CARSMART 车网互联

领域驱动设计与模型驱动开发

钟玮军

2015年3月

致谢:

此培训材料借鉴了来自参考文献以及互联网的大量资料，部分资料的参考来源未能尽数列举，谨在此对那些在网络中无私分享自己知识的人表达我的衷心感谢!

培训内容

- 领域驱动设计简介
- 领域通用语言
- 领域驱动设计的构造块
- 领域驱动设计编程实践
- CQRS架构
- 模型驱动开发

领域驱动设计思想的发展

- 2002年Martin Fower在其出版《企业应用架构模式》中，归纳总结了40多种企业应用架构的设计模式。其中所提到的多种设计模式和概念，如事务脚本、活动记录和领域模型等，对业界产生了深远的影响。
- 2004年著名建模专家Eric Evans发表了他最具影响力的著名书籍：《Domain-Driven Design –Tackling Complexity in the Heart of Software》（中文译名：领域驱动设计—软件核心复杂性应对之道），书中提出了“领域驱动设计(简称DDD)”的概念。
- 2010年Greg Young在“CQRS, Task Based UIs, Event Sourcing agh!”一文中对Betrand Meyer的CQS模式进行改造，提出CQRS模式。
- 此后Jimmy Nilsson的《Applying Domain-Driven Design and Patterns》、Abel Avram和Floyd Marinescu合作的《Domain-Driven Design Quickly》、Dan Haywood的《Domain-Driven Design Using Naked Objects》、以及Vaughn Vernon的《Implementing Domain-Driven Design》等书籍的出版，丰富了领域驱动设计的实践和指导。

领域驱动设计是什么

- 领域驱动设计事实上针对是OOAD的一个扩展和延伸，DDD基于面向对象分析与设计技术，对技术框架进行了分层规划，同时对每个类进行了策略和类型的划分。
  - It’s a set of proven modeling techniques especially targeted to complex applications.
  - It’s a set of principles and practices supporting the development process.
  - It’s a set of patterns that support a clean and coherent view of the domain model.
  - It’s a set of pragmatic strategies allowing applications to scale in size and complexity maintaining their integrity.

领域驱动设计的特性

分层架构

- 成熟、清晰的分层架构
- 领域对象与现实世界的业务映射
- 明确的职责划分

复用

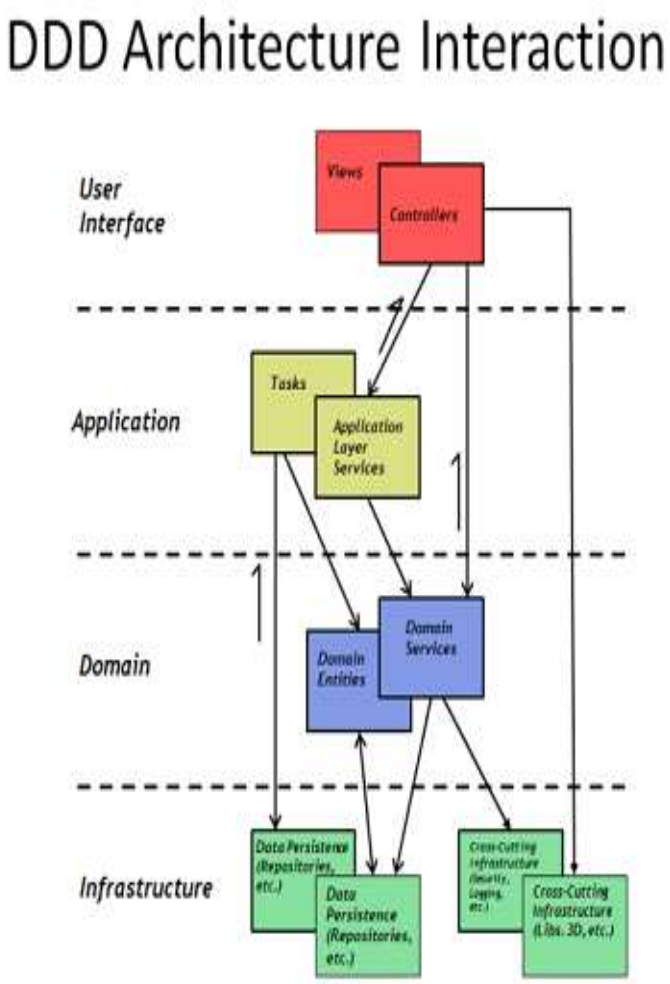
- 领域对象是核心
- 领域对象复用：完整的业务对象描述
- 设计复用：设计基于领域对象而非数据库

使用场景

- 具备复杂业务逻辑的软件开发
- 对设计和开发人员要求较高
- 不适用普通CRUD的业务
- 软件的维护性和扩展性良好 (Testable)

领域驱动设计分层规划（一）

- **领域驱动设计分层规划**



用户界面/展现层	负责向用户展现信息以及解释用户命令。展示层的组件实现用户与应用交互的功能。一般建议用MVC，MVP或者MVVM模式来分隔这些组件为子层
应用层	很薄的一层，用来协调应用的活动，实现协调应用的“通道”，例如事务、执行单位操作、调用应用程序的任务。它不包含业务逻辑。它不保留业务对象的状态，但它保有应用任务的进度状态。 类似于Façade模式，调用领域层和基础设施层来完成应用的用例。
领域层	本层包含关于领域的信息。这是业务软件的核心所在。在这里保留业务对象的状态，对业务对象和它们状态的持久化被委托给了基础设施层。
基础设施层	本层作为其他层的支撑库存在。它提供了层间的通信，实现对业务对象的持久化，包含对用户界面层的支撑库等作用。

领域驱动设计分层规划（二）

- 领域驱动设计是对传统N层架构模式的继承和发展

Traditional N-Layer architecture (Logical)

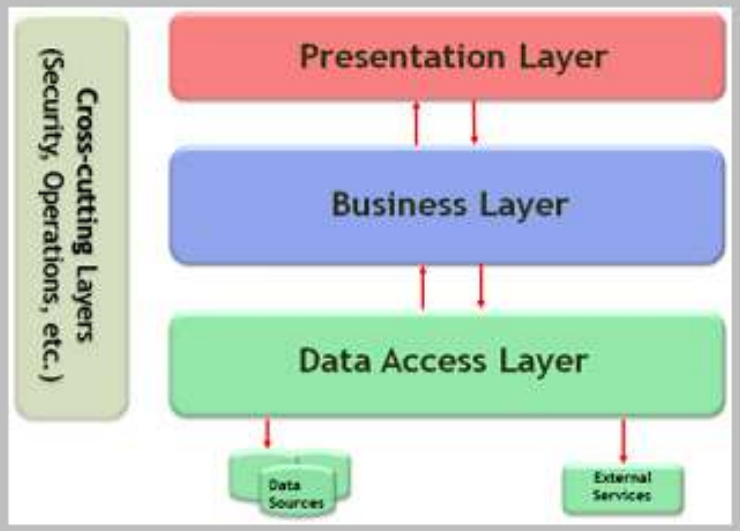


Figure 1.- *Traditional N-Layer architecture (Logical)*

3-Tier Architecture (Physical)

