актора:

- Реализация метода calculatePay() определяется бухгалтерией.
- Реализация метода reportHours() определяется и используется отделом по работе с персоналом.
- Реализация метода save() определяется администраторами баз данных.

Поместив исходный код этих трех методов в общий класс Employee, разработчики объединили перечисленных акторов.

Принципы дизайна SOLID

В результате такого объединения действия сотрудников бухгалтерии могут затронуть что-то, что требуется сотрудникам отдела по работе с персоналом.

Принцип единственной ответственности требует разделять код, от которого зависят разные акторы.

Признак 2: слияния.

Слияния — обычное дело для исходных файлов с большим количеством разных методов. Эта ситуация особенно вероятна, если эти методы отвечают за разных акторов.

Принципы дизайна SOLID

Например, представим, что коллектив администраторов баз данных решил внести простое исправление в схему таблицы Employee. Представим также, что сотрудники отдела по работе с персоналом пожелали немного изменить формат отчета.

Два разных разработчика, возможно, из двух разных команд, извлекли класс Employee из репозитория и внесли изменения. К сожалению, их изменения оказались несовместимыми. В результате потребовалось выполнить слияние.

В нашем примере процедура слияния поставила под удар администраторов баз данных и отдел по работе с персоналом.

Принципы дизайна SOLID

Решения

Существует много решений этой проблемы. Но каждое связано с перемещением функций в разные классы.

Наиболее очевидным, пожалуй, является решение, связанное с отделением данных от функций. Три класса используют общие данные EmployeeData — простую структуру без методов. Каждый класс включает только исходный код для конкретной функции. Эти три класса никак не зависят друг от друга. То есть любое непреднамеренное дублирование исключено. Недостаток такого решения — разработчик теперь должен создавать экземпляры трех классов и следить за ними.

Принципы дизайна SOLID

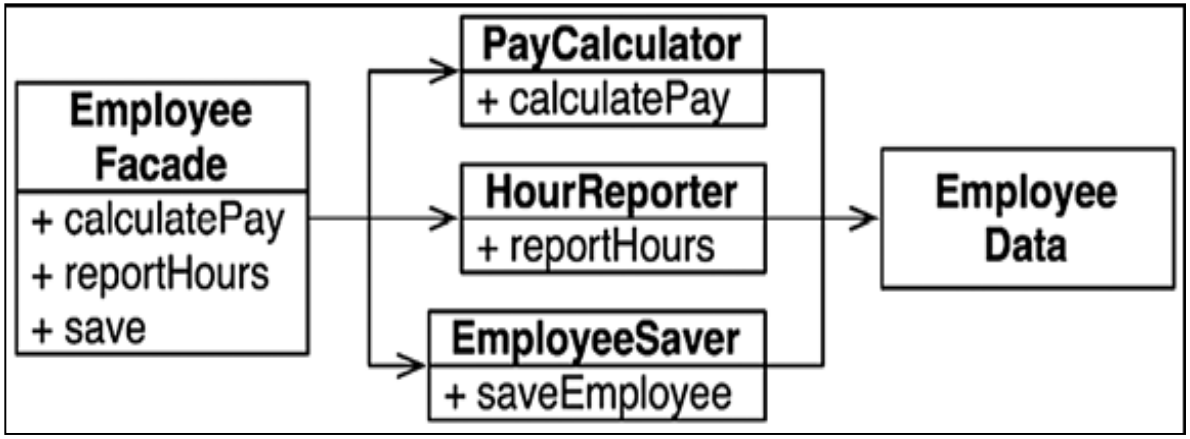

Рис. 2.3 - Шаблон «Фасад»

Эта проблема часто решается применением шаблона проектирования «Фасад» (Facade), как показано на рис. 2.3. Класс EmployeeFacade содержит очень немного кода и отвечает за создание экземпляров трех классов и делегирование вызовов методов.

Принципы дизайна SOLID

Принцип открытости/закрытости (OCP) был сформулирован Бертраном Мейером в 1988 году. Он гласит:

Программные сущности должны быть открыты для расширения и закрыты для изменения.

Иными словами, должна иметься возможность расширять поведение программных сущностей без их изменения.

Рассмотрим пример.

