Архитектура приложения

Литература

Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2021. — 352 с.: ил.

ASP.NET Core Application Architecture – электронный ресурс.

Режим доступа:

https://dotnet.microsoft.com/en-us/learn/aspnet/architecture

Дата доступа 09.2022

Введение

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Введение

Если бы строители строили здания так, как программисты пишут программы, то первый попавшийся дятел уничтожил бы цивилизацию.

(Дже́ральд Ва́йнберг — американский учёный, автор и преподаватель психологии и антропологии разработки программного обеспечения)

Введение

Цель архитектуры программного обеспечения — уменьшить человеческие трудозатраты на создание и сопровождение системы. (Роберт Мартин)

Введение

Что такое архитектура ПО?

- это совокупность важнейших решений об организации программной системы (википедия)
- отображает организацию или структуру системы и дает объяснение того, как она ведет себя. (https://datascience.eu/ru/компьютерное-зрение/архитектура-программного-обеспечени/)
- это структура программы или вычислительной системы, которая включает программные компоненты, видимые снаружи свойства этих компонентов, а также отношения между ними (https://dic.academic.ru/dic.nsf/ruwiki/146913)

Введение

Как видим, четкого определения нет, но можно описать критерии, которые определяют программное средство с «правильной» архитектурой:

- **Эффективность системы**.
- **Гибкость системы.**
- **Расширяемость системы**.
- **Масштабируемость процесса разработки**.
- **Тестируемость.**
- **Возможность повторного использования**.
- **Хорошо структурированный, читаемый и понятный код.**
- **Сопровождаемость**.

Введение

Когда речь идет о построении архитектуры программы, создании ее структуры, под этим, главным образом, подразумевается **декомпозиция** программы на подсистемы (функциональные модули, сервисы, слои, подпрограммы) и организация их **взаимодействия** друг с другом, с пользователем программы, с другими программами и внешними устройствами.

Введение

При декомпозиции особое внимание уделяется минимизации связей (зависимостей) между отдельными компонентами.

Это позволит вести независимую разработку компонентов.

Кроме того, модификация (рефакторинг) одного компонента не повлечет модификации связанных с ним других компонентов (либо модификация будет минимальной), и тем более не потребует модификации всего приложения.

Принципы SOLID

Принципы SOLID

SOLID — это аббревиатура пяти основных принципов проектирования в объектно-ориентированном программировании

- **S**ingle responsibility — принцип единственной ответственности
- **O**pen-closed — принцип открытости / закрытости
- **L**iskov substitution — принцип подстановки Барбары Лисков
- **I**nterface segregation — принцип разделения интерфейса
- **D**ependency inversion — принцип инверсии зависимостей

Принципы SOLID

Single Responsibility Principle (SRP) — принцип единственной ответственности

Каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс, а все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

Принципы SOLID

Single Responsibility Principle (SRP) — принцип единственной ответственности (по Р. Мартину)

Модуль должен иметь одну и только одну причину для изменения.

Или

Модуль должен отвечать за одного и только за одного актора.

(актор – actor – объект/пользователь модуля)

Принципы SOLID

Single Responsibility Principle (SRP) — принцип единственной ответственности (по Р. Мартину)

Модуль должен иметь одну и только одну причину для изменения.

Или

Модуль должен отвечать за одного и только за одного актора.

(актор – actor – объект/пользователь модуля)

Принципы SOLID

Open-closed Principle (OCP) — принцип открытости / закрытости

Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

Принципы SOLID

Open-closed Principle (OCP) — принцип открытости / закрытости

Цель OCP — 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.

Принципы SOLID

Liskov substitution principle (LSP) — принцип подстановки Барбары Лисков

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

Принципы SOLID

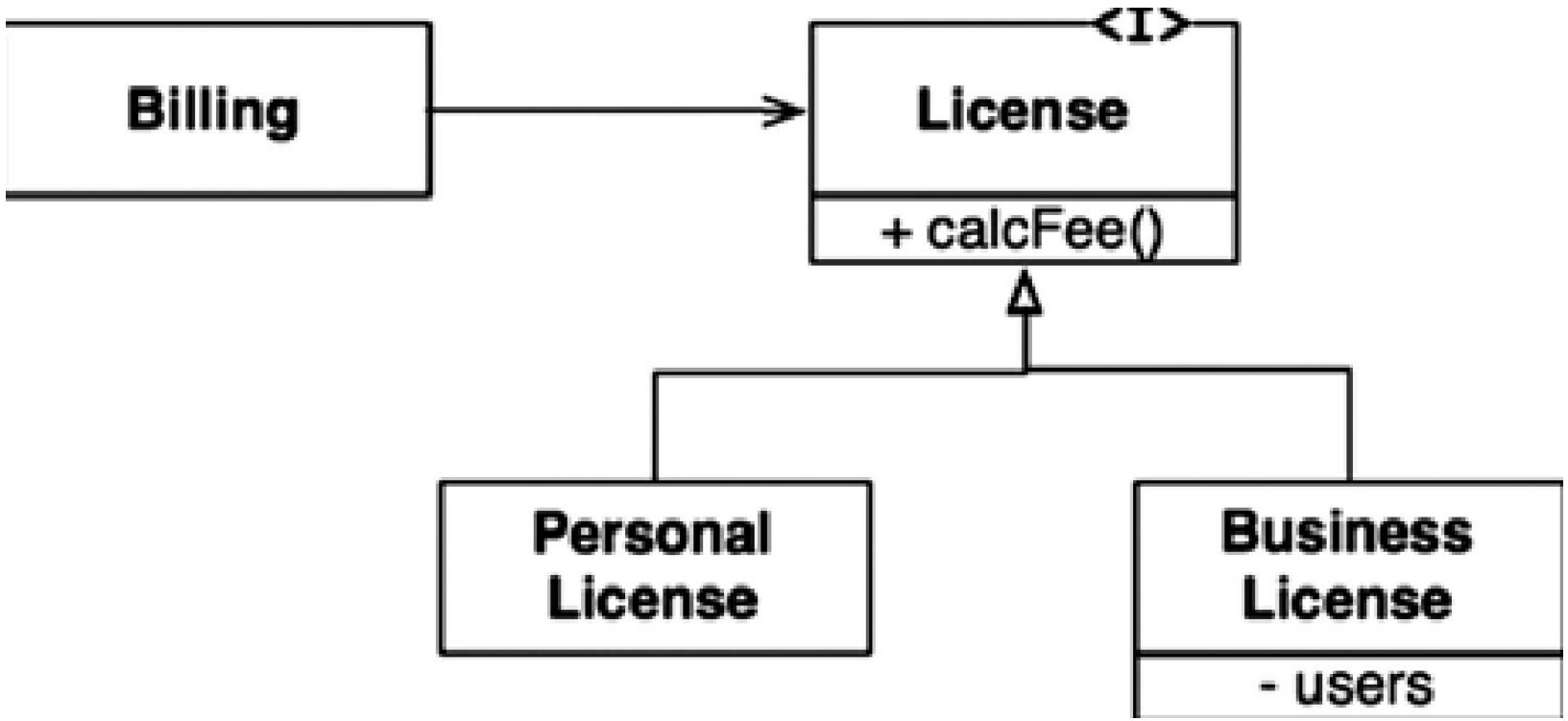
Liskov substitution principle (LSP) — принцип подстановки Барбары Лисков

Оригинальное определение Барбары Лисков:

«В том случае, если q(x) — свойство, верное по отношению к объектам x некого типа T, то свойство q(y) тоже будет верным относительно ряда объектов y, которые относятся к типу S, при этом S — подтип некого типа T».

