



Domain Driven Design

Как правильно
проектировать
сервисы?

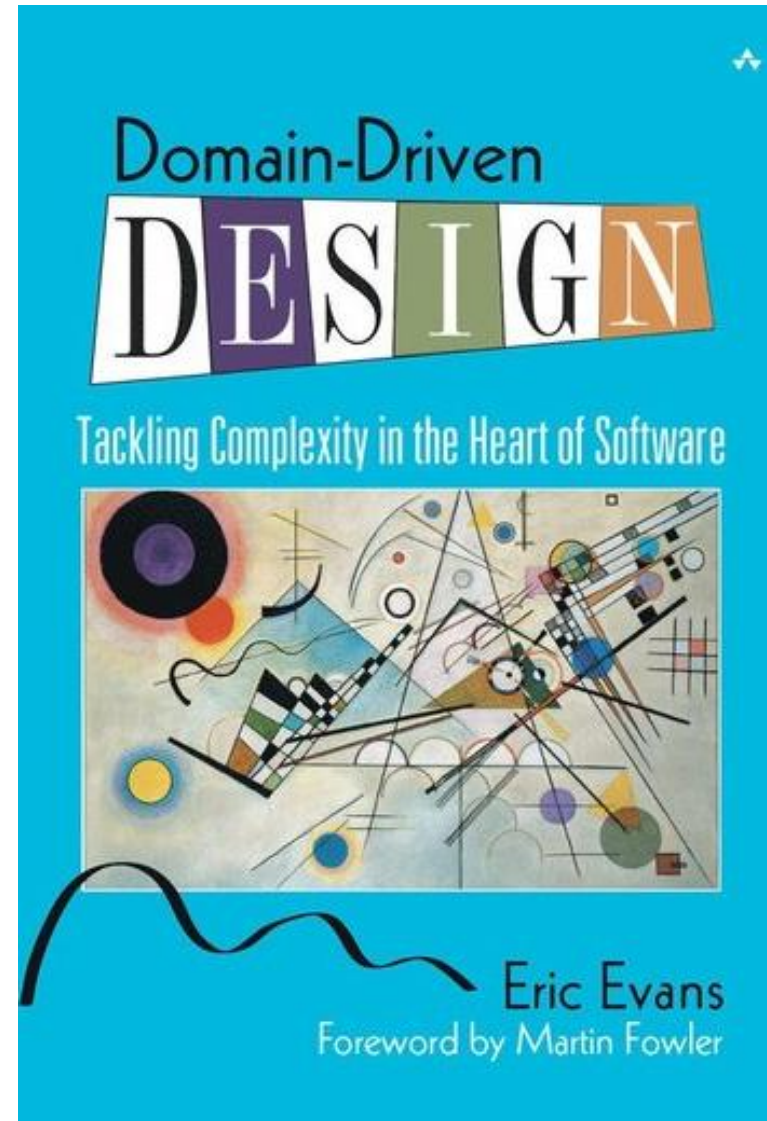


Domain Driven DESIGN

Сокращенно DDD

Придумано Eric Evans

“Domain-Driven Design: Tackling Complexity in the Heart of Software”



Домен

— это некоторая область объединяющая в себе процессы или объекты по некоторым правилам

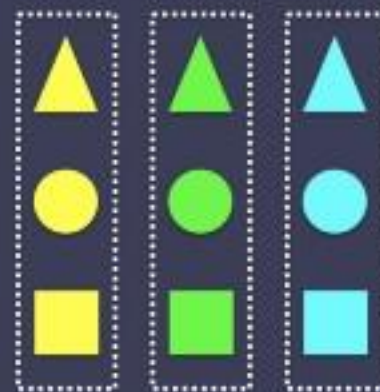
Домены субъективны и не являются
взаимоисключающими.

Одни и те же понятия
могут существовать
во многих разных
областях.

DEFINING DOMAIN BOUNDARIES



Shape Domains



Colour Domains



Other Possible
Domains

Определение

https://en.wikipedia.org/wiki/Domain-driven_design

DDD - это подход к разработке программного обеспечения для сложных задач путем соединения реализации с эволюционирующей моделью предметной области.

Предпосылкой проектирования, ориентированного на домен (DDD), является следующее:

- основной фокус проекта находится в решении задач какой-либо предметной области
- большая сложность проекта
- потребность в активном сотрудничестве между техническими специалистами и экспертами домена для итеративного уточнения концептуальной модели, которая решает конкретные проблемы домена

Почему стоит заниматься DDD?

- ✓ DDD помогает объединить экспертов и разработчиков в одну команду
- ✓ Разработчики могут помочь экспертам лучше понять то что им нужно

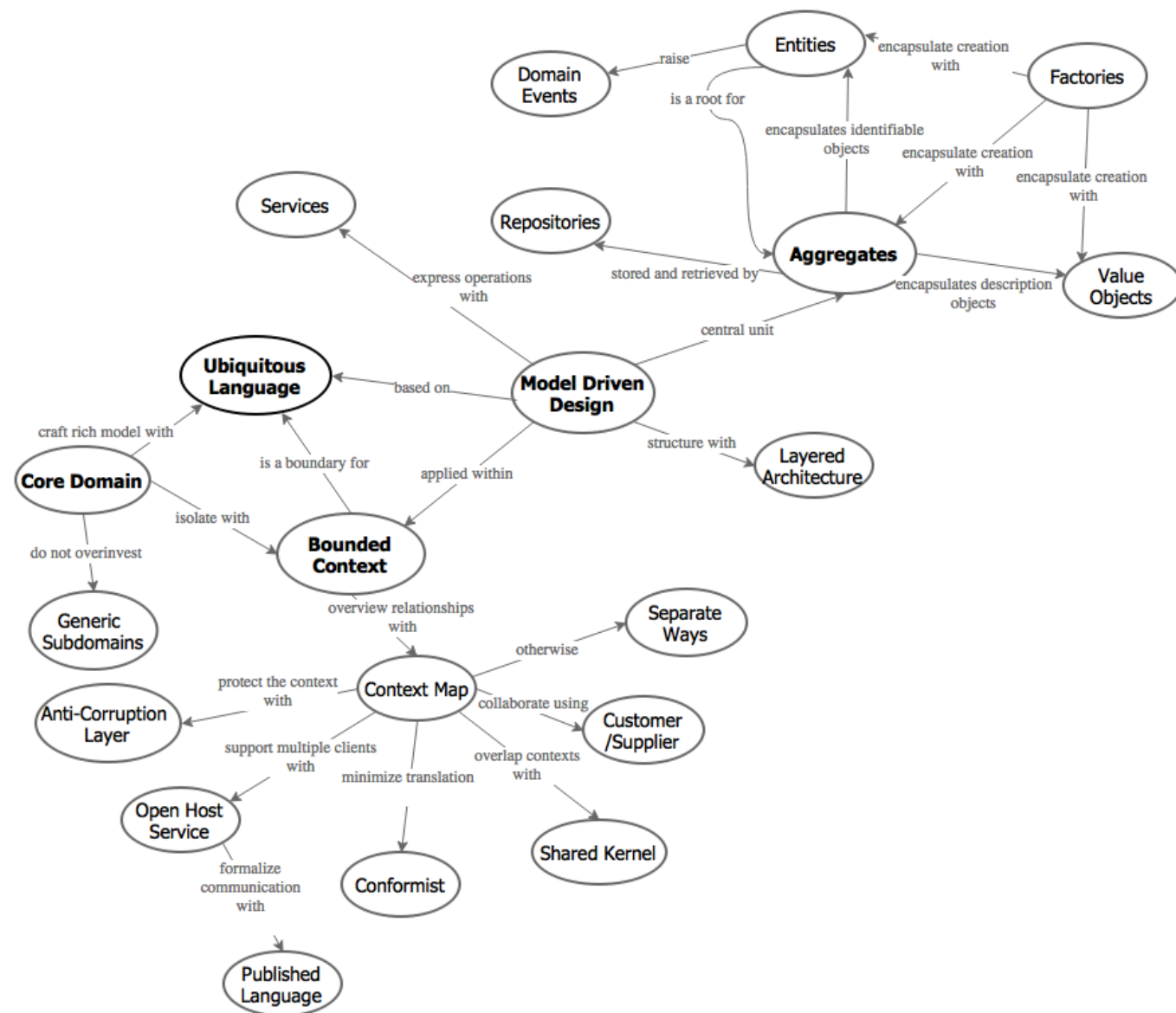
Как это работает?

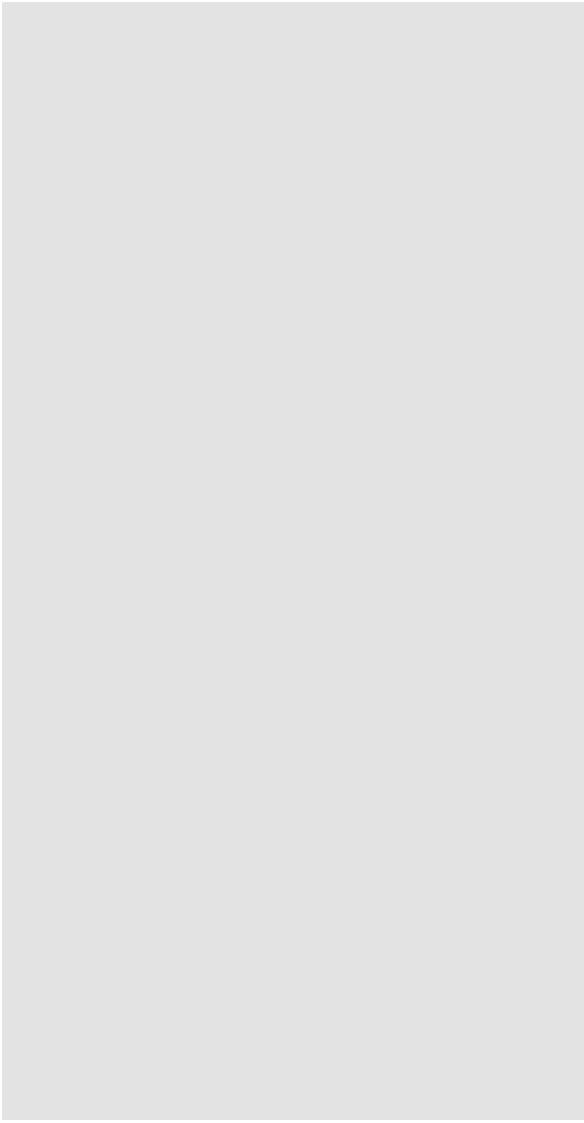

- **Не нужны переводчики** между экспертами домена, разработчиками программного обеспечения и программным обеспечением
- **Дизайн - это код, а код - это дизайн.**

Уровни DDD

- **Стратегический дизайн** определяет части программного обеспечения, которые требуют больше усилий для разработки, и кто должен быть вовлечен в процесс.
- **Тактический дизайн** помогает нам построить одну конкретную модель, которая может содержать со сложными бизнес-правилами и дальнейшей эволюцией программного обеспечения.

Элементы



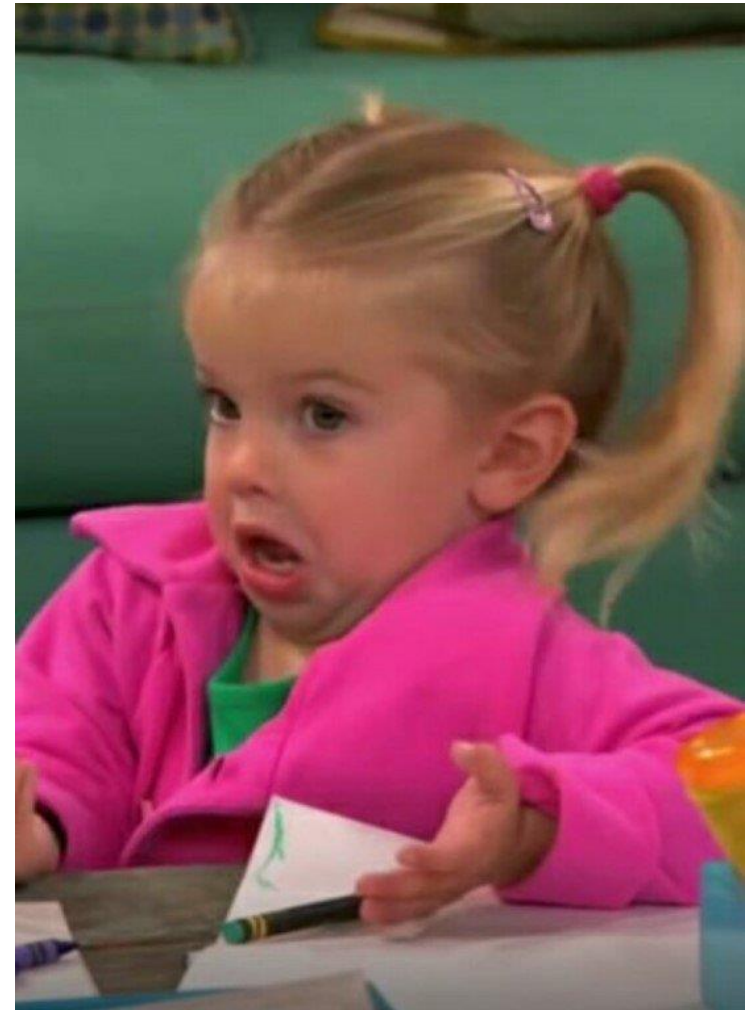


Ubiquitous
Language
повсеместный язык
общения

Пример

Как клиента могут называть в телекоме?

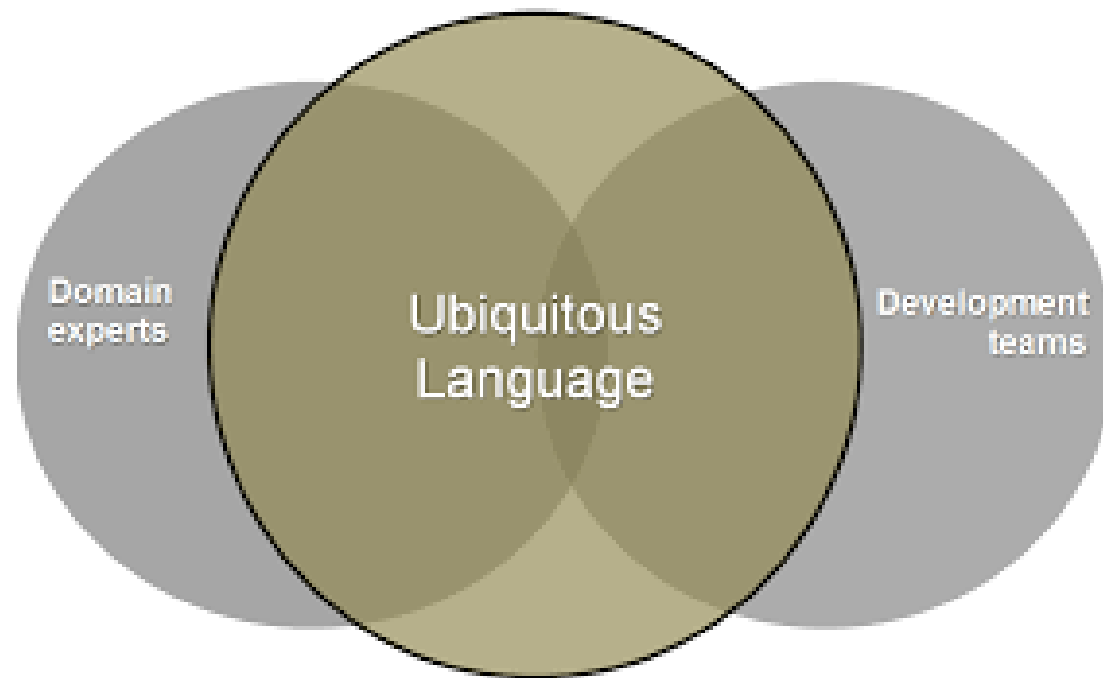
- В домене CRM – **Client**
- В домене управления заказами – **Party**
- В домене управления сетью - **Subscriber**
- В биллинговом домене - **Account**



Ubiquitous Language

Используется во всех частях проекта и на всех этапах

Единый язык общения между пользователями и разработчиками



Bounded Context

В доменно-ориентированном проектировании (DDD) **ограниченный контекст** — это определенная граница, внутри которой применяется конкретная модель предметной области.

