Разделение ответственностей

Денис С. Мигинский

Принцип разделения ответственностей (напоминание)

Разделение ответственности (separation of concerns, SoC) – программа должны состоять из функциональных блоков, как можно меньше дублирующих функциональность друг друга *(Э. Дейкстра)*.

Concern – ответственность, функциональность

Связь с требованиями

Функциональность программы проистекает из **требований** к ней.

Требования бывают:

(разделение по сути требований)

- Функциональные
- Нефункциональные

(разделение по происхождению)

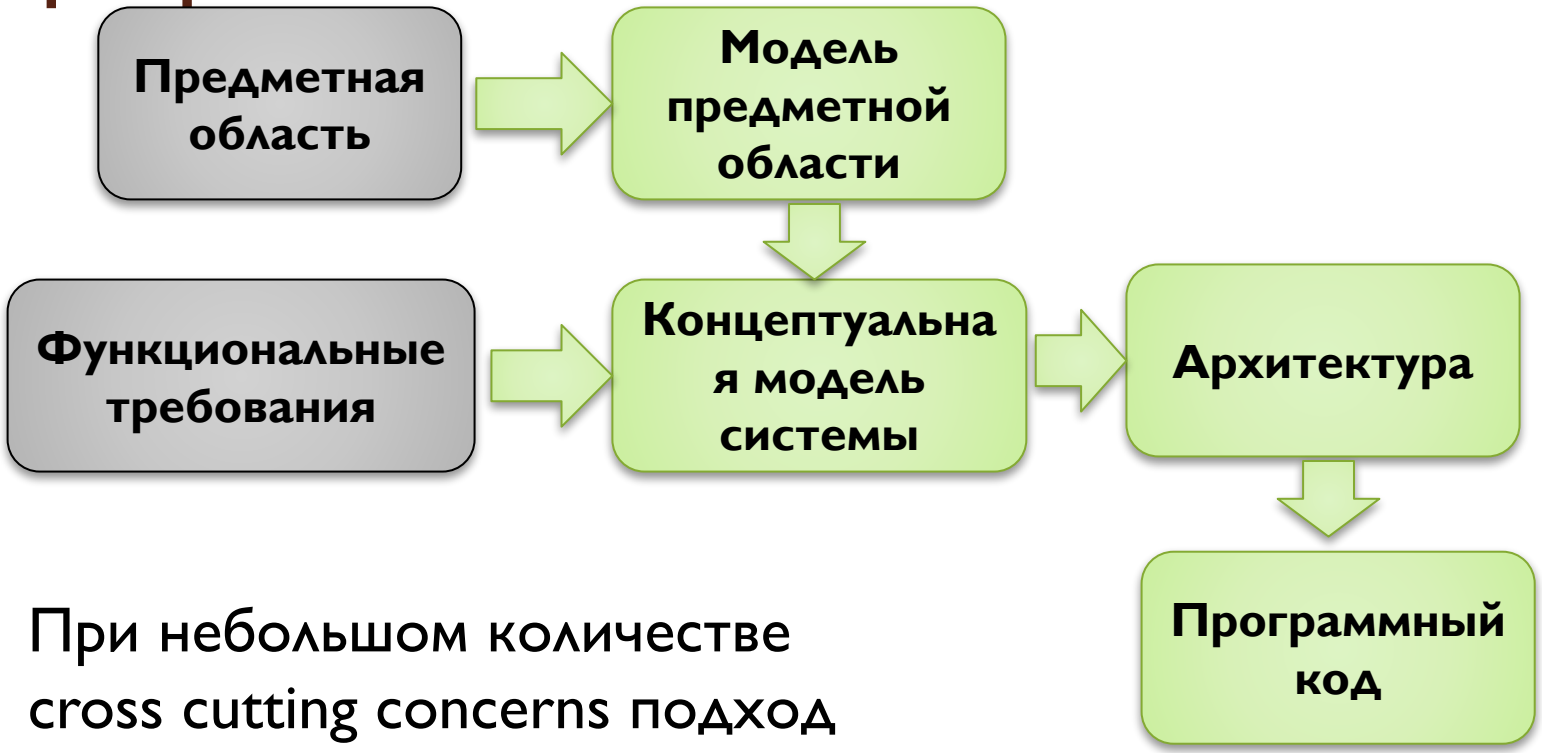
- Требования заказчика
- Внутренние требования

Виды ответственностей

Core concerns (ответственности 1-го класса) – бизнес-логика приложения, следует непосредственно из функциональных требований

Cross-cutting concerns (CCC, ответственности 2-го класса, сквозная функциональность) – функциональность, не относящаяся к бизнес-логике. Следует в основном из нефункциональных требований, а также из внутренних требований (расширяемость, сопровождаемость и т.д.). Имеет свойство тесно «переплетаться» с основной функциональностью.