Принципы SOLID

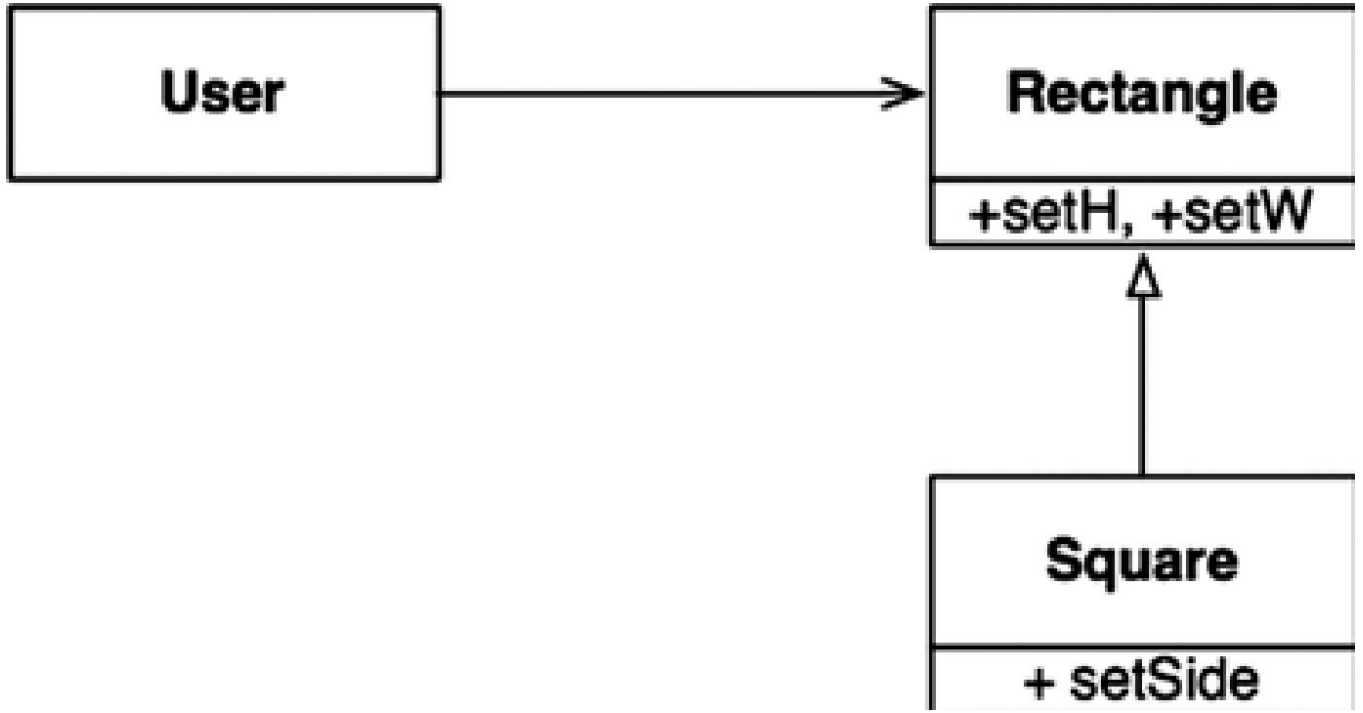
Структура соответствует принципу LSP: поведение Billing не зависит от конкретного типа license



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Принципы SOLID

Нарушение принципа LSP: поведение User зависит от конкретного типа, т.к. в прямоугольнике можно изменять стороны независимо, а в квадрате - нет



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

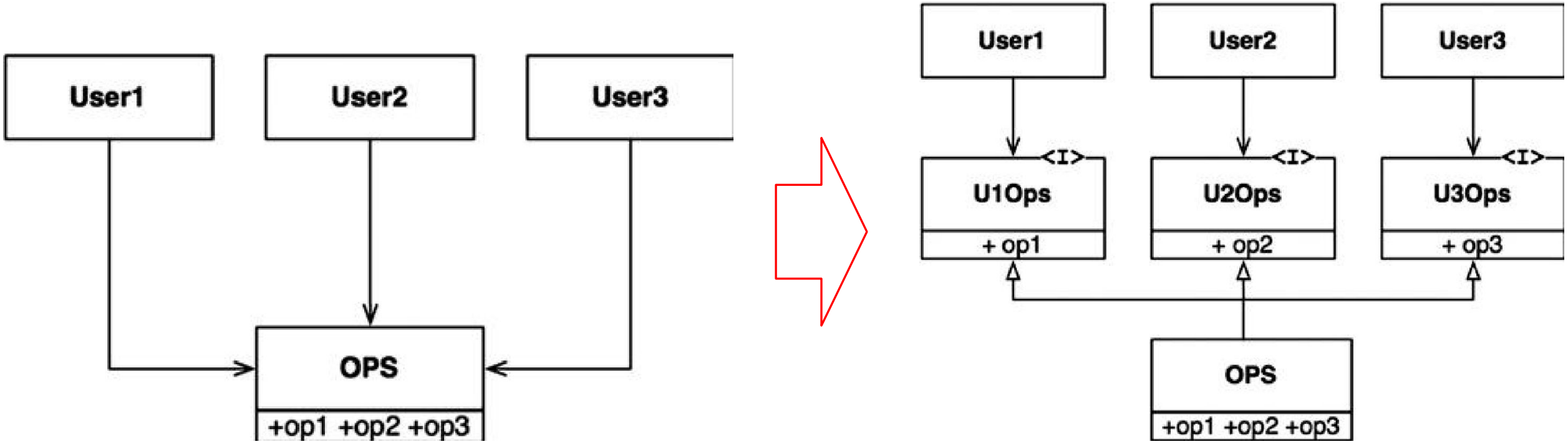
Принципы SOLID

Interface segregation principle (ISP) — принцип разделения интерфейса

Клиенты не должны зависеть от методов, которые они не используют. То есть если какой-то метод интерфейса не используется клиентом, то изменения этого метода не должны приводить к необходимости внесения изменений в клиентский код.

Принципы SOLID

Interface segregation principle (ISP) — принцип разделения интерфейса



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Принципы SOLID

Dependency inversion — принцип инверсии зависимостей

Модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

Принципы SOLID

Dependency inversion — принцип инверсии зависимостей

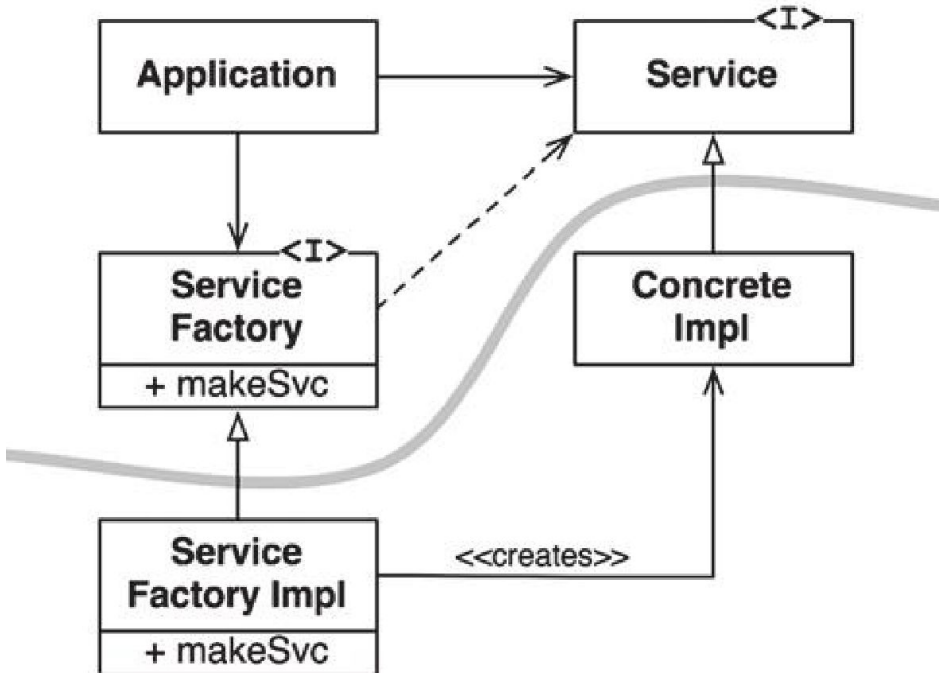
Каждое изменение абстрактного интерфейса вызывает изменение его конкретной реализации. Изменение конкретной реализации, напротив, невсегда сопровождается изменениями и даже обычно не требует изменений в соответствующих интерфейсах.