Он определяет область действия, в которой конкретная модель действительна и актуальна.

Ограниченные контексты помогают установить четкие границы между различными частями системы, гарантируя, что каждая часть имеет свой собственный четко определенный контекст, язык и правила.

Ubiquitous Language

- Язык должен быть основан на модели домена, которую строит команда. В языке **нет технических терминов**, которые эксперт не может понять.
- Язык **должен развиваться** вместе с развитием домена.
- Для каждого **ограниченного контекста** (bounded context) существует один язык Ubiquitous.
- Применение одного языка Ubiquitous Language вообще ко всему – не рекомендованная практика. **Для разных экспертов – разные языки.**

Use case	Commit a backlog item to sprint
Плохо	<pre>sprint = sprints.GetSprint(sprintName); backLogItem = backLog.GetItem(itemId); backLogItem.SprintId = sprint.Id; backLogItem.DevelopmentTeam = sprint.DevelopmentTeam; backLogItem. ...</pre>
Лучше	<pre>sprint = sprints.GetSprint(sprintName); backLogItem = backLog.GetItem(itemId); backLogItem.CommitTo(sprint);</pre>

Язык есть не только в общении/документации но и в программном коде

Создание языка

Рисуем картинки физической и концептуальной области и обозначаем их названиями.

Создаем глоссарий терминов с простыми определениями.

Собираем обратную связь от членов команды. Постоянно совершенствуем язык.





Инструменты

Strategic Design

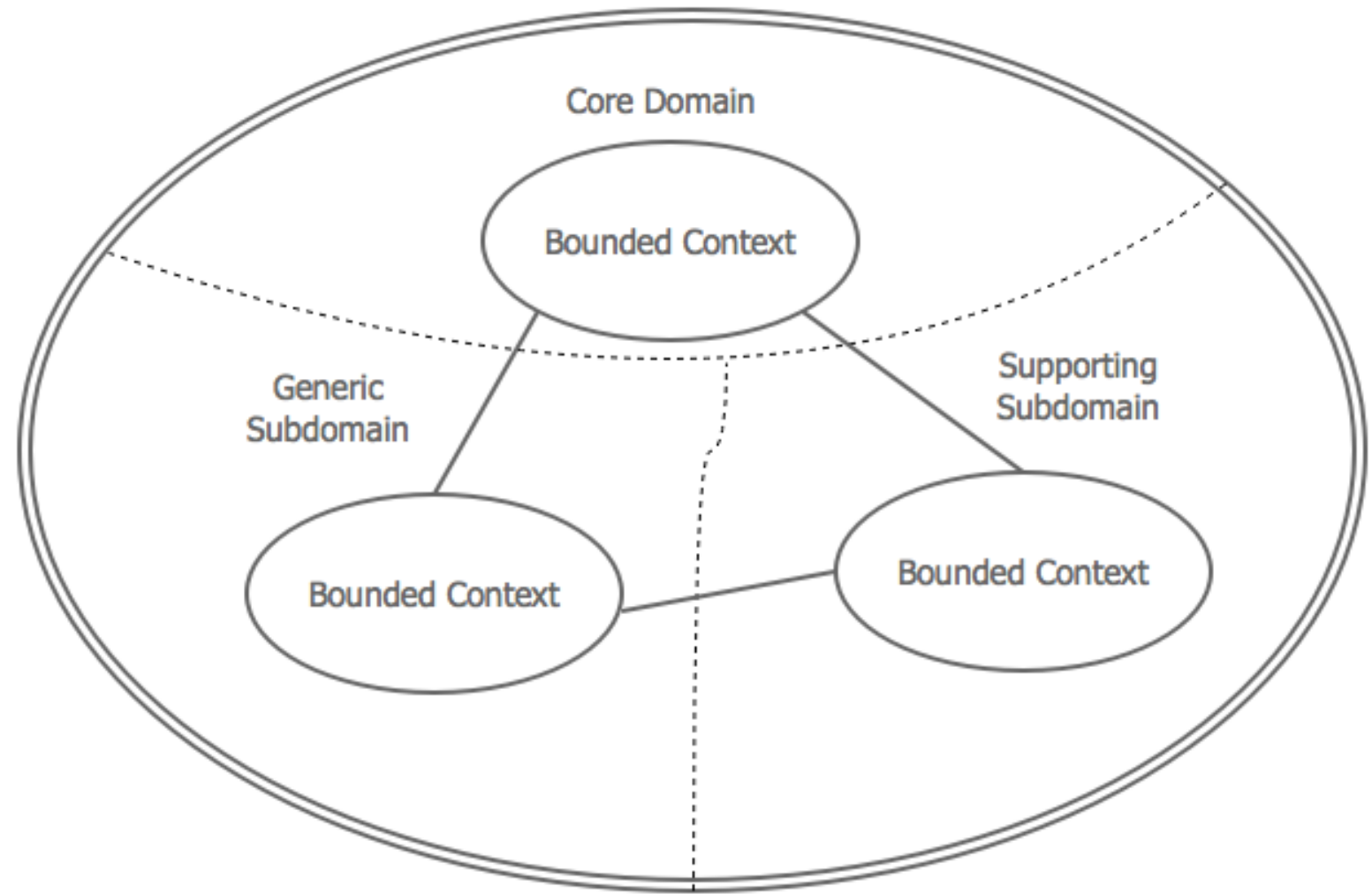


Паттерны

- **Bounded Context** – центральный паттерн в Domain-Driven Design. DDD имеет дело с большими моделями, разделяя их на различные «Ограниченные Контексты».
- **Context Map** – карта ограниченных контекстов и отношений между ними

Domains/ Subdomains

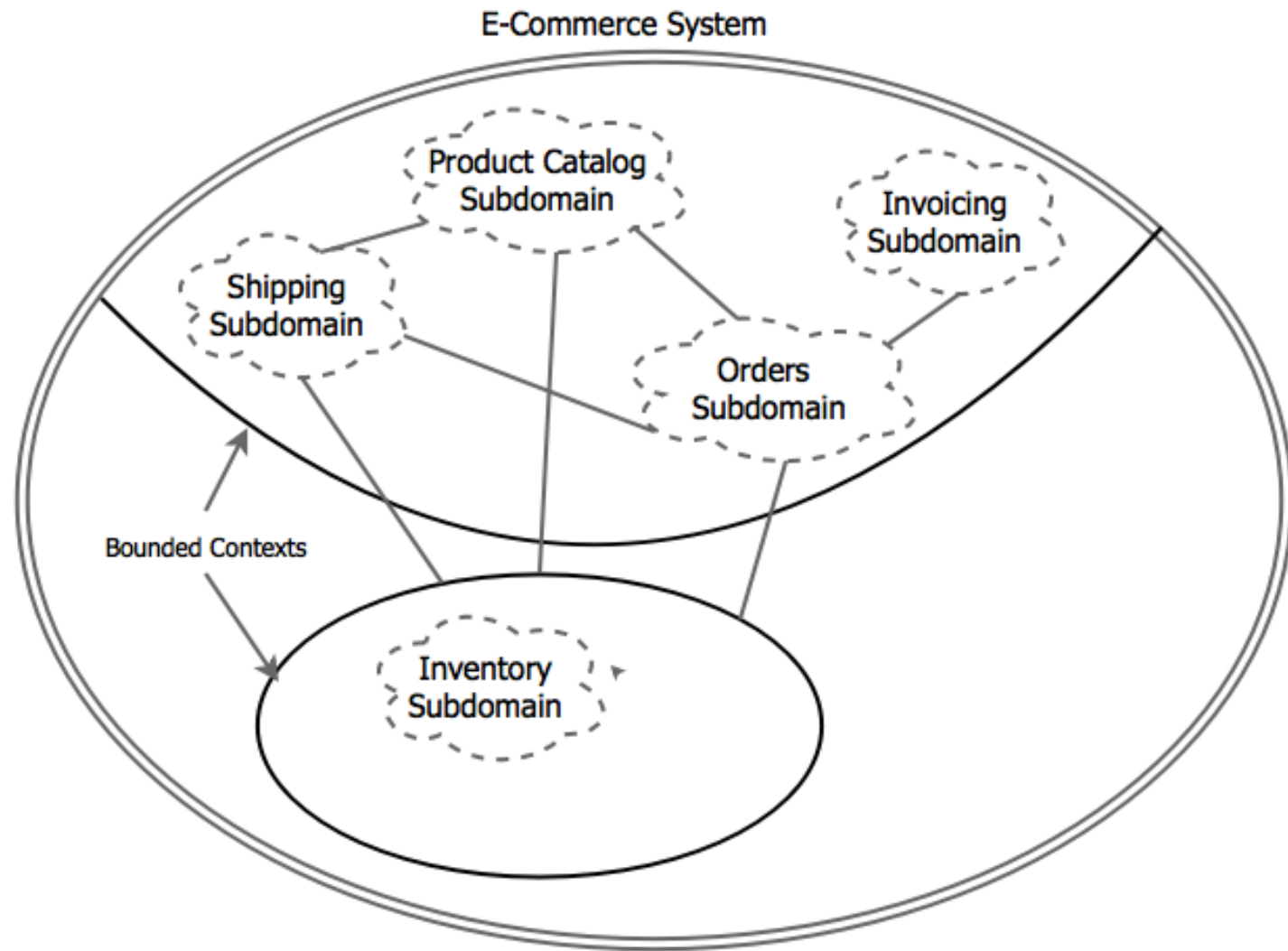
Bounded Contexts



Когда нужен Strategic Design

- Системы становятся слишком **сложными**, поскольку состоят из множества компонентов и подсистем
- Ведение одной большой модели домена для нескольких подсистем становится **сложным**
- Некоторые части системы становятся более **важными**, чем другие.
- Над разработкой системы работают **несколько команд**

Без
стратегического
дизайна



Без стратегического дизайна

- Два Bounded Contexts (subsystems) существуют в E-Commerce System
- Несколько **неявных** доменных моделей соединены в одну
 - *Product Catalog*
 - *Orders*
 - *Invoicing*
 - *Shipping*

Чем плоха одна доменная модель для всего приложения?

- **Одна модель должна удовлетворять все потребности домена**, поэтому она может быть либо слишком сложной, либо слишком общей и лишенной какого-либо поведения.
- В различных подсистемах **могут существовать конфликтующие концепции**, которые будут препятствовать развитию друг друга
- Например Customer может иметь **разные атрибуты в разных системах**:
 - **Catalog Subsystem**: previous purchases, loyalty, available products, discount, shipping options
 - **Orders Subsystem**: shipping address, billing address, payment terms.

Что же такое Strategic Design?

- DDD - это **способ уменьшение сложности**, одна монолитная модель обычно слишком сложна
- Стратегическое проектирование - это **набор техник**, которые должны управлять ростом сложных приложений.
- Основной принцип, которому следует стратегическое проектирование, - **"Разделяй и властвуй"**,
- Поэтому стратегическое проектирование DDD заключается в разбиении проблемного домена на под-домены, каждый из которых реализуется в виде отдельной модели отдельной командой.

Типы ПОД-ДОМЕНОВ

1. **Core Domains** - области, которые отличают уникальное продуктивное предложение вашей компании от предложения конкурентов. Основные области - это причина, по которой вы сами пишете это программное обеспечение
2. **Generic Subdomains** – под-домены которые не формируют отдельных приложений (Ex: Notifications, Logging, Reporting).
3. **Supporting Subdomains** - вспомогательные средства для основной области и системы

1 Core Domain - выбор

Чтобы определить основной домен, необходимо ответить на следующие вопросы:

- Какие части продукта сделают его успешным?
- Почему эти части системы важны?
- Что делает вашу систему достойной создания?

1 Core Domain - работа

- Выделите лучших разработчиков для работы над основным доменом
- Вкладывайте большую часть усилий в реализацию основного домена
- Развивайте основной домен в течение всего срока службы, чтобы победить других конкурентов
- Рассматривайте основной домен как продукт, а не как проект

2 Generic Subdomains

- Общие поддомены для многих крупных систем (безопасность, протоколирование, отправка электронной почты)
- Не являются ключевыми для бизнеса и не дают вам конкурентного преимущества
- Не имеет смысла тратить время на их создание, возможные варианты:
 - Покупка готового решения
 - Выделить младших разработчиков для их создания

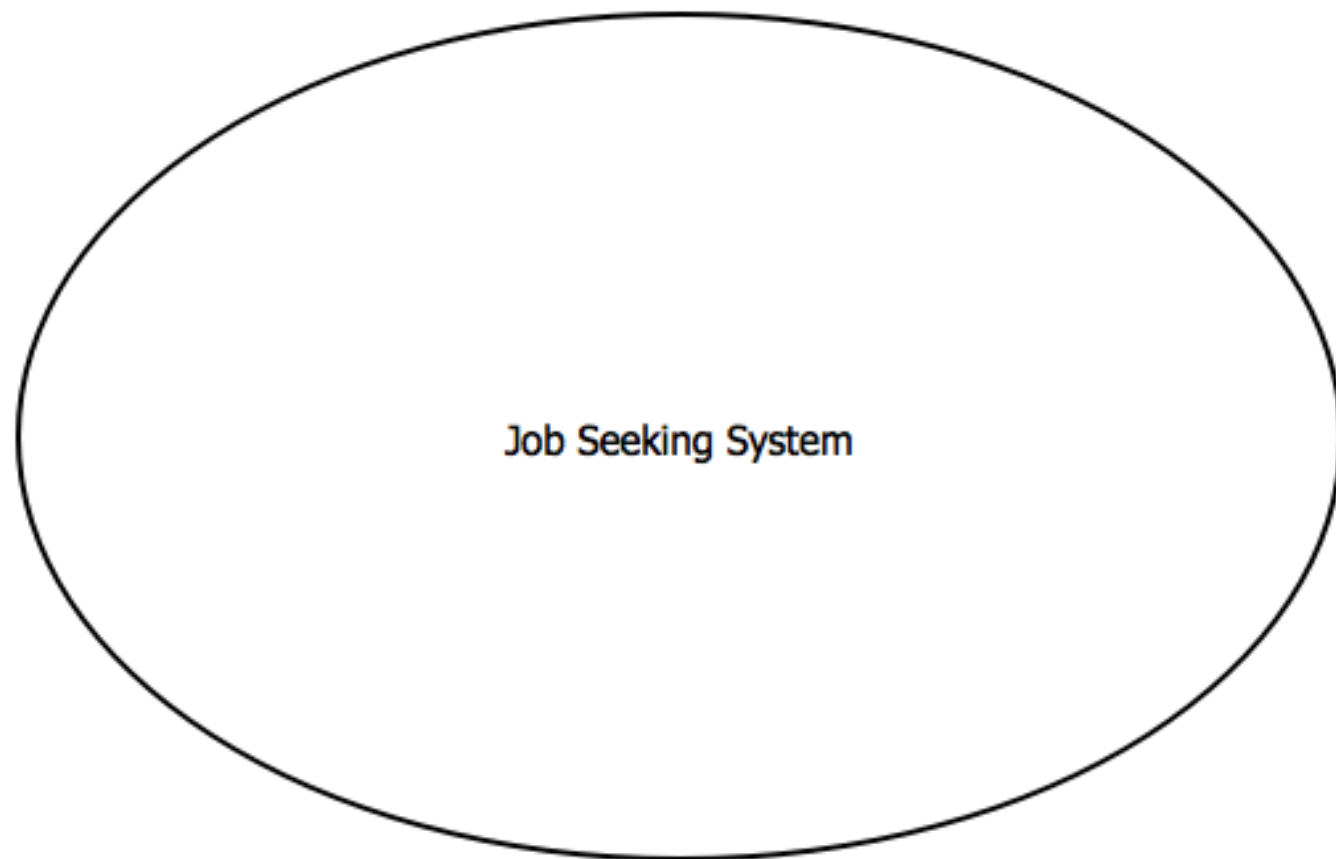
3 Supporting Subdomains

- Поддержка непосредственно основных областей
- Если есть возможность приобрести готовое решение, сделайте это.

