

# Лекция 9: Архитектурные стили

Юрий Литвинов  
y.litvinov@spbu.ru

01.11.2024

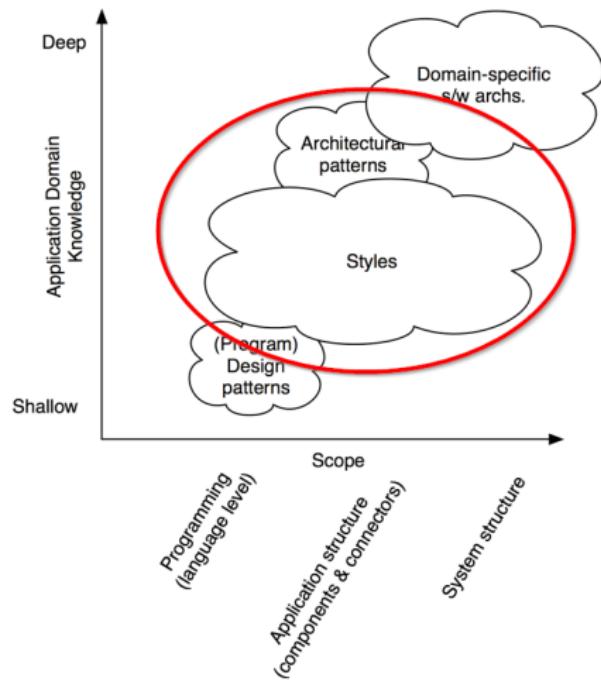
# Архитектурные шаблоны и стили

Архитектурный стиль — набор решений, которые

1. применимы в выбранном контексте разработки,
2. задают ограничения на принимаемые архитектурные решения, специфичные для определённых систем в этом контексте,
3. приводят к желаемым положительным качествам получаемой системы.

Архитектурный шаблон — именованный набор ключевых проектных решений по эффективной организации подсистем, применимых для повторяемых технических задач проектирования в различных контекстах и предметных областях

