Лекция 5

SOLID

Принципы SOLID

- Принципы SOLID определяют объединение функций и структур данных
- Цель принципов – создание программных сущностей среднего уровня:
  - *Терпимы к изменениям*
  - *Просты и понятны*
  - *Образуют основу для компонентов, которые могут использоваться во многих программных средствах*

SOLID

- **S** – Single Responsibility Principle (SRP) – принцип единственной ответственности
- **O** – Open-Closed Principle (OCP) – принцип открытости/закрытости
- **L** – Liskov Substitution Principle (LSP) – принцип подстановки Барбары Лисков
- **I** – Interface Segregation Principle (ISP) – принцип разделения интерфейсов
- **D** – Dependency Inversion Principle (DIP) – принцип инверсии зависимости

Single Responsibility Principle

Принцип единственной ответственности

- Ошибочное определение:

  «Функция должна делать что-то одно и только одно»

Принцип единственной ответственности

- Ошибочное определение:

  «Функция должна делать что-то одно и только одно»

- Традиционное определение:

  «Модуль должен иметь одну и только одну причину для изменения»

Принцип единственной ответственности

- Ошибочное определения:

  «Функция должна делать что-то одно и только одно»

- Традиционное определение:

  «Модуль должен иметь одну и только одну причину для изменения»

- Более точное определение:

  «Модуль должен отвечать за одного и только за одного пользователя или заинтересованное лицо»

Принцип единственной ответственности

- Ошибочное определения:

  «Функция должна делать что-то одно и только одно»

- Традиционное определение:

  «Модуль должен иметь одну и только одну причину для изменения»

- Более точное определение:

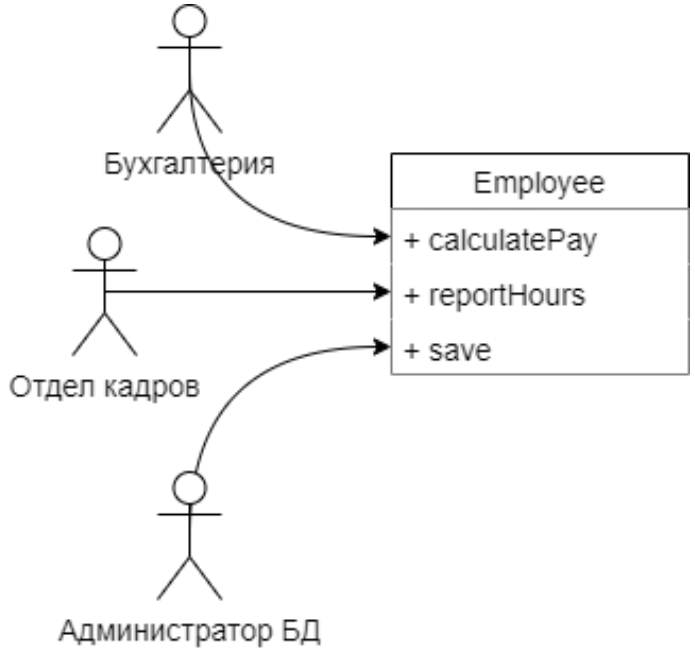
  «Модуль должен отвечать за одного и только за одного пользователя или заинтересованное лицо»

- Самое точное определение:

  «Модуль должен отвечать за одного и только одного актора»

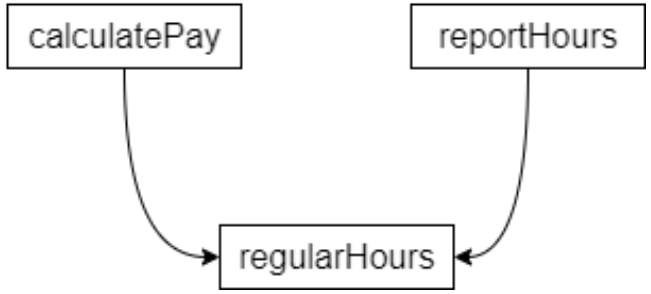
Непреднамеренное дублирование

- Несколько акторов определяют разное поведение в одном модуле
- *calculatePay* и *reportHours* используют один алгоритм, вынесенный в *regularHours*



Проблема:

1. Бухгалтерия запросила изменения в *calculatePay*
2. Для изменения необходимо изменение *regularHours*
3. Изменение влияет на *reportHours*, хотя его изменения не требовались
4. Отдел кадров получает неправильные результаты, думая, что результаты правильные



Слияние

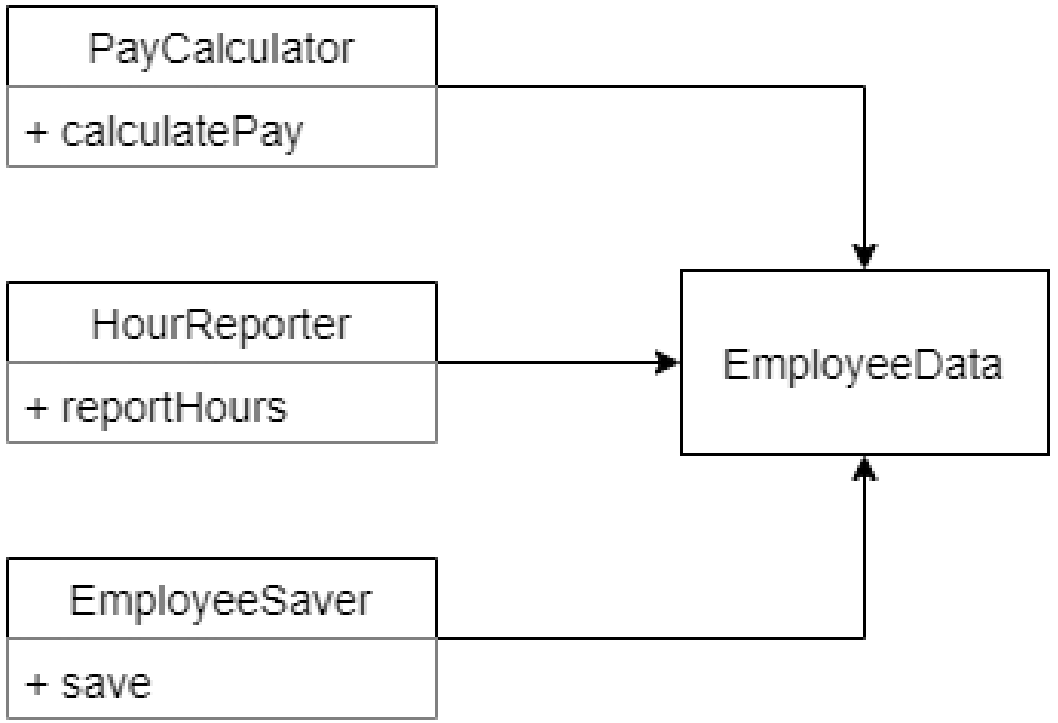
- Возникает в случаях, если в одном файле находится множество методов/классов, отвечающих за разных акторов