Пример:

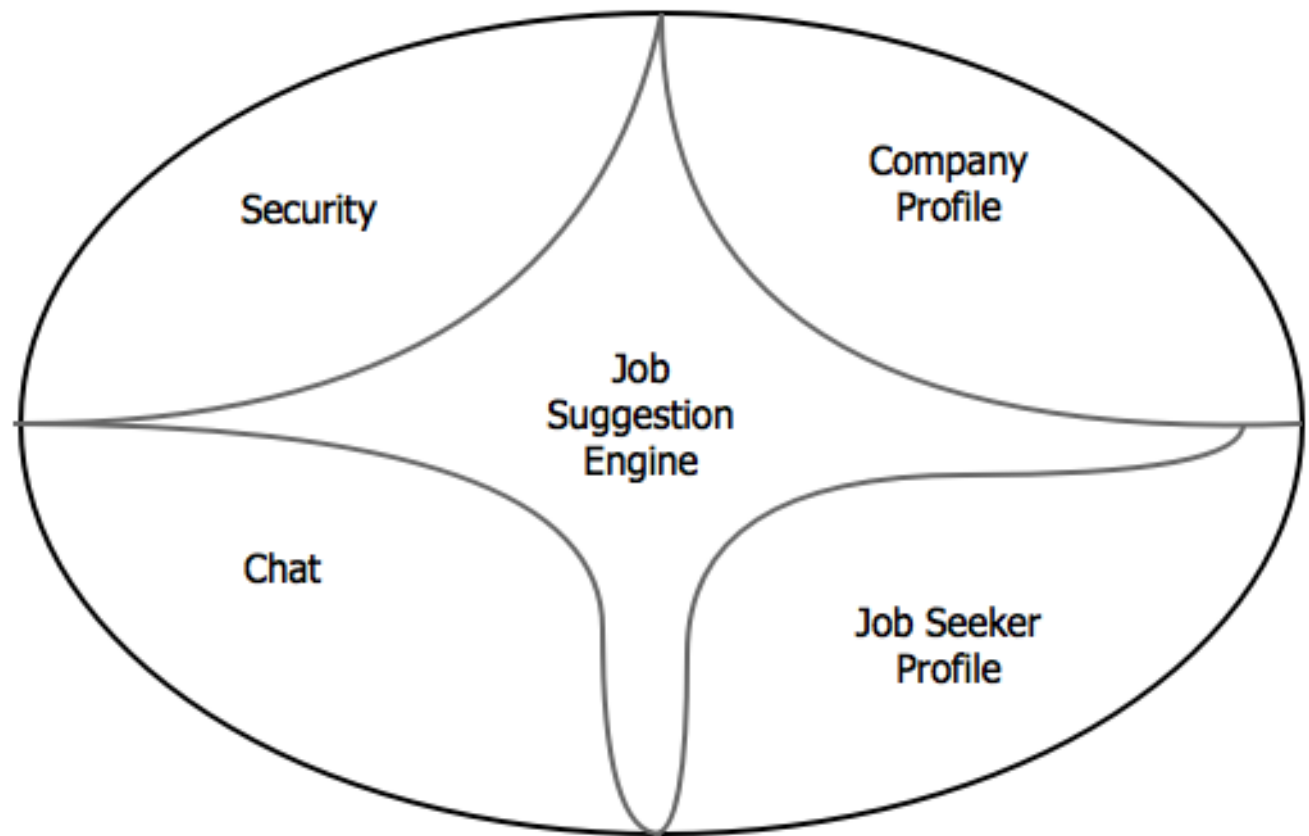
Каталог товаров Amazon является поддерживающим под-доменом для основного домена - рекомендательного движка

Пример выделения домена



Пример выделения домена

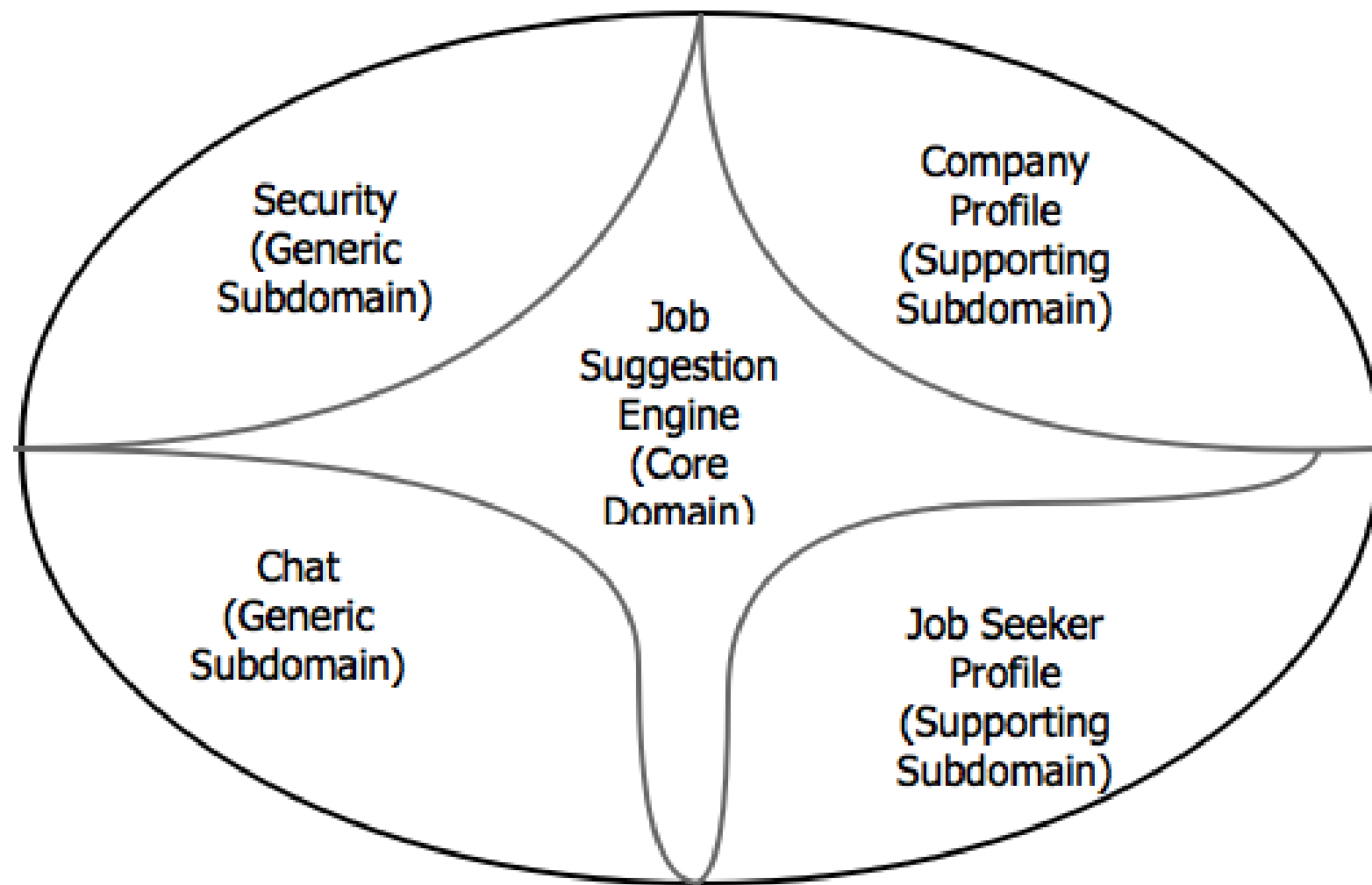
Identifying the Subdomains



Пример выделения домена

- **Что является жизненно важными частями системы для успеха бизнеса?**
Механизм предложения вакансий - анализирует существующие предложения о работе и делает предложения на основе профиля соискателей. Основной домен
- **Каковы вспомогательные поддомены для основного домена Job Suggestion Engine?**
Профиль компании
Профиль соискателя
- **Каковы общие поддомены?**
Чат - может быть готовым коммерческим или бесплатным решением
Безопасность - может быть готовым коммерческим или бесплатным решением

Пример выделения домена



Bounded Contexts

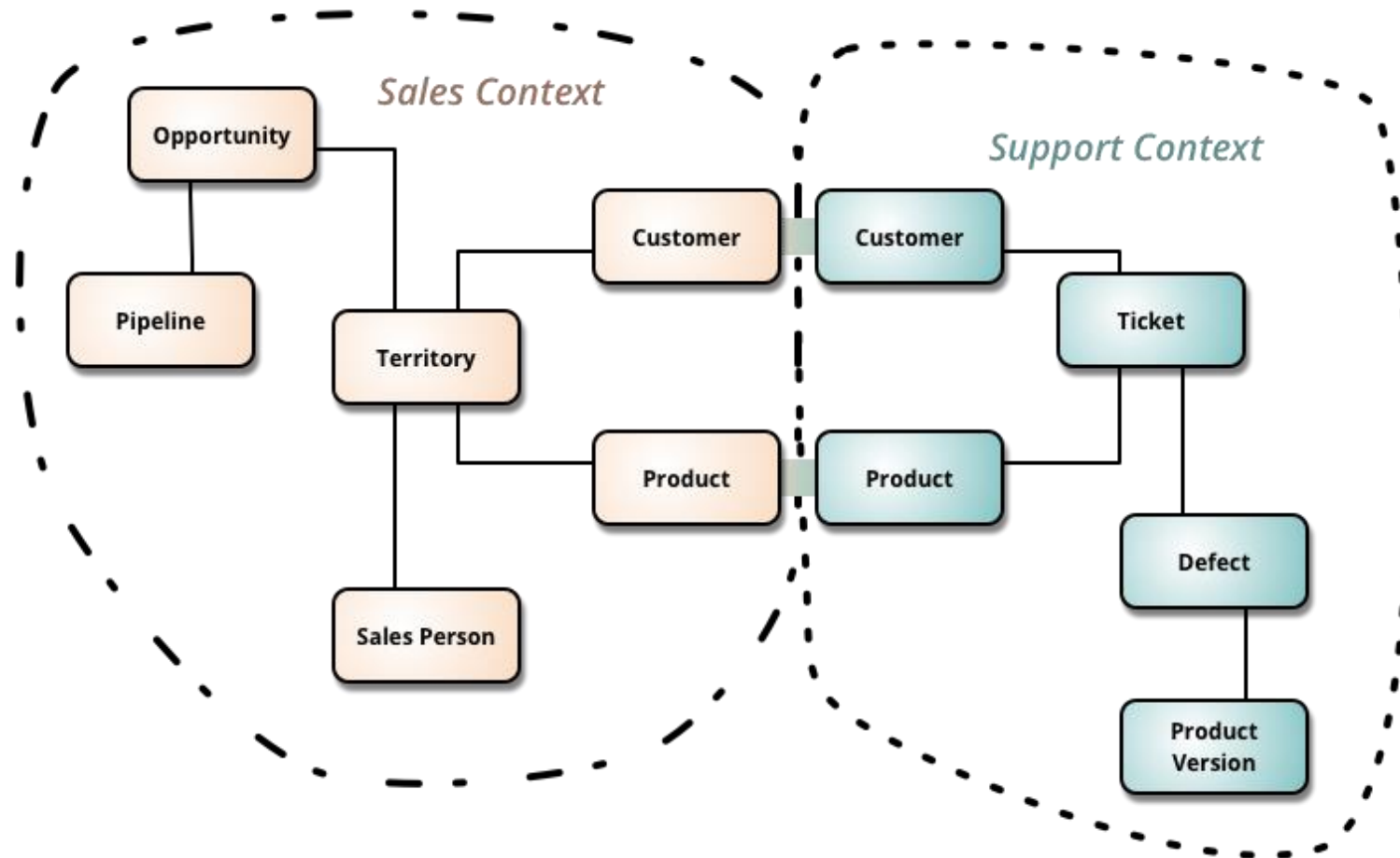


Bounded Contexts

- Центральный паттерн проектирования DDD
- Граница между языком и моделью домена (DM)
- Физическое разделение доменных моделей
- Если используется тактический DDD, он имеет свой собственный набор агрегатов, сущностей, значений, событий, сервисов, репозиториях, БД (хотя это не обязательно).
- Управляется ОДНОЙ командой (допустимо, чтобы одна команда управляла несколькими Ограниченными Контекстами).

Пример

<http://martinfowler.com/bliki/BoundedContext.html>



Context Maps

A close-up photograph of a map with several pushpins. The map is light-colored with blue lines and text. The pushpins are dark blue and are pinned to various locations on the map. The text on the map includes 'Esporões', 'Rabacal', 'Carvalho', 'Coriscada', 'Ervedosa', 'Juiz', 'Oateira', 'Vieira', and 'Coriscada'. The pushpins are arranged in a way that suggests they are marking specific points of interest or locations. The background is a soft, out-of-focus blue.

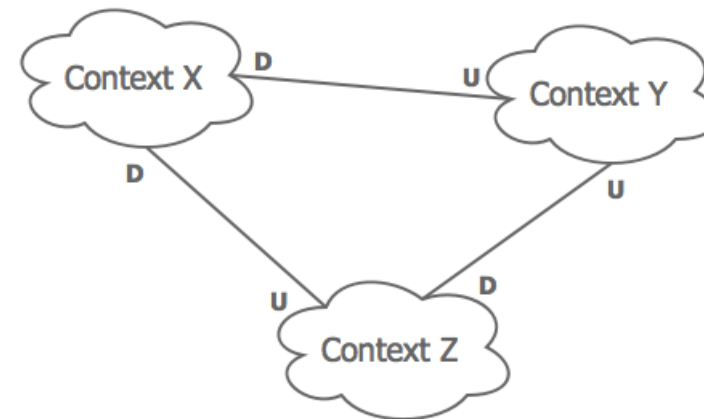
Обзор

Контекстные карты отражают **технические и организационные отношения** между Ограниченными Контекстами.

Контекстные карты должны отражать **реальное** положение вещей и должны меняться только при изменении кода.

Между связанными контекстами существует **несколько моделей отношений**

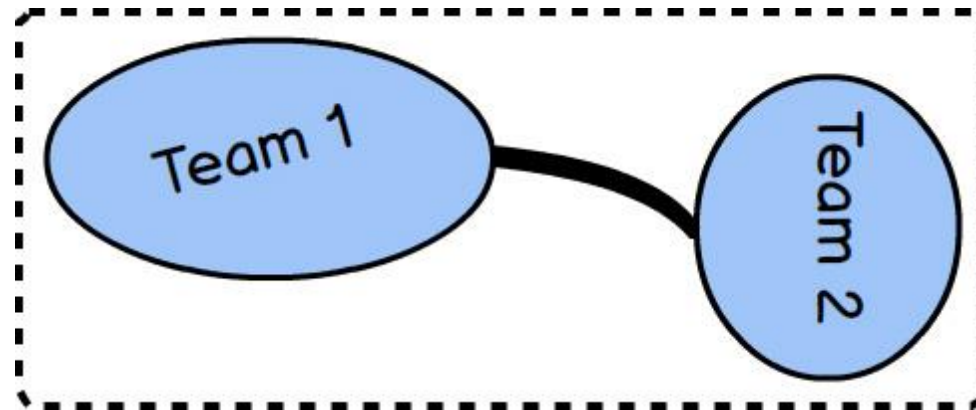
Это не технические интеграционные паттерны взаимодействия между Контекстами.