Представьте, что у нас есть финансовая сводка. Содержимое страницы прокручивается, и отрицательные значения выводятся красным цветом.

Принципы дизайна SOLID

Теперь допустим, что заинтересованные лица попросили нас представить ту же информацию в виде распечатанного отчета.

Очевидно, что для этого придется написать новый код. Но как много старого кода придется изменить?

В программном обеспечении с хорошо проработанной архитектурой таких изменений должно быть очень немного. В идеале их вообще не должно быть.

Применяя принцип единственной ответственности, можно прийти к потоку данных, изображенному на рис. 2.4.

Принципы дизайна SOLID

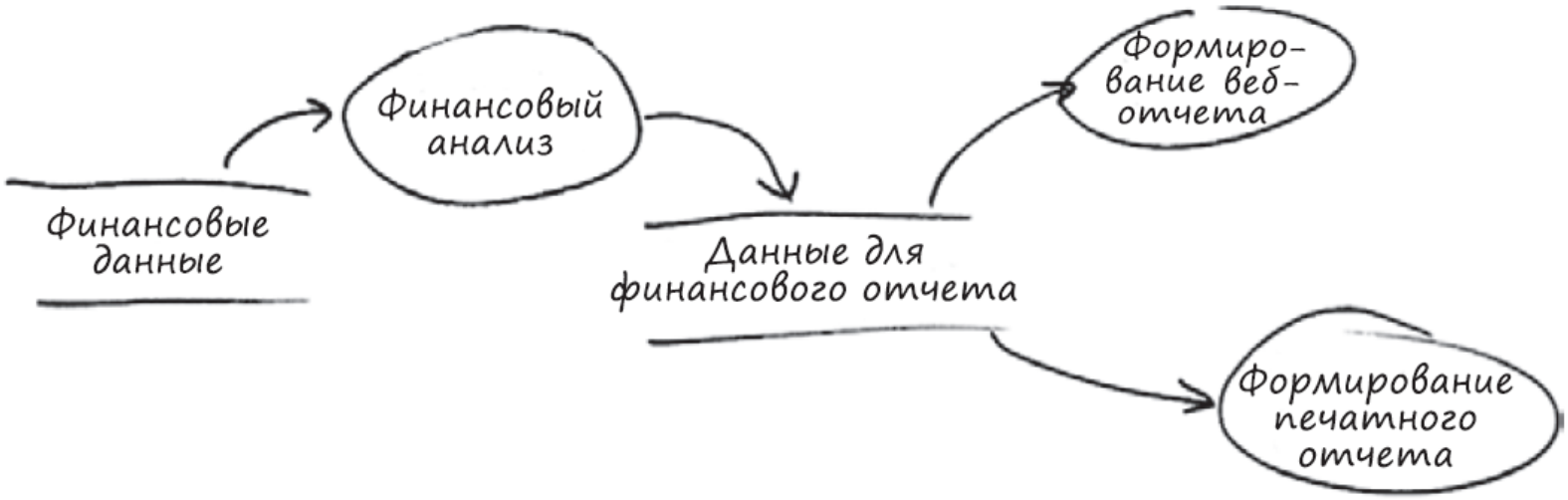

Рис 2.4 - Результат применения принципа единственной ответственности.

В создание отчета вовлечены две отдельные ответственности: вычисление данных для отчета и представление этих данных в форме веб-отчета или распечатанного отчета.

Принципы дизайна SOLID

Принцип открытости/закрытости — одна из движущих сил в архитектуре систем. Его цель — сделать систему легко расширяемой и обезопасить ее от влияния изменений. Эта цель достигается делением системы на компоненты и упорядочением их зависимостей в иерархию, защищающую компоненты уровнем выше от изменений в компонентах уровнем ниже.

LSP - Чтобы понять идею, известную как **принцип подстановки Барбары *Лисков***, рассмотрим несколько примеров.

Принципы дизайна SOLID

Представьте, что у нас есть класс с именем License, как показано на рис. 2.5. Этот класс имеет метод с именем calcFee(), который вызывается приложением Billing. Существует два «подтипа» класса License: PersonalLicense и BusinessLicense. Они реализуют разные алгоритмы расчета лицензионных отчислений.