То есть интерфейсы менее изменчивы, чем реализации.

Принципы SOLID

Dependency inversion principle (DIP) — принцип инверсии зависимостей

Для реализации DIP часто используют абстрактную фабрику



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Программные компоненты

Программные компоненты

Возвращаясь к декомпозиции программы, возникает вопрос, по какому принципу следует объединять код в отдельные компоненты.

Принцип эквивалентности повторного использования и выпусков

REP: Reuse/Release Equivalence Principle — принцип эквивалентности повторного использования и выпусков;

- Классы и модули, составляющие компонент, должны принадлежать **связной группе**.
- Классы и модули, объединяемые в компонент, должны ***выпускаться*** **вместе**.

Под выпуском понимается публикация новой обновленной версии

Принцип согласованного изменения

CCP: Common Closure Principle — принцип согласованного изменения

В один компонент должны включаться классы, изменяющиеся **по одним причинам** и в **одно время**. В разные компоненты должны включаться классы, изменяющиеся в разное время и по разным причинам.

Принцип совместного повторного использования

CRP: Common Reuse Principle — принцип совместного повторного использования.

В компонент должны включаться классы и модули, используемые совместно

Классы, включаемые в компонент, должны быть неотделимы друг от друга — чтобы нельзя было зависеть от одних и не зависеть от других.

Принцип совместного повторного использования

Принцип совместного повторного использования является обобщенной версией принципа разделения интерфейсов (ISP). Принцип ISP советует не создавать зависимостей от классов, методы которых не используются.

Диаграмма противоречий для определения связности компонентов

Три принципа связности компонентов вступают в противоречие друг с другом.

Принципы эквивалентности повторного использования (REP) и согласованного изменения (CCP) являются *включительными*: оба стремятся сделать компоненты как можно крупнее.

Принцип повторного использования (CRP) — *исключительный*,

стремящийся сделать компоненты как можно мельче.

Диаграмма противоречий для определения связности компонентов

Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

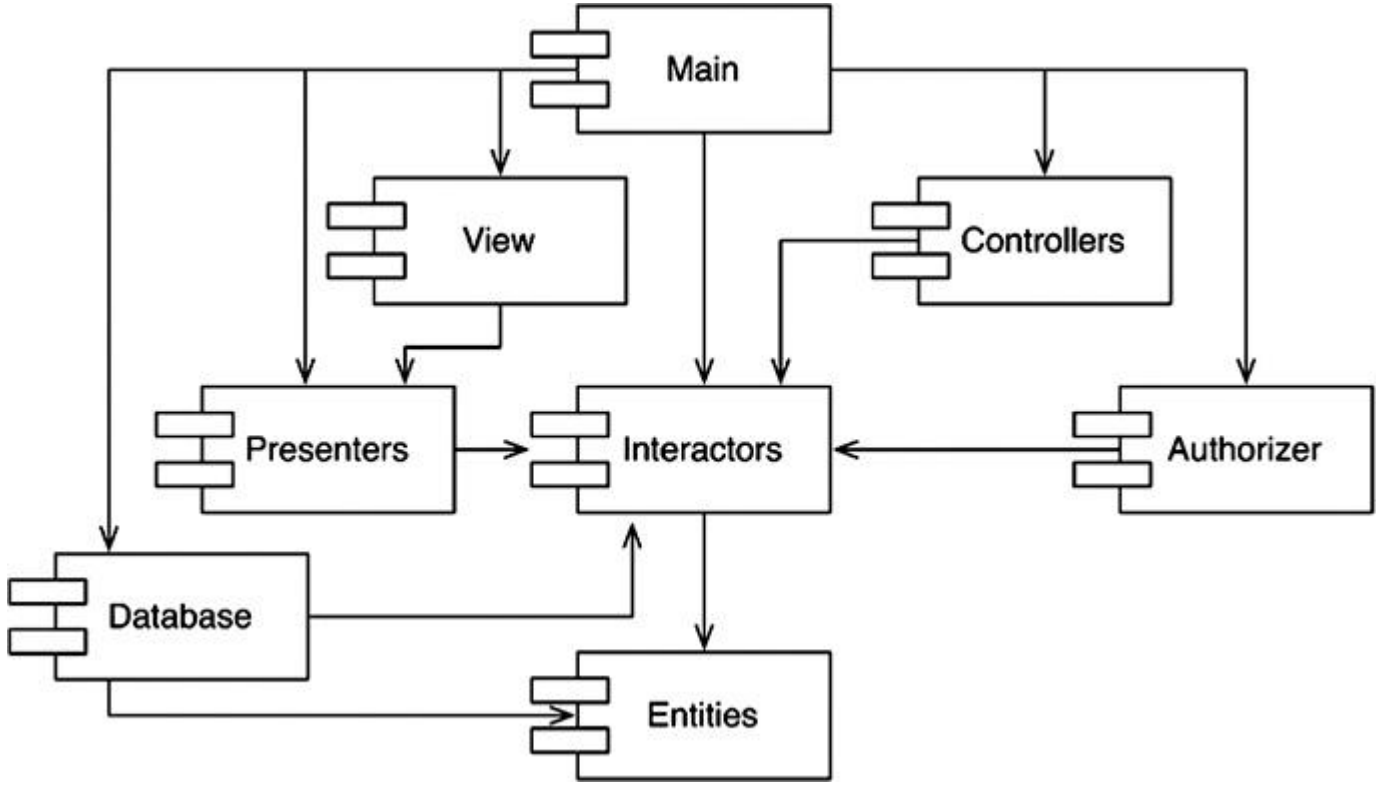
Взаимодействие компонентов

Когда один компонент использует другой компонент, то первый компонент зависит от второго:

В правильно спроектированнной структуре зависимостей компонентов *не должно быть циклических зависимостей*.