# Архитектурные шаблоны и стили, классификация



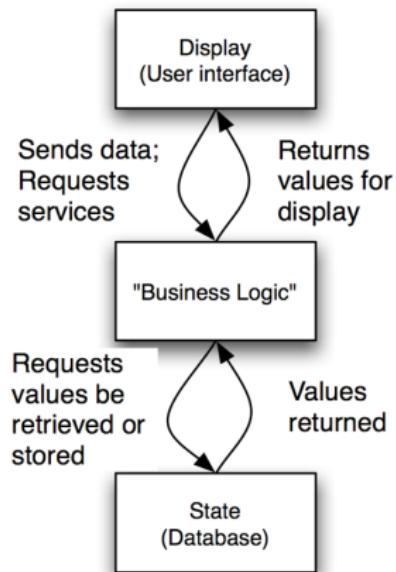
© N. Medvidovic

# Пример: трёхзвенная архитектура

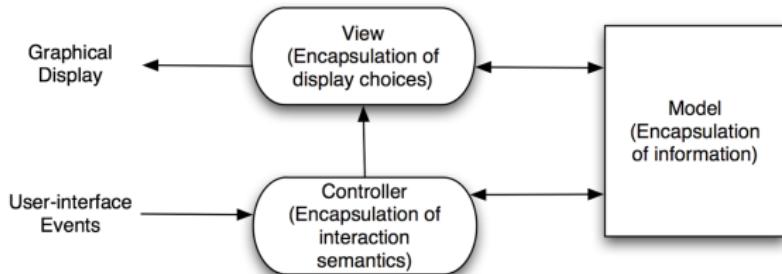
## State-Logic-Display

### Примеры применения

- ▶ Бизнес-приложения
- ▶ Многопользовательские игры
- ▶ Веб-приложения



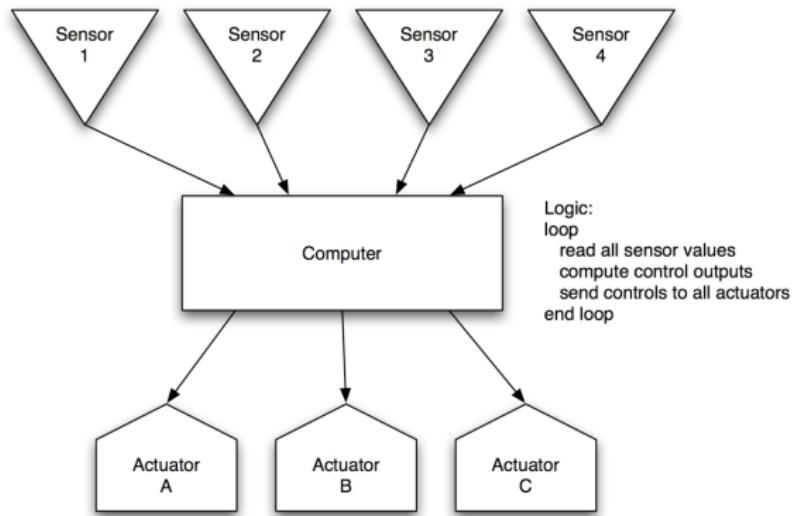
# Пример: Model-View-Controller



© N. Medvidovic

- ▶ Разделяет данные, представление и взаимодействие с пользователем
- ▶ Если в модели что-то меняется, она оповещает представление (представления)
- ▶ Через контроллер проходит всё взаимодействие с пользователем
  - ▶ Естественное место для паттерна “Команда” и Undo/Redo

## Пример: Sense-Compute-Control



© N. Medvidovic

- ▶ Применяется во встроенных системах и робототехнике

# Архитектурные стили

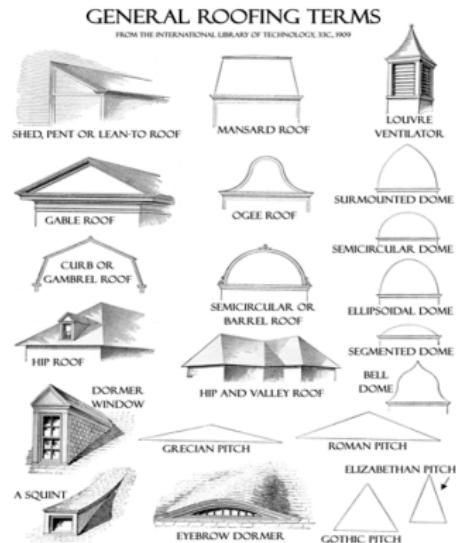
- ▶ Именованная коллекция архитектурных решений
- ▶ Менее узкоспециализированные, чем архитектурные паттерны



© N. Medvidovic

# Архитектурные стили

- ▶ Одна система может включать в себя несколько архитектурных стилей
- ▶ Понятие стиля применимо и к подсистемам



© N. Medvidovic

# Преимущества использования стилей

- ▶ Переиспользование архитектуры
  - ▶ Для новых задач можно применять хорошо известные и изученные решения
- ▶ Переиспользование кода
  - ▶ Часто у стилей бывают неизменяемые части, которые можно один раз реализовать
- ▶ Упрощение общения и понимания системы
- ▶ Упрощение интеграции приложений
- ▶ Специфичные для стиля методы анализа
  - ▶ Возможны благодаря ограничениям на структуру системы
- ▶ Специфичные для стиля методы визуализации

# “Сырой” объектно-ориентированный стиль

- ▶ Компоненты — объекты
- ▶ Соединители — сообщения и вызовы методов
- ▶ Инварианты:
  - ▶ Объекты отвечают за своё внутреннее состояние
  - ▶ Реализация скрыта от других объектов
- ▶ Преимущества:
  - ▶ Декомпозиция системы в набор взаимодействующих агентов
  - ▶ Внутреннее представление объектов можно менять независимо
  - ▶ Близко к предметной области
- ▶ Недостатки:
  - ▶ Побочные эффекты при вызове методов
  - ▶ Объекты вынуждены знать обо всех, от кого зависят