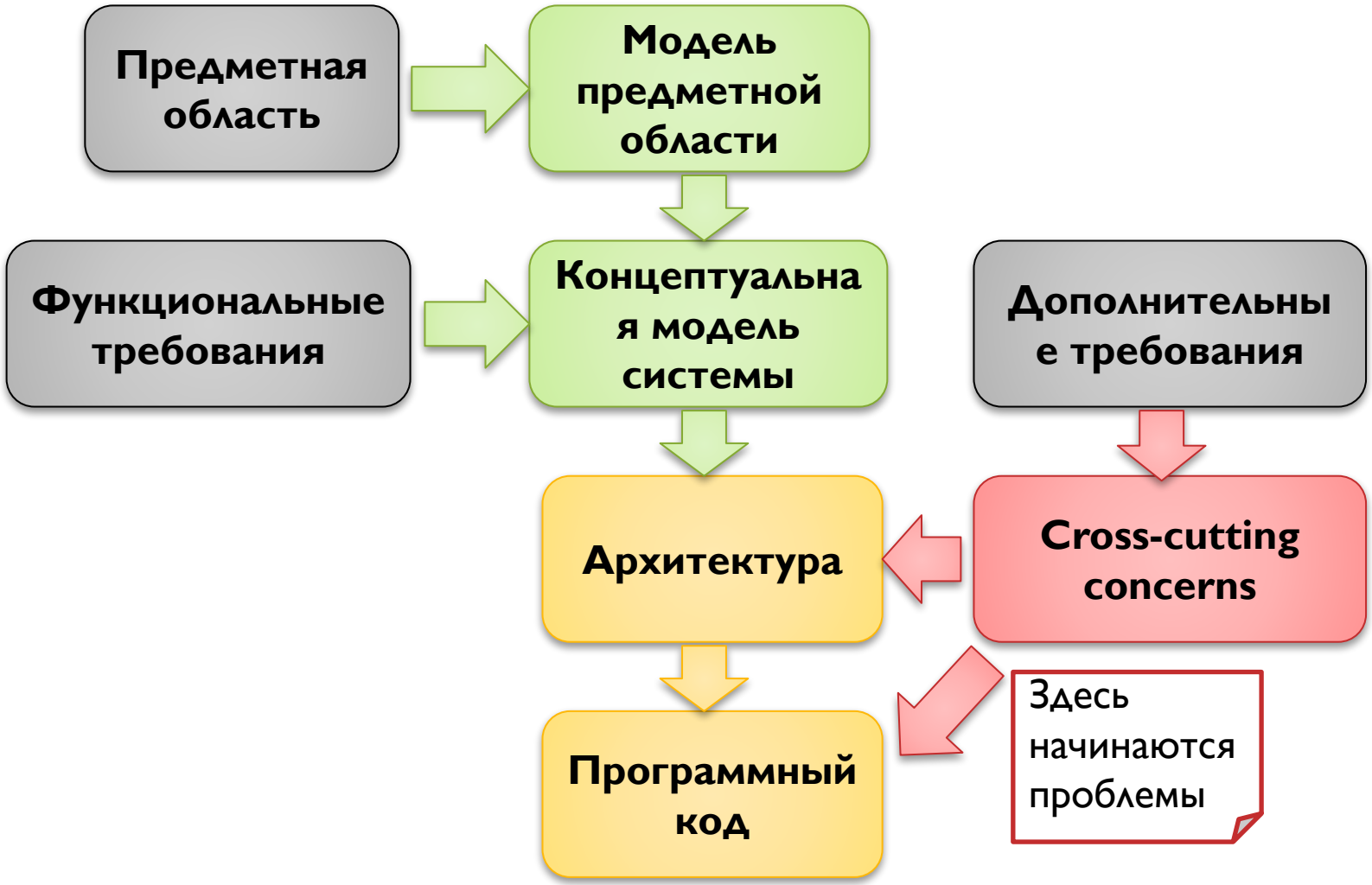
Роль ООП (как методологии) в разработке ПО



При небольшом количестве cross cutting concerns подход обычно **работает эффективно**.

Основная причина: объектный способ мышления естественен для человека.

ООП + ССС


Предметная область
Модель предметной области
Функциональные требования
Концептуальная модель системы
Дополнительные требования
Архитектура
Cross-cutting concerns
Программный код
Здесь начинаются проблемы

Нефункциональные требования

- Платформа («железо», ОС, языки, библиотеки)
- Производительность (performance)
- Масштабируемость (scalability)
- Распределение по физическим узлам
- Простота поддержки, расширения, повторного использования модулей (maintainability, extensibility, reusability)
- Поддержка реального времени
- …

Cross-cutting concerns

- Инициализация, конфигурация
- Управление жизненным циклом объектов
- Оптимизация (в т.ч. кэширование)
- Персистентность
- Журналирование
- Транзакции
- Многопоточность и синхронизация
- Безопасность
- …

Ограничения распространенных парадигм (или их общепринятых реализаций)

Структурное программирование: процедура не может быть разделена на взаимно-независимые составляющие

Объектно-ориентированное программирование:

диспетчеризация методов (динамический полиморфизм) лишь немного ослабляет проблему СП.