Этот дизайн соответствует принципу подстановки Барбары Лисков, потому что поведение приложения Billing не зависит от использования того или иного подтипа.

Оба подтипа могут служить заменой для типа License.

Принципы дизайна SOLID

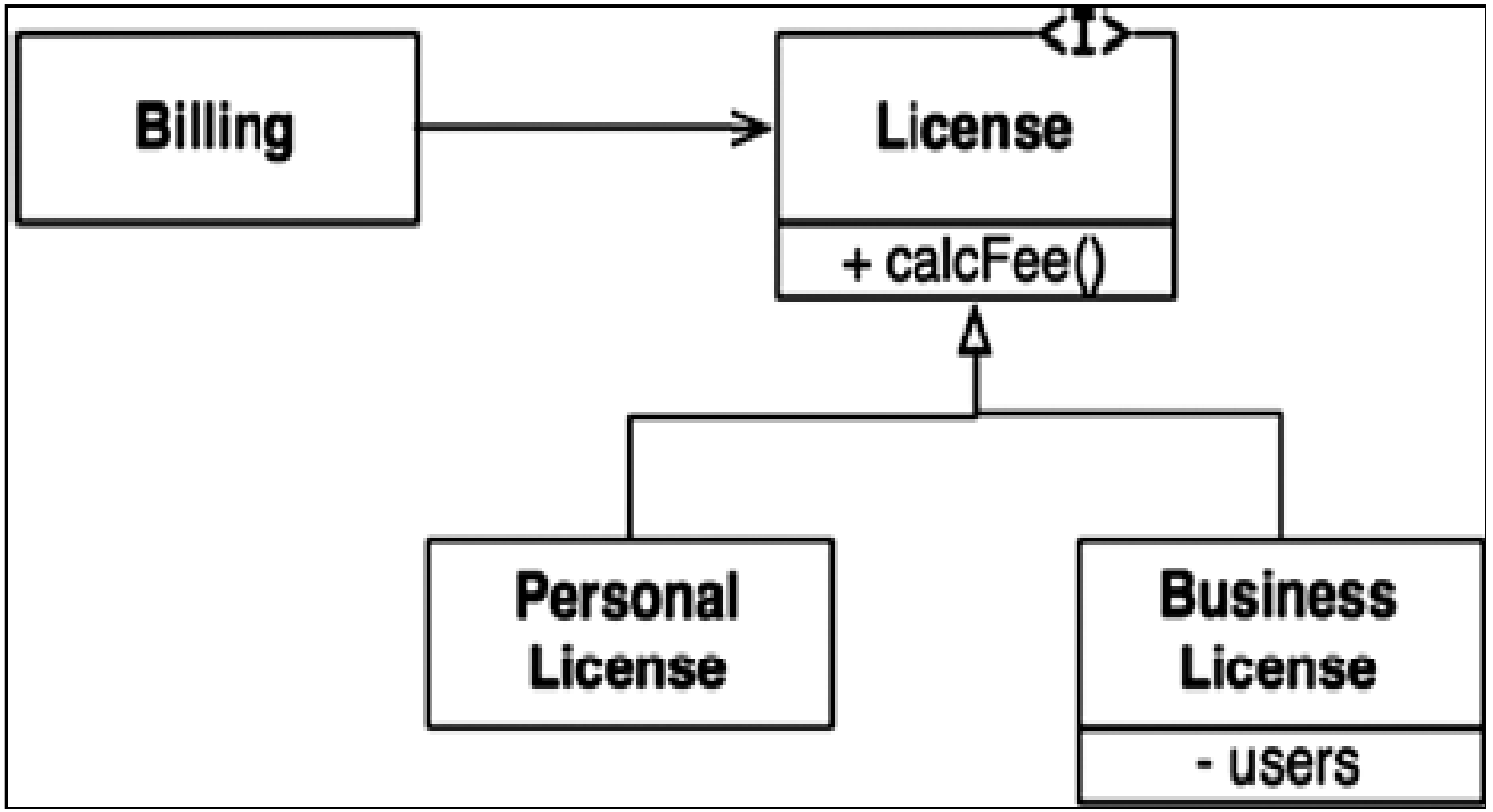

Рис. 2.5 - Класс License и его производные, соответствующие принципу LSP

Принципы дизайна SOLID

Классическим примером нарушения принципа подстановки Барбары Лисков может служить известная проблема квадрат/прямоугольник (рис. 2. 6).