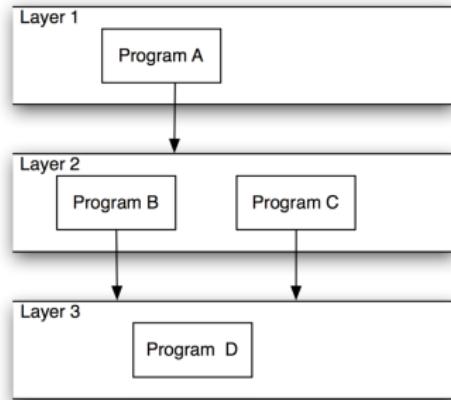
# Слоистый стиль

Layered style

- ▶ Иерархическая организация системы
  - ▶ “Многоуровневый клиент-сервер”
  - ▶ Каждый слой предоставляет интерфейс для использования слоями выше
- ▶ Каждый слой работает как:
  - ▶ Сервер — предоставляет функциональность слоям выше
  - ▶ Клиент — использует функциональность слоёв ниже
- ▶ Соединители — протоколы взаимодействия слоёв
- ▶ Пример — операционные системы, сетевые стеки протоколов

# Слоистый стиль, подробности

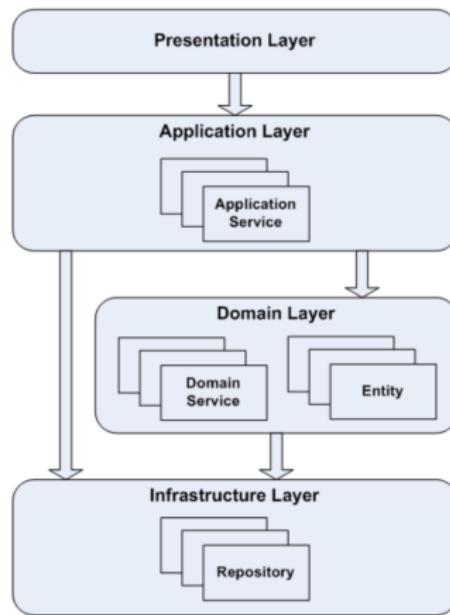
- ▶ Преимущества:
  - ▶ Повышение уровня абстракции
  - ▶ Лёгкость в расширении
  - ▶ Изменения в каждом уровне затрагивают максимум два соседних
  - ▶ Возможны разные реализации уровня, если они удовлетворяют интерфейсу
- ▶ Недостатки:
  - ▶ Не всегда применим
  - ▶ Проблемы с производительностью



© N. Medvidovic

# Пример

## Стандартные слои в Domain-Driven Design



© [http://uniknow.github.io/AgileDev/site/0.1.8-SNAPSHOT/parent/ddd/core/layered\\_architecture.html](http://uniknow.github.io/AgileDev/site/0.1.8-SNAPSHOT/parent/ddd/core/layered_architecture.html)

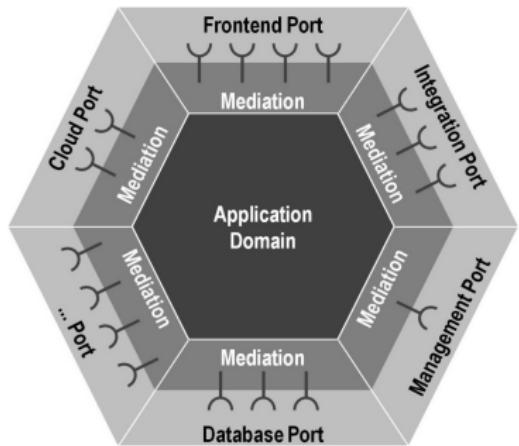
# “Клиент-сервер”

- ▶ Компоненты — клиенты и серверы
- ▶ Серверы не знают ничего о клиентах, даже их количество
- ▶ Клиенты знают только про сервера и не могут общаться друг с другом
- ▶ Соединители — сетевые протоколы