Класс является неделимой сущностью.

Способы борьбы с ССС

- Композиция разных парадигм
- Расширенные объектные модели
  - интроспекция
  - вспомогательные методы
  - динамические модели
  - Meta-Object Protocol
- Аспектно-ориентированное программирование
  - аннотация кода (внутреннее или внешнее)
  - инструментирование байт-кода
  - динамические контексты/«контекстный полиморфизм»
  - Inversion of Control
- Мета-программирование
  - кодогенерация (в т.ч. макросы, препроцессоры)
  - метамодели/формальные онтологии
  - Language-Oriented Programming

Аспектно-ориентированное программирование

Как парадигма программирования:

- Расщепление классов
- Расщепление методов
- «Контекстный полиморфизм»

Как методология: архитектура строится в терминах аспектов, которые могут пересекать границы классов и методов. Классы сохраняются, но они уже не обязаны являться единицами модульности.

Hollywood principle

Don’t call us, we’ll call you

Цикл обработки сообщений WinAPI

```
/* Так начиналось 99% программ образца 90х под Windows.
   Подсветка синтаксиса оригинала сохранена.*/
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine,
                    int nCmdShow)
{
        MSG msg;
        while(GetMessage(&msg, NULL, 0, 0) > 0) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
        }
        return msg.wParam;
}
```

Инверсия управления (Inversion of Control, IoC, Hollywood Principle)

Задача: использовать альтернативный механизм запуска и/или исполнения приложения (вычислительную модель)

Решение: framework реализует “main” с соответствующей вычислительной моделью, пользовательский код явной точки входа не содержит.

Основные области применения IoC:

Управление жизненным циклом объектов и связей (Dependency Injection)

Интерфейсы пользователя (событийно-ориентированные системы)

Альтернативные вычислительные модели (автоматы, продукции, потоки данных и т.д.)

Понятие framework

Framework вводит новые или изменяет существующие механизмы абстрагирования.

В частности, практически любая реализация IoC является framework'ом.

Примеры:

- Инструментарий для любого языка программирования (runtime, компилятор)
- Расширения языка: объектные модели, ленивые вычисления
- Библиотеки для управления многопоточностью, распределенными вычислениями и т.д.
- Инструменты управления транзакциямию персистентностью
- См. список CCC

IoC и жизненный цикл объекта

Garbage Collector: вместо явного уничтожения объект «забывается» и далее обрабатывается некоторой внешней сущность (GC)

Можно ли тот же самый объект применить еще раз?

Можно ли просто получить объект, не конструируя его явно?

Инициализация зависимостей без применения Dependency Injection

```
public interface ITool{ …}
public class Fork implements ITool{ … }

//non-DI class
public class Person1{
    ITool tool;
    public Person1(){
        //создаем вилку самостоятельно
        tool = new Fork(…);
    }
}
```

Тривиальный Dependency Injection (DI) на основе конструктора

```
//constructor-based DI class
public class Person2{
    ITool tool;
    //требуем некоторый инструмент
    public Person2(ITool tool){
        this.tool = tool;
    }
}
…
  public static void main(){
      ITool tool = new Fork(…);
      //выдаем инструмент (вилку) в пользование
      Person2 person = new Person2(tool);
  }
```

Преимущества DI

- Агрегирующие классы (т.е. те, для которых работает DI) не обязаны зависеть от конкретных классов включаемых объектов
- Можно явно указать какие зависимости будут общими для нескольких агрегирующие классов. При этом агрегирующие классы могут об этом не знать.
- Агрегирующие классы не зависят от «божественной» сущности, которая реализует DI

Проблемы с тривиальным DI

- DI реализует «божественная» сущность, зависящая от всей системы
- Явно позволяет реализовать только статическое связывание. Что делать, если инструмент сломался и его нужно заменить?

DI на основе контейнера

- Контейнер (сущность, реализующая IoC) универсален и не зависит от конкретных классов реализации. Зависимость устанавливается внешними конфигурационными файлами. Сам контейнер работает через интроспекцию (Java Reflection и т.д.)

- Зависимости могут внедряться опосредовано через proxy. Это позволяет подменять зависимости в соответствии с контекстом (поток, сессия и т.д.)

- Возможно конфигурировать жизненный цикл отдельных объектов (типично singleton, session, call)

DI: Spring framework example

```
import javax.inject.Inject;

//container-based DI class
public class Person3 implements IPerson{
   //literally: I need a tool
   @Inject ITool tool;
}

//applicationContext.xml:
//declarative representation of aspect for
//instantiating classes over software system
…
<bean id="person" class="Person3"/>
<bean id="tool" scope="prototype" class="Fork">
   <property name="teethNum" value="4"/>
</bean>
```