- Например, 2 параллельных изменения в одном файле могут оказаться несовместимы

Решение 1

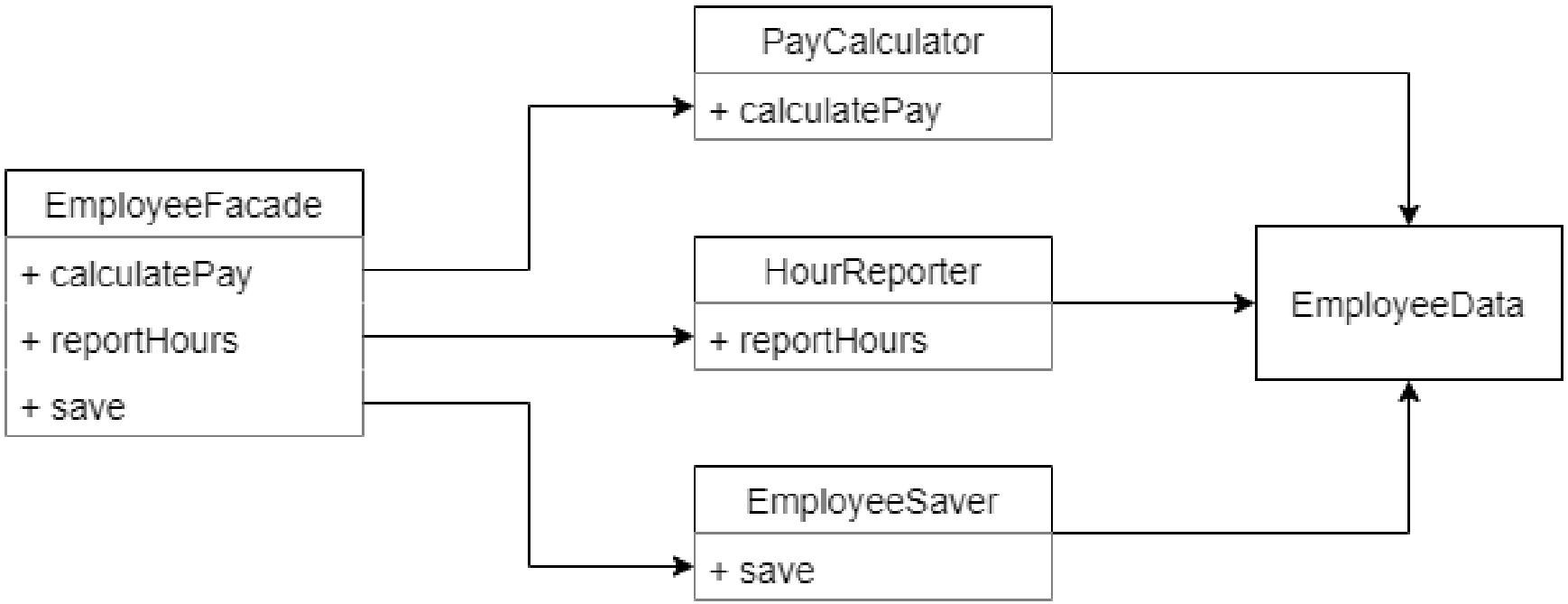
1. Отделение данных от функций

2. Отдельный класс для отдельных функций каждого актора



Решение 2

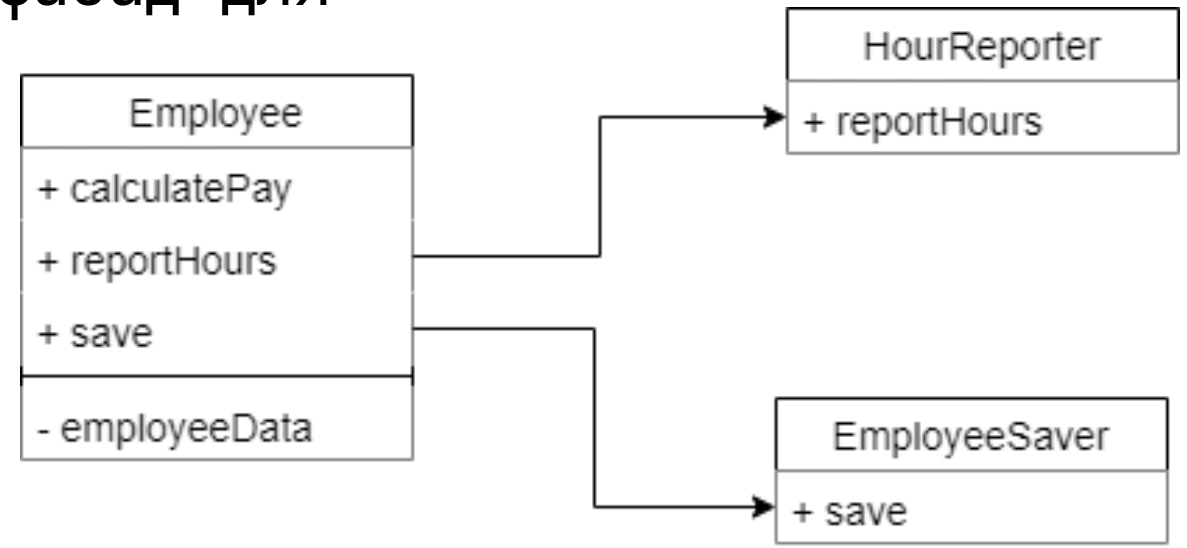
Объединение классов одним фасадом



Решение 3

1. Сохранить наиболее важные методы вместе с данными

2. Использовать класс, как фасад для низкоуровневой логики