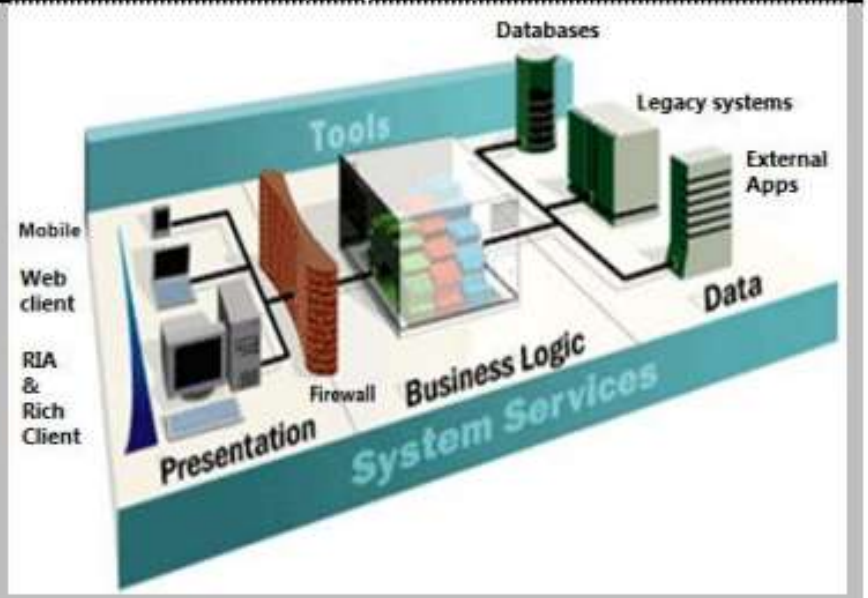
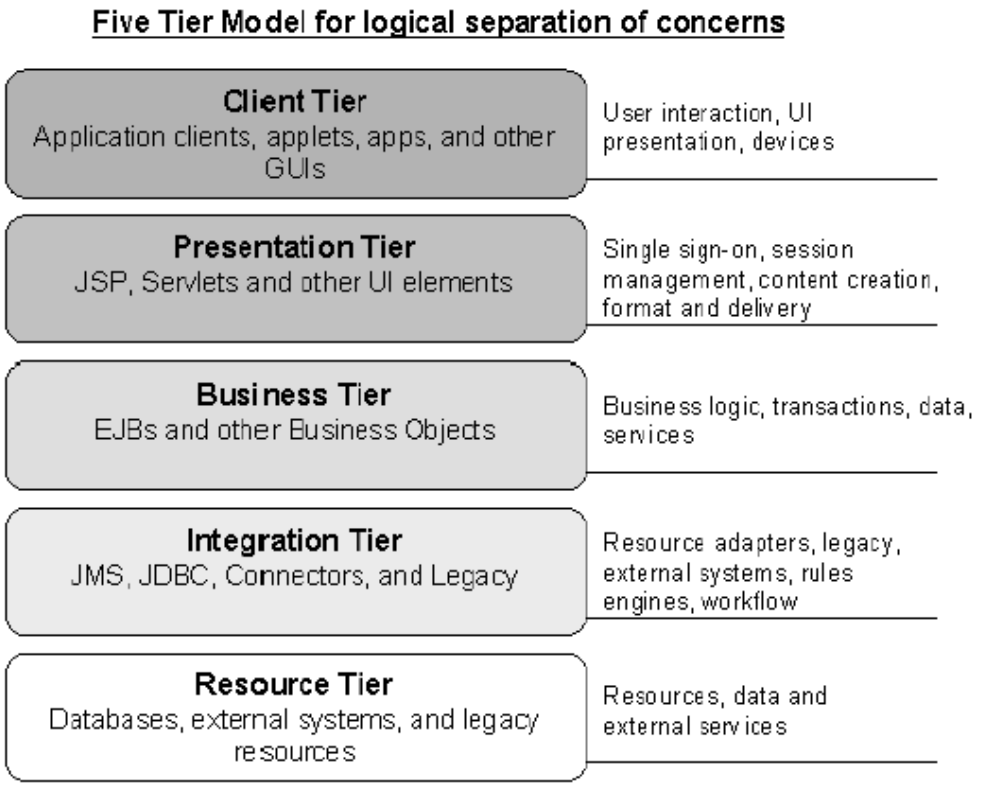


Figure 2.- *Tier architecture (Physical)*

领域驱动设计分层规划（三）

- **领域驱动设计是对传统N层架构模式的继承和发展**

例：J2EE参考分层架构

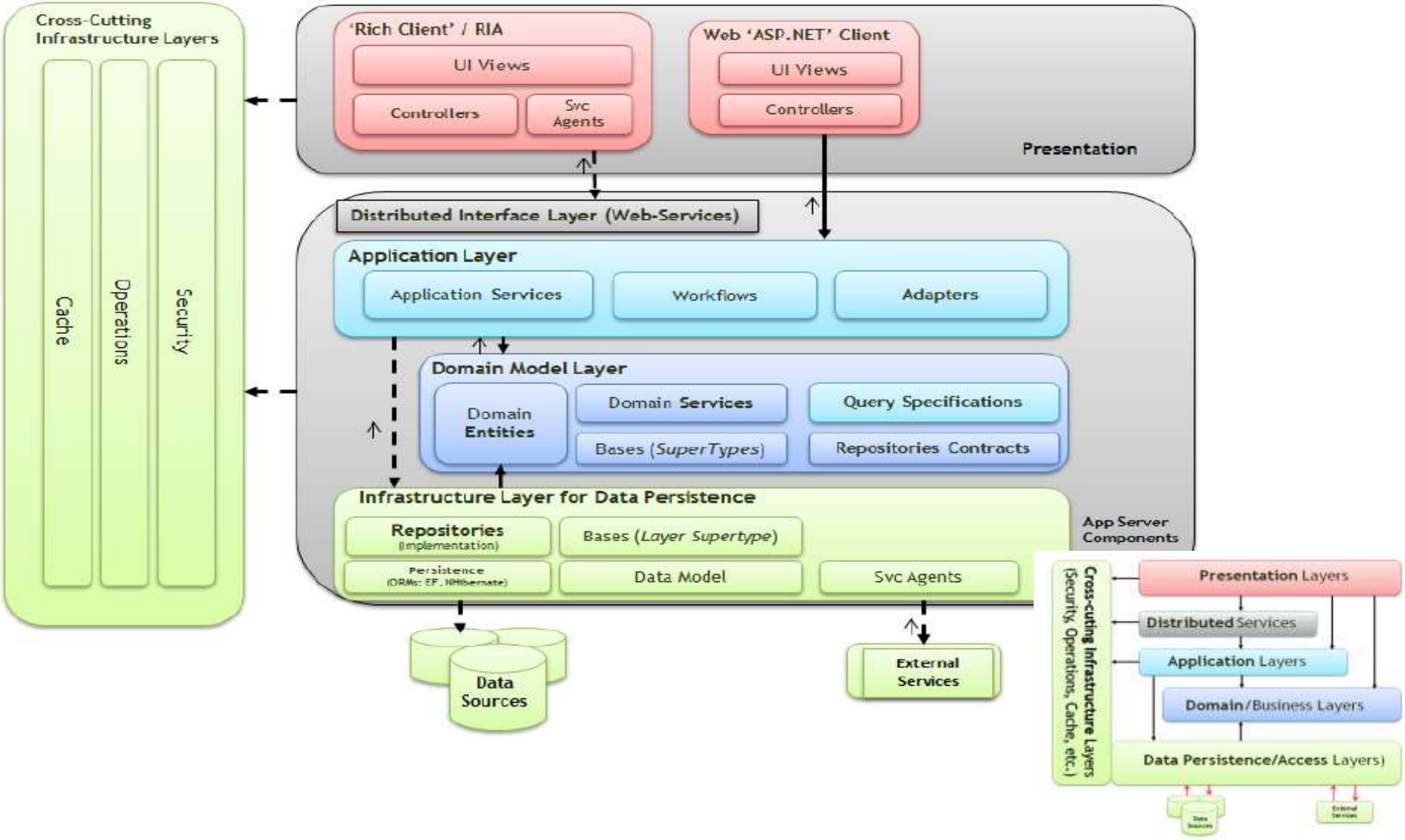


《Core J2EE Patterns》

传统J2EE或Spring+Hibernate等事务性编程模型只关心数据，这些数据对象除了简单sette/getter方法外，没有任何业务方法，被比喻成“失血模型”。