# Гексагональная архитектура

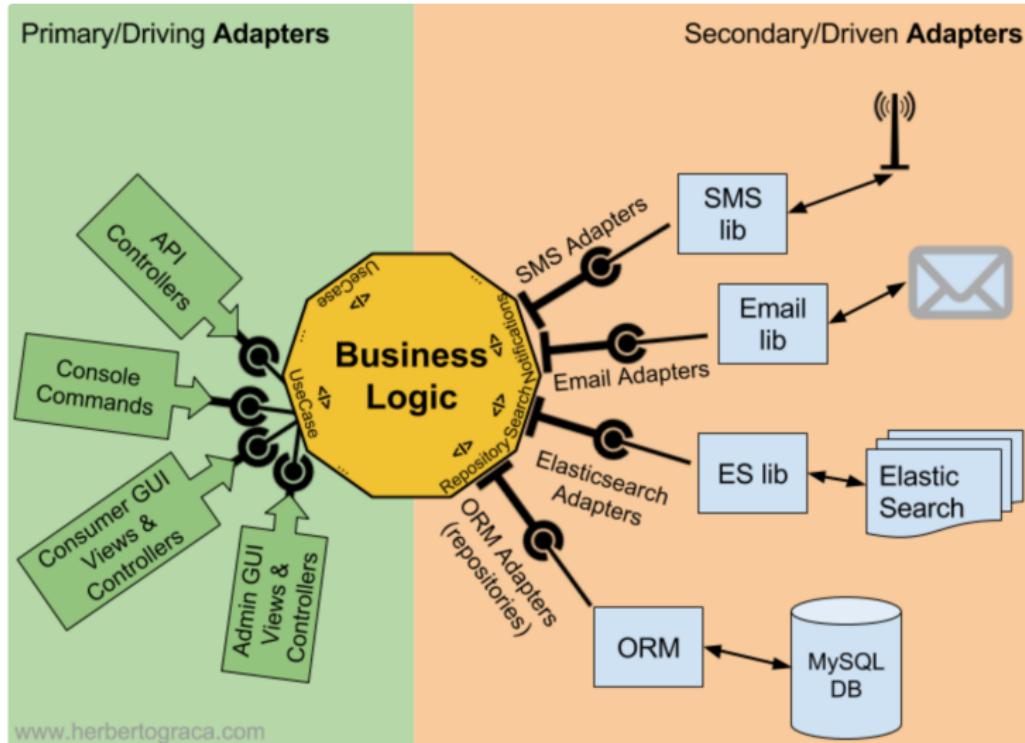
“Порты и адаптеры”

- ▶ Другая точка зрения на уровни: самый нижний — уровень предметной области
- ▶ Всё остальное поставляется ему как внешние зависимости
- ▶ Активно используется Dependency Inversion
- ▶ Порт — по сути, интерфейс, предоставляемый или потребляемый
- ▶ Адаптер — паттерн “Адаптер” для “подгонки” интерфейсов



© B Butzin et al, Microservices Approach for the Internet of Things

# Подробности



© <https://herbertograca.com/2017/09/14/ports-adapters-architecture/>

# Плюсы и минусы

Плюсы:

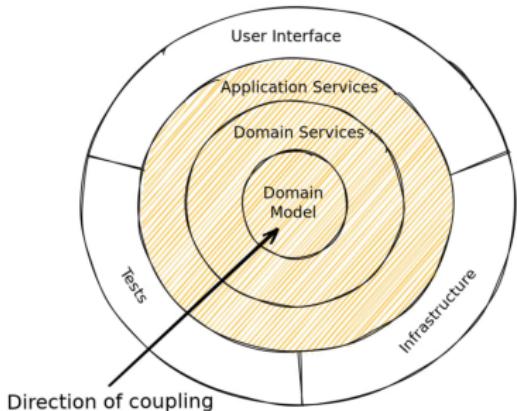
- ▶ Изоляция механизмов доставки
- ▶ Изоляция вспомогательных механизмов
- ▶ Лёгкость тестирования, моки
- ▶ Чистая бизнес-логика и модель предметной области
  - ▶ Максимальная простота
  - ▶ Возможность валидации и конвертирования данных

Минусы:

- ▶ Довольно тяжеловесна
- ▶ Непонятно, что делать с фреймворками
- ▶ Не очень подробна