Open-Closed Principle

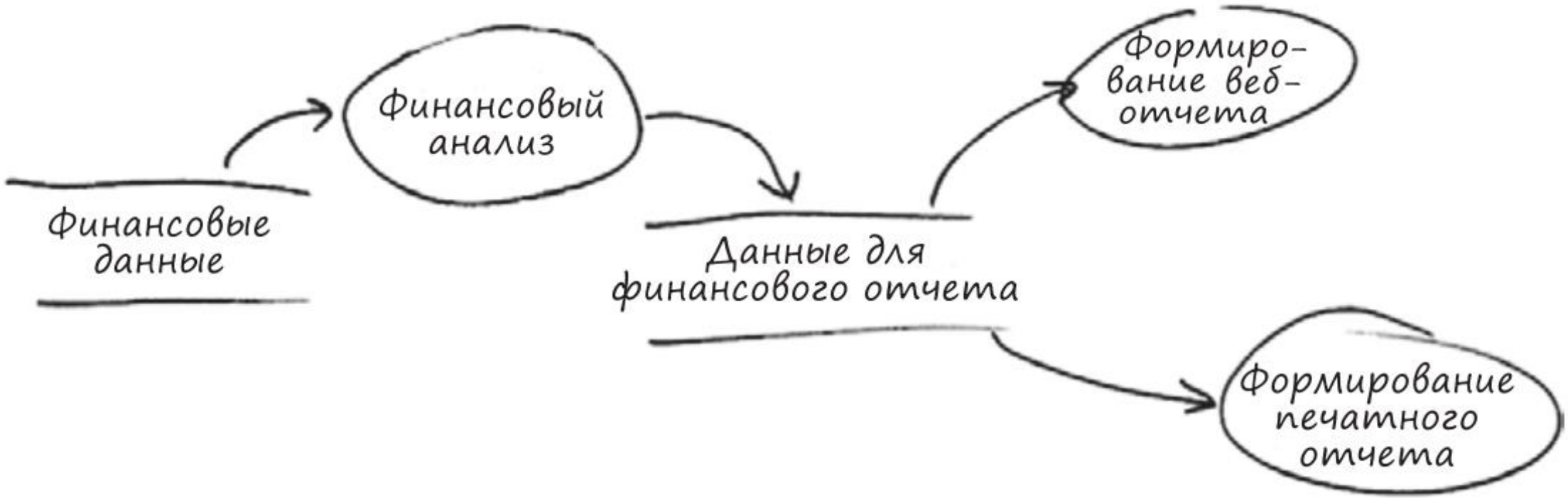
Принцип открытости/закрытости

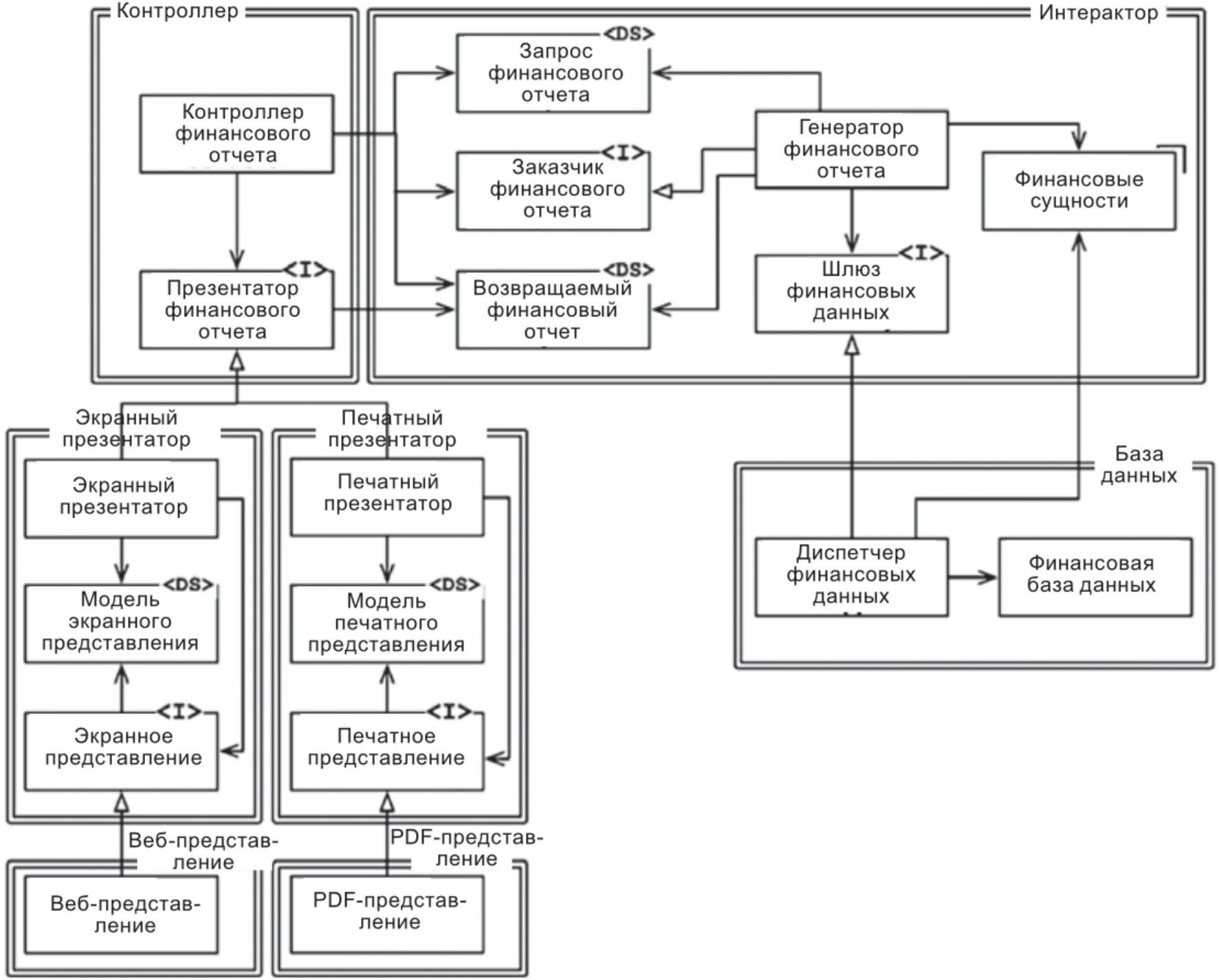
- Определение

  «Программные сущности должны быть открыты для расширения и закрыты для изменения»

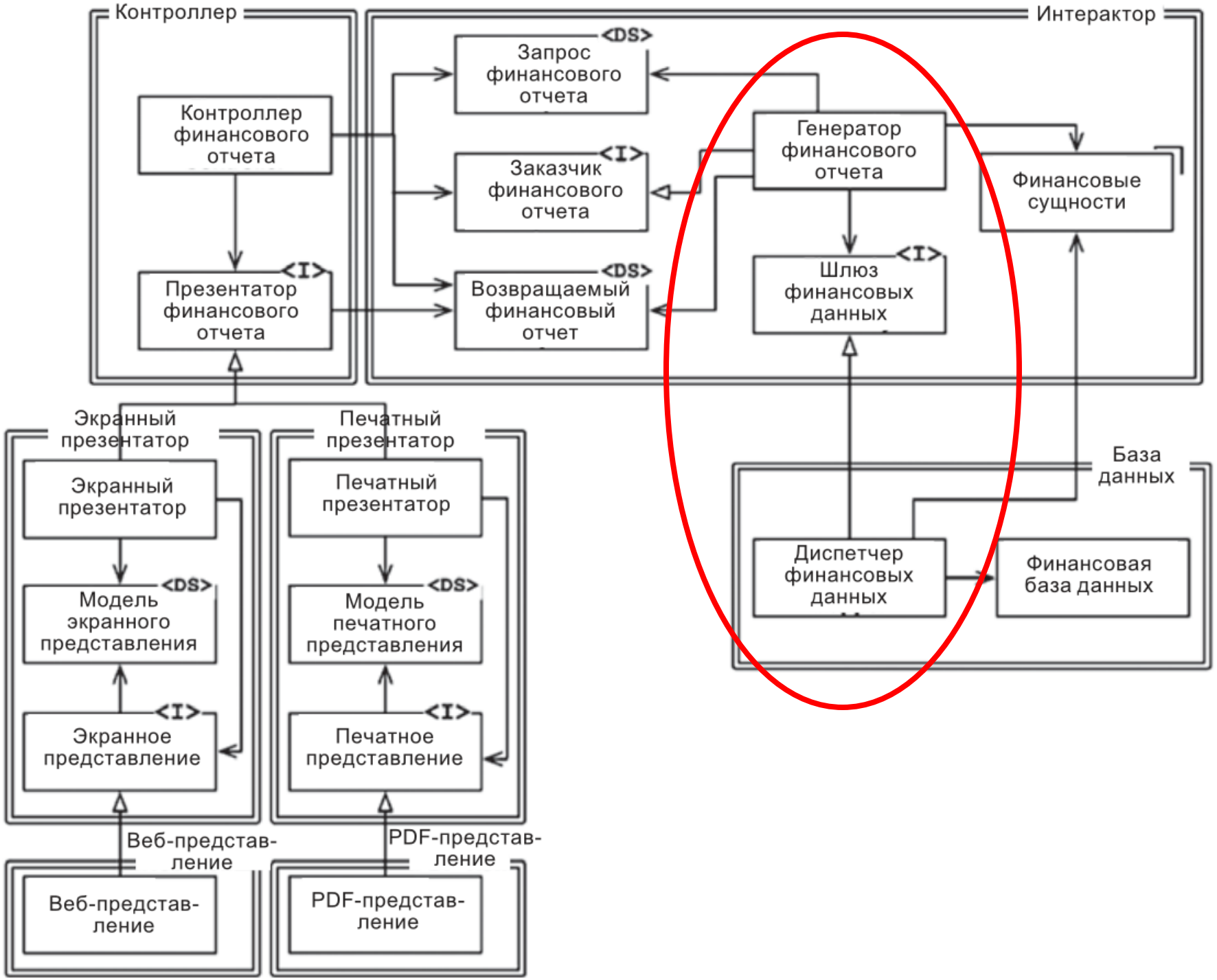
- Если простое расширение функциональности ведет к значительным изменениям в программном обеспечении – архитектура плохая