Bounded Contexts Relationships

Partnership

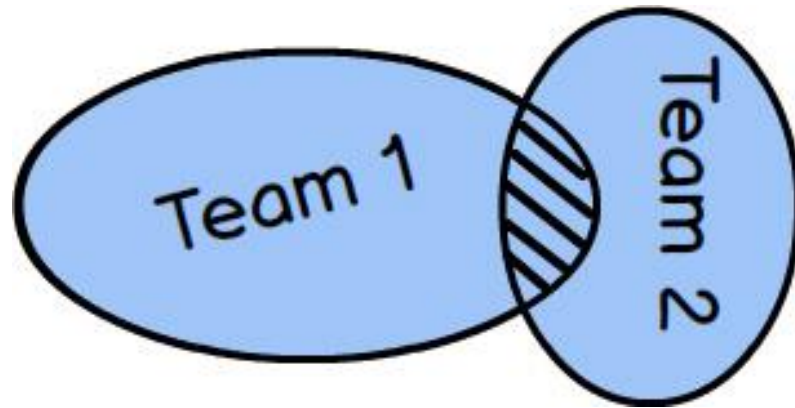
- Команды в двух контекстах будут преуспевать или терпеть неудачу вместе. Команды находятся в отношениях сотрудничества.
- Команды координируют свои процессы планирования и интеграционную деятельность.
- Команды должны координировать развитие своих интерфейсов, чтобы учесть потребности разработки обеих систем.
- Взаимозависимые функции должны быть завершены для одного и того же релиза.



Bounded Contexts Relationships

Shared Kernel

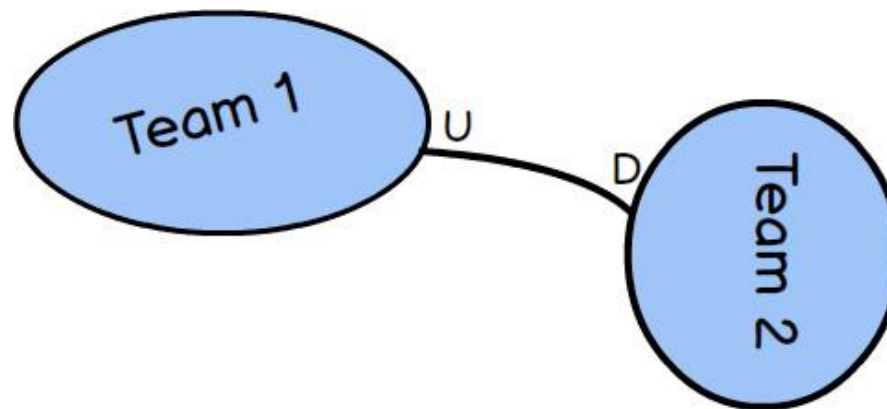
- **Совместное использование** части модели и связанного кода для облегчения интеграции.
- Команды тесно **работают над одним и тем же приложением**.
- Применимо, если есть два ограниченных контекста в пределах одного под-домена (чего обычно следует избегать).
- Отмечается некоторое подмножество модели домена, которым команды согласны поделиться.
- Ядро должно быть небольшим. Общий код не должен быть изменен без согласия участвующих команд.



Bounded Contexts
Relationships

Customer-Supplier Development

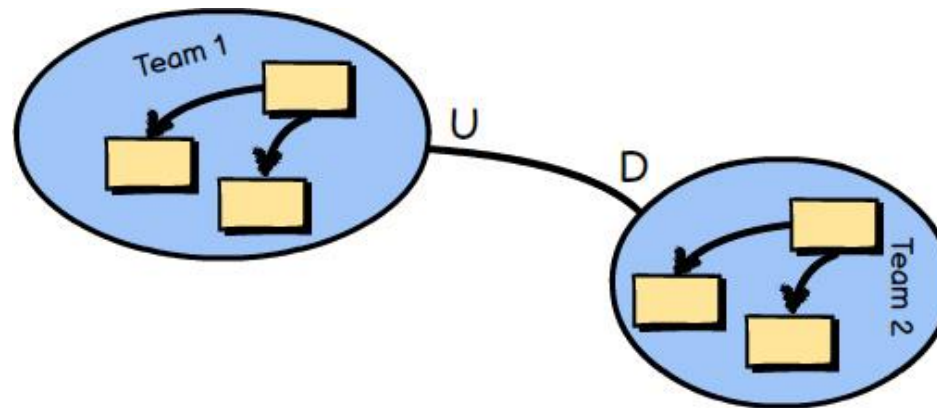
- Две команды находятся в отношениях **поставщик/upstream (U)** – **потребитель/downstream (D)**
- Команда **поставщик** может быть успешной независимо от команды **потребителя**.
- Команда **потребитель** зависит от данных или поведения конца **поставщика**.
- Команда **поставщик** просит команду **потребителя** определить приоритетность необходимых работ.
- Команда **потребитель** согласовывает даты выпуска необходимых функций с командой **поставщиком**.



Bounded Contexts Relationships

Conformist

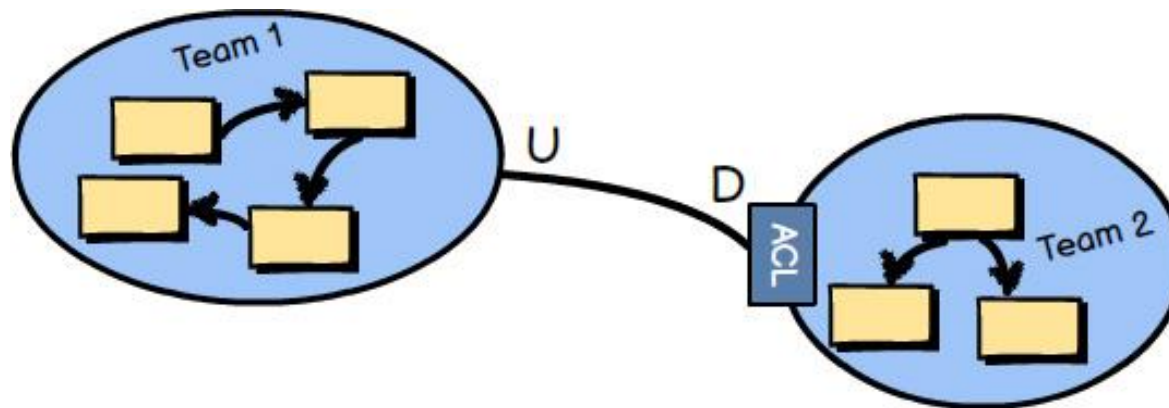
- Две команды разработчиков имеют отношения между поставщиком (U) и потребителем (D).
- Команда поставщик не может сотрудничать с командой потребителем по вопросам интеграции.
- Команда потребитель устраняет сложности перевода и интеграции между ограниченными контекстами, придерживаясь модели команды поставщика потока.

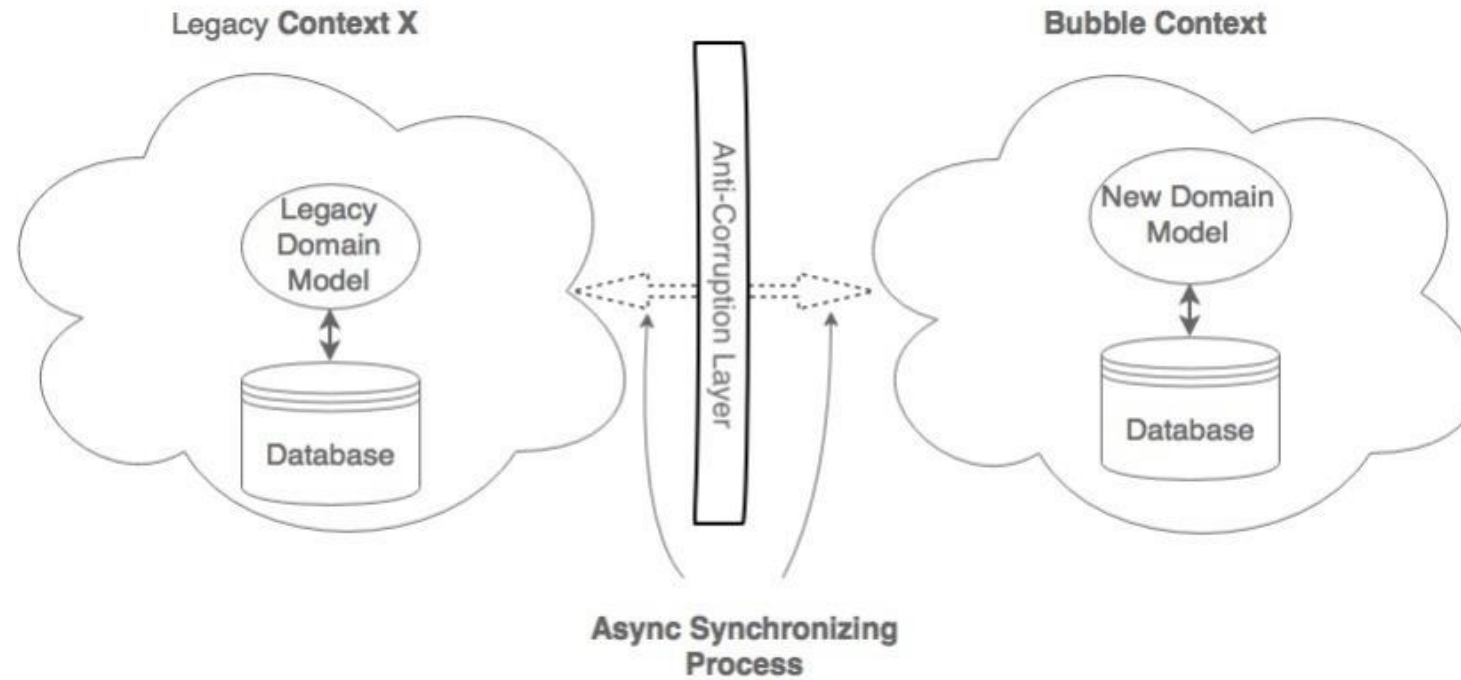


Bounded Contexts Relationships

Anticorruption Layer

- Разные команды работают над разными моделями с разными языками Ubiquitous Languages. Если не быть осторожным во время интеграции и импортируете слишком много «иностранных» концепций, ваша собственная доменная модель может быть повреждена.
- Как потребитель, создайте изолирующий слой, чтобы обеспечить вашей системе функциональность системы поставщика в терминах вашей собственной доменной модели.
- Этот антикоррупционный слой общается с другой системой через ее общедоступный интерфейс, не требуя практически никаких изменений в другой системе. Этот слой переводит в одном или обоих направлениях локальные понятия во внешние.



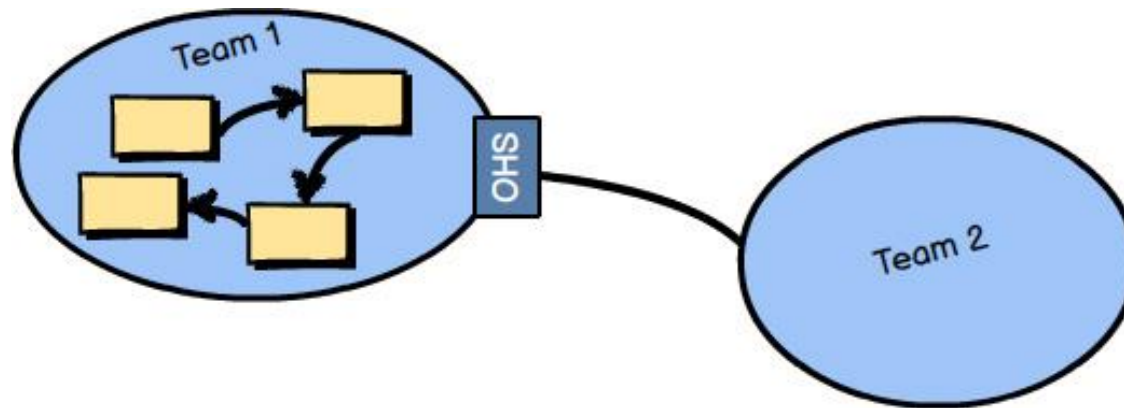


Пример интеграции с legacy

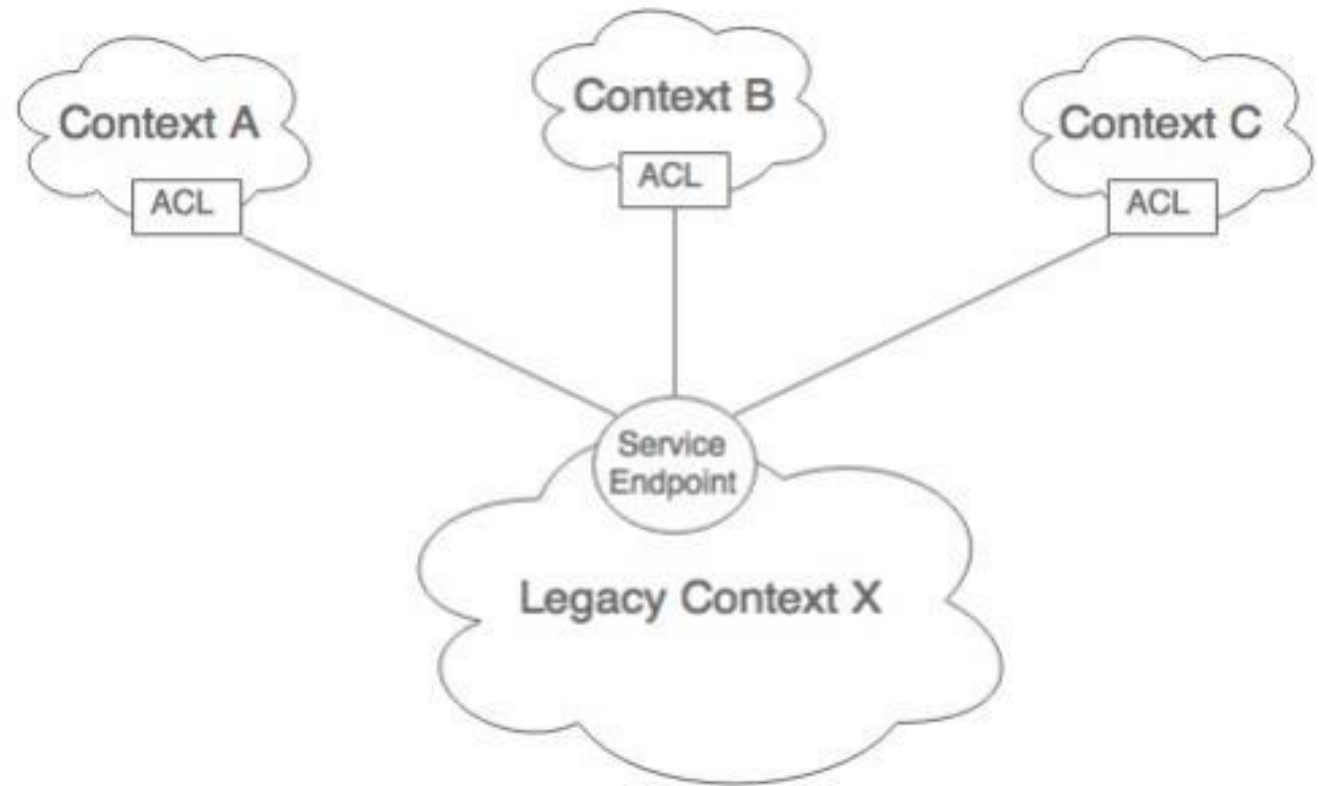
Bounded Contexts Relationships

Open Host Service

- Несколько систем могут иметь схожую логику преобразования данных при взаимодействии с Ограниченным Контекстом со сложным протоколом.
- Это усложняет процессы обслуживания, что приводит к дублированию кода на каждом последующем этапе.
- Определите упрощенную версию протокола, который предоставляет доступ к вашей подсистеме как к набору сервисов (может быть реализован через механизмы обмена сообщениями).
- Общий протокол должен быть простым и последовательным.
- Совершенствуйте и расширяйте протокол для обработки новых требований к интеграции