В этом примере класс Square (представляющий квадрат) неправильно определен как подтип класса Rectangle (представляющего прямоугольник), потому что высоту и ширину прямоугольника можно изменять независимо; а высоту и ширину квадрата можно изменять только вместе.

Принципы дизайна SOLID

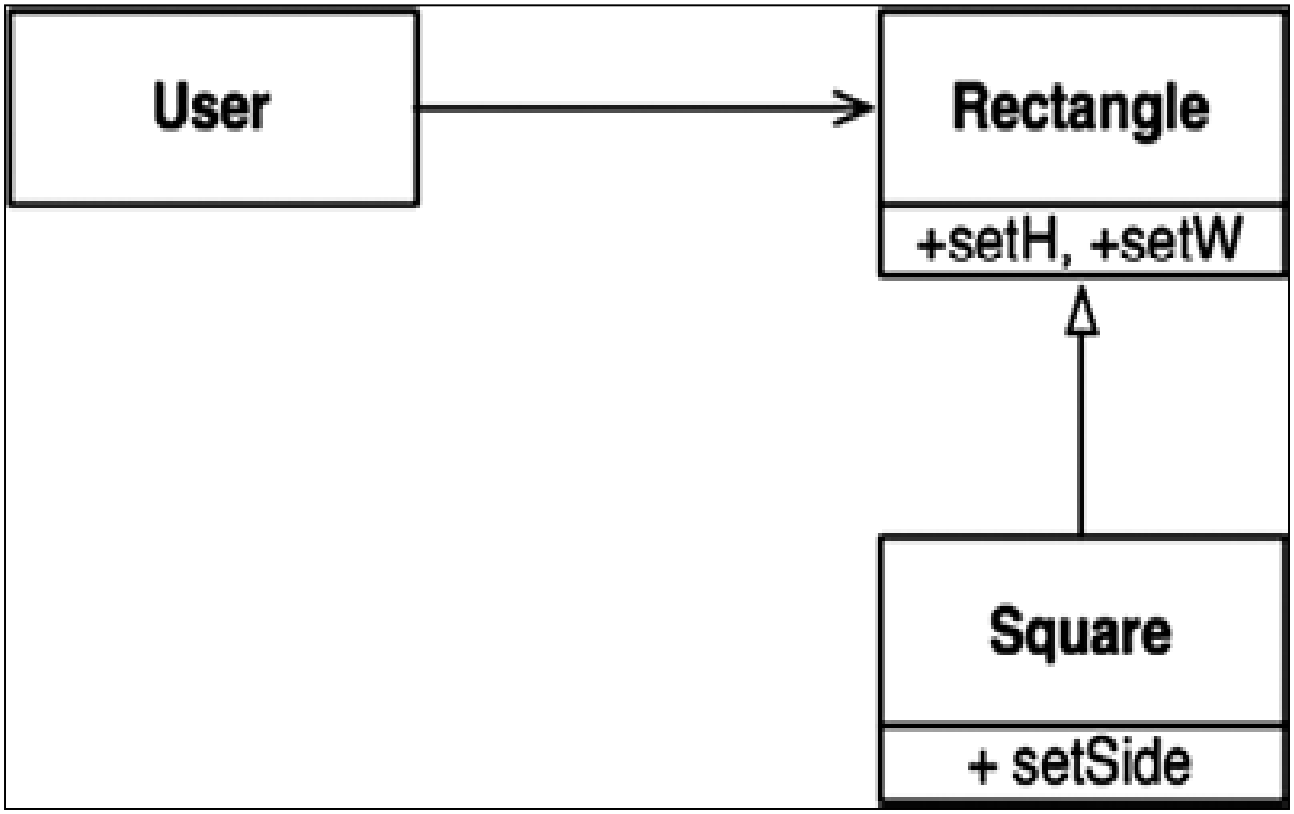

Рис 2.6 - Известная проблема квадрат/прямоугольник

Принципы дизайна SOLID

На заре объектно-ориентированной революции принцип LSP рассматривался как руководство по использованию наследования. Но со временем LSP был преобразован в более широкий принцип проектирования программного обеспечения, который распространяется также на интерфейсы и реализации.