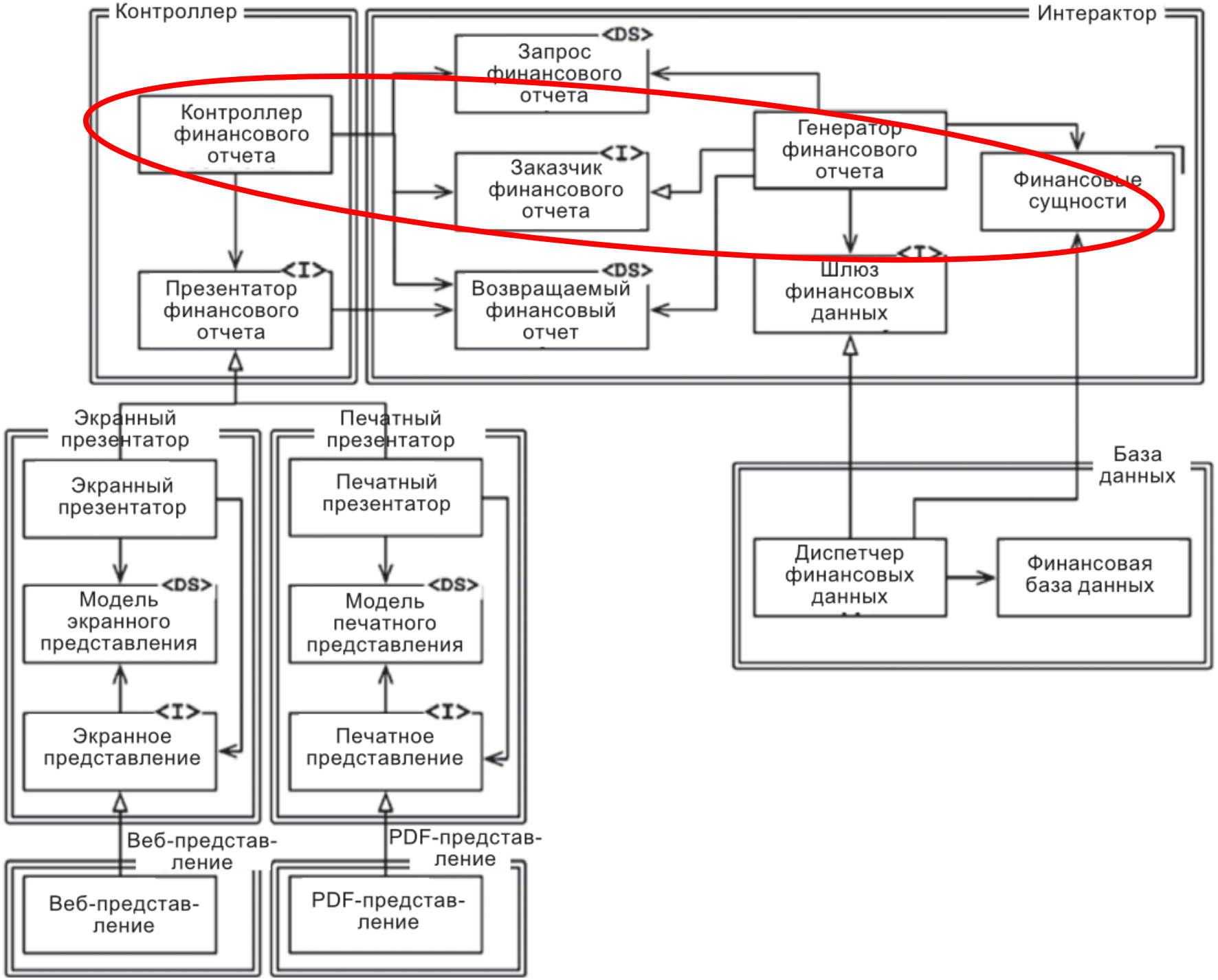
Пример – DFD


Финансовые данные
Финансовый анализ
Данные для финансового отчета
Формиро-вание веб-отчета
Формирование печатного отчета


Контроллер
Интерактор
Контроллер финансового отчета
Презентатор финансового отчета
<I>
Запрос финансового отчета
<DS>
Заказчик финансового отчета
<I>
Возвращаемый финансовый отчет
<DS>
Генератор финансового отчета
Шлюз финансовых данных
<I>
Финансовые сущности
Экранный презентатор
Экранный презентатор
Модель экранного представления
<DS>
Экранное представление
<I>
Печатный презентатор
Печатный презентатор
Модель печатного представления
<DS>
Печатное представление
<I>
База данных
Диспетчер финансовых данных
Финансовая база данных
Веб-представ-ление
Веб-представ-ление
PDF-представ-ление
PDF-представ-ление