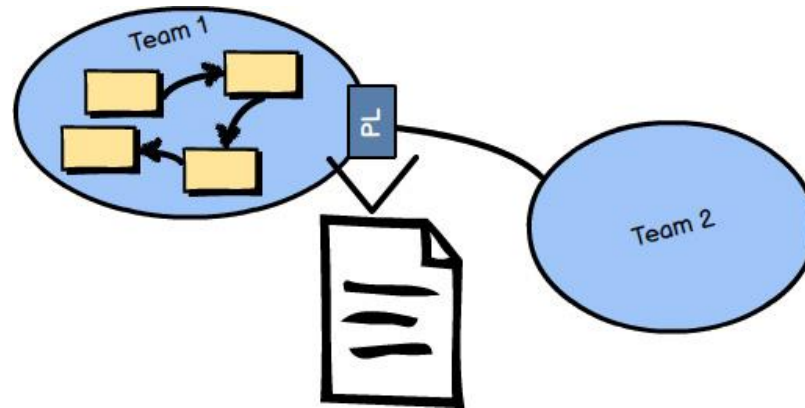
пример с
legacy



Bounded Contexts
Relationships

Published Language

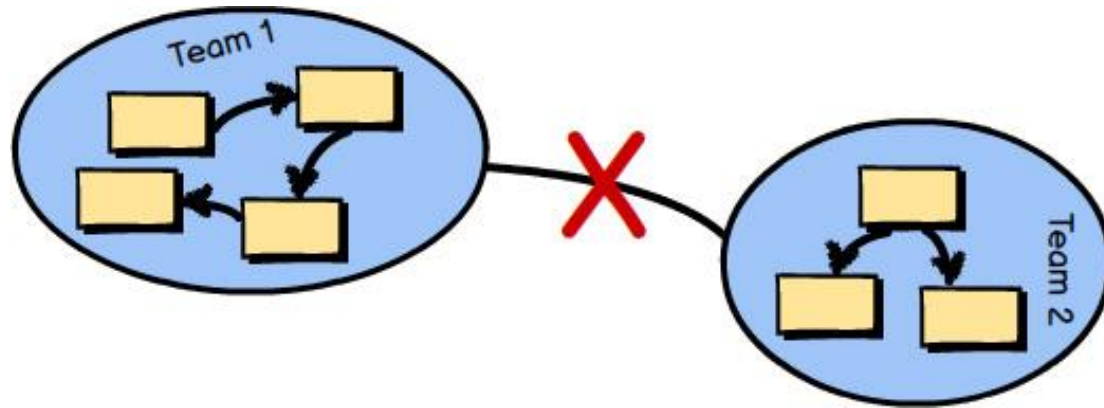
- Общий язык необходим для обработки перевода между моделями двух ограниченных контекстов.
- Цель Published Language - обеспечить хорошо документированную общую понятийную модель, которая может использоваться между разными доменами.
- Публикуемый язык (PL) часто сочетается с OHS.
- Пример – единая модель данных для взаимодействия между контекстами



Bounded Contexts Relationships

Separate Ways

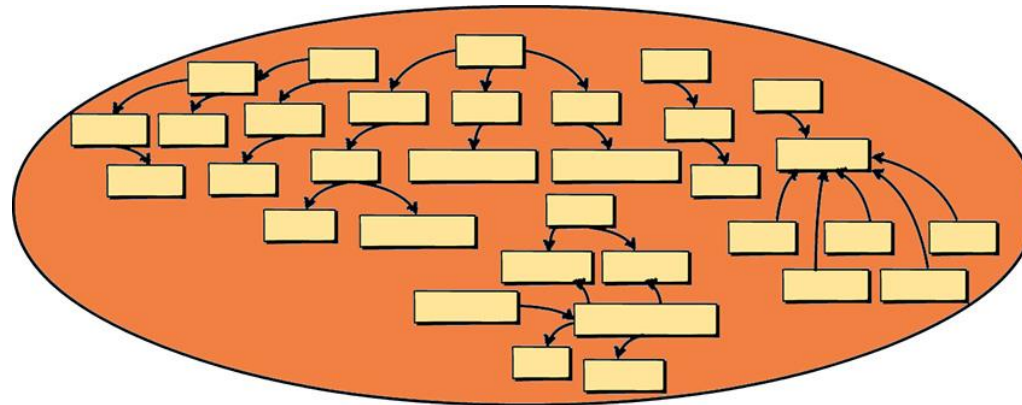
- Если два контекста не имеют существенной взаимосвязи, их можно полностью изолировать друг от друга.
- Интеграция всегда обходится дорого, и если она не приносит существенной пользы, не делайте ее.



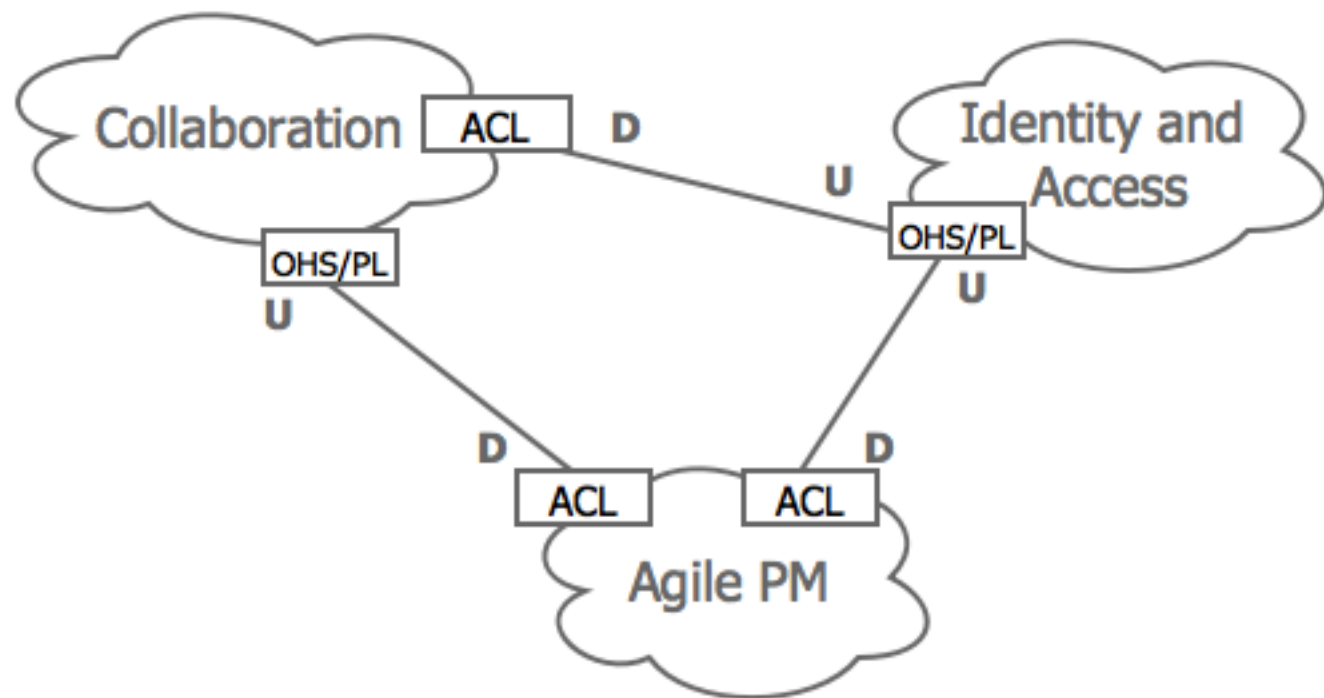
Bounded Contexts Relationships

Big Ball of Mud

- Части систем, часто больших, где модели смешаны, а границы размыты.
- Постройте границу (фасад) вокруг всего этого беспорядка и обозначьте его как Большой клубок грязи.
- Не пытайтесь применять сложное моделирование в этом контексте.



Пример



Зачем нужен Context Maps



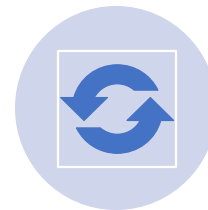
Помогает понять
ответственность и
обязанности.



Помогает выявить места,
где может быть путаница,
в рабочих процессах.

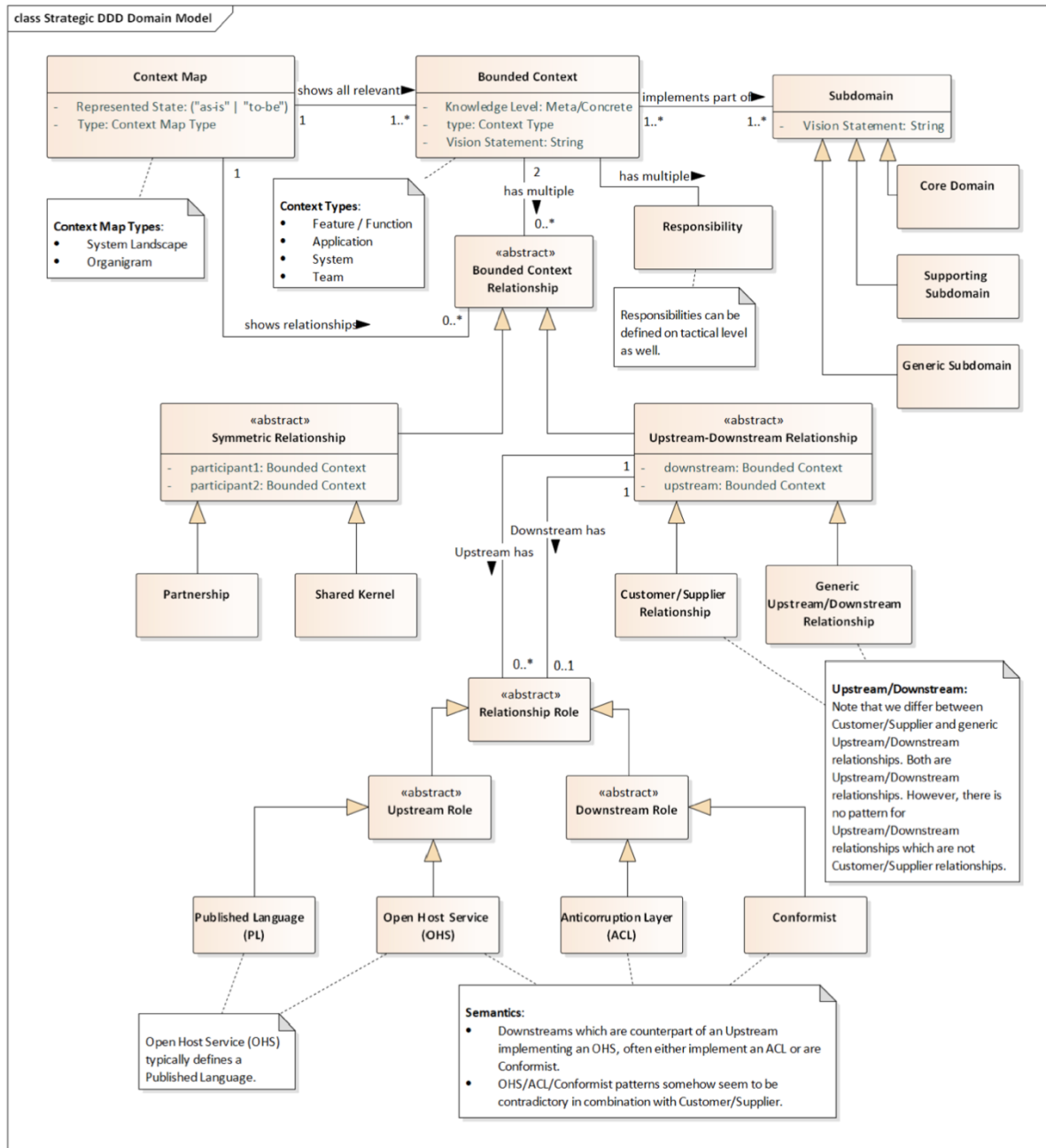


Упрощает общение
между командами.

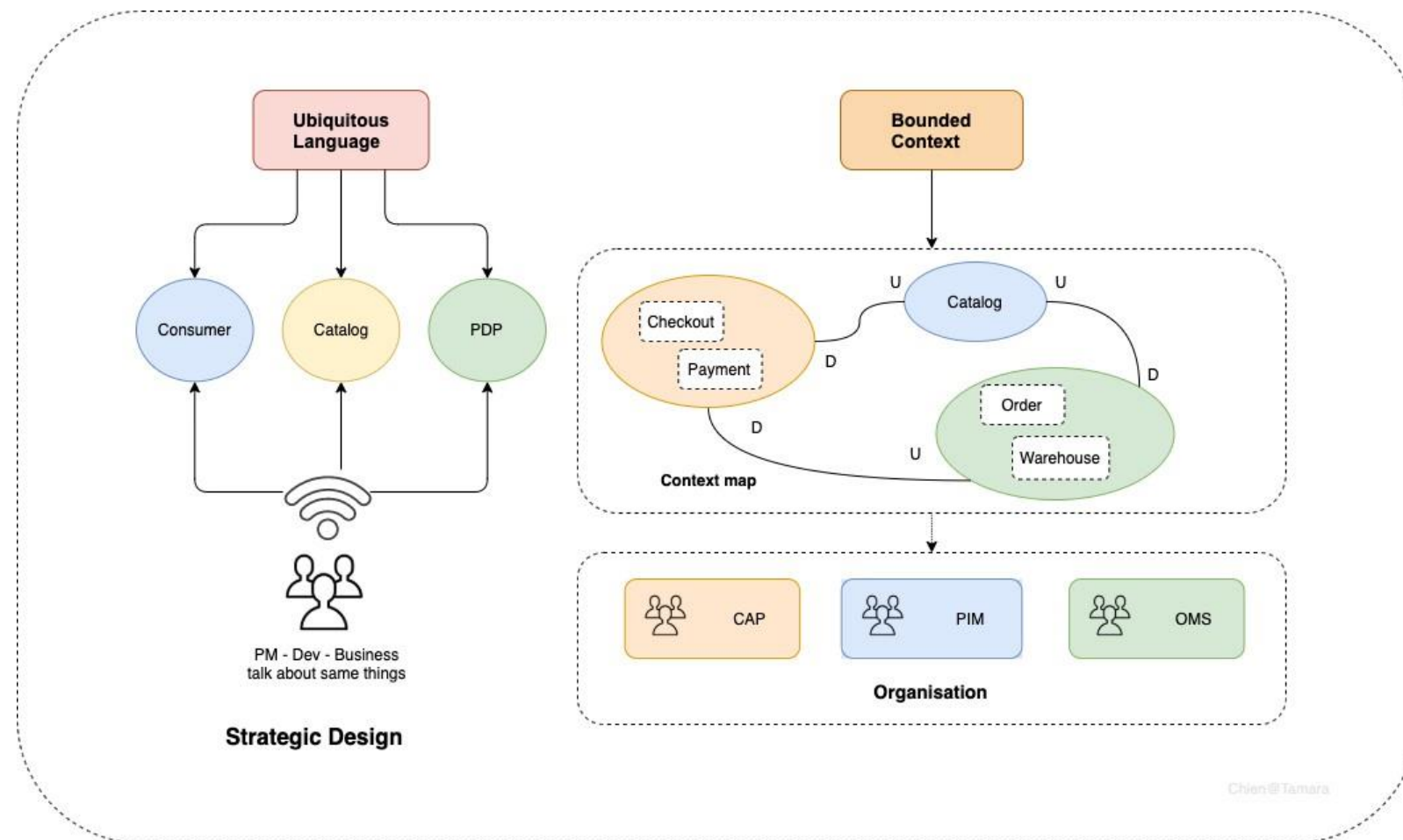


Облегчает процесс
объяснения командам
как устроена система.