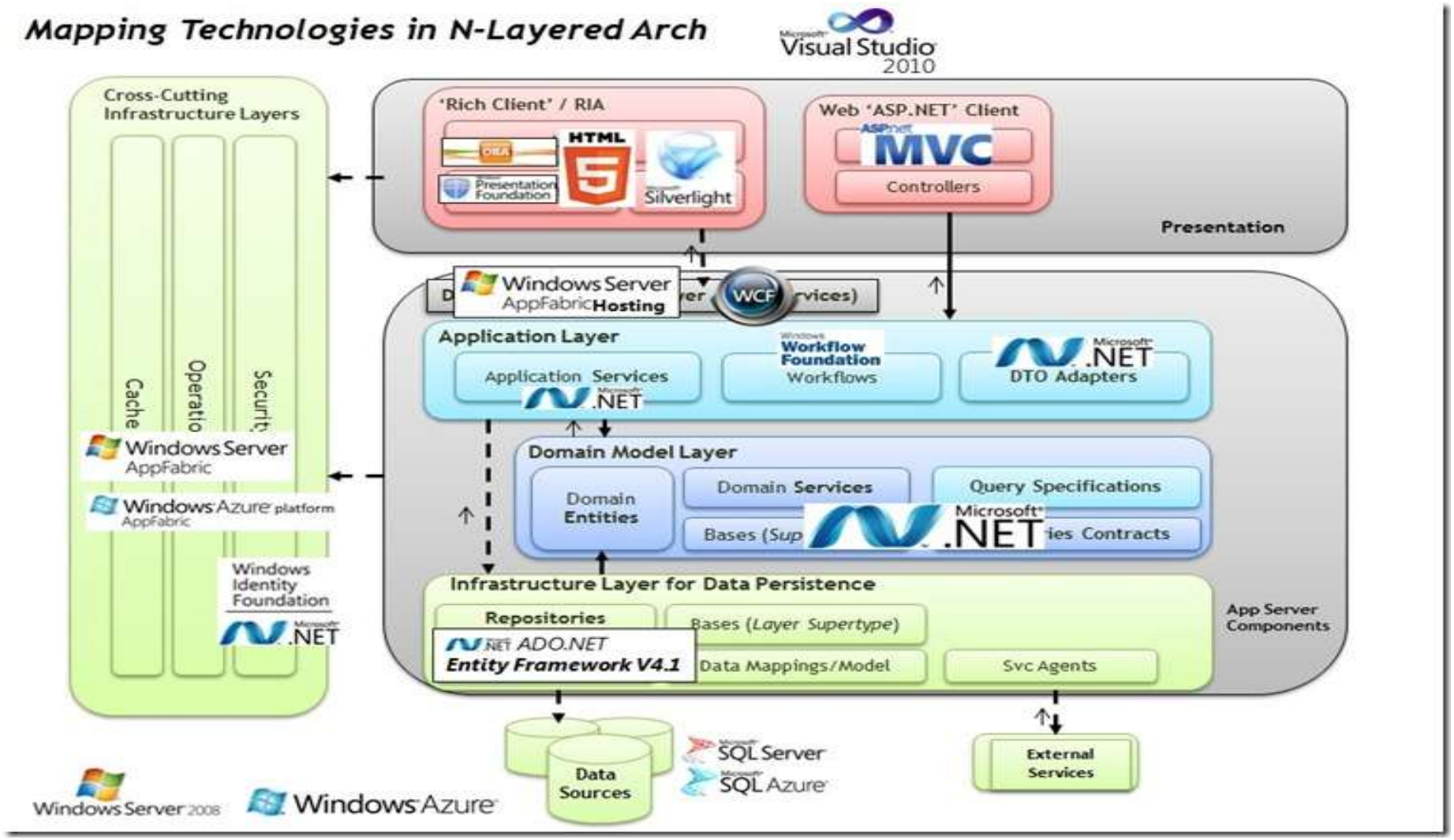
领域驱动设计分层规划（四）

- 分布式领域驱动设计



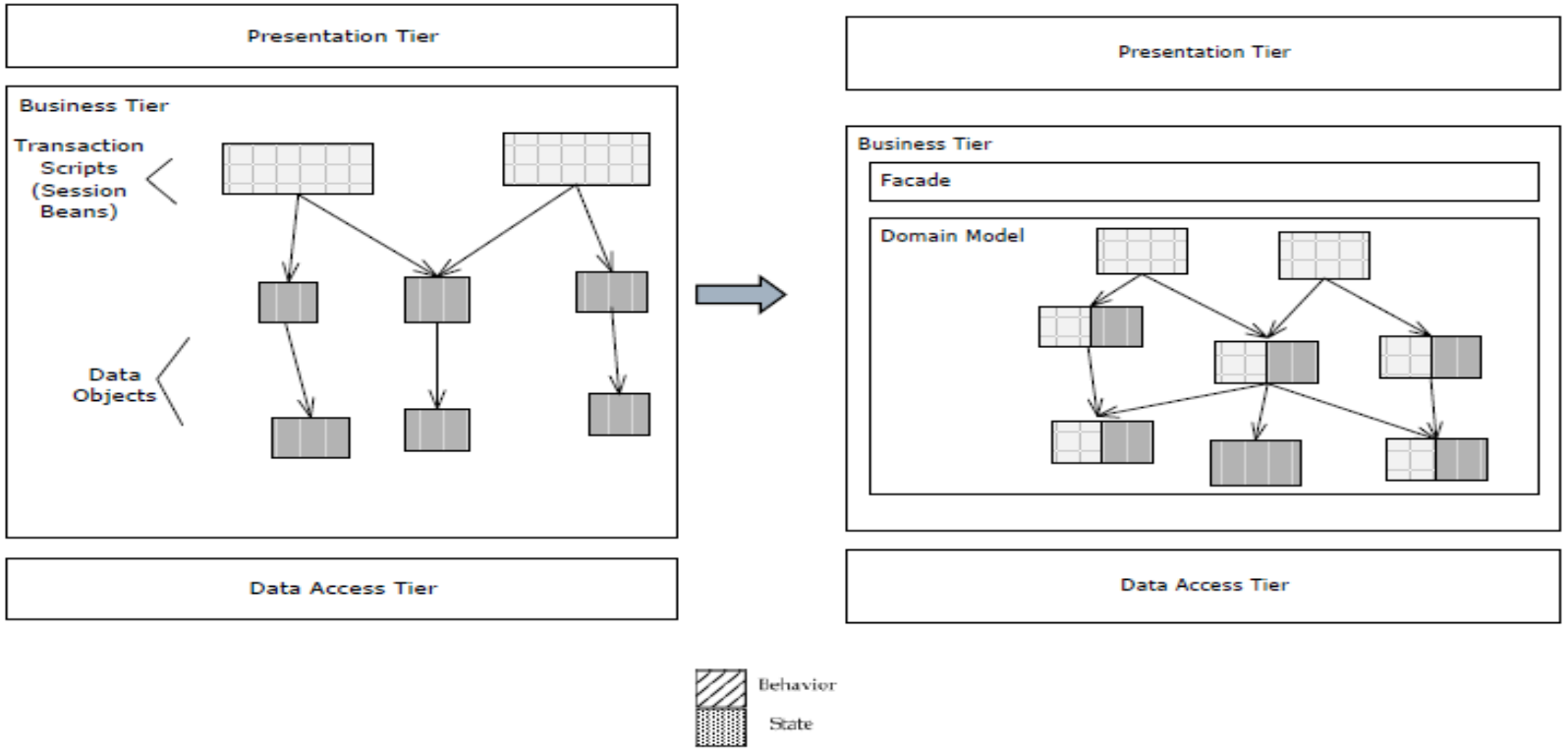
领域驱动设计分层规划（五）

- 分布式领域驱动设计与DotNET技术架构体系之间的关系映射



面向对象分析与设计技术

- **面向过程vs.面向对象**



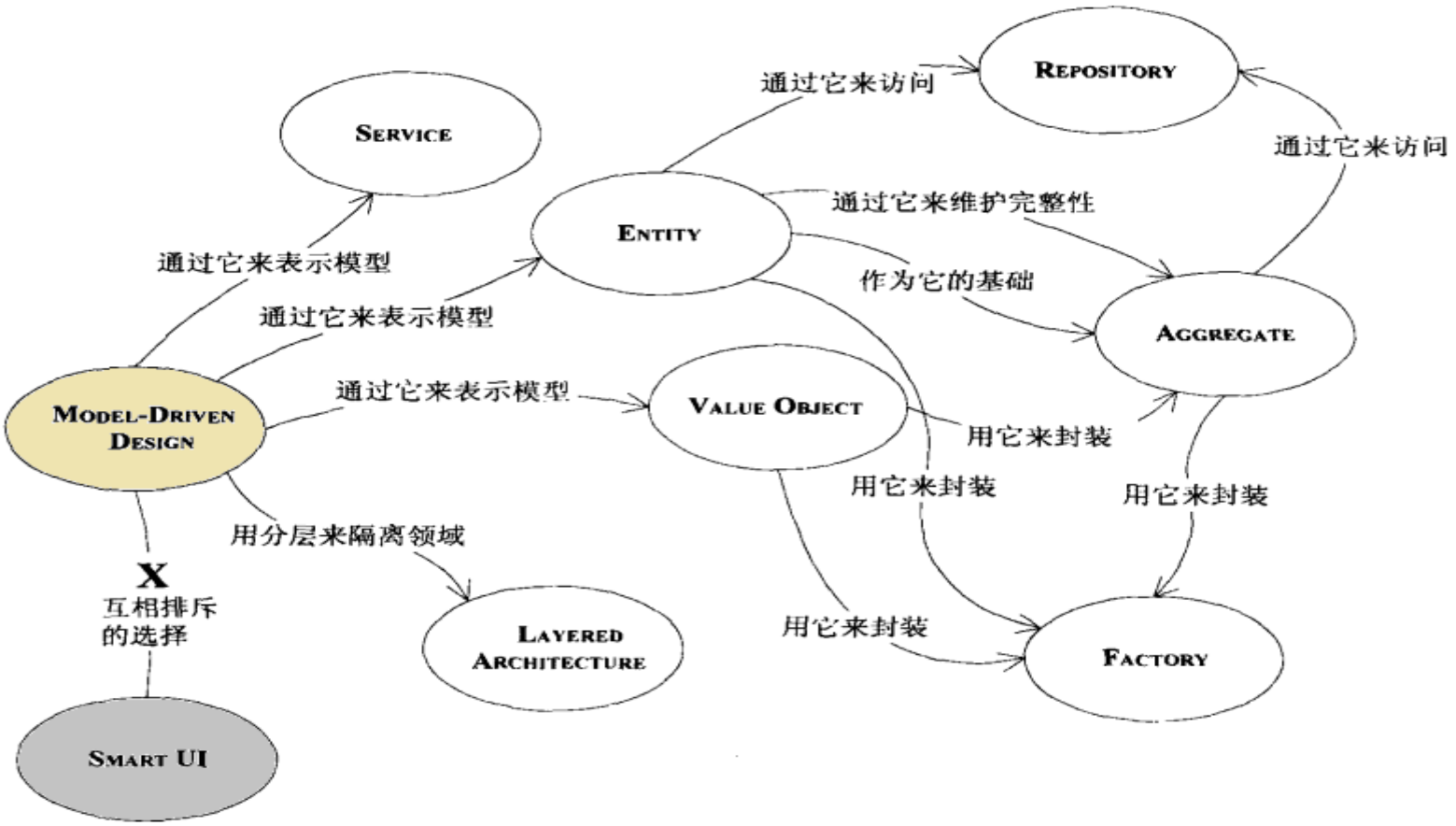
事务脚本模式把业务逻辑组织成单个过程，在过程中直接调用数据库，业务逻辑在服务(Service)层处理。