Подразумеваемые интерфейсы могут иметь множество форм. Это могут быть интерфейсы в стиле Java, реализуемые несколькими классами.

Или это может быть набор служб, соответствующих общему интерфейсу REST.

Принципы дизайна SOLID

Во всех этих и многих других ситуациях применим принцип LSP, потому что существуют пользователи, зависящие от четкого определения интерфейсов и замещаемости их реализаций.

Принцип подстановки Барбары Лисков может и должен распространяться до уровня архитектуры. Простое нарушение совместимости может вызвать загрязнение архитектуры системы значительным количеством дополнительных механизмов.

Происхождение названия **принципа разделения интерфейсов (Interface Segregation Principle; ISP)** наглядно иллюстрирует схема на рис. 2.7.

Принципы дизайна SOLID

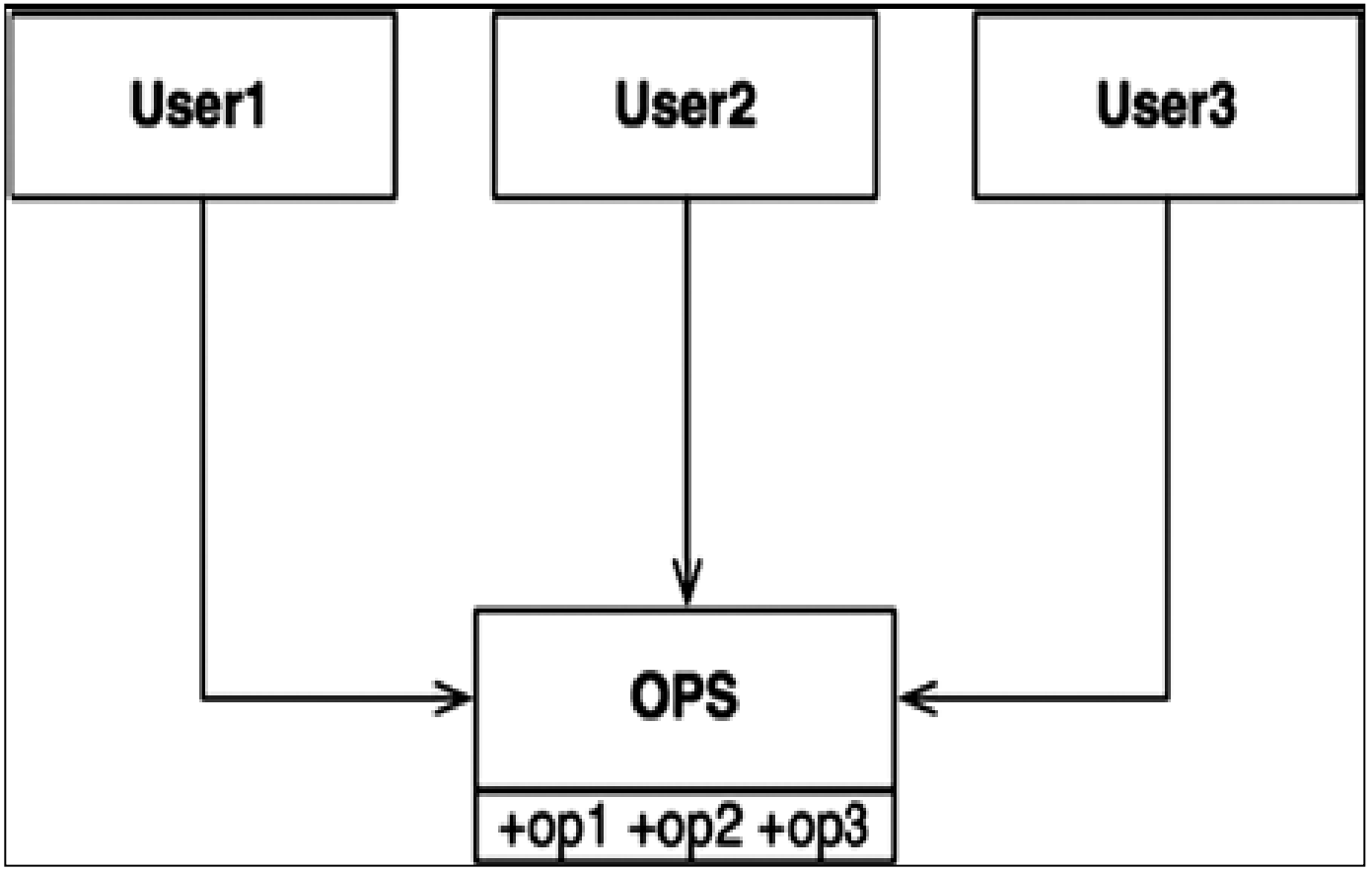

Рис. 2.7 - Принцип разделения интерфейсов.

Принципы дизайна SOLID

В данной ситуации имеется несколько классов, пользующихся операциями в классе OPS. Допустим, что User1 использует только операцию op1, User2 — только op2 и User3 — только op3.

Теперь представьте, что OPS — это класс, написанный на таком языке, как Java. Очевидно, что в такой ситуации исходный код User1 непреднамеренно будет зависеть от op2 и op3, даже при том, что он не пользуется ими. Эта зависимость означает, что изменения в исходном коде метода op2 в классе OPS потребуют повторной компиляции и развертывания класса User1, несмотря на то что для него ничего не изменилось.

Принципы дизайна SOLID

Эту проблему можно решить разделением операций по интерфейсам, как показано на рис. 2.8.

Если снова представить, что этот интерфейс реализован на языке со строгим контролем типов, таком как Java, исходный код User1 будет зависеть от U1Ops и op1, но не от OPS. То есть изменения в OPS, которые не касаются User1, не потребуют повторной компиляции и развертывания User1.

Очевидно, что описание выше в значительной степени зависит от типа языка. Языки со статическими типами, такие как Java, вынуждают программистов создавать объявления, которые должны импортироваться или подключаться к исходному коду пользователя как-то иначе.

Принципы дизайна SOLID