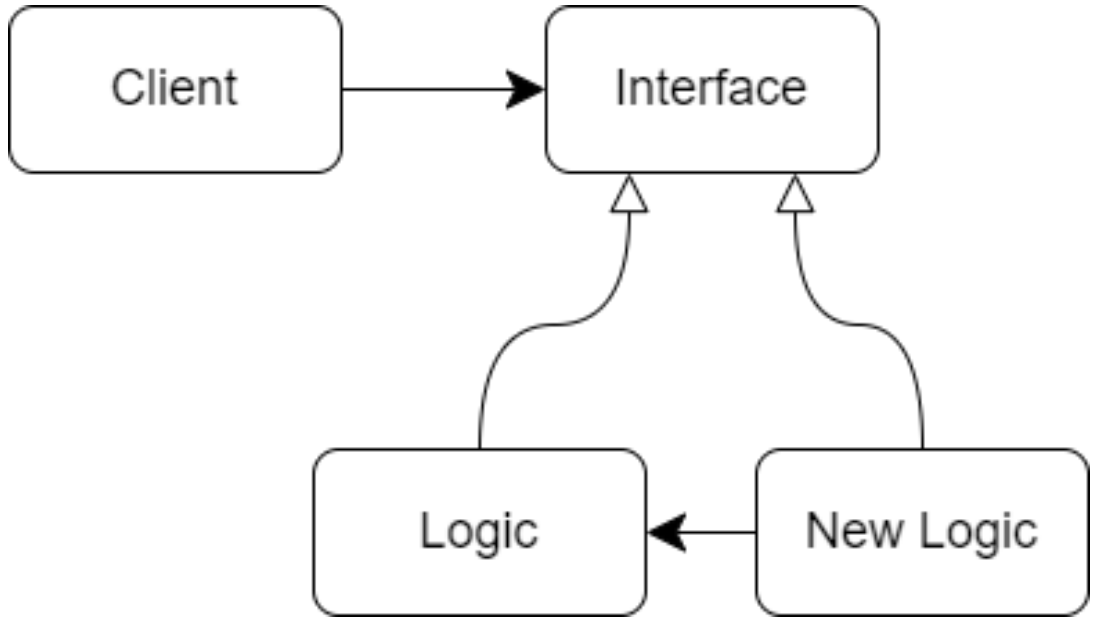

Контроллер финансового отчета
Интерактор финансового отчета
Экранный презентатор
Печатный презентатор
Финансовая база данных
Веб-представ-ление
PDF-представ-ление


Контроллер
Интерактор
Контроллер финансового отчета
Презентатор финансового отчета
<I>
Запрос финансового отчета
<DS>
Заказчик финансового отчета
<I>
Возвращаемый финансовый отчет
<DS>
Генератор финансового отчета
Шлюз финансовых данных
<I>
Финансовые сущности
Экранный презентатор
Экранный презентатор
Модель экранного представления
<DS>
Экранное представление
<I>
Печатный презентатор
Печатный презентатор
Модель печатного представления
<DS>
Печатное представление
<I>
База данных
Диспетчер финансовых данных
Финансовая база данных
Веб-представ-ление
PDF-представ-ление
Веб-представ-ление
PDF-представ-ление


Контроллер
Интерактор
Запрос финансового отчета
<DS>
Контроллер финансового отчета
Генератор финансового отчета
Заказчик финансового отчета
<I>
Финансовые сущности
Презентатор финансового отчета
<I>
Возвращаемый финансовый отчет
<DS>
Шлюз финансовых данных
<I>
Экранный презентатор
Печатный презентатор
База данных
Экранный презентатор
Печатный презентатор
Диспетчер финансовых данных
Финансовая база данных
Модель экранного представления
<DS>
Модель печатного представления
<DS>
Экранное представление
<I>
Печатное представление
<I>
Веб-представ-ление
PDF-представ-ление
Веб-представ-ление
PDF-представ-ление

Соблюдение OCP

- Клиент работает через интерфейс
- Добавление функциональности через создание новой сущности, а не изменение старой
- Новая сущность может делегировать часть работы старой сущности



Liskov Substitution Principle

Принцип подстановки Барбары Лисков

- Определение:

  «Здесь требуется что-то вроде следующего свойства подстановки: если для каждого объекта o1 типа S существует такой объект o2 типа T, что для всех программ P, определенных в терминах T, поведение P не изменяется при подстановке o1 вместо o2, то S является подтипом T»

Принцип подстановки Барбары Лисков

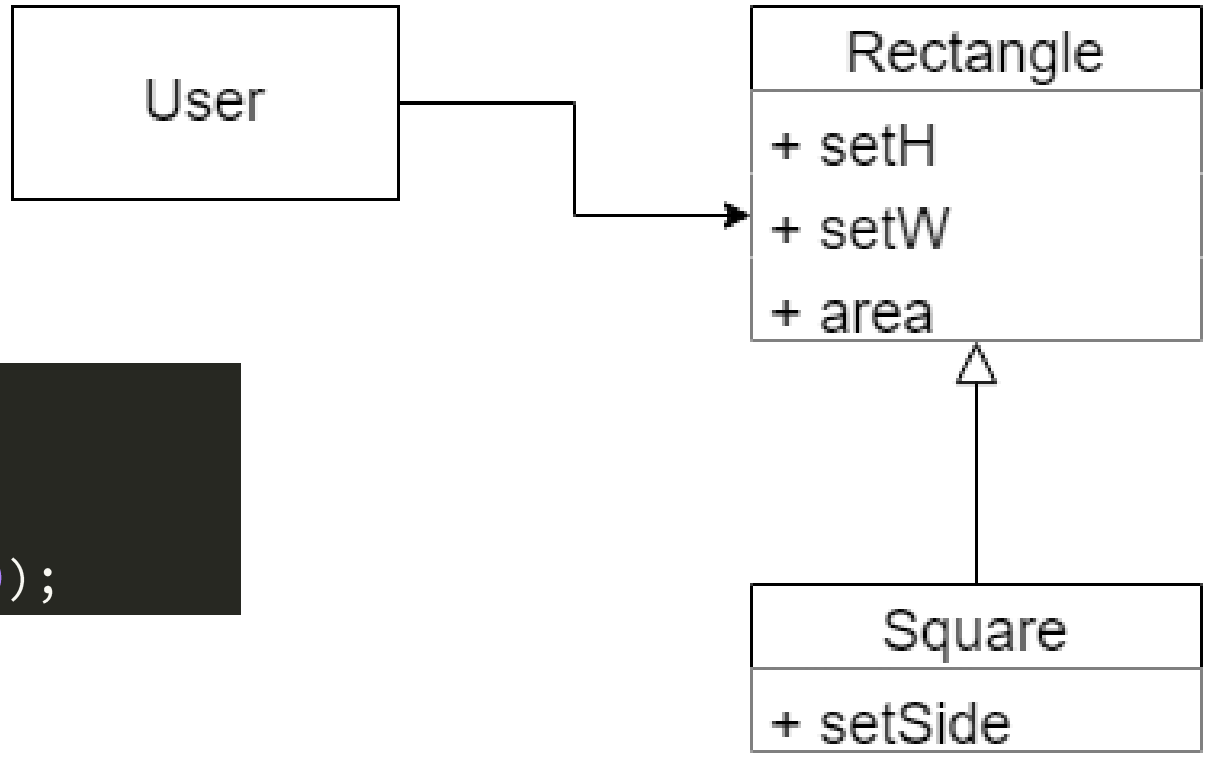
- Определение:

  «Здесь требуется что-то вроде следующего свойства подстановки: если для каждого объекта o1 типа S существует такой объект o2 типа T, что для всех программ P, определенных в терминах T, поведение P не изменяется при подстановке o1 вместо o2, то S является подтипом T»

- Простое определение:

  «Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом»

Проблема квадрат/прямоугольник




```
Rectangle r = ...;
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

Interface Segregation Principle

Принцип разделения интерфейсов

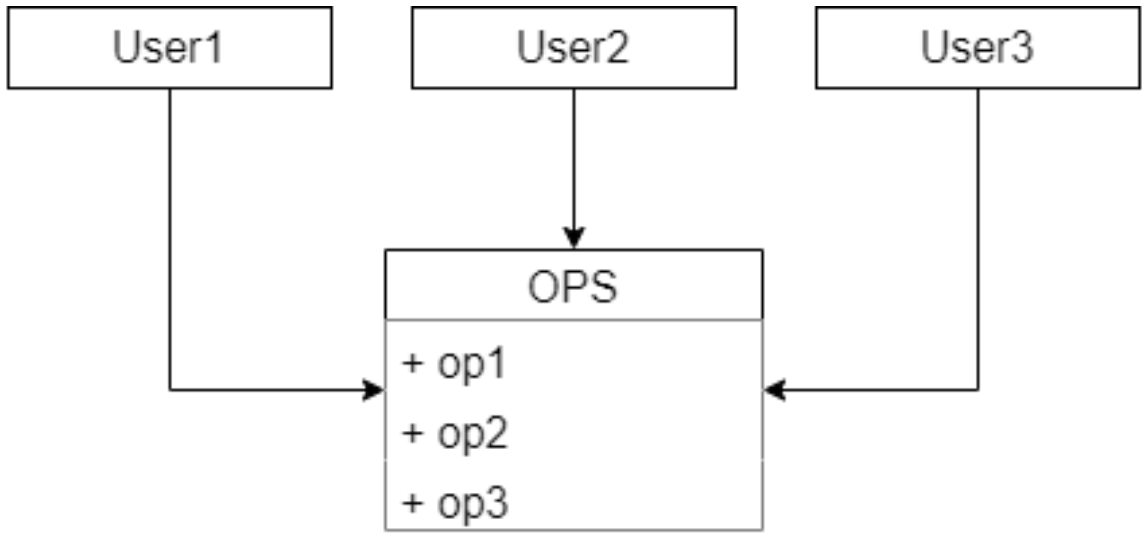
- Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения

- Разделение интерфейса облегчает использование и тестирование модулей

Проблема единого интерфейса

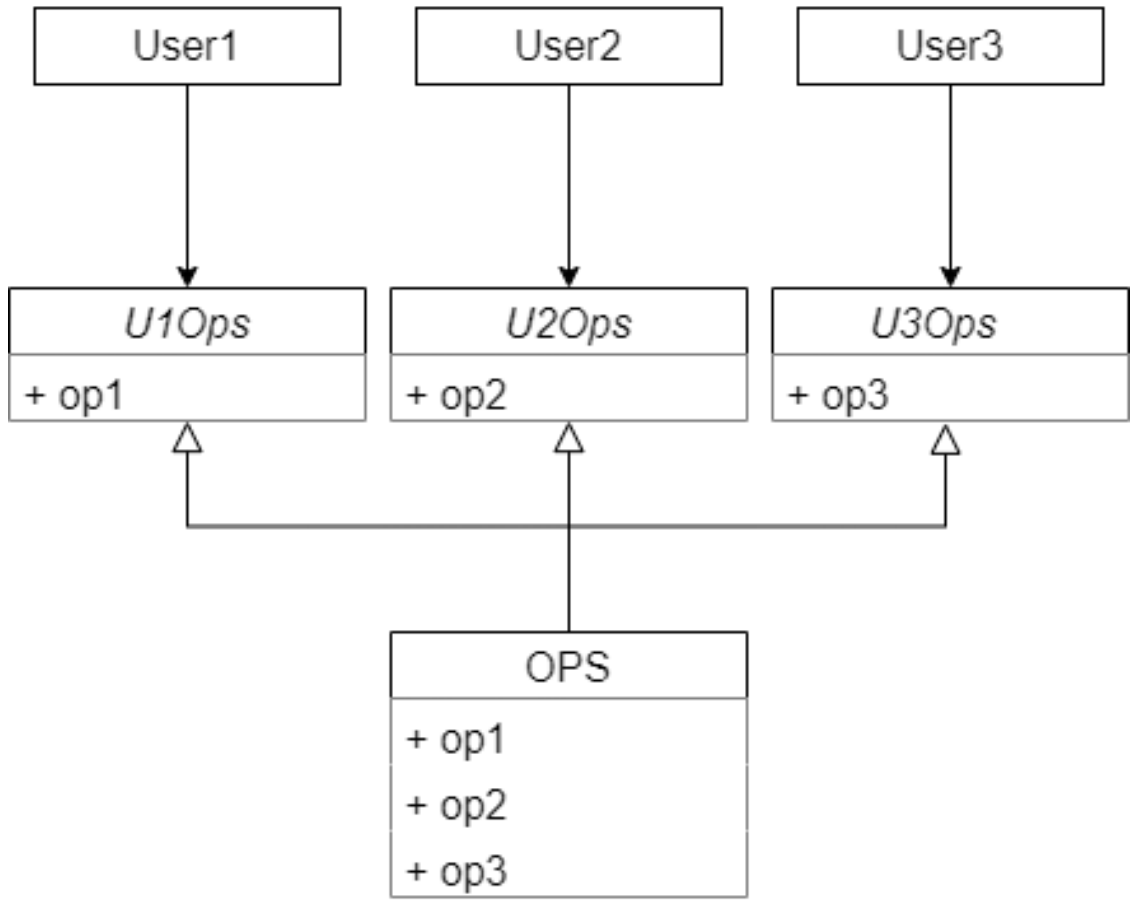
- User1 использует только op1
- User2 использует только op2
- User3 использует только op3

- Изменение op1 косвенно влияет на User2 и User3 и может требовать их повторного развертывания