Метамодель



Итого



Тактический дизайн

паттерны

Tactical Design

– набор принципов и паттернов, которые используются при моделировании с помощью **Bounded Context**. Тактический дизайн **базируется на принципах OOP и OOD**.

Object Oriented Programming Principles

- ООР эволюция структурного программирования
- У объектов есть **данные и поведение**
- Принципы– абстракция, **инкапсуляция** , наследование, полиморфизм

OOP Principle - Encapsulation

- **Encapsulation** - сокрытие реализации данных путем ограничения доступа к аксессорам и мутаторам.
- **Accessor** - это публичный метод, который используется для запроса объекта о самом себе. Обычно это может быть часть `get` свойства (C#, Java, etc.).
- **Mutator** – это публичный метод, который используется для изменения состояния объекта.

Encapsulation

Объект контролирует свою целостность (инварианты), не позволяя пользователям устанавливать внутренние данные компонента в противоречивое состояние.

Реализация контракта объекта находится в самом объекте, что:

- предотвращает дублирование кода
- улучшает читаемость кода и облегчает сопровождение

Ограничение доступа к внутренним компонентам объекта дает нам свободу рефакторинга, поскольку клиенты объекта не связаны с его внутренними компонентами..

Пример: Нарушаем принцип инкапсуляции *example_o*

```
class User
{
private:
    std::string user_name;
    std::string password;
    int auth_retry_count = 0;
    bool account_is_locked = false;

public:
    std::string &UserName() { return user_name; };
    std::string &Password() { return password; };
    int &AuthAccountRetry() { return auth_retry_count; };
    bool &AccountIsLocked() { return account_is_locked; };

    const std::string &GetUserName() { return user_name; };
    const std::string &GetPassword() { return password; };
    int GetAuthAccountRetry() { return auth_retry_count; };
    bool GetAccountIsLocked() { return account_is_locked; };
};
```

```
class UserSecurityService
{
private:
    const int AuthRetryCountLimit = 5;

public:
    bool AuthenticateUser(const std::string &userName, const std::string &password)
    {
        User user; // = _userRepository.GetUserByName(userName);

        if (user.GetAccountIsLocked())
            throw std::logic_error("User account is locked.");
        std::string hashedPassword; // = _passwordEncryptor.EncryptPassword(password);
        if (user.GetPassword() == hashedPassword){
            user.AuthAccountRetry() = 0;
            return true;
        }
        if (user.GetPassword() != hashedPassword) user.AuthAccountRetry()++;
        if (user.GetAuthAccountRetry() == AuthRetryCountLimit) user.AccountIsLocked() = true;

        // _userRepository.Save(user);
        return false;
    }
};
```

Пример: Нарушаем принцип инкапсуляции

У пользователя нет никакого поведения, только данные.

Пользователь не контролирует свои инварианты:

- Когда аккаунт не заблокирован, его Auth Retry Count < 5
- Когда AuthRetryCount равен 5, аккаунт заблокирован.
- Пароль пользователя всегда должен быть зашифрован
- Любые действия с аккаунтом пользователя запрещены, если аккаунт заблокирован.

Эти правила применяются где-то еще. Поэтому состояние пользователя может быть повреждено. Это непременно приведет к ошибкам и нагрузке на обслуживание, когда логика приложения усложнится.

Рефакторинг example_1

```
class User
{
private:
    const int auth_retry_count_limit = 5;
    std::string user_name;
    std::string password;
    int auth_retry_count = 0;
    bool account_is_locked = false;
    void ResetAuthRetryAttempts(){ auth_retry_count = 0;};

public:
    bool Authenticate(const std::string &raw_psassword, PasswordEncryptor &encryptor){
        if (account_is_locked) throw std::logic_error("User account is locked.");
        std::string hashed_password = encryptor.EncryptPassword(raw_psassword);
        if (password == hashed_password){
            ResetAuthRetryAttempts();
            return true;
        }
        if (password != hashed_password) RegisterNewFailedAttempt();
        return false;
    }

    void UnlockAccount(){
        if (!account_is_locked) return;
        account_is_locked = false;
        auth_retry_count = 0;
    }

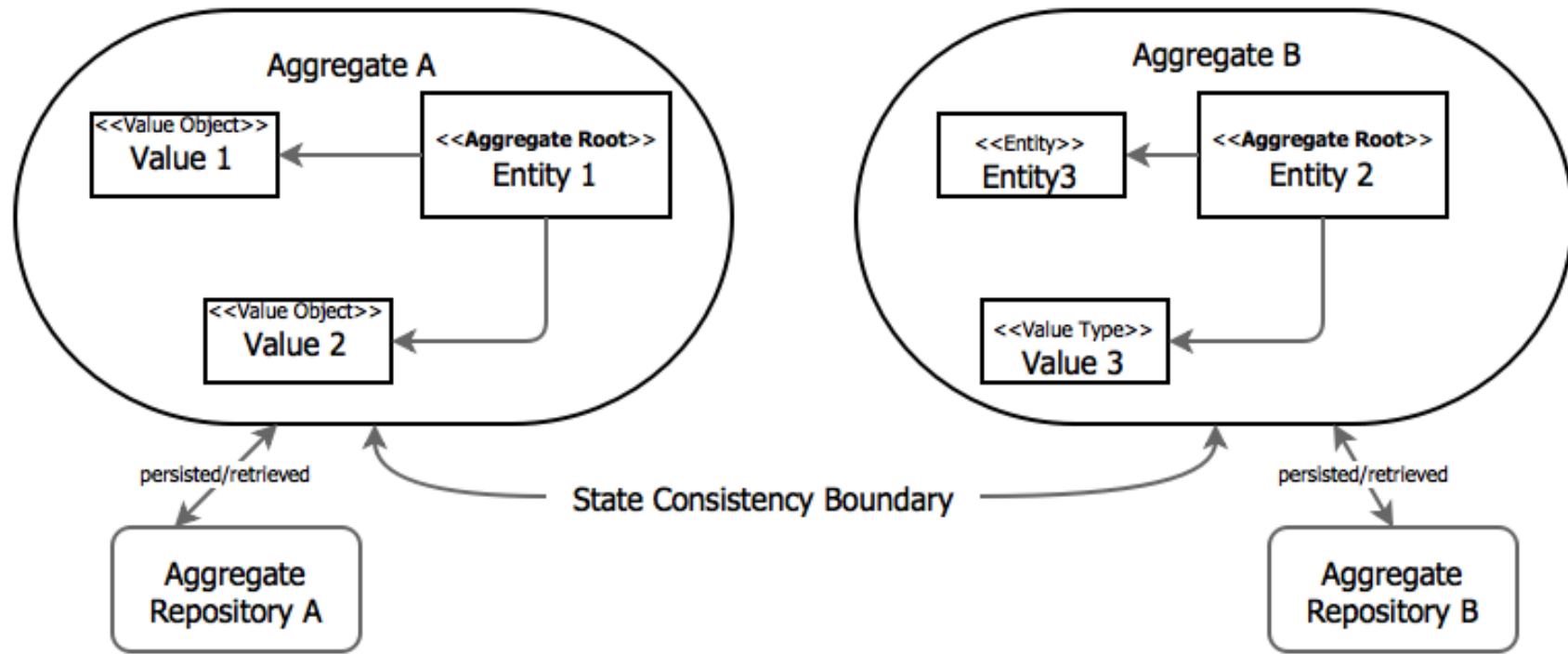
    void RegisterNewFailedAttempt(){
        auth_retry_count++;
        if (account_is_locked == auth_retry_count_limit) account_is_locked = true;
    }
};
```



Агрегаты

Aggregate – основной элемент DDD

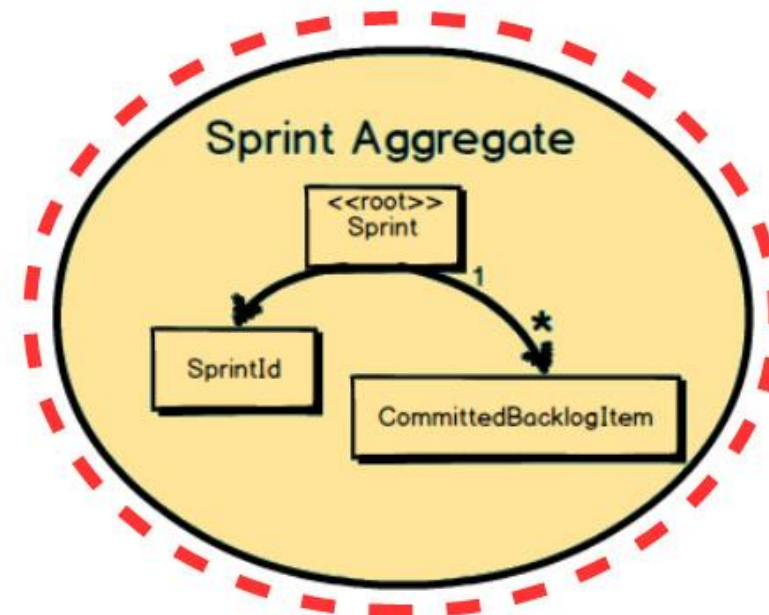
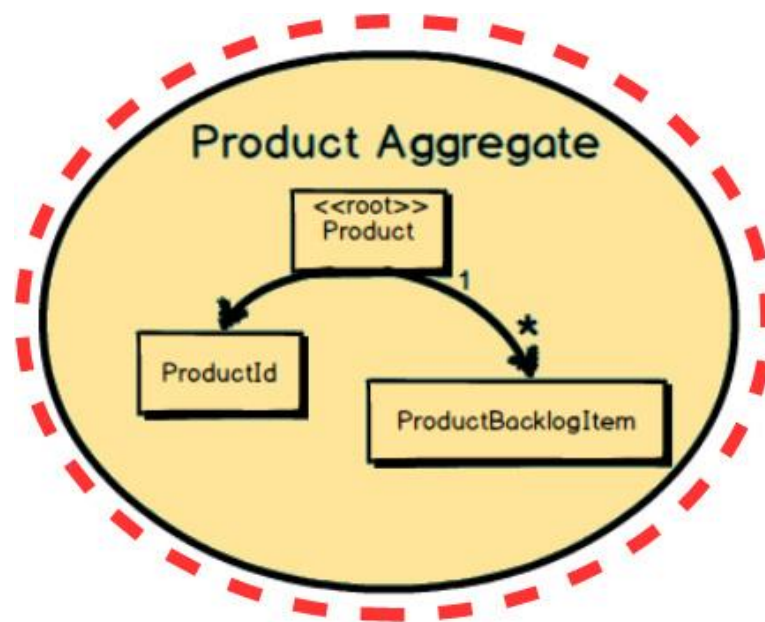
- **Aggregate** – главный паттерн тактического дизайна.
 - Всегда есть **Aggregate Root**, который **Entity**
 - Состоит из **Entities, Value Objects**
 - Сохраняет свое состояние в **Repositories**
 - Может генерировать **Domain Events**
 - Может быть создано с помощью **Factories**
 - Обрабатывается **Services**
- **Services** бывают
 - Application
 - Domain
 - Infrastructure
- **Services** располагаются в **Modules**

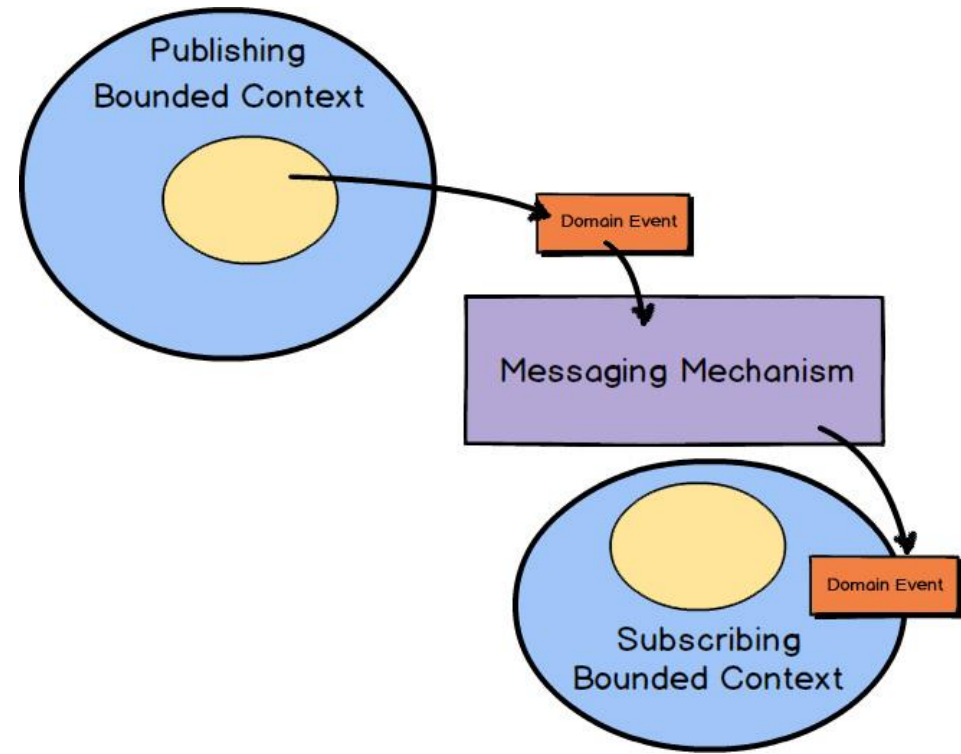
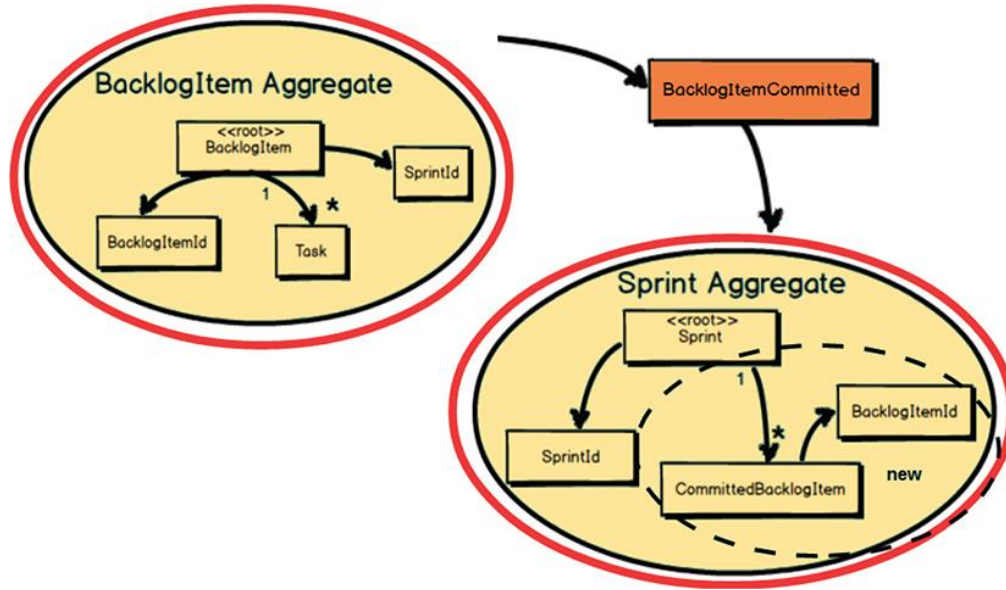


Aggregates

Правила выделения агрегатов

1. Храните бизнес-инварианты внутри границ агрегата.
2. Проектируйте небольшие агрегаты.
3. Ссылайтесь на другие агрегаты только по идентификатору.
4. Обновляйте другие агрегаты, используя конечную согласованность.