В языках с динамической типизацией, таких как Ruby или Python, подобные объявления отсутствуют в исходном коде — они определяются автоматически во время выполнения. То есть в исходном коде отсутствуют зависимости, вынуждающие выполнять повторную компиляцию и развертывание.

Это главная причина, почему системы на языках с динамической типизацией получаются более гибкими и с меньшим количеством строгих связей.

Этот факт ведет нас к заключению, что принцип разделения интерфейсов является проблемой языка, а не архитектуры.

Принципы дизайна SOLID

Принцип инверсии зависимости (DIP) утверждает, что наиболее гибкими получаются системы, в которых зависимости в исходном коде направлены на абстракции, а не на конкретные реализации.

В языках со статической системой типов, таких как Java, это означает, что инструкции use, import и include должны ссылаться только на модули с исходным кодом, содержащим интерфейсы, абстрактные классы и другие абстрактные объявления. Никаких зависимостей от конкретных реализаций не должно быть. То же правило действует для языков с динамической системой типов, таких как Ruby или Python.

Принципы дизайна SOLID

Исходный код не должен зависеть от модулей с конкретной реализацией. Однако в этих языках труднее определить, что такое конкретный модуль. В частности, это любой модуль, в котором реализованы вызываемые функции.

Очевидно, что принять эту идею за правило практически невозможно, потому что программные системы должны зависеть от множества конкретных особенностей. Например, String в Java — это конкретный класс и его невозможно сделать абстрактным. Зависимости исходного кода от конкретного модуля java.lang.String невозможно и не нужно избегать.

Принципы дизайна SOLID

С другой стороны, класс String очень стабилен. Изменения в этот класс вносятся крайне редко и жестко контролируются. Программистам и архитекторам не приходится беспокоиться о частых и непредсказуемых изменениях в String.

По этим причинам можно игнорировать фундамент операционной системы и платформы, рассуждая о принципе инверсии зависимости. Мы терпим эти конкретные зависимости, потому что уверенно можем положиться на их постоянство. Мы должны избегать зависимости от неустойчивых конкретных элементов системы. То есть от модулей, которые продолжают активно разрабатываться и претерпевают частые изменения.