# Луковая архитектура

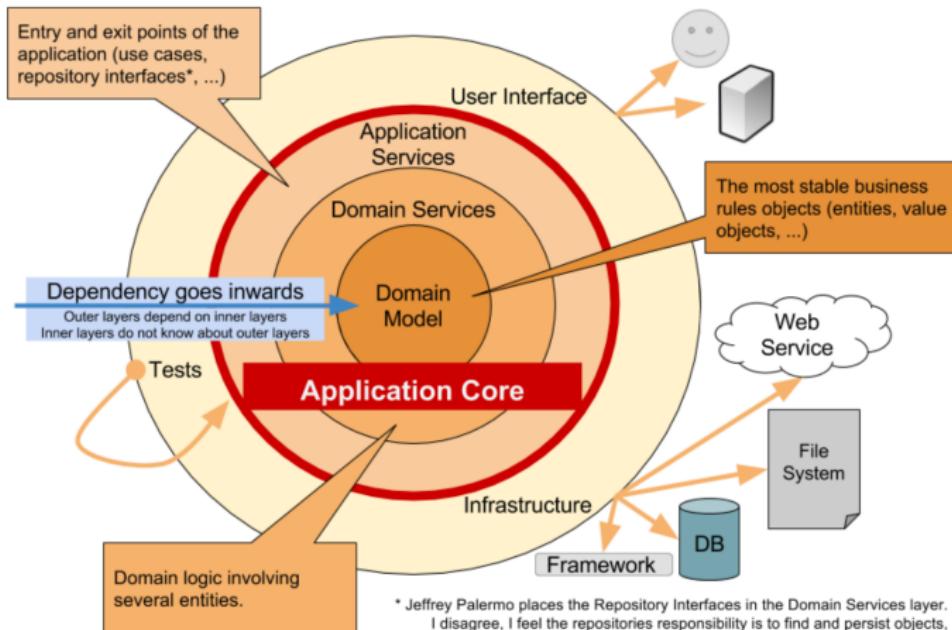
- ▶ Дальнейшее развитие гексагональной — определяет внутреннюю структуру ядра
- ▶ Внутренние слои не знают о внешних, доменная модель вообще ни о ком не знает
- ▶ Внутренние слои определяют интерфейсы, внешние их реализуют
- ▶ Уровневость нестрогая — слой может использовать все слои под ним



© <https://dev.to/barrymcauley/onion-architecture>

3fgl

# Подробности

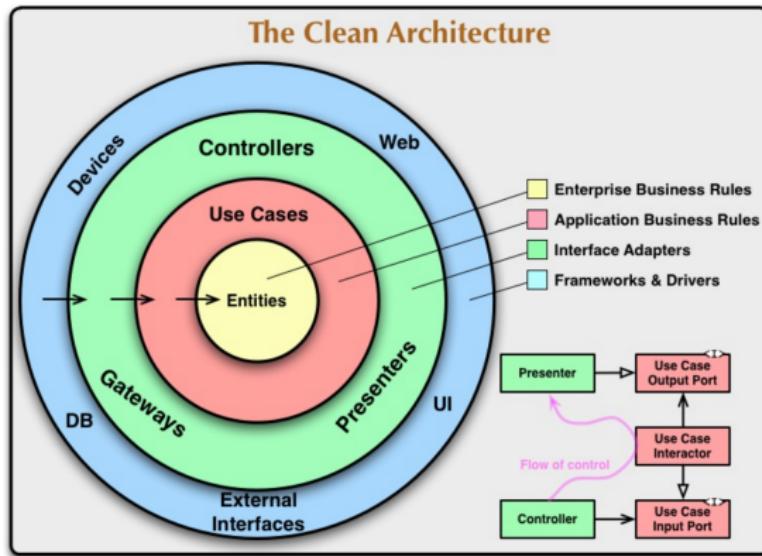


[www.herbertograca.com](http://www.herbertograca.com)

\* Jeffrey Palermo places the Repository Interfaces in the Domain Services layer.  
I disagree, I feel the repositories responsibility is to find and persist objects,  
so their interface is an entry/exit port, just like the use case interfaces,  
and therefore should not be known by our domain layers.