Обновление зависимых агрегатов

Микросервисы

- Удобно делать микросервисы, вокруг ограниченного контекста, работающего с выбранным агрегатом.
- Это обеспечивает хорошую изоляцию и независимость сервисов.

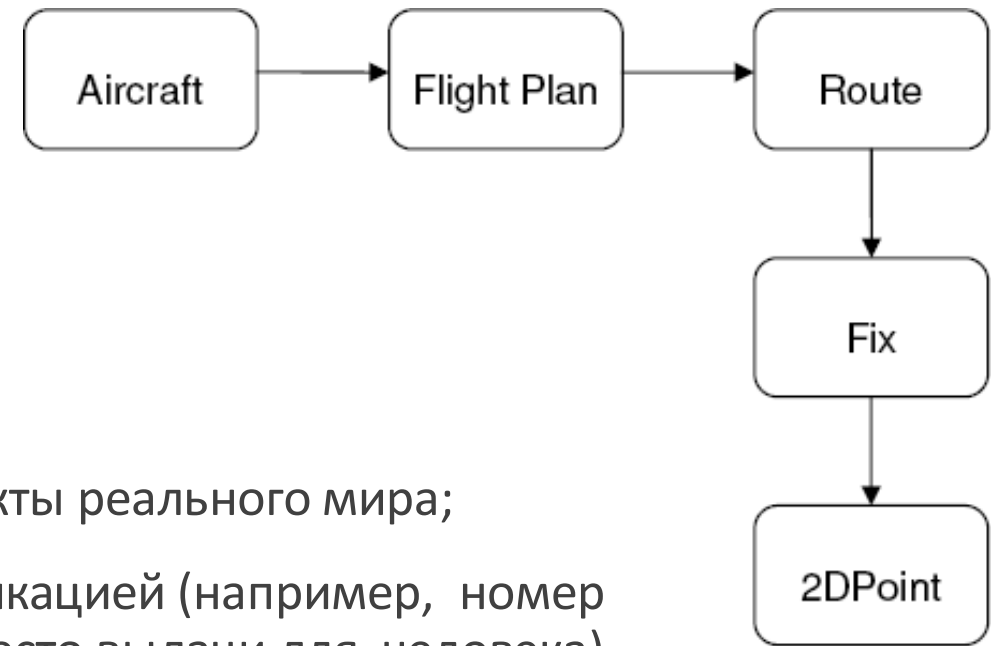
Entity

"When an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity. Define a means of distinguishing each object regardless of its form or history." (Evans*)

Taken from: http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf

Shared under Creative Commons Attribution 4.0 International License - <https://creativecommons.org/licenses/by/4.0/>

Сущности – объекты реального мира



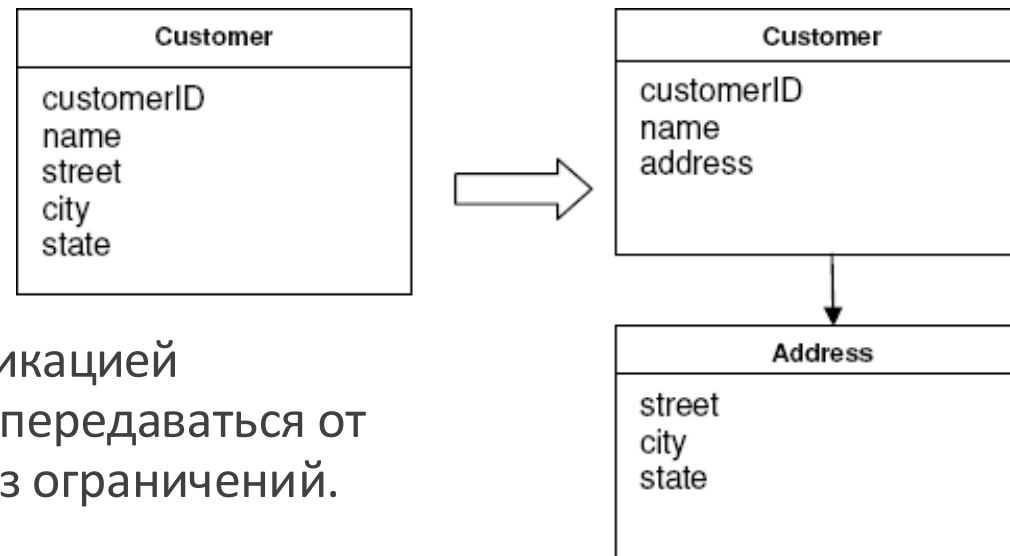
1. Представляют объекты реального мира;
2. Обладают идентификацией (например, номер паспорта + дата и место выдачи для человека)
3. Обладают явно выраженным жизненным циклом (процедурой создания, удаления, изменения состояния)
4. Обладают поведением.

VALUE OBJECT

When you care **only about the attributes** and logic of an element of the model, classify it as a value object. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the value object as immutable. Make all operations Side-effect-free Functions that don't depend on any mutable state. **Don't give a value object any identity and avoid the design complexities necessary to maintain entities.** (Evans*)

Taken from: http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf
Shared under Creative Commons Attribution 4.0 International License - <https://creativecommons.org/licenses/by/4.0/>

Объект- значение



1. Не обладают идентификацией
2. Могут копироваться и передаваться от функции к функции без ограничений.
3. Не обладают жизненным циклом Создаются когда надо и удаляются, когда ни кем не используются
4. Зачастую не меняют своих атрибутов Могут быть доступны для коллективного доступа из разных частей программы.
5. Атрибуты составляющие Value Object должны быть концептуально полными (соответствовать какой-либо абстракции)

Характеристики объекта- значения

- Он измеряет, количественно определяет или описывает что-либо в данной области
- Он может оставаться неизменным
- Он моделирует концептуальное целое, составляя связанные атрибуты как единое целое
- Он полностью заменяем при изменении измерения или описания.
- Его можно сравнивать с другими с помощью равенства значений
- Типичные примеры: Деньги, Электронная почта, Цвет, Uniqueld



Anemic Domain Model Anti-Pattern

Overview

- Термин, введенный Мартином Фаулером
- Anemic Domain Models (ADM) - это модели, объекты домена которых не выражают никакого поведения.
- Симптомы ADM:
 - Объекты имеют только геттеры и сеттеры
 - Большая часть (вся) бизнес-логики реализована в других объектах Service/Managers.
- Может использоваться только в CRUD-задачах

ADM Example

example_2

```
void saveClient( const std::string& client_id,
                const std::string& client_first_name,
                const std::string& client_last_name,
                const std::string& street_address,
                const std::string& phone_number,
                const std::string& email_address) {
    Client client; // = clientDao.readClient(client_id);
    if (!client_first_name.empty()) client.setClientFirstName(client_first_name);
    if (!client_last_name.empty()) client.setClientLastName(client_last_name);
    if (!street_address.empty()) client.setStreetAddress(street_address);
    if (!phone_number.empty()) client.setPhoneNumber(phone_number);
    if (!email_address.empty()) client.setEmailAddress(email_address);
    //clientDao.saveClient(client);
}
```

Ошибки

- Эксперты по доменам не могут здесь помочь, потому что код понятен только разработчикам.
- Интерфейс `saveClient()` не объясняет что в клиенте должно поменяться
- Реализация `saveClient()` добавляет скрытую сложность. Клиент может быть "сохранен" некорректно в неправильных ситуациях. Ограничения отсутствуют или скрыты (например, в базе данных).
- Клиентский "объект домена" является просто держателем данных.

Двигаемся в сторону
DDD
example_3

```
struct Client {  
    void changeName(const std::string& firstName, const std::string& lastName) {}  
    void relocateTo(const Address& address) {}  
    void changePhoneNumber(const Telephone& telephone){}  
    void emailAddress(const EmailAddress& emailAddress){}  
};  
  
struct ClientService {  
    void changeClientPhoneNumber(const std::string& clientId, const Telephone& telephone) {  
        Client client; // = clientRepository.getById(clientId);  
        client.changePhoneNumber(telephone);  
    }  
};
```

Factories

Создаем объекты

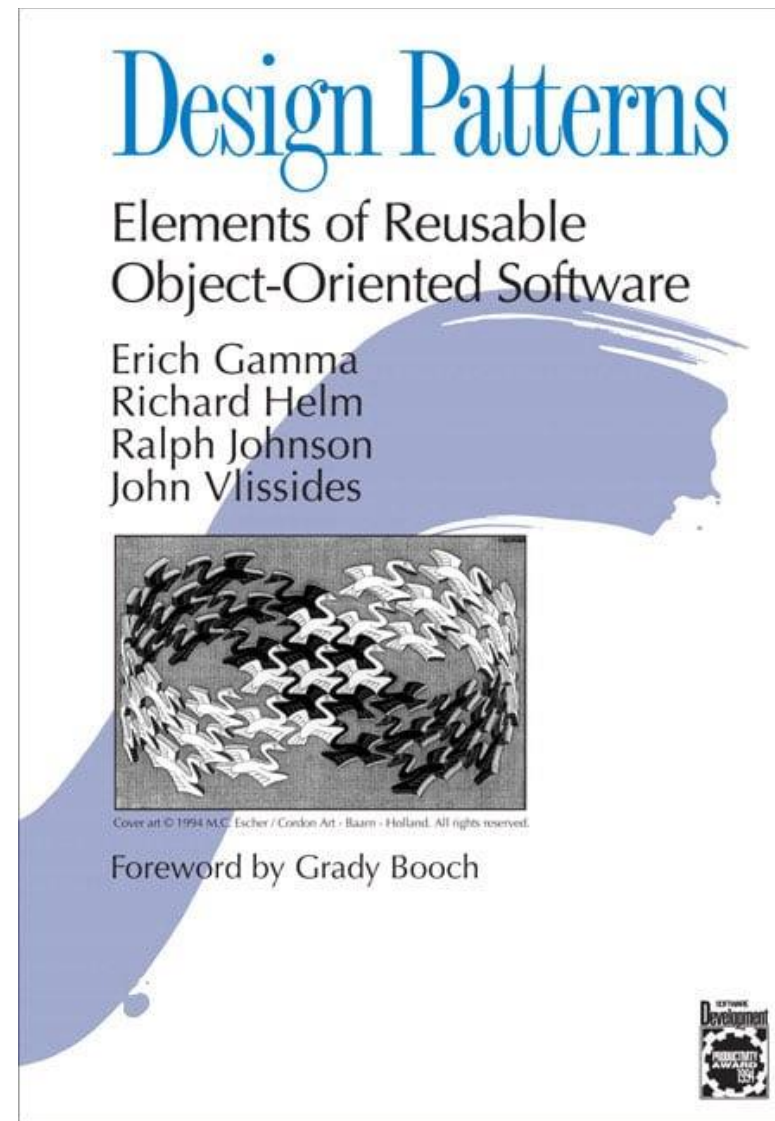


Описание

Shift the responsibility for creating instances of complex objects and AGGREGATES to a separate object, which may itself have no responsibility in the domain model but is still part of the domain design. Provide an interface that encapsulates all complex assembly and does not require the client to reference the concrete classes of the objects being instantiated. Create entire AGGREGATES as a piece, enforcing their invariants. (Evans*)

GoF patterns - Создание объектов

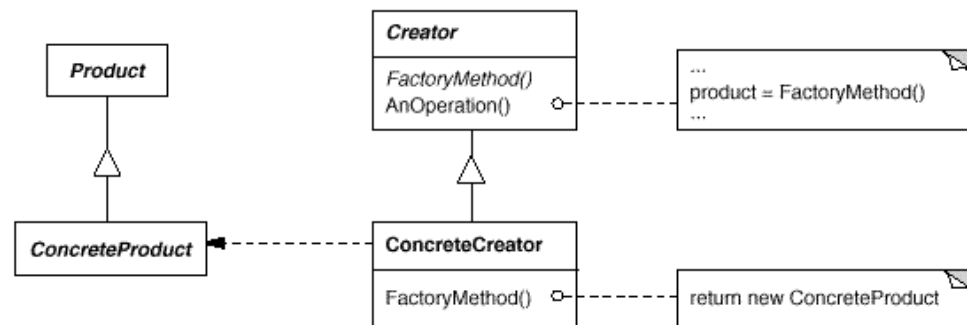
- Описывают способы как можно создавать объекты в программе.
- Борется с основным видом зависимости между модулями – знанием о существовании модуля и месте расположения.
- Самое серьезное препятствие лежит в жестко зашитой в код информации о том, какие классы инстанцируются.
- С помощью порождающих паттернов можно различными способами избавиться от явных ссылок на конкретные классы из кода.
- Паттерны:
 - Abstract Factory
 - Builder
 - **Factory Method**
 - Prototype
 - Singleton



Паттерны проектирования

создание объектов / Factory Method

- **Описание задачи**
Внутри метода класса нам нужно создавать экземпляр другого класса. При этом в наследниках, нам возможно понадобится создавать другие классы.
- **Описание решения**
Конструктор класса вызывается в специальном методе, который может быть переопределен в наследнике.
- **Когда применять**
Хорошо все вызовы **new** оформлять в виде отдельных методов, где это возможно.
- **Пример**
Приложение может использовать различные библиотеки для доступа к базе данных. Создание экземпляра объекта-библиотеки может определяться конфигурацией.



Factory Method

- Две основных разновидности паттерна.
 - Во-первых, это случай, когда класс Creator является абстрактным и не содержит реализации объявленного в нем фабричного метода.
 - Вторая возможность: Creator – конкретный класс, в котором по умолчанию есть реализация фабричного метода. Редко, но встречается и абстрактный класс, имеющий реализацию по умолчанию;
- Параметризованные фабричные методы.

Это еще один вариант паттерна, который позволяет фабричному методу создавать разные виды продуктов. Фабричному методу передается параметр, который идентифицирует вид создаваемого объекта. Все объекты, получающиеся с помощью фабричного метода, разделяют общий интерфейс Product..