Взаимодействие компонентов

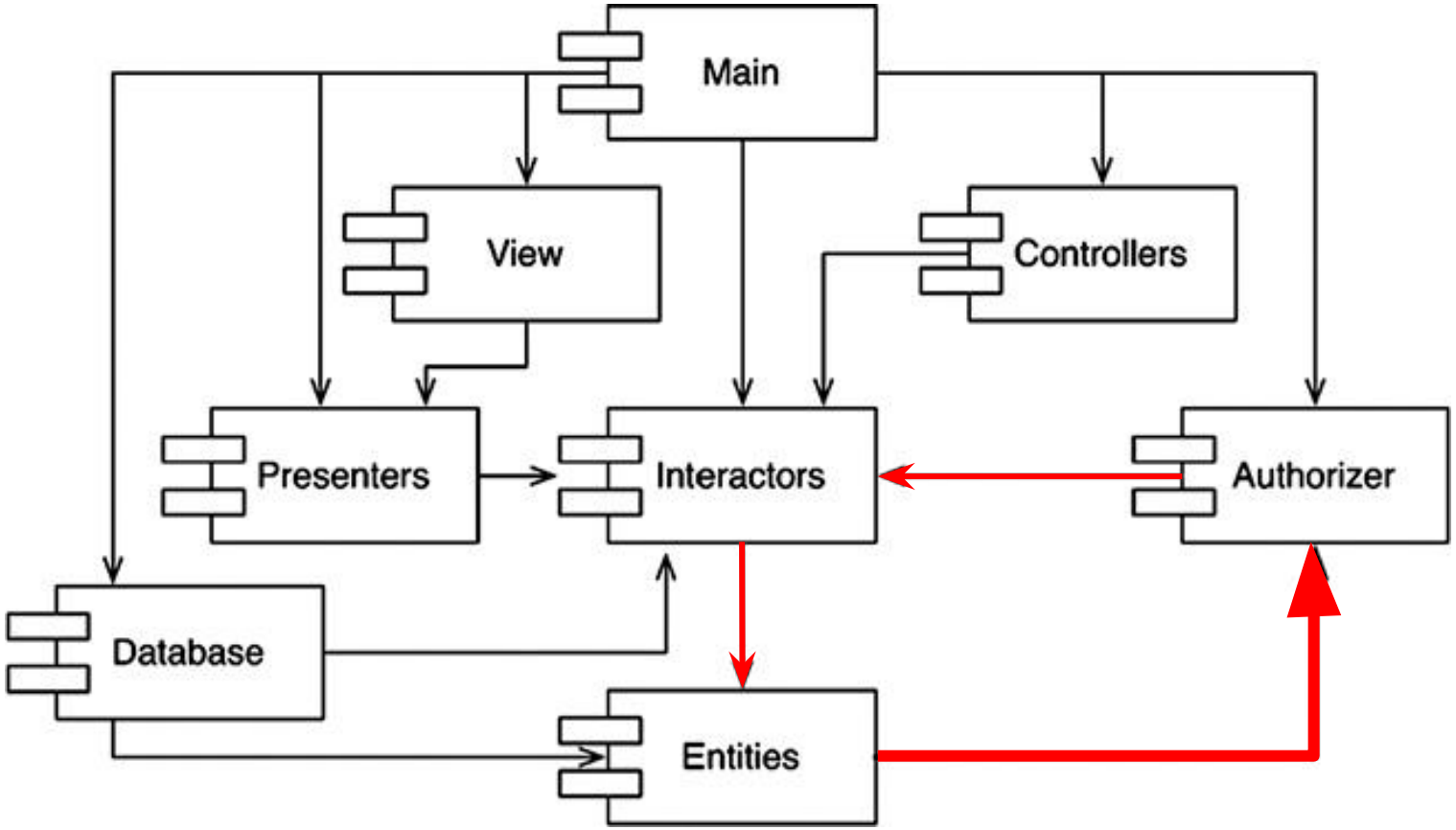
Диаграмма компонентов *без циклических зависимостей*.



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

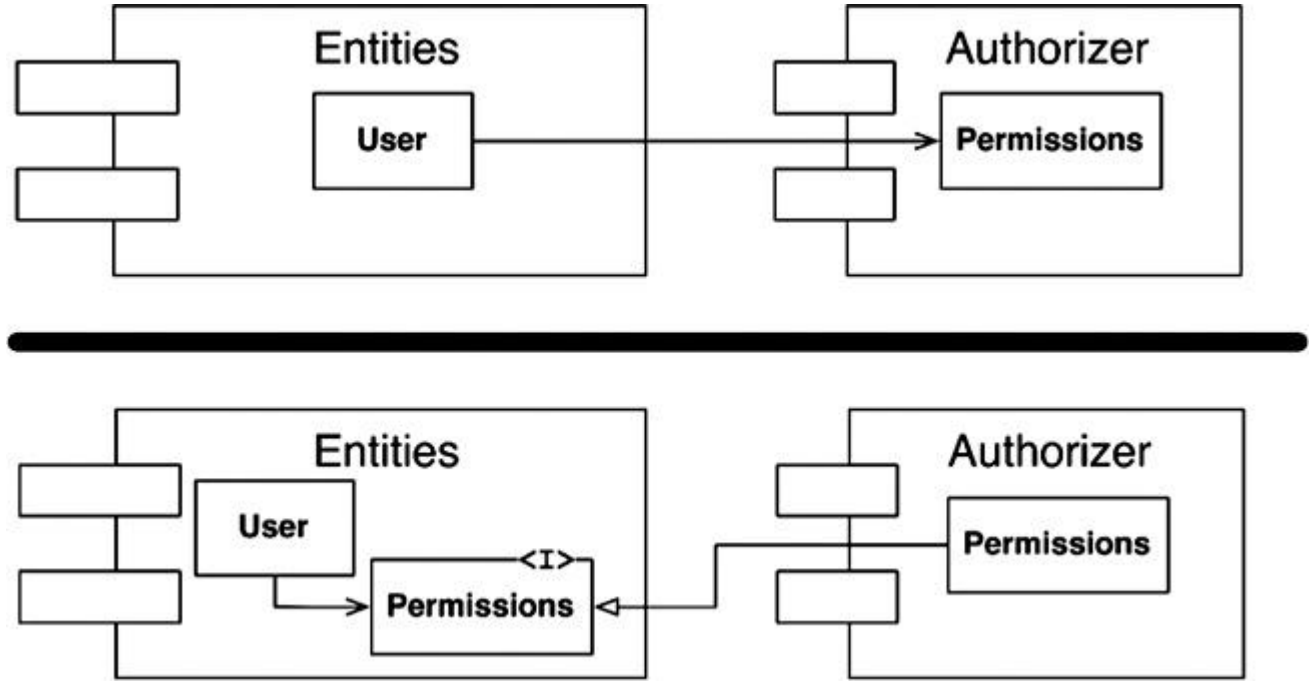
Пример неправильной организации зависимостей



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

Пример устранения циклической ссылки



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

Принцип устойчивых зависимостей (SDP)

Зависимости должны быть направлены в сторону устойчивости.

Компоненты, с большим трудом поддающиеся изменению, не должны зависеть от любых изменчивых компонентов. Иначе изменчивый компонент тоже трудно будет изменять.