事务脚本模式的特点是简单容易理解，面向过程设计。对于少量逻辑的业务应用来说，事务脚本模式简单自然，性能良好，容易理解，而且一个事务的处理不会影响其他事务。

不过缺点也很明显，对于复杂的业务逻辑处理力不从心，难以保持良好的设计，事务之间的冗余代码不断增多，通过复制粘贴方式进行复用。可维护性和扩展性变差。

对类的策略和类型的划分

- **按照策略和类型对类进行划分**



对类进行StereoType(“构造型”)划分的好处在于：

（1）指导设计

（2）帮助命令对象

（3）辅助理解

六边形架构

- 以领域模型为核心的六边形架构

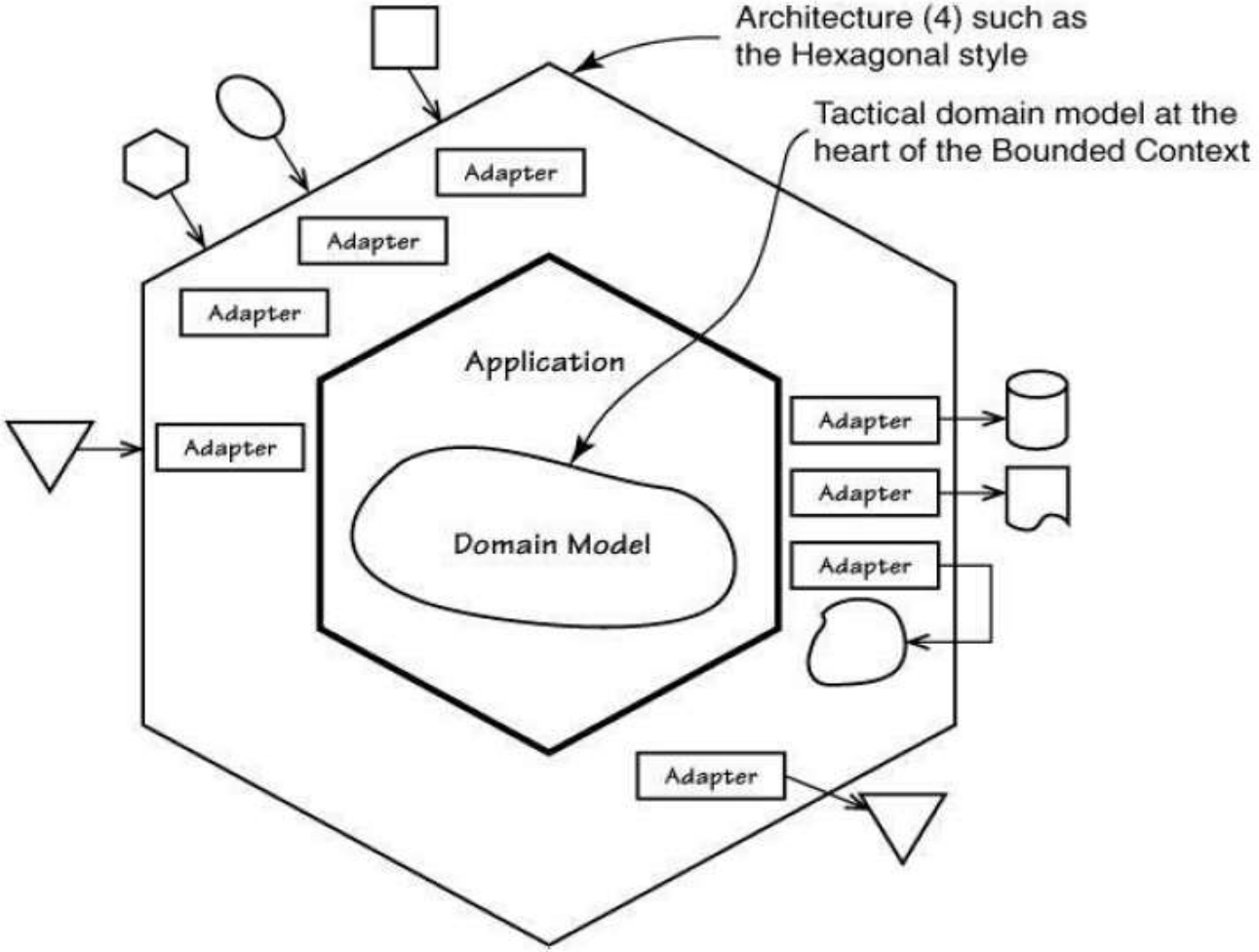
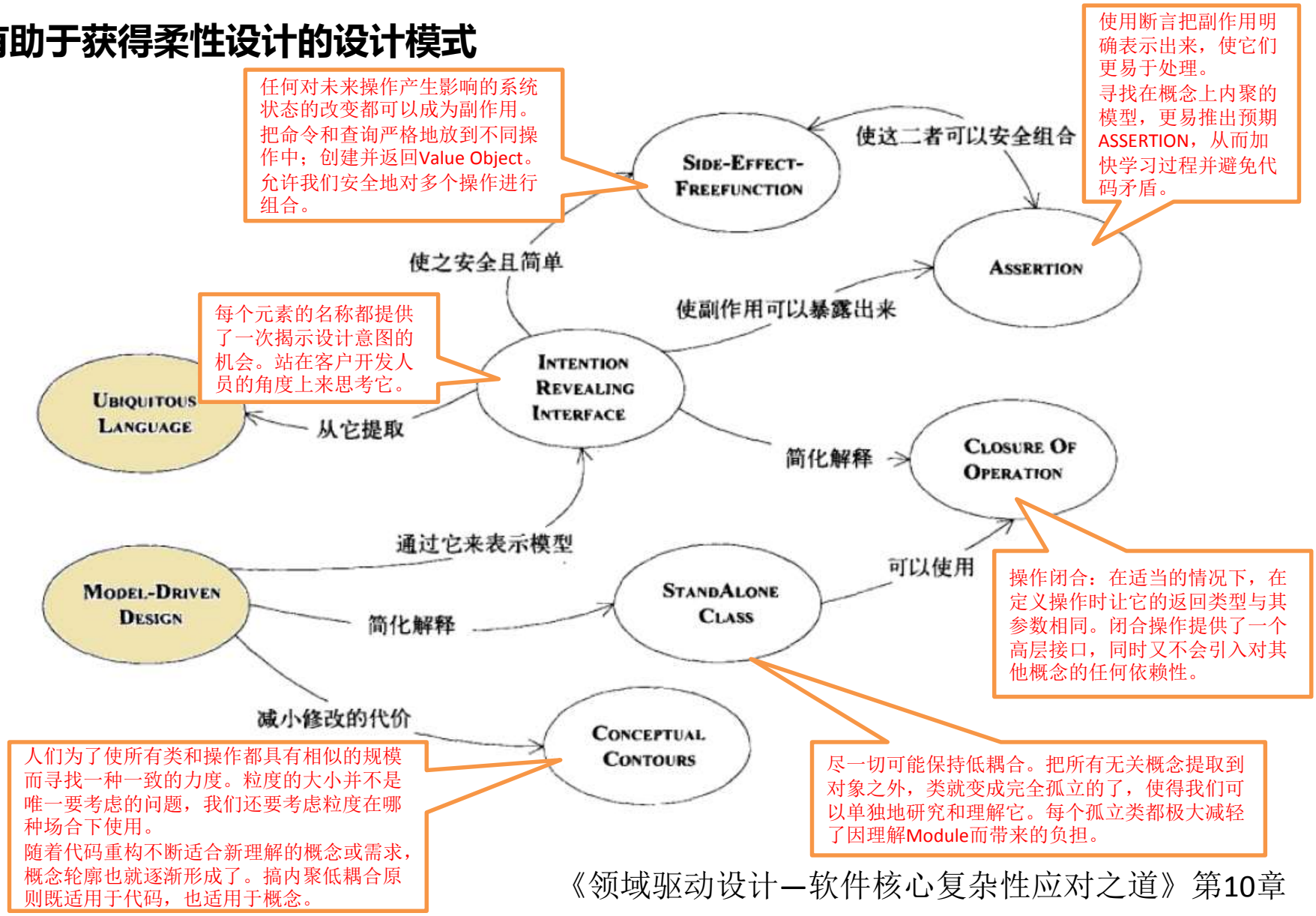