Factory method

1. В Агрегате (в корне)
2. Отдельный сервис
3. Отдельный объект - фабрика

Factory в агрегате

example_4

```
struct Forum : public Entity
{
    std::string tenant(){
        return "current_tenant";
    }

    std::string forumId(){
        return "forum id";
    }

    Discussion startDiscussionFor(
        std::string aSubject,
        const Author& anAuthor,
        ForumIdentityService& aForumIdentityService)
    {
        return Discussion(
            tenant(),
            forumId(),
            aForumIdentityService.nextDiscussionId(),
            anAuthor,
            aSubject);
    }
};
```

Фабрика как отдельный сервис

example_5

```
struct TranslatingCollaboratorService : CollaboratorService
{
    Author authorFrom(Tenant aTenant, const std::string& anIdentity)
    {
        return Author(aTenant,anIdentity);
    }
};
```

https://github.com/VaughnVernon/IDDD_Samples/blob/master/idd_collaboration/src/main/java/com/saasovation/collaboration/port/adaptor/service/TranslatingCollaboratorService.java

Слои программного обеспечения

Слои

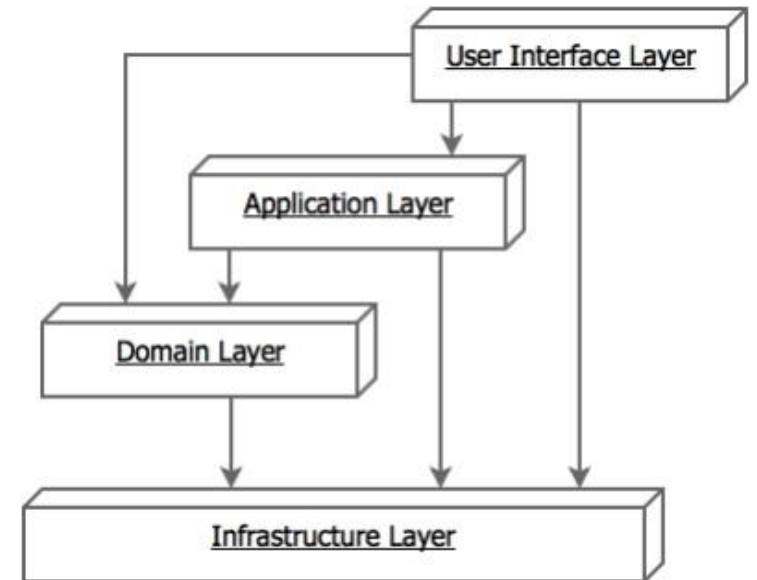
Isolate the expression of the domain model and the business logic, and **eliminate any dependency** on infrastructure , user interface, or even application logic that is not business logic. Partition a complex program into layers. Develop a design within each layer that is cohesive and that depends only on the layers below. (Evans*)

Слои программного обеспечения

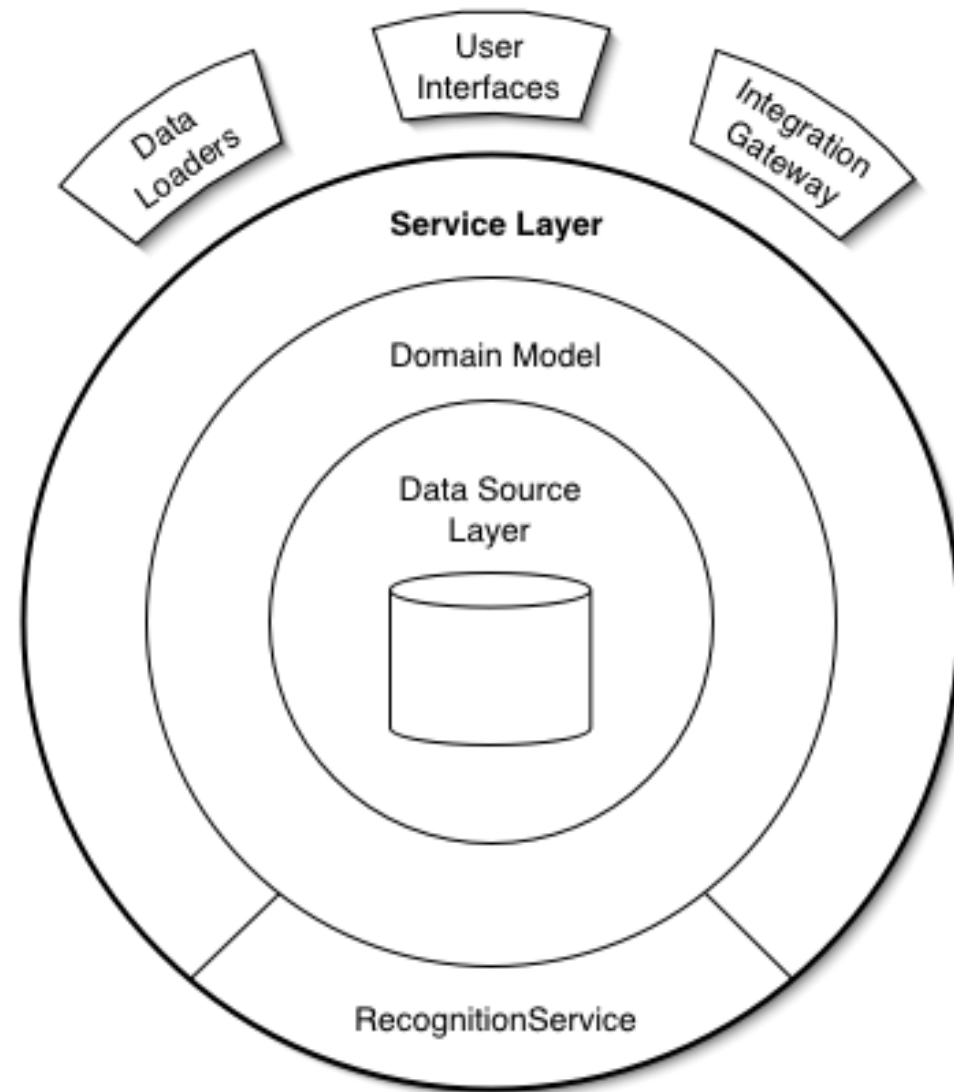
Разделение программного обеспечения на слои согласно выполняемым функциям с точки зрения программного обеспечения.

Например:

- ▲ Слой пользовательского интерфейса;
- ▲ Слой бизнес-логики;
- ▲ Слой информации о домене сущностей;
- ▲ Слой доступа к данным;



Слои



Свойства

- Поддерживает многоуровневые приложения
- Обычно используется в веб-приложениях, корпоративных и настольных приложениях.
- Каждый слой может соединяться только с самим собой и ниже
- Строгая архитектура слоев - позволяет связываться только со слоем, расположенным непосредственно под ним
- Расслабленная архитектура слоев - позволяет любому слою более высокого уровня связываться с любым слоем, расположенным ниже него
- Большинство систем основано на архитектуре расслабленных слоев
- Нижние слои могут не связываться с верхним слоем косвенно (наблюдатель, посредник)

Слой: Пользовательский интерфейс

- Пользовательский интерфейс – служит для взаимодействия с пользователем
- **Он не должен содержать доменную/бизнес-логику**
- Допускает **валидацию запросов** без глубокого знания правил домена
- Может **представлять клиентский API системы**, если пользователем является другая система

Слой: сервисов

- Построен на основе прикладных сервисов
- Выражает сценарии использования на модели
- Принимает параметры от уровня пользовательского интерфейса
- Прикладные сервисы - прямые клиенты доменной модели
- Обеспечивают функциональность отчетов для своих клиентов
- Не содержит бизнес-логики, только логику рабочего процесса, координируя обращения к Domain Layer (доменные сервисы, агрегаты)
- Управляет персистентностью транзакций, безопасностью, отправкой уведомлений о событиях в другие системы, составлением отправки электронных писем и т.д.
- Этот слой должен быть тонким, иначе это может привести к утечке доменной логики и анемичности доменной модели.

Сервисы

Основные свойства:

- ▲ Описывают процессы, работающие с Entity и Value Object в терминах предметной области
- ▲ Не имеют состояния;
- ▲ Не заменяют операции, которые принадлежат Entity, а дополняет их. Обычно сервис является важным процессом с точки зрения домена;

Разделяют сервисы, принадлежащие уровню домена и сервисы, принадлежащие инфраструктуре (например, доступ к данным)

Слой: Доменный слой

- Содержит только бизнес-логику
- Агрегаты, сущности, объекты значений, доменные службы, фабрики, доменные события

Слой: Инфраструктурный слой

- Различные инфраструктурные сервисы
- Работа с базами данных
- Нотификации
- Работа с сетью
- ...

Все кладем в модули



Modules

Состоят из элементов, логически связанных друг с другом.



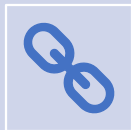
Агрегаты

Группа связанных объектов, которые могут рассматриваться как одно целое



Фабрики

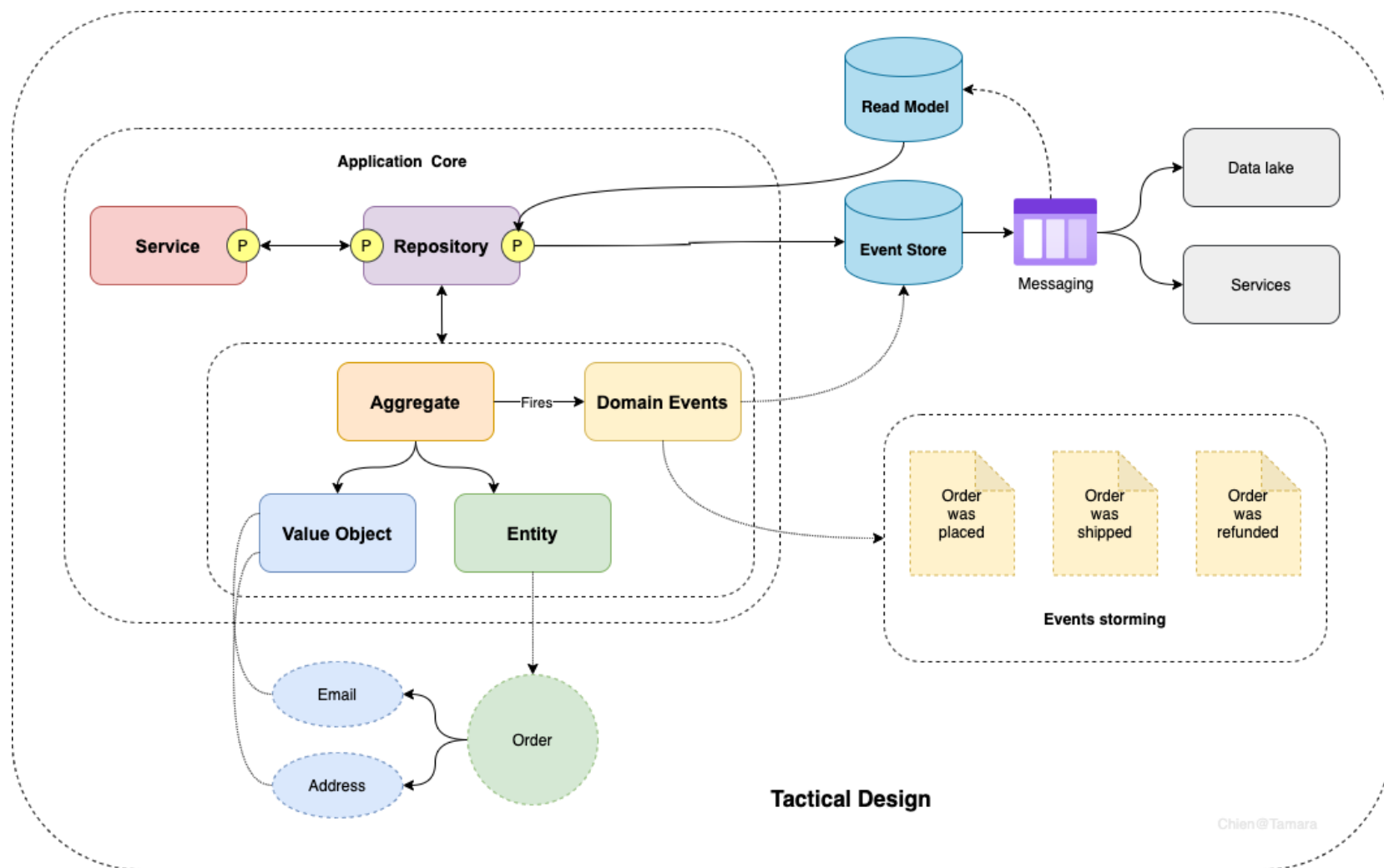
Инкапсулирую процесс создания сложных объектов или группы объектов



Репозитории

Реализуют логику получения ссылки на объекты предметной области по разным критериям

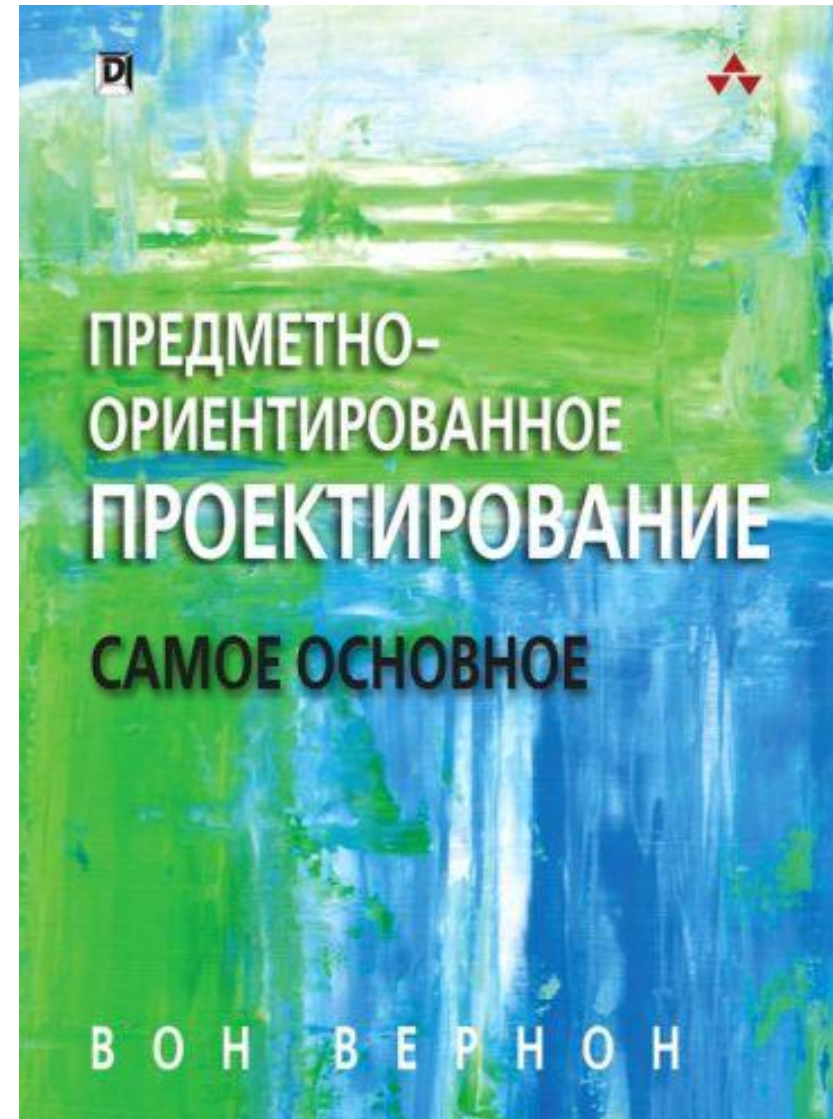
Итого



Итого

- DDD - это подход к разработке программного обеспечения, **основной акцент в котором делается на домене.**
- DDD подразумевает **тесное сотрудничество** между разработчиками и экспертами домена
- DDD в значительной степени **основан на повсеместном языке**
- DDD подразумевает стратегическое проектирование, ограниченный контекст является центральным паттерном DDD
- DDD поставляется с набором тактических инструментов для реализации домена в ограниченном контексте
- Анемичная модель домена - это анти-паттерн, очень часто используемый в неправильных местах

Что почитать



На сегодня все

ddzuba@yandex.ru