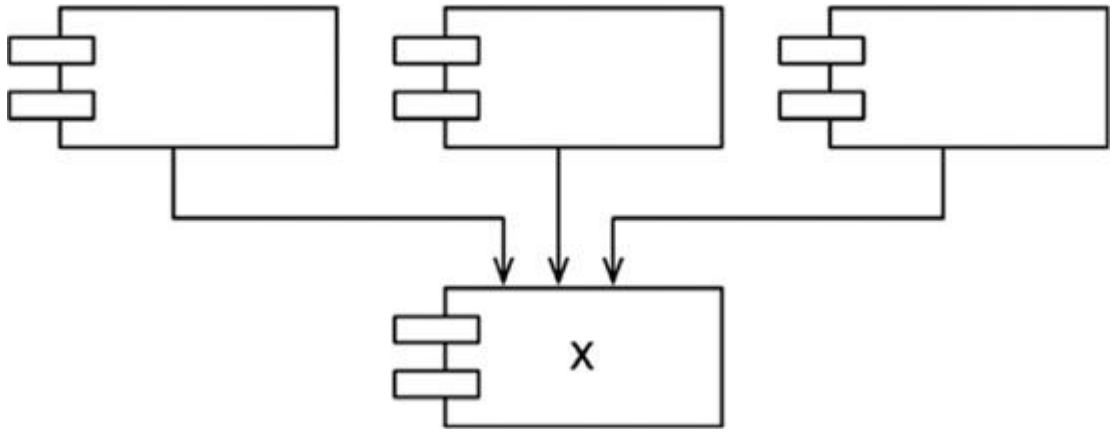
Взаимодействие компонентов

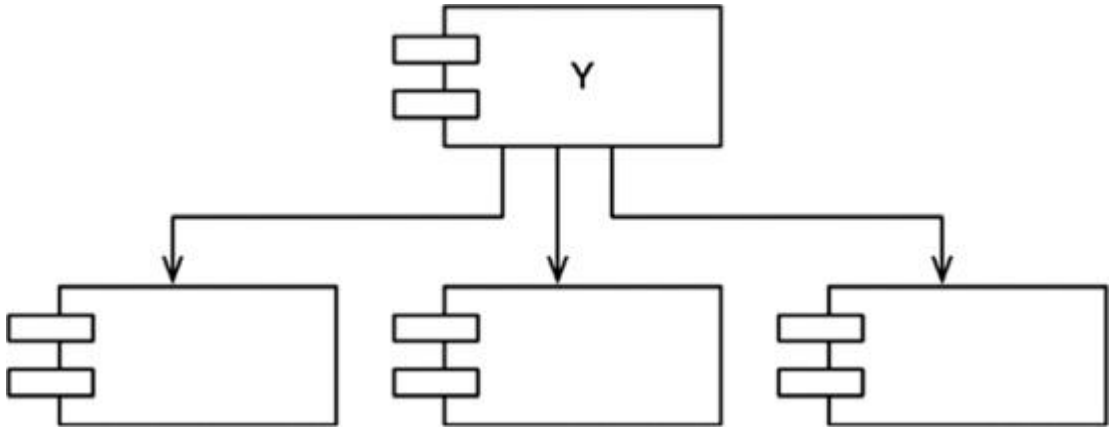
Принцип устойчивых зависимостей

Компонент с множеством входящих зависимостей очень устойчив, потому что согласование изменений со всеми зависящими компонентами требует значительных усилий.

Взаимодействие компонентов

Принцип устойчивых зависимостей





X – очень устойчивый компонент, **Y** – очень неустойчивый компонент

Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

Метрики устойчивости

Fan-in (*число входов*): количество входящих зависимостей. Эта метрика определяет количество классов вне данного компонента, которые зависят от классов внутри компонента.

Fan-out (*число выходов*): количество исходящих зависимостей. Эта метрика определяет количество классов внутри данного компонента, зависящих от классов за его пределами.

неустойчивость:

***I* = *Fan-out* ÷ (*Fan-in* + *Fan-out*).**

Значение этой метрики изменяется в диапазоне [0, 1].

I = **0** соответствует **максимальной устойчивости** компонента,

I = **1** — **максимальной неустойчивости**.

Взаимодействие компонентов

Если все компоненты в системе будут иметь максимальную устойчивость, такую систему невозможно будет изменить.

Структура компонентов должна проектироваться так, чтобы в ней имелись и устойчивые, и неустойчивые компоненты.

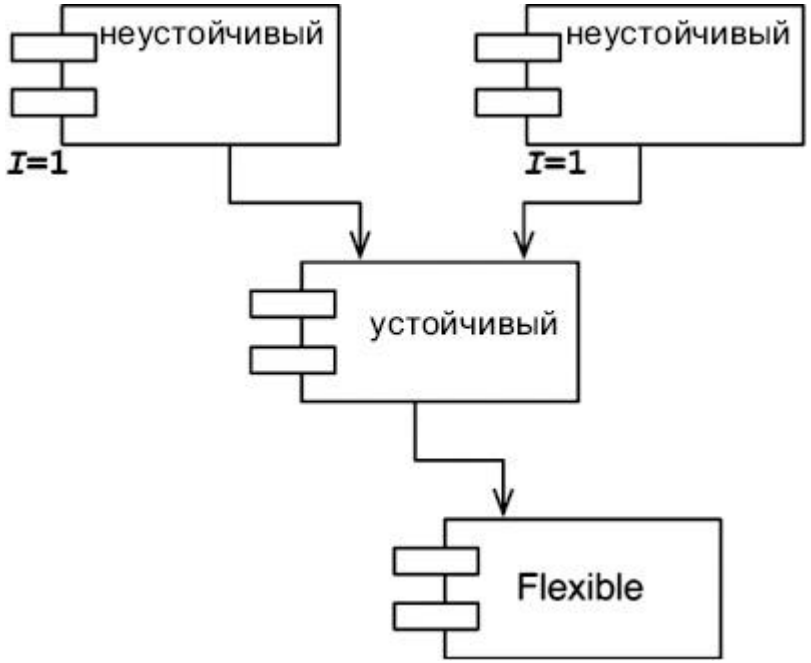
Взаимодействие компонентов

Изменяемые компоненты располагаются на диаграмме вверху и зависят от устойчивого компонента внизу.

Размещение неустойчивых компонентов в верхней части диаграммы — общепринятое и очень удобное соглашение, потому что любые стрелки, направленные *вверх*, ясно покажут нарушение принципа устойчивых зависимостей

Взаимодействие компонентов

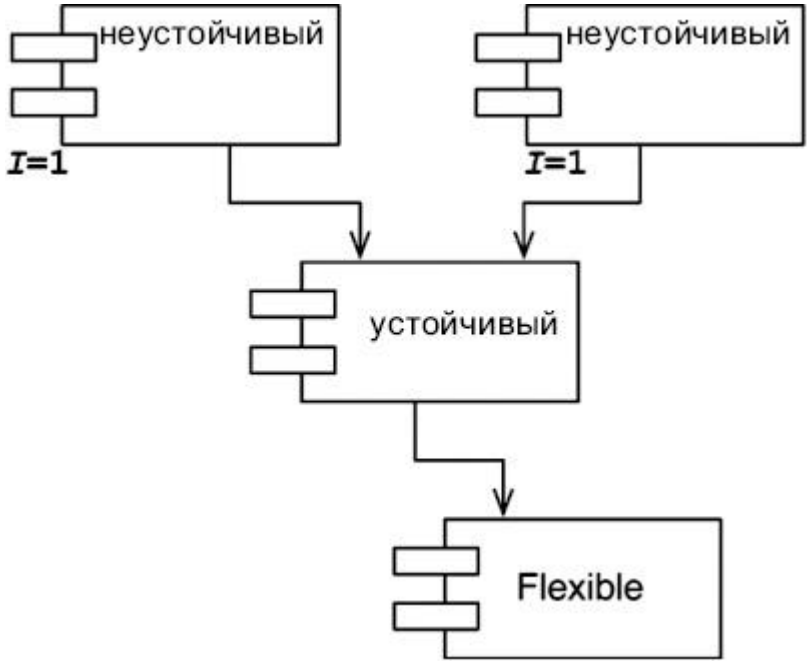
Нарушение принципа SDP



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

Нарушение принципа
SDP



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Взаимодействие компонентов

Принцип устойчивости абстракций

Устойчивость компонента пропорциональна его абстрактности.

Если высокоуровневые правила поместить в устойчивые компоненты, это усложнит изменение исходного кода, реализующего их. Чтобы компонент с максимальной устойчивостью ($I = 0$) был гибким настолько, чтобы он сохранял устойчивость при изменениях, используется принцип открытости/закрытости (OCP). Этому принципу соответствуют, например, *Абстрактные классы*.