Figure G.3. The Hexagonal Architecture with the domain model at the heart of the software

领域驱动设计中的设计模式

- **有助于获得柔性设计的设计模式**



《领域驱动设计—软件核心复杂性应对之道》第10章

培训内容

- 领域驱动设计简介
- 领域通用语言
- 领域驱动设计的构造块
- 领域驱动设计编程实践
- CQRS架构
- 模型驱动开发

使用通用语言的重要性

- Talking different languages makes projects fail.
  - Programmers speak using technical jargon (design patterns, acronyms, geeky in-jokes)
  - Domain experts use terminology specific to their field of expertise
  - Computers speak programming languages
  - 大家必须妥协

领域驱动设计的关键点

- 关注核心领域(Core Domain)
- 领域专家和软件从业者共同开发模型
- 在一个明确的限界上下文(Bounded Context)中使用领域通用语言(ubiquitous language)

通用语言（一）

- 通用语言（UBIQUITOUS LANGUAGE）是团队共享的语言。领域专家和开发者使用相同的通用语言进行交流。事实上，团队中每个人都使用相同的通用语言。不管你在团队中的角色如何，只要你是团队的一员，你都将使用通用语言。
  - 通用语言是团队自己创建的公用语言。团队中同时包含领域专家和软件开发人员。
  - 通用语言更多地是关于业务本身如何思考和运作的，领域专家对通用语言有很大影响。不同领域专家会在概念和术语上产生分歧，甚至也会犯错，当领域专家和开发者一起创建领域模型的时候，他们有时会达成一致，有时会做一些妥协，但最终目的都是为了创造最适合项目的通用语言。团队成员们妥协的绝对不应是通用语言的质量，而是概念、术语和含义。最初的一致并不表示始终一致，通用语言也会随着时间推移而不断演化改变。
  - 领域驱动设计的一个核心思想就是使用基于模型的共同语言。因为模型是软件满足领域的共同点，它很适合作为这种通用语言的构造基础。使用模型作为语言的核心骨架，要求团队在进行所有的交流都是使用一致的语言，在代码中也是这样。在共享知识和推敲模型时，团队会使用语言、文字和图形。这儿需要确保团队使用的语言在所有的交流形式中看上去都是一致的，这种语言被称为“通用语言（Ubiquitous Language）”。
  - 通用语言的词汇表包括类名称和主要操作。语言中包含术语，有些术语用来讨论模型中已经明确的规则，还有一些术语则来自施加于模型上的高级组织原则。最后，团队一致应用于领域模型的模式名称使这种语言更为丰富。模型之间的关系成为所有语言都具有的组合规则，词和短语的意义反映了模型的语义。

通用语言（二）

- 在应用通用语言时，应注意：
  - 将模型作为语言的中心。确保团队在所有交流活动和代码中坚持使用这种语言。在画图、写东西特别是讲话时也要使用这种语言。
  - 通过尝试不同的表示方法（它们反映了不同模型）来消除难点。然后重构代码，并对类、方法和模块重新命名，以便与新模型相一致。解决交谈中的术语混淆问题，就像我们对普通词汇形成一个公认的理解一样。
  - 要认识到UBIQUITOUS LANGUAGE中的更改就是对模型的更改。
  - 领域专家应该避免使用拗口或无法表达领域理解的术语或结构，开发人员应该密切监视那些将会妨碍设计的有歧义和不一致的地方
- 有了通用语言，模型就不仅仅是一个设计工作了。它成为开发人员和领域专家共同完成的每项工作中的不可或缺的部分。语言以动态形式传递知识。使用这种语言进行讨论能够更清楚地表达图和代码背后的真实含义。
- 通用语言是那些不以代码形式出现的设计方面的主要载体，这些方面包括把整个系统组织在一起的比例结构、定义了不同系统和模型之间关系的Bounded Context，以及在模型和设计中使用的其他模式。

通用语言的应用

- 通用语言贯穿于项目的各个环节
  - User Stories
  - Project Meetings
  - Team Emails
  - Instant Messages
  - Schedule Plan
  - Software Documents
- 在限界上下文中，保持语言的一致性（如口语、图形（如UML图等）、文字、代码等）。

通用语言的应用示例（一）

- User Stories

NO

When User logs on with valid credentials, an empty panel is displayed.

YES

When Player logs on with valid credentials, an empty board game is displayed.

(from a Tic Tac Toe Game software example)

通用语言的应用示例（二）

- Code Example

NO

. Integer i = new Integer();
. String char1 = new String();
. public class GameDAO() { }
. catch (Exception e)

NO

. Ambiguities
. Inconsistencies
. Synonyms
. Abbreviations

YES

. String realMeaningOfMyString = new String();

. public class ScoreDataLoader() { }

. catch (Exception NotLoggedInException)

YES

. Clarity
. Precision
. Reuse
. Full Names

A class BEFORE and AFTER Ubiquitous Language

```
package tictactoe.client.userInterface;

/**
* Add the string O or X to a cell in the grid.
*/
public class ShowCellGrid{

public static void displayUser (Grid grid, Cell cell) {

        if (!Initialization.flag
        && Initialization.gameStatus.getSequence() == null
        && isEmpty(grid, cell)) {

        Initialization.flag = true;

        String mk= showString(Initialization.gameStatus
        .getCurrentUser().getUserString());

        grid.setHTML(cell.getRowIndex(), cell.getCellIndex(), mk);

        Initialization.gameStatus.getStatus()[cell.getRowIndex()][cell
        .getCellIndex()] = Initialization.gameStatus
        .getCurrentUser();

        GameEnd.checkEnd(Initialization.gameStatus,
        cell.getRowIndex(), cell.getCellIndex());
        }

(...)
}
```

```
package tictactoe.client.userInterface;

/**
* Performs a move in the game.
*/
public class PlayerMove {

/**
* When the player clicks in a cell, the game draws an O or a X on the
* game grid depending on which player's turn it is.
*/
public static void makeMove (GameGrid gameGrid, Cell cell) {

        if (!GameInitialization.waitingMoveFlag
        && GameInitialization.currentGameStatus.getSequenceWinner() ==
        null && isCellEmpty(gameGrid, cell)) {

        GameInitialization.waitingMoveFlag = true;

        String marker =
        showPlayerIcon(GameInitialization.currentGameStatus
        .getCurrentPlayer().getPlayerIcon());

        gameGrid.setHTML(cell.getRowIndex(), cell.getCellIndex(), marker);

        GameInitialization.currentGameStatus.getGameMoves()[cell.getRow
        Index()][cell.getCellIndex()] =
        GameInitialization.currentGameStatus.getCurrentPlayer();

        CheckWinner.checkForWinner(GameInitialization.currentGameStatu
        s, cell.getRowIndex(), cell.getCellIndex());
        }

(...)
}
```

(Excerpted from a Tic Tac Toe Game source code)

Which one would a Stakeholder better understand?