© <https://herbertograca.com/2017/09/21/onion-architecture/>

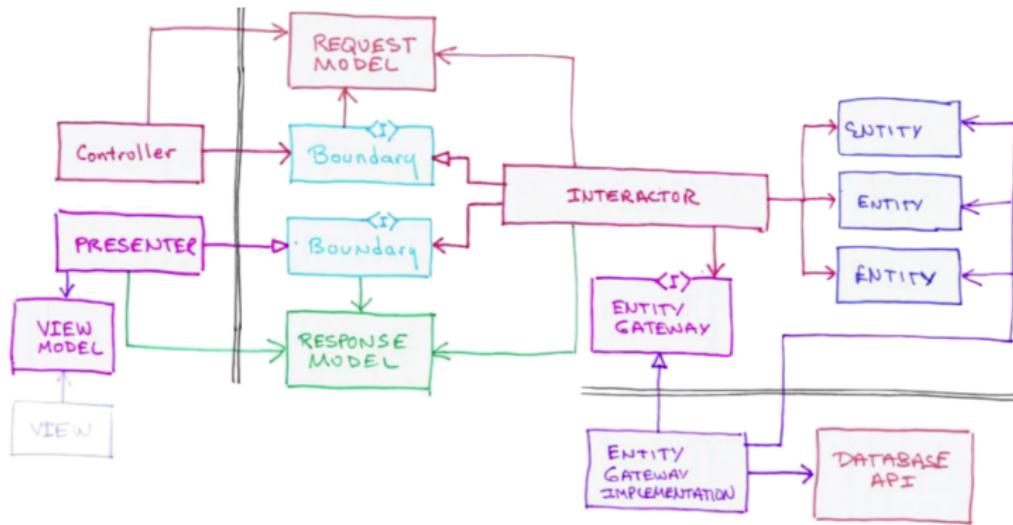
# Чистая архитектура



© <https://herbertograca.com/2017/09/28/clean-architecture-standing-on-the-shoulders-of-giants/>

- ▶ Дальнейшее развитие луковой — определяет поток управления

# Чистая архитектура, обработка запроса

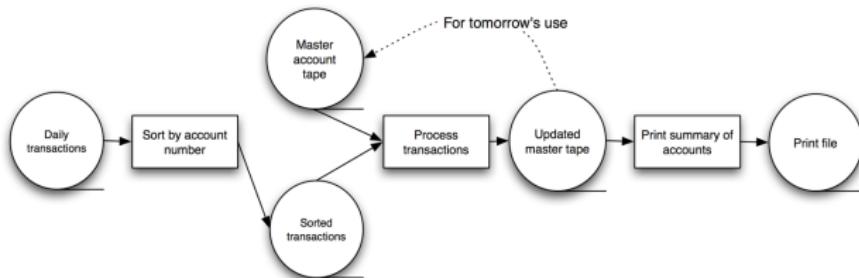


© <https://herbertograca.com/2017/09/28/clean-architecture-standing-on-the-shoulders-of-giants/>

# Пакетная обработка

- ▶ Система строится как набор отдельных программ, выполняющихся последовательно
- ▶ Данные стандартным для ОС способом передаются от программы к программе
  - ▶ Pipes, named pipes, файлы
- ▶ Данные — в явном виде всё, необходимое для работы

Типичен для финансовых систем глубокой древности (“Прадедушка стилей”)



© N. Medvidovic

# Каналы и фильтры

## Pipes and filters

- ▶ Компоненты — это фильтры, преобразующие данные из входных каналов в данные в выходных каналах
- ▶ Соединители — каналы
- ▶ Инварианты:
  - ▶ Фильтры независимы (не имеют разделяемого состояния)
  - ▶ Фильтры не знают о фильтрах до или после них
- ▶ Вариации:
  - ▶ Конвейеры — линейные последовательности фильтров
  - ▶ Ограниченные каналы — где канал это очередь с ограниченным количеством элементов
  - ▶ Типизированные каналы — где каналы отличаются по типу передаваемых данных