Взаимодействие компонентов

Принцип устойчивости абстракций

Мерой абстрактности компонента служит метрика *A*

Nc: число классов в компоненте.

Na: число абстрактных классов и интерфейсов в компоненте.

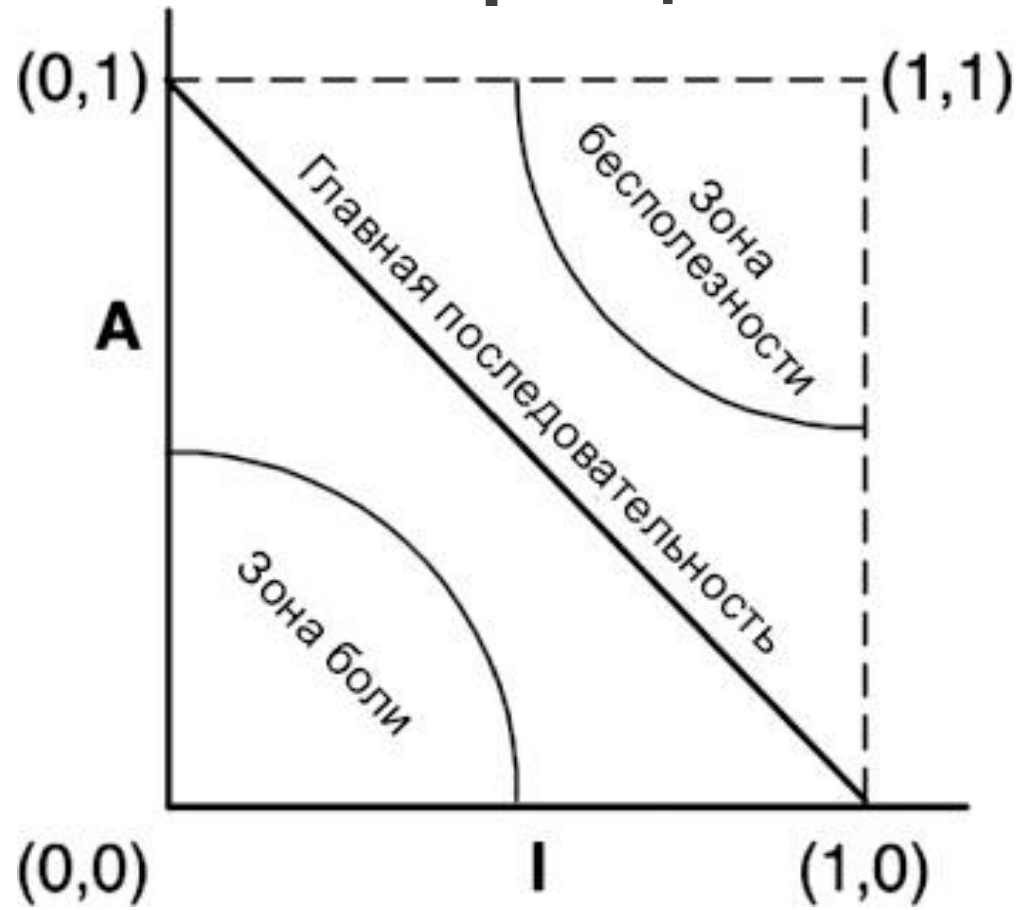
$$A = Na \div Nc.$$

Значение метрики *A* изменяется в диапазоне от 0 до 1.

0 означает полное отсутствие абстрактных классов в компоненте, а **1** означает, что компонент не содержит ничего, кроме абстрактных классов

Взаимодействие компонентов

Принцип устойчивости абстракций



Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения

Многоуровневая (Layered) архитектура

Layered architecture

Компоненты в шаблоне многоуровневой архитектуры организованы в горизонтальные слои, каждый уровень выполняет определенную роль в приложении (например, логику представления или бизнес-логику).

Layered architecture

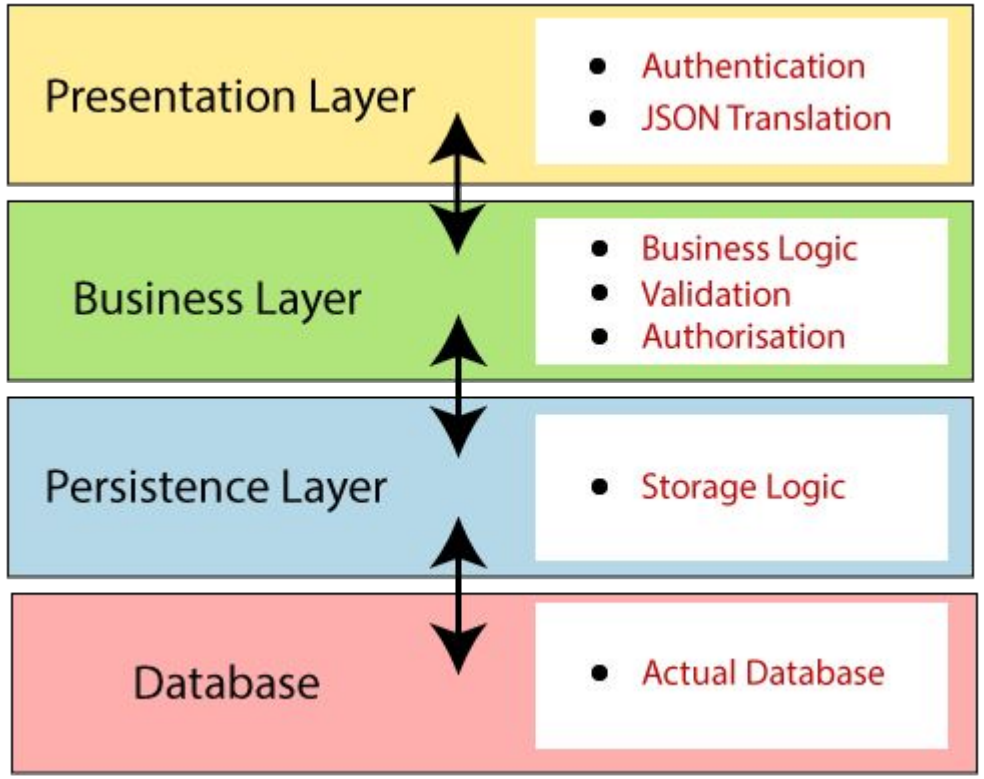
Обычно выделяют четыре стандартных уровня:

- Представления (UI, Presentation),
- Бизнеса (BL),
- Сохраняемости (Persistence)
- Базы данных (DL)

В некоторых случаях бизнес-уровень и уровень сохраняемости объединяются в один бизнес-уровень.

Более крупные и сложные бизнес-приложения могут содержать пять или более уровней.

Layered architecture



Layered architecture

- Модели баз данных находятся в нижней части архитектуры.
- Слои могут взаимодействовать только со слоями на один уровень ниже
- Только части приложения должны быть абстрагированы
- Сильная связь между слоями

Layered architecture

Модели баз данных находятся в нижней части архитектуры.

База данных – неустойчивый компонент. Мы можем менять классы репозиториев, тип базы данных и т.д. Это повлечет изменение остальных слоев.