Show Cell Grid
Add the String O or X to a cell in the grid.

Display User
Is Empty

Player Move
Performs a move in the game.

Make Move
When the player clicks in a cell, the game draws an O or a X on the game grid depending on which player's turn it is.
Is Cell Empty
The Player can select a cell only if it wasn't already selected.

(Excerpted from a Tic Tac Toe Game source code)

模型的统一

- 模型的内部一致性又叫做“统一”，这样每个术语都不会有模棱两可的意义，也不会有规则冲突。除非模型在逻辑上是一致的，否则它就没有意义。
- 识别限界上下文中的不一致：**重复的概念**和**假同源**
  - 重复的概念是指两个模型元素（以及伴随的实现）实际上表示同一个概念。每当这个概念的信息发生改变时，都必须要更新两个地方。每次由于新的知识导致一个对象被修改时，也必须重新分析和修改另一个对象。如果不进行实际的重新分析，结果就会出现同一个概念的两个版本，它们遵守不同的规则，甚至不同的数据。更重要的是，团队成员必须学习同一操作的两种方法，以及保持这两种方法同步的各种方式。
  - 假同源是指使用相同术语（或已实现的对象）的两个人认为他们是在谈论同一件事情，但实际上并不是这样。但是，当两个定义都与同一个领域方面相关，而只是在概念上稍有区别时，这种冲突更难以发现。假同源会导致开发团队互相干扰对方的代码，也可能导致数据库中含有奇怪的矛盾，还会引起团队沟通的混淆。
- 注意用词词汇
  - 注意正确用词，不要歪曲词义
  - 开发人员经常习惯于使用增/删/改/查（CRUD）此类动词词汇，也许有时候它们也确实属于通用语言，但大多数情况下，它们并不能正确反映业务，用词上混淆了业务概念。

模型的分裂

- 在理想的世界中，我们可以有一种把整个企业领域包含进来的单一模型；这个模型将是统一的，没有任何相互矛盾或相互重叠的术语定义；每个有关领域的逻辑声明都将是一致的。但大型系统开发并不是这样理想。
- 大型系统领域模型的完全统一是不可行的，也不是一种经济有效的做法。我们可以采用**限界上下文**（Bounded Context）定义每个模型的应用范围，采用**上下文映射**（Context Map）给出项目上下文以及它们之间关系的总体视图。
  - 任何一个大型项目都会存在多个模型。而当基于不同模型的代码被组合到一起后，软件就会出现bug、变得不可靠和难以理解。团队成员之间的沟通变得混乱。人们往往弄不清楚一个模型不应该在哪个上下文中使用。
  - 明确地定义模型所应用的上下文。根据团队的组织、软件系统的各个部分的用法以及物理表现（代码和数据库模式等）来设置模型的边界。在这些边界中严格保持模型的一致性，而不要受到边界之外问题的干扰和混淆。在Context中，要保证模型在逻辑上统一，而不用考虑它是不是适用于边界之外的情况。在其他Context中，会使用其他的模型，这些模型具有不同的术语、概念、规则和UBIQUITOUS LANGUAGE的技术行话。
  - 定义Bounded Context：视察项目的现状，而不是它的理想状态。