Принципы дизайна SOLID

Каждое изменение абстрактного интерфейса вызывает изменение его конкретной реализации. Изменение конкретной реализации, напротив, не всегда сопровождается изменениями и даже обычно не требует изменений в соответствующих интерфейсах. То есть интерфейсы менее изменчивы, чем реализации.

Действительно, хорошие дизайнеры и архитекторы программного обеспечения всеми силами стремятся ограничить изменчивость интерфейсов. Они стараются найти такие пути добавления новых возможностей в реализации, которые не потребуют изменения интерфейсов. Это основа проектирования программного обеспечения.

Принципы дизайна SOLID

Как следствие, стабильными называются такие архитектуры, в которых вместо зависимостей от переменчивых конкретных реализаций используются зависимости от стабильных абстрактных интерфейсов. Это следствие сводится к набору очень простых правил:

- Не ссылайтесь на изменчивые конкретные классы. Ссылайтесь на абстрактные интерфейсы. Это правило применимо во всех языках, независимо от устройства системы типов. Оно также накладывает важные ограничения на создание объектов и определяет преимущественное использование шаблона «Абстрактная фабрика».

Принципы дизайна SOLID

- Не наследуйте изменчивые конкретные классы. Это естественное следствие из предыдущего правила, но оно достойно отдельного упоминания. Наследование в языках со статической системой типов является самым строгим и жестким видом отношений в исходном коде; следовательно, его следует использовать с большой осторожностью. Наследование в языках с динамической системой типов влечет меньшее количество проблем, но все еще остается зависимостью, поэтому дополнительная предосторожность никогда не помешает.

Принципы дизайна SOLID

- Не переопределяйте конкретные функции. Конкретные функции часто требуют зависимостей в исходном коде. Переопределяя такие функции, вы не устраняете эти зависимости — фактически вы наследуете их. Для управления подобными зависимостями нужно сделать функцию абстрактной и создать несколько ее реализаций.
- Никогда не ссылайтесь на имена конкретных и изменчивых сущностей. В действительности это всего лишь перефразированная форма самого принципа.

Принципы дизайна SOLID

Чтобы соблюсти все эти правила, необходимо предусмотреть особый способ создания изменчивых объектов. Это объясняется тем, что практически во всех языках создание объектов связано с образованием зависимостей на уровне исходного кода от конкретных определений этих объектов.

В большинстве объектно-ориентированных языков, таких как Java, для управления подобными нежелательными зависимостями можно использовать шаблон «Абстрактная фабрика».

Рисунок 2.8 демонстрирует, как работает такая схема.