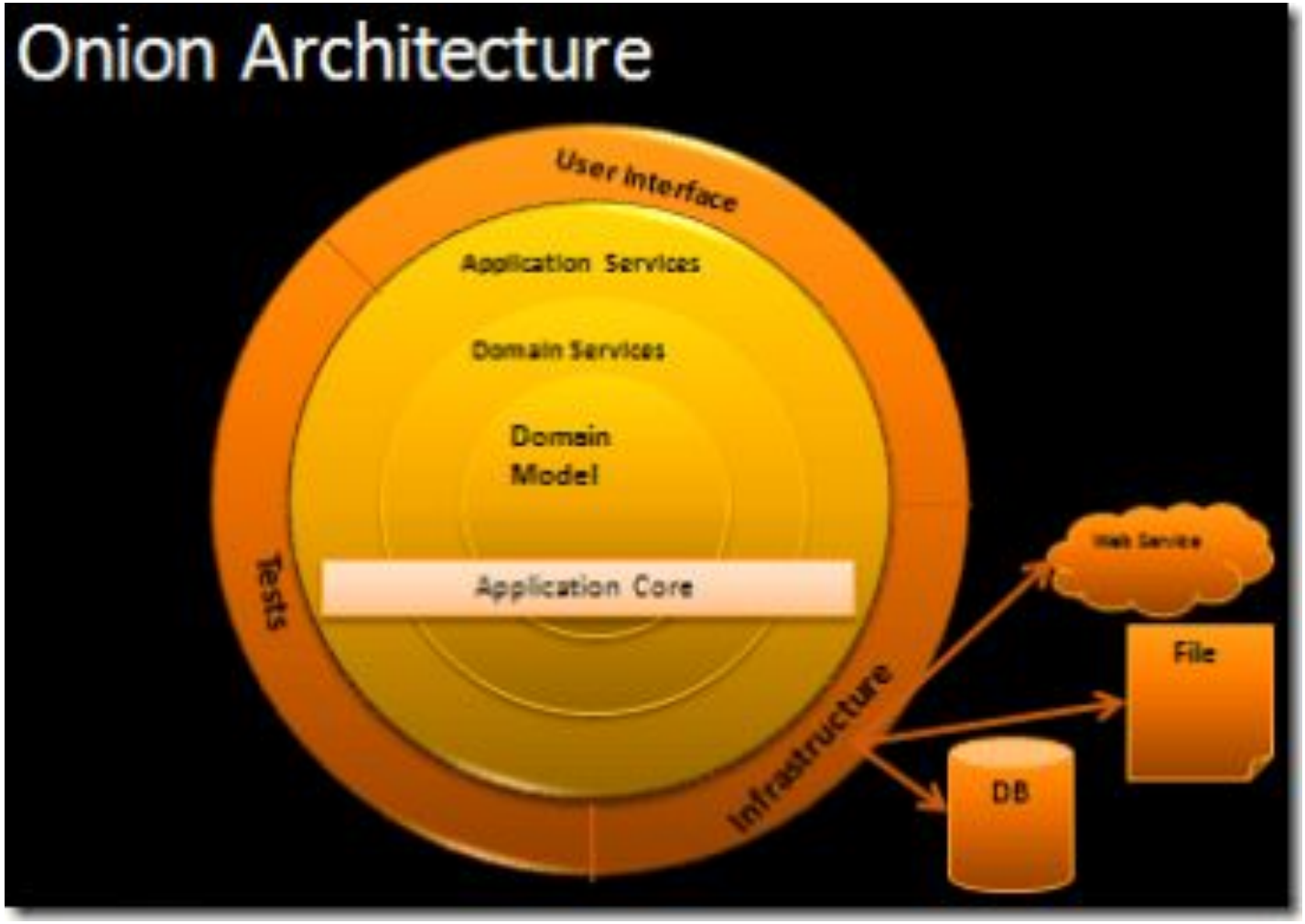
Луковая (onion) архитектура

Луковая (onion) архитектура

Луковая архитектура была представлена Джеффри Палермо в 2008 году

(https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/)

Луковая (onion) архитектура



Луковая (onion) архитектура

Луковый слой решает проблему жесткой связи, с которой мы столкнулись в n-уровневой архитектуре.

Эта архитектура имеет:

- слой домена,
- слой сервисов домена,
- слой приложения
- слой представления.

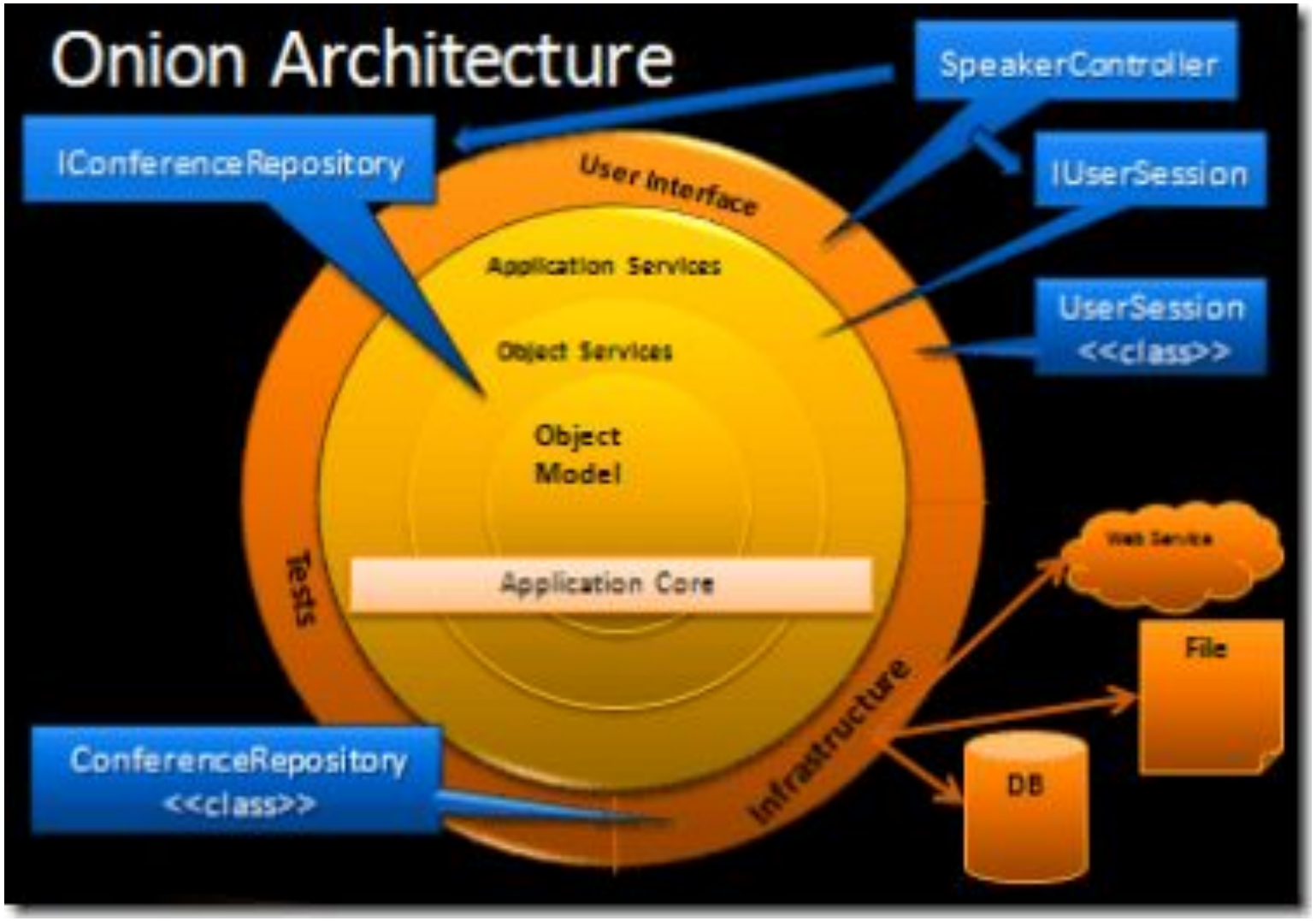
Луковая (onion) архитектура

- Модель домена находится в середине архитектуры
- Внешние слои могут взаимодействовать только с внутренними слоями.
- Сильно зависит от DIP (DIP/DI/IoC)
- Слабая связь между слоями
- Слои могут взаимодействовать с несколькими внутренними слоями
- Уровни инфраструктуры могут взаимодействовать друг с другом