领域、子域和限界上下文

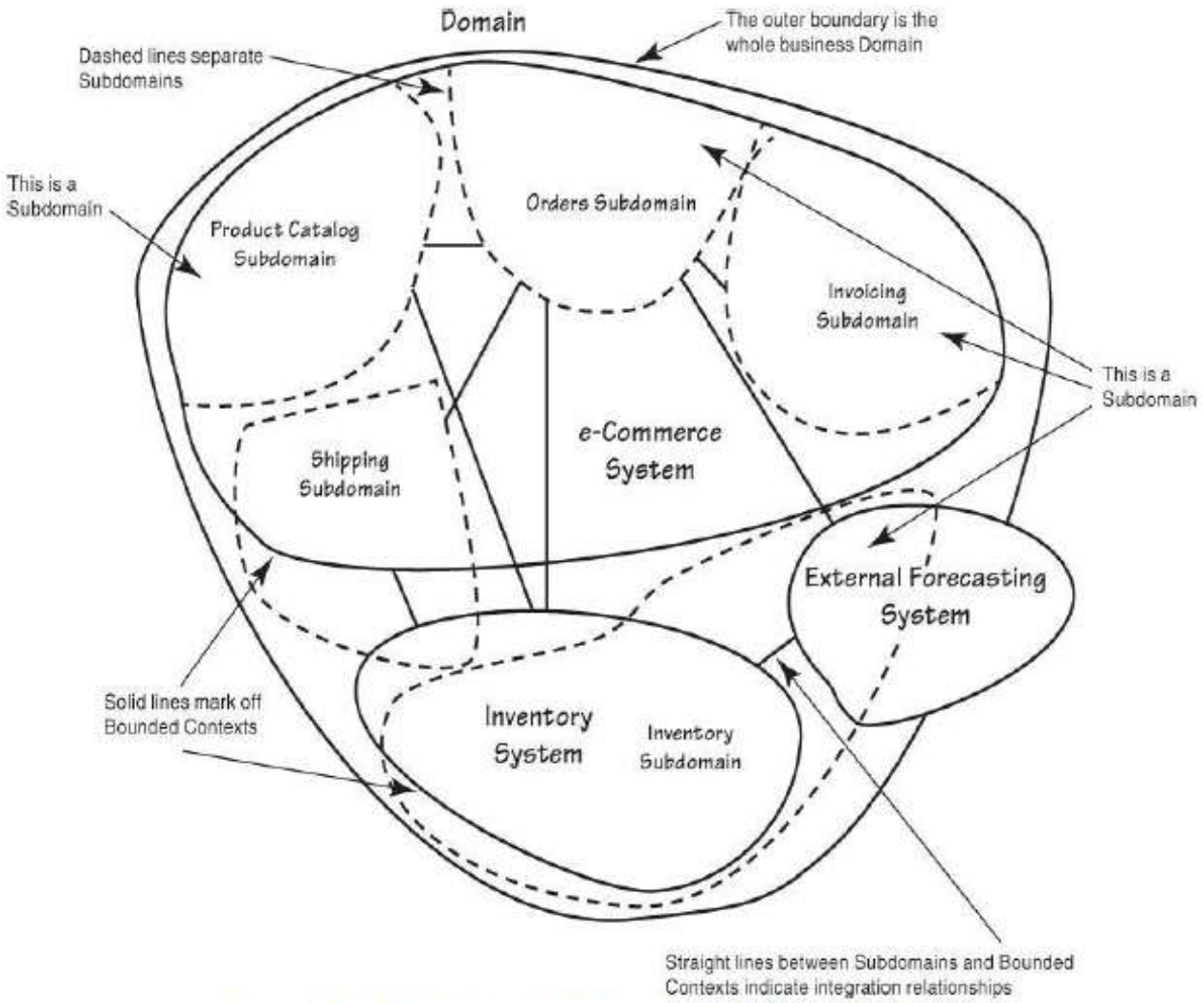


Figure 2.1. A Domain with Subdomains and Bounded Contexts

核心域、支撑域和通用域

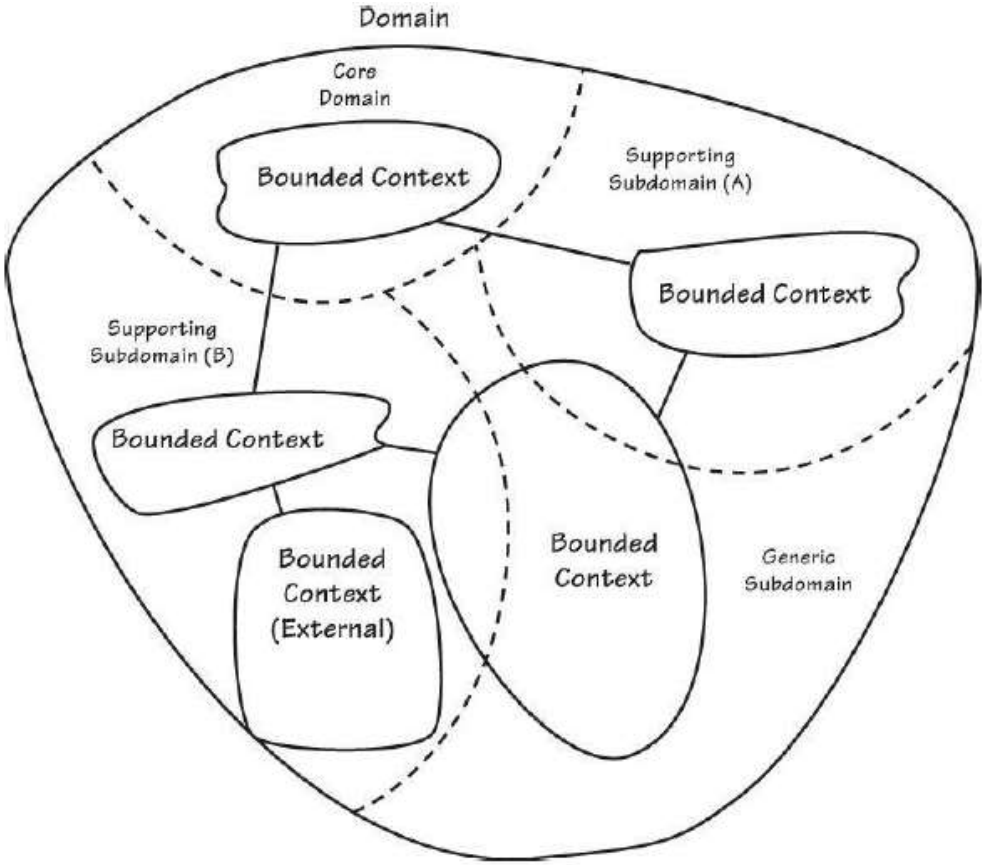


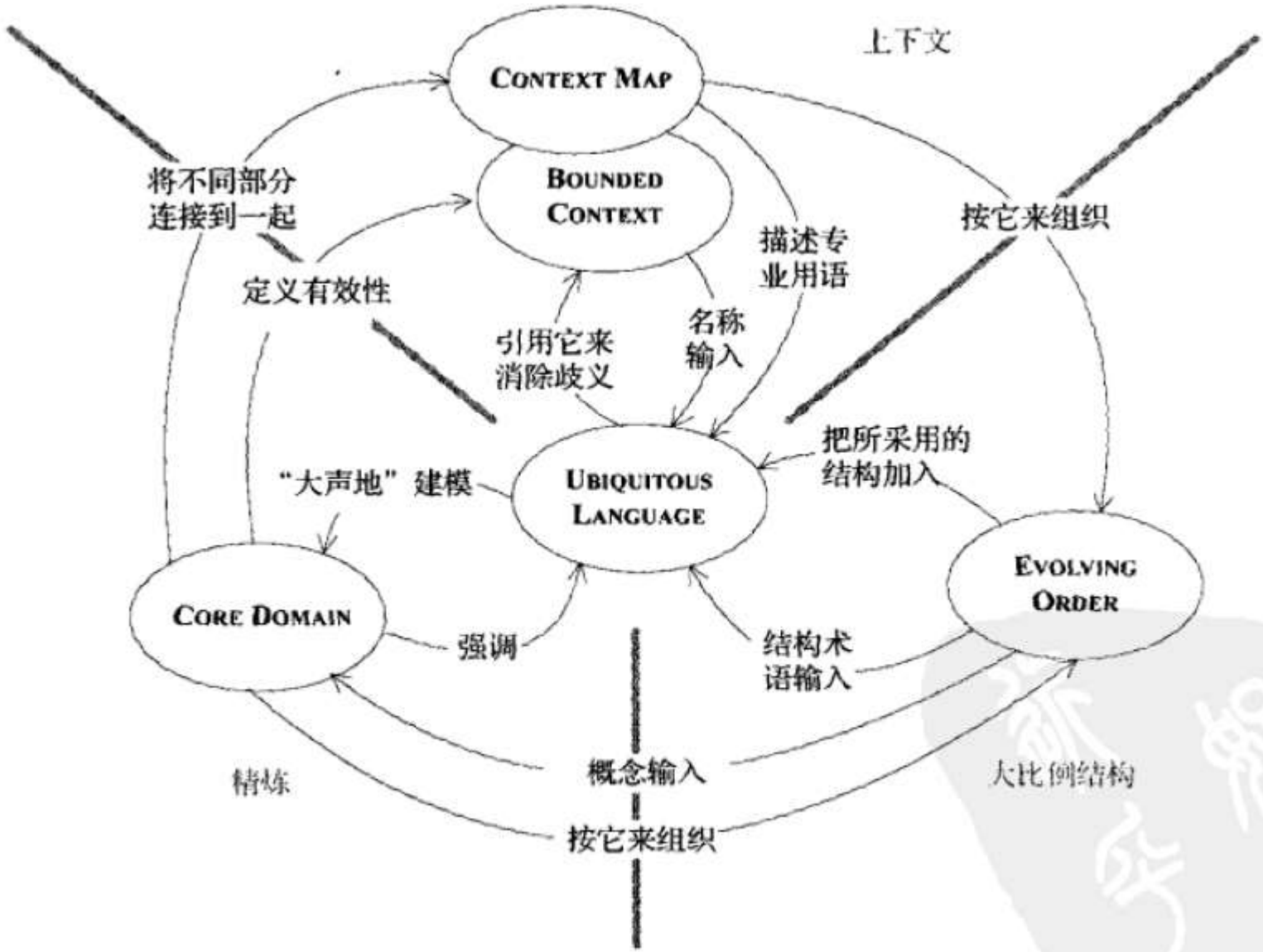
Figure 2.2. An abstract business Domain that includes Subdomains and Bounded Contexts

- A **Core Domain** is a part of the business Domain that is of primary importance to the success of the organization. It is of utmost importance to the ongoing success of the business.
- If a domain models some aspect of the business that is essential, yet not Core, it is a **Supporting Subdomain**.
- if a domain captures nothing special to the business, yet is required for the overall business solution, it is a **Generic Subdomain**.
- Focus on the core domain

战术建模与战略建模

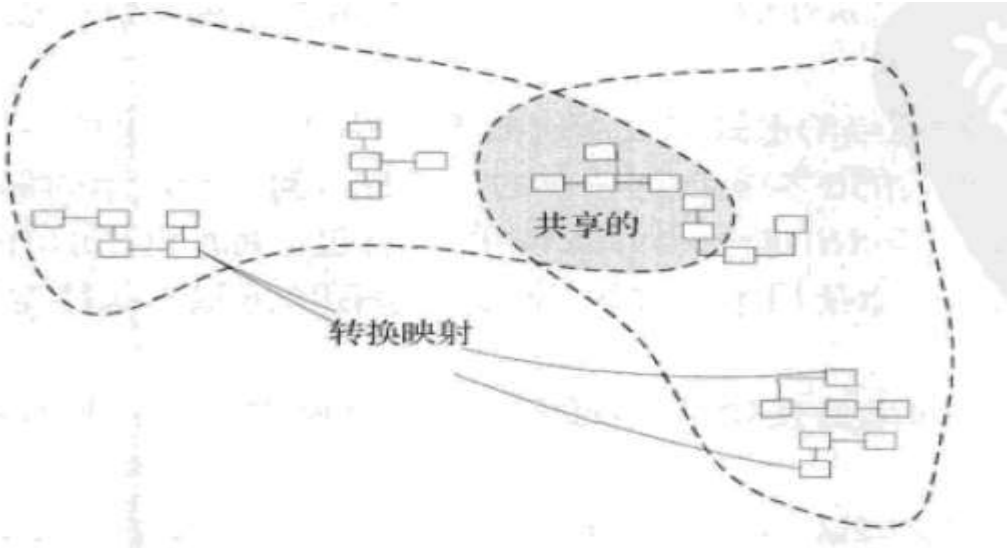

push state change with
DOMAIN EVENTS
access with
REPOSITORIES
access with
SERVICES
express change with
express model with
ENTITIES
encapsulate with
express identity with
act as root of
AGGREGATES
encapsulate with
express state &
computation with
MODEL-DRIVEN
DESIGN
VALUE
OBJECTS
encapsulate with
encapsulate with
model gives structure to
isolate domain
expressions with
define model within
LAYERED
ARCHITECTURE
encapsulate with
FACTORIES
UBIQUITOUS LANGUAGE
cultivate rich
model with
keep model
unified by
CONTINUOUS
INTEGRATION
names
enter
SHARED
KERNEL
SHARED
KERNEL
CORE DOMAIN
BOUNDED CONTEXT
interdependent
contexts form
overlap allied
contexts through
CUSTOMER
/SUPPLIER
relate allied contexts as
work in
autonomous, clean
assess/overview
relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
GENERIC
SUBDOMAINS
segregate the
conceptual messes
free teams
to go
support multiple
clients through
OPEN HOST
SERVICE
translate and insulate
unilaterally with
loosely couple
contexts
through
formalize
BIG BALL OF MUD
ANTI-
CORRUPTION
LAYER
SEPARATE
WAYS
PUBLISHED
LANGUAGE