# Каналы и фильтры, подробности

- ▶ Преимущества:
  - ▶ Поведение системы — это просто последовательное применение поведений компонентов
  - ▶ Легко добавлять, заменять и переиспользовать фильтры
    - ▶ Любые два фильтра можно использовать вместе
  - ▶ Широкие возможности для анализа
    - ▶ Пропускная способность, задержки, deadlock-и
  - ▶ Широкие возможности для параллелизма
- ▶ Недостатки:
  - ▶ Последовательное исполнение
  - ▶ Проблемы с интерактивными приложениями
  - ▶ Пропускная способность определяется самым “узким” элементом

# Blackboard

- ▶ Два типа компонентов:
  - ▶ Центральная структура данных — та самая “Blackboard”
  - ▶ Компоненты, работающие с blackboard
- ▶ Инварианты:
  - ▶ Управление системой осуществляется только через состояние доски
  - ▶ Компоненты не знают друг о друге и не имеют своего состояния
- ▶ Системы переписывающих правил — некоторые задачи ИИ, трансляторы, графовые грамматики, алгорифмы Маркова

# Событийный стиль

- ▶ Оповещение о событии вместо явного вызова метода
  - ▶ “Слушатели” могут подписаться на событие
  - ▶ Система при наступлении события сама вызывает все зарегистрированные методы слушателей
- ▶ Компоненты имеют два вида интерфейсов — методы и события
- ▶ Два типа соединителей:
  - ▶ Явный вызов метода
  - ▶ Неявный вызов по наступлению события
- ▶ Инварианты:
  - ▶ Те, кто производит события, не знают, кто и как на них отреагирует
  - ▶ Не делается никаких предположений о том, как событие будет обработано и будет ли вообще

# Событийный стиль, преимущества и недостатки

- ▶ Преимущества:
  - ▶ Переиспользование компонентов
    - ▶ Очень низкая связность между компонентами
  - ▶ Лёгкость в конфигурировании системы
    - ▶ Как во время компиляции, так и во время выполнения
- ▶ Недостатки:
  - ▶ Зачастую неинтуитивная структура системы
  - ▶ Компоненты не управляют последовательностью вычислений
  - ▶ Непонятно, кто отреагирует на запрос и в каком порядке придут ответы
  - ▶ Тяжело отлаживаться
  - ▶ Гонки даже в однопоточном приложении

# Издатель-подписчик

## Publish-subscribe

- ▶ Подписчики регистрируются, чтобы получать нужные им сообщения или данные. Издатели публикуют сообщения, синхронно или асинхронно.
- ▶ Компоненты: издатели, подписчики, “маршрутизаторы”
- ▶ Соединители: как правило, сетевые протоколы, часто механизм наподобие паттерна “Наблюдатель”
- ▶ Данные: подписки, нотификации, публикуемая информация
- ▶ Топология: подписчики подключаются к издателям напрямую, либо через посредников
- ▶ Преимущества: очень низкая связность между компонентами, при этом высокая эффективность распределения информации

# Событийная шина

- ▶ Независимые компоненты посылают и принимают события, передаваемые по шинам
- ▶ Компоненты: независимые генераторы или потребители событий
- ▶ Соединители: шины событий (хотя бы одна)
- ▶ Данные: события и связанные с ними данные, посылаемые пошине
- ▶ Топология: компоненты общаются только с шинами событий, не друг с другом
- ▶ Варианты: push- и pull-режимы работы с шиной
- ▶ Преимущества: лёгкость масштабирования и добавления новой функциональности, эффективно для распределённых приложений
- ▶ Event Sourcing

# Peer-to-peer

- ▶ Состояние и поведение распределены между компонентами, которые могут выступать как клиенты и как серверы
- ▶ Компоненты: имеют своё состояние и свой поток управления
- ▶ Соединители: как правило, сетевые протоколы
- ▶ Элементы данных: сетевые сообщения
- ▶ Топология: сеть (возможно, с избыточными связями между компонентами), может динамически меняться
- ▶ Преимущества:
  - ▶ Хорош для распределённых вычислений
  - ▶ Устойчив к отказам
  - ▶ Если протокол взаимодействия позволяет, легко масштабируется