Луковая (onion) архитектура

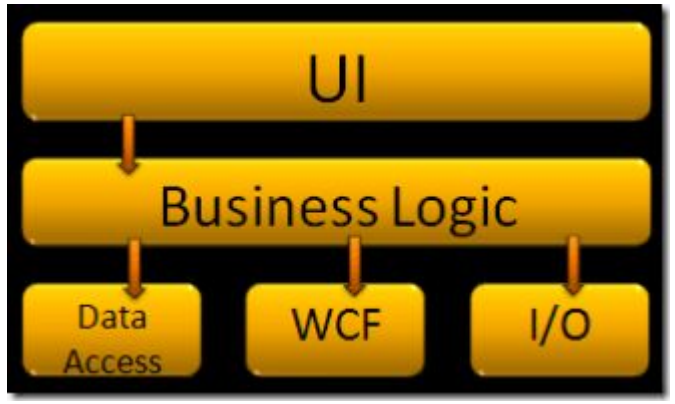
- Домен описывает корпоративные бизнес-правила (предметную область)
- Domain Services - абстрактные репозитории (без деталей реализации, такие как соединение с базой данных, - это будет описано на верхних уровнях)
- Application services определяют бизнес-процессы приложения.
- На самом внешнем уровне находятся пользовательский интерфейс, подключения к внешней инфраструктуре и автоматизированные тесты.

Луковая (onion) архитектура

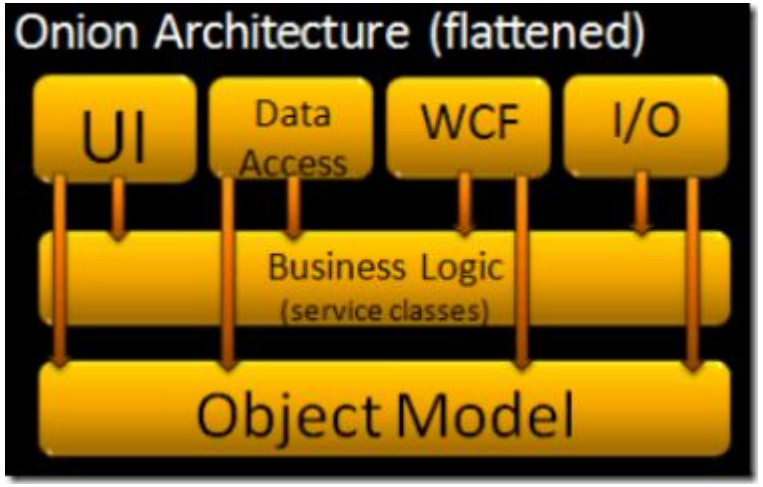


Луковая (onion) архитектура

3-слойная архитектура



Onion архитектура



Чистая (Clean) архитектура

Чистая (Clean) архитектура

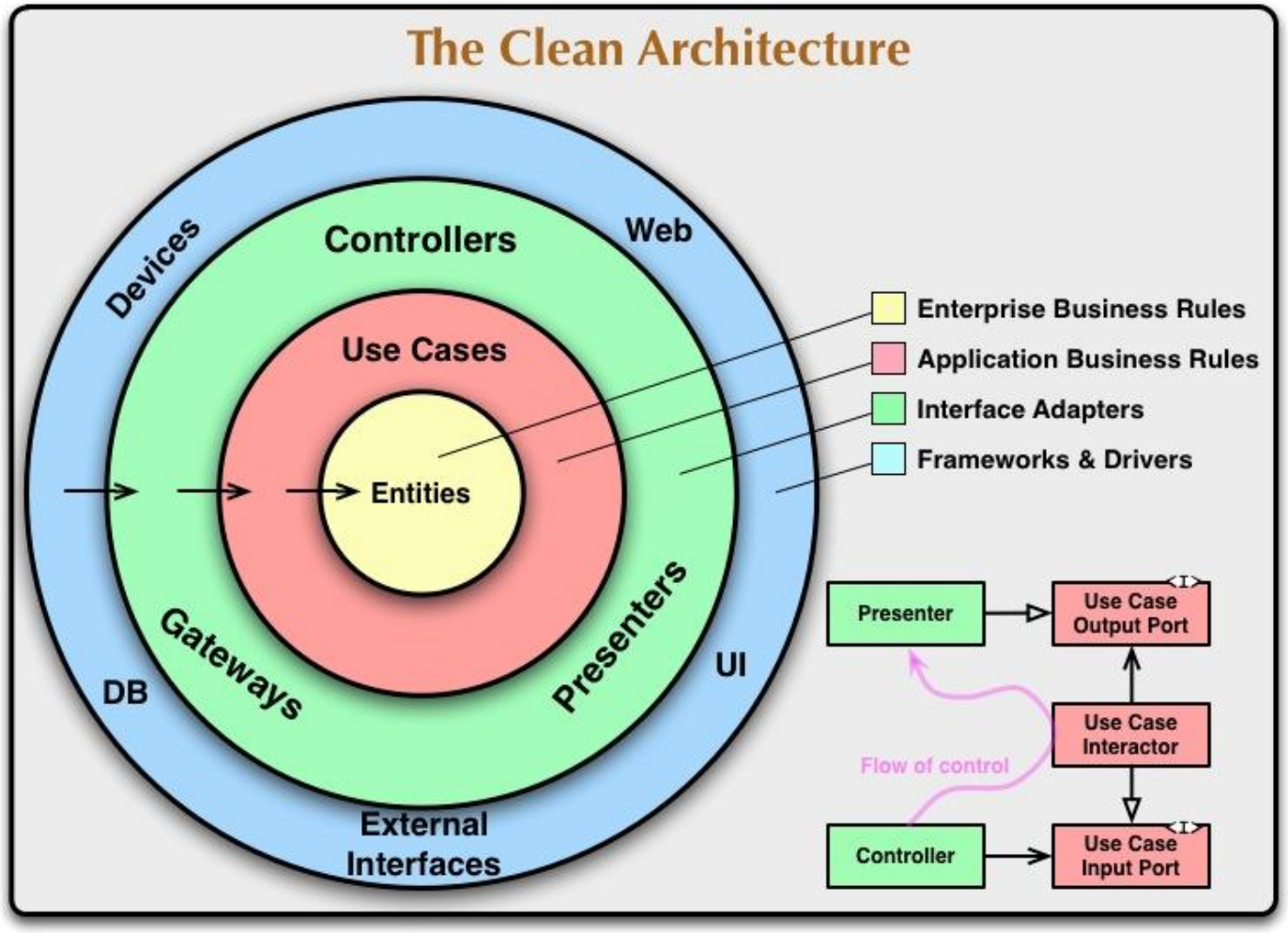
Чистая архитектура была представлена Робертом «Uncle Bob» Мартином в 2012 году

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Чистая (Clean) архитектура

Чистая архитектура базируется на Луковой архитектуре, но с некоторыми отличиями.

Чистая (Clean) архитектура



Чистая (Clean) архитектура

Чистая архитектура базируется на Луковой архитектуре, но с некоторыми отличиями.