Принципы дизайна SOLID

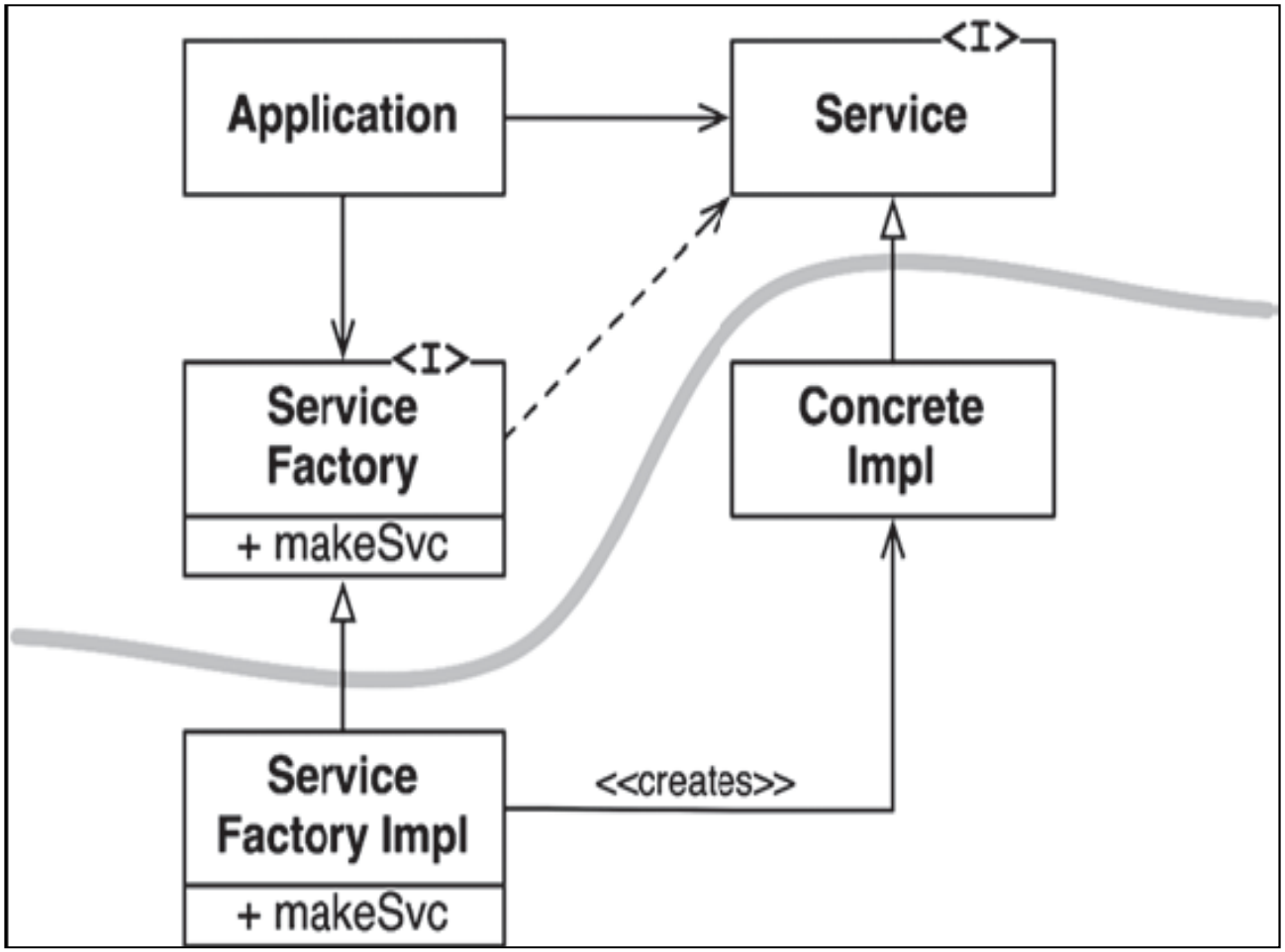

Рис. 2.8 - Использование шаблона «Абстрактная фабрика» для управления зависимостями

Принципы дизайна SOLID

Приложение Application использует конкретную реализацию ConcreteImpl через интерфейс Service. Однако приложению требуется каким-то образом создавать экземпляры ConcreteImpl.

Чтобы решить эту задачу без образования зависимости от ConcreteImpl на уровне исходного кода, приложение вызывает метод makeSvc интерфейса фабрики ServiceFactory. Этот метод реализован в классе ServiceFactoryImpl, наследующем ServiceFactory. Эта реализация создает экземпляр ConcreteImpl и возвращает его как экземпляр интерфейса Service.

Извилистая линия на рис. 2.8 обозначает архитектурную границу.

Принципы дизайна SOLID

Она отделяет абстракцию от конкретной реализации. Все зависимости в исходном коде пересекают эту границу в одном направлении — в сторону абстракции. Извилистая линия делит систему на два компонента: абстрактный и конкретный.

Абстрактный компонент содержит все высокоуровневые бизнес-правила приложения. Конкретный компонент содержит детали реализации этих правил.

Обратите внимание, что поток управления пересекает извилистую линию в направлении, обратном направлению зависимостей в исходном коде.

Принципы дизайна SOLID

Зависимости следуют в направлении, противоположном направлению потока управления — именно поэтому принцип получил название принципа инверсии зависимости.

Конкретный компонент ConcreteImpl на рис. 2.8 имеет единственную зависимость, то есть он нарушает принцип DIP. Это нормально. Полностью устранить любые нарушения принципа инверсии зависимости невозможно, но их можно сосредоточить в узком круге конкретных компонентов и изолировать от остальной системы.

Большинство систем будет содержать хотя бы один такой конкретный компонент.