Решение

- Предоставить каждому User собственный интерфейс



Dependency Inversion Principle

Принцип инверсии зависимости

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций

- Например, в разных языках инструкции *use*, *import*, *include* должны ссылаться на модули только с интерфейсами

- Изменение интерфейса влечет изменения конкретной реализации, но изменение конкретной реализации не всегда требует изменения интерфейса

Правила для соблюдения

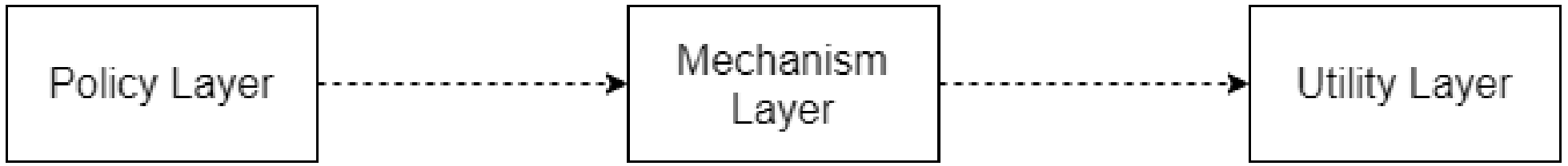
- Не ссылайтесь на изменчивые конкретные классы
- Не наследуйте изменчивые конкретные классы
- Не переопределяйте конкретные функции
- Не ссылайтесь на имена конкретных и изменчивых сущностей

Проблема

Mechanism Layer зависит от Utility Layer

Policy Layer зависит от Mechanism Layer

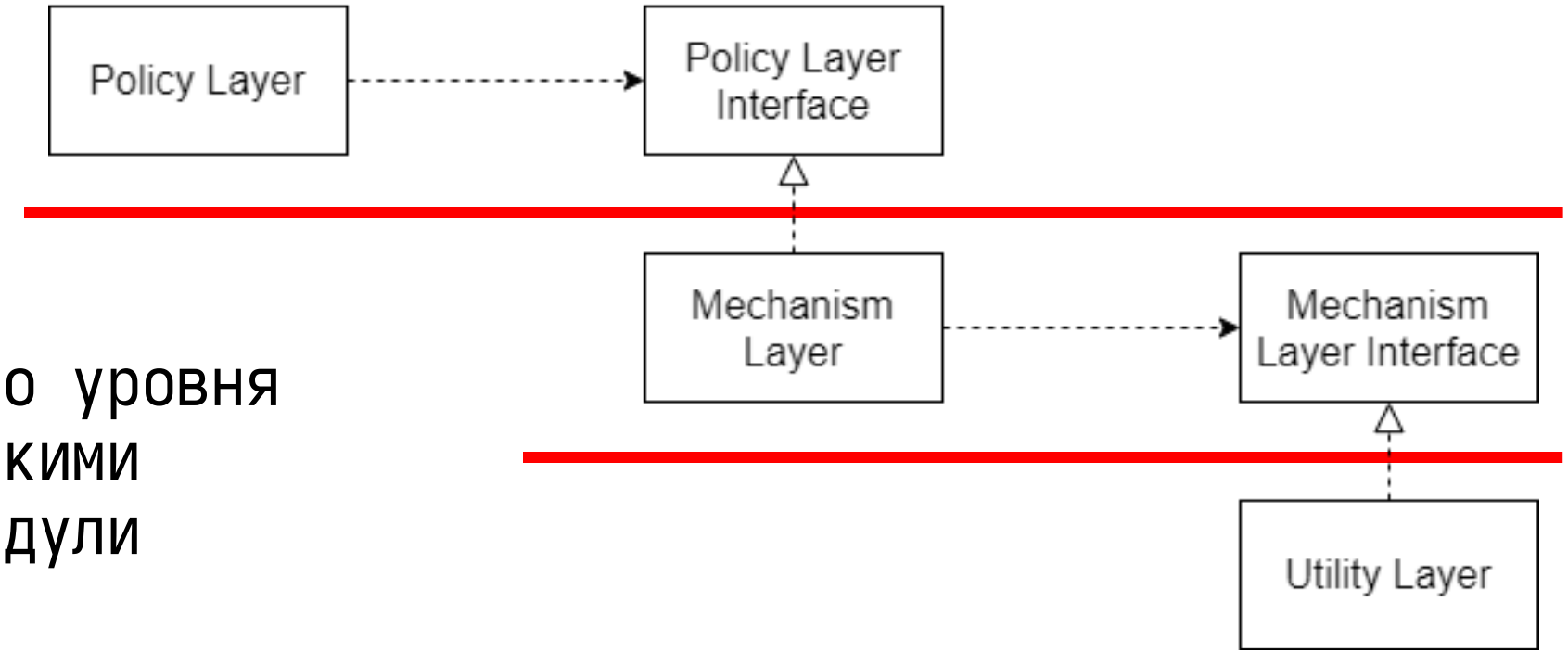
Policy Layer зависит от Utility Layer



Верхний уровень

Нижний уровень

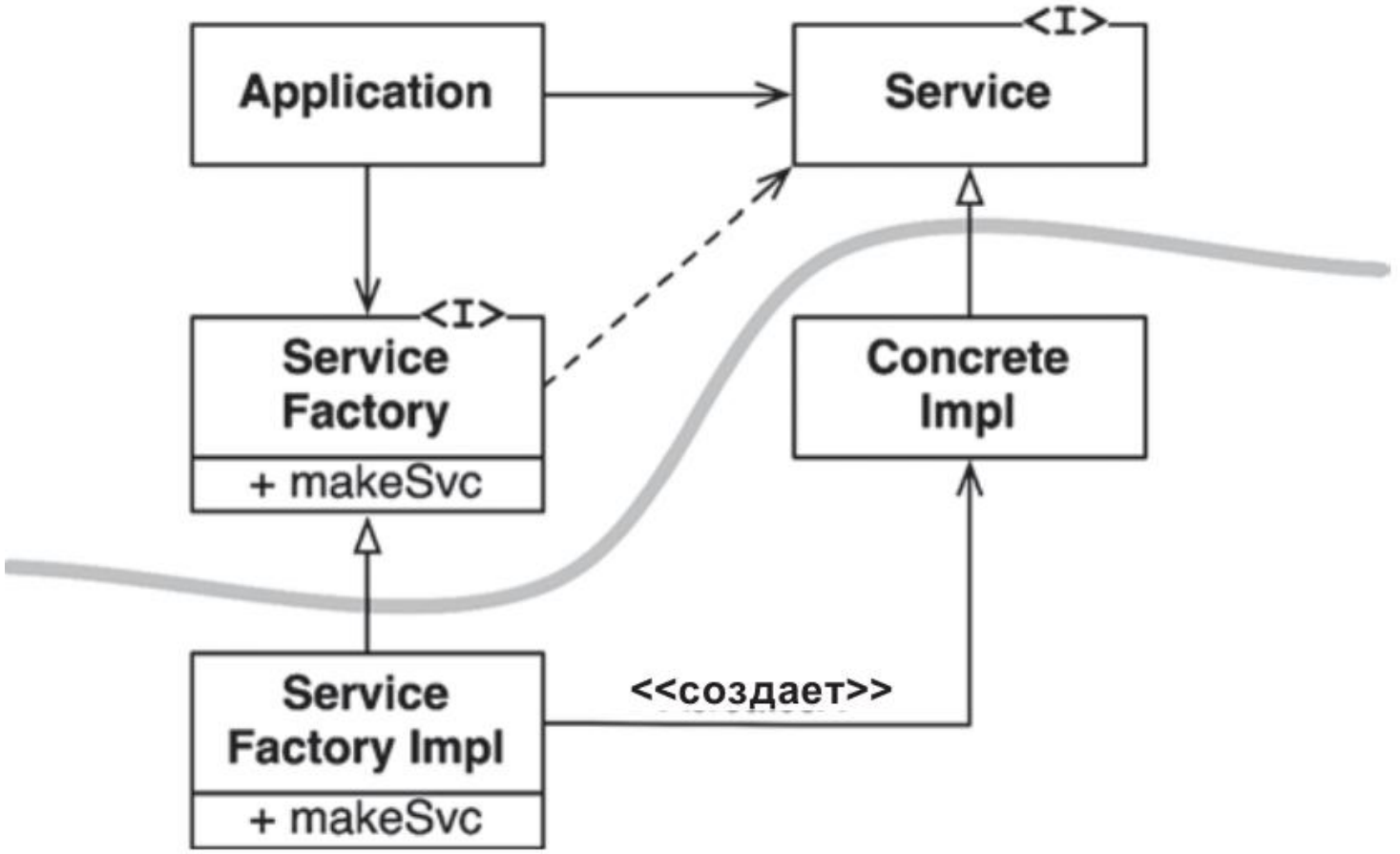
Решение



Модули верхнего уровня диктуют то, какими должны быть модули нижнего уровня

Использование фабрик

Архитектурна граница – разделяет абстрактные и конкретные компоненты



Архитектурные паттерны

Архитектурные паттерны

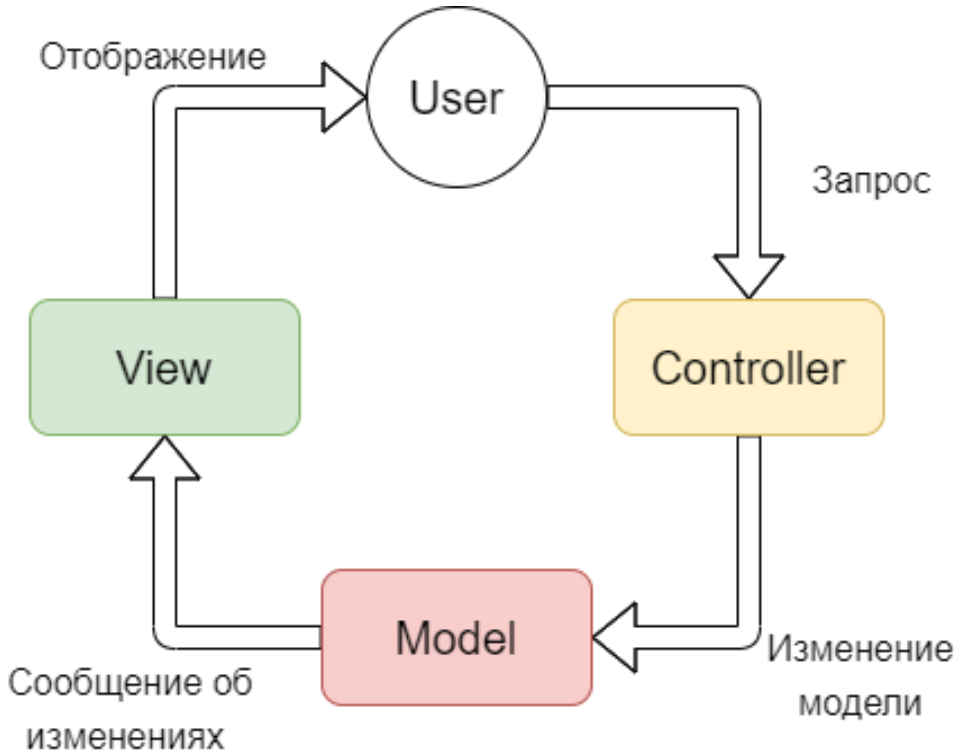
- Позволяют достигнуть:
  - Масштабируемости
  - Поддерживания
  - Отказоустойчивости

- Применимы не только в ООП парадигме

Model View Controller (MVC)

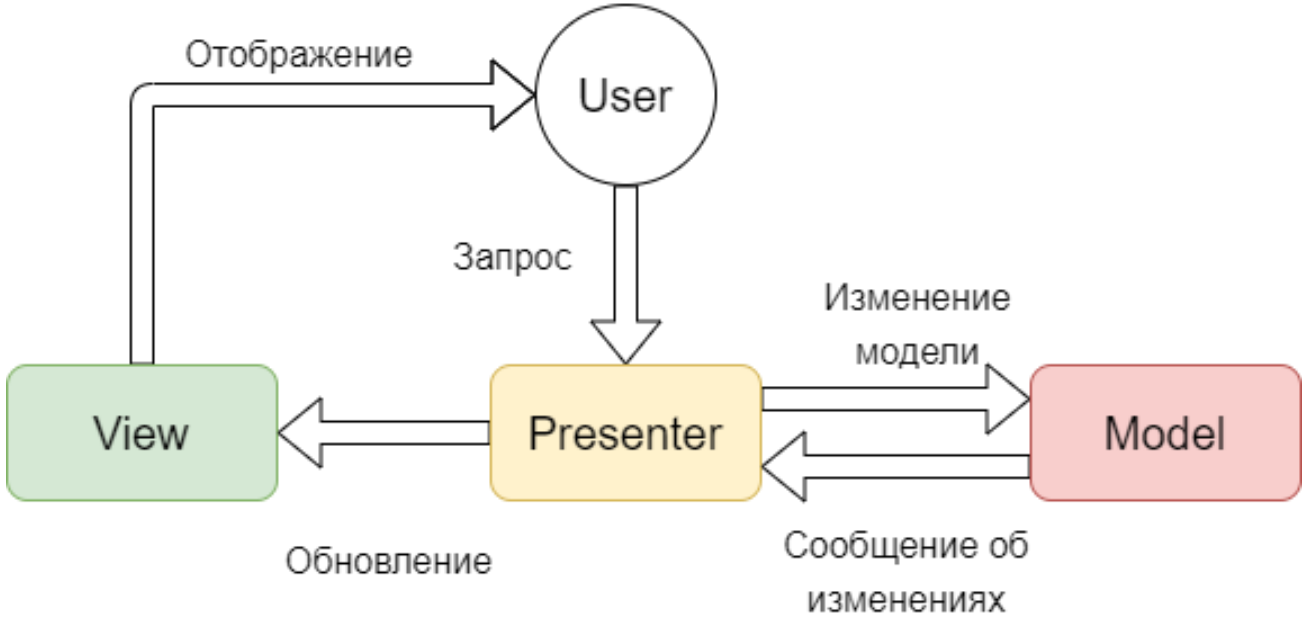
- View и Controller зависит от Model
- Controller обычно не зависит от View

- Model напрямую «общается» с View



Model View Presenter (MVP)

- В отличии от MVC Model не «общается» с View напрямую
- Presenter работает с View через интерфейс
- Presenter проверяет изменения Model



Model View ViewModel (MVVM)

- ViewModel получает только команду
- ViewModel сам получает необходимые данные
- Вся логика отображения данных находится в ViewModel
- ViewModel подписывается на изменение Model