领域驱动设计的综合应用



共享内核(Shared Kernel)

- 当不同团队开发一些紧密相关的应用程序时，如果团队之间不进行协调，即使短时间内能够取得快速进展，他们开发出的产品也可能互相不适合，最后可能不得不在转换层上花费大量时间，而且得到的产品也五花八门。
  - 从领域模型中选出两个团队都同意共享的一个子集。当然，除了模型的这个子集以外，这还包括与该模型部分相关的代码子集，或数据库设计的子集。这部分明确共享的内容具有特殊的状态，而且一个团队在没与另一个团队商量的情况下不应擅自更改它。
  - 功能系统要经常进行集成，但集成的频率应该比团队中Continuous Integration的频率低一些。在进行这些集成的时候，两个团队都要运行测试。
  - Shared Kernel通常是Core Domain，或是一组Generic Subdomain（通用子领域），也可能二者兼有。



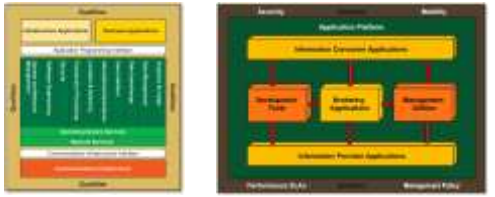
企业架构方法与领域驱动设计

- 两者都强调Business和IT的高度统一，很多企业架构方法对于领域驱动设计“战略设计”的具体实施办法具有详实的指导意义。如TOGAF V9构件：

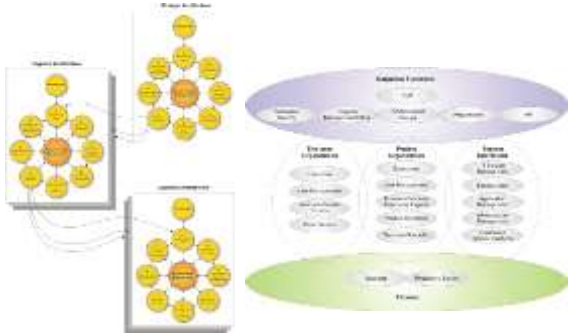
1. 架构开发方法

3. 架构内容框架

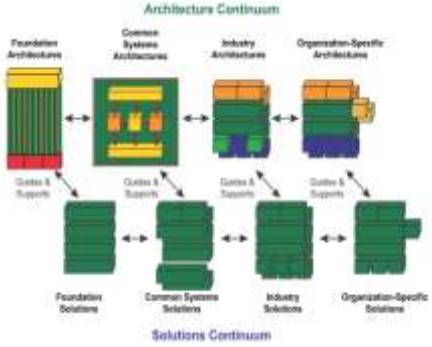
5. 参考模型



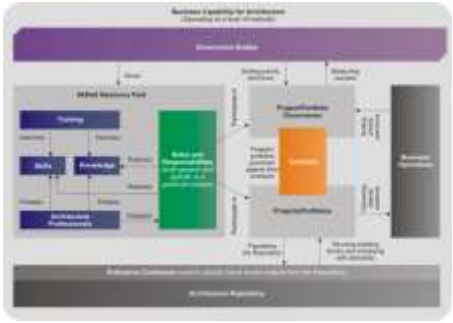
2. 架构开发指引和技术



4. 企业连续系列



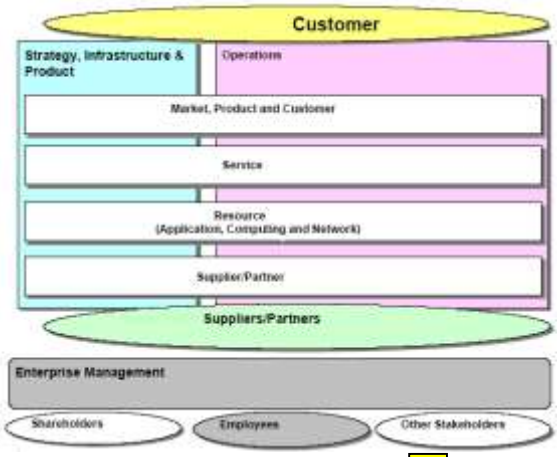
6. 架构能力框架



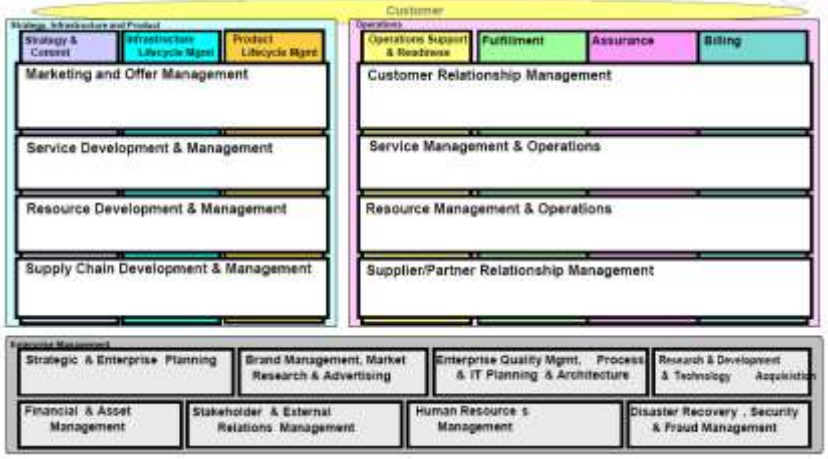
eTOM业务建模

业务流程解耦/分解

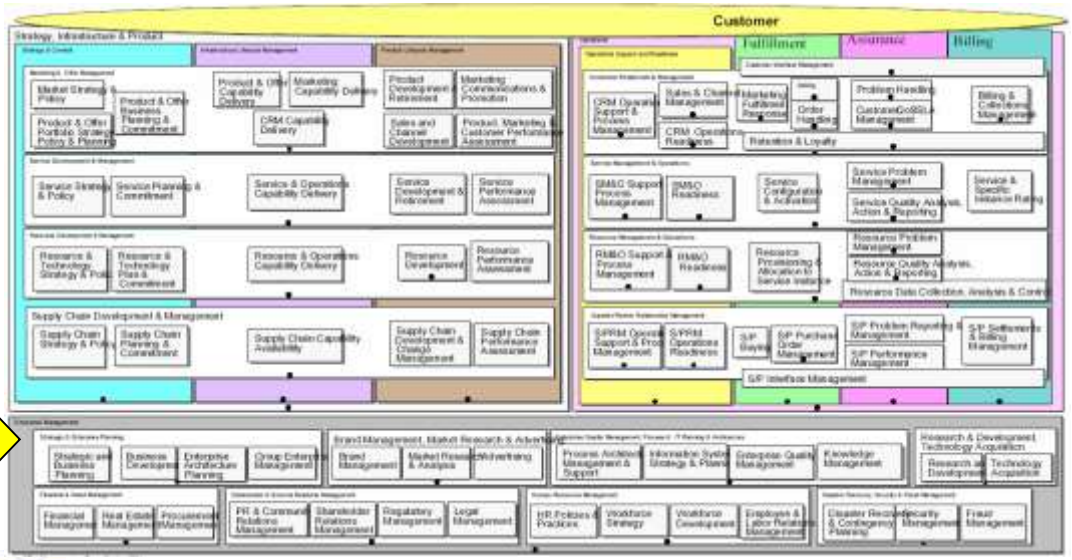
Level 0 Processes



Level 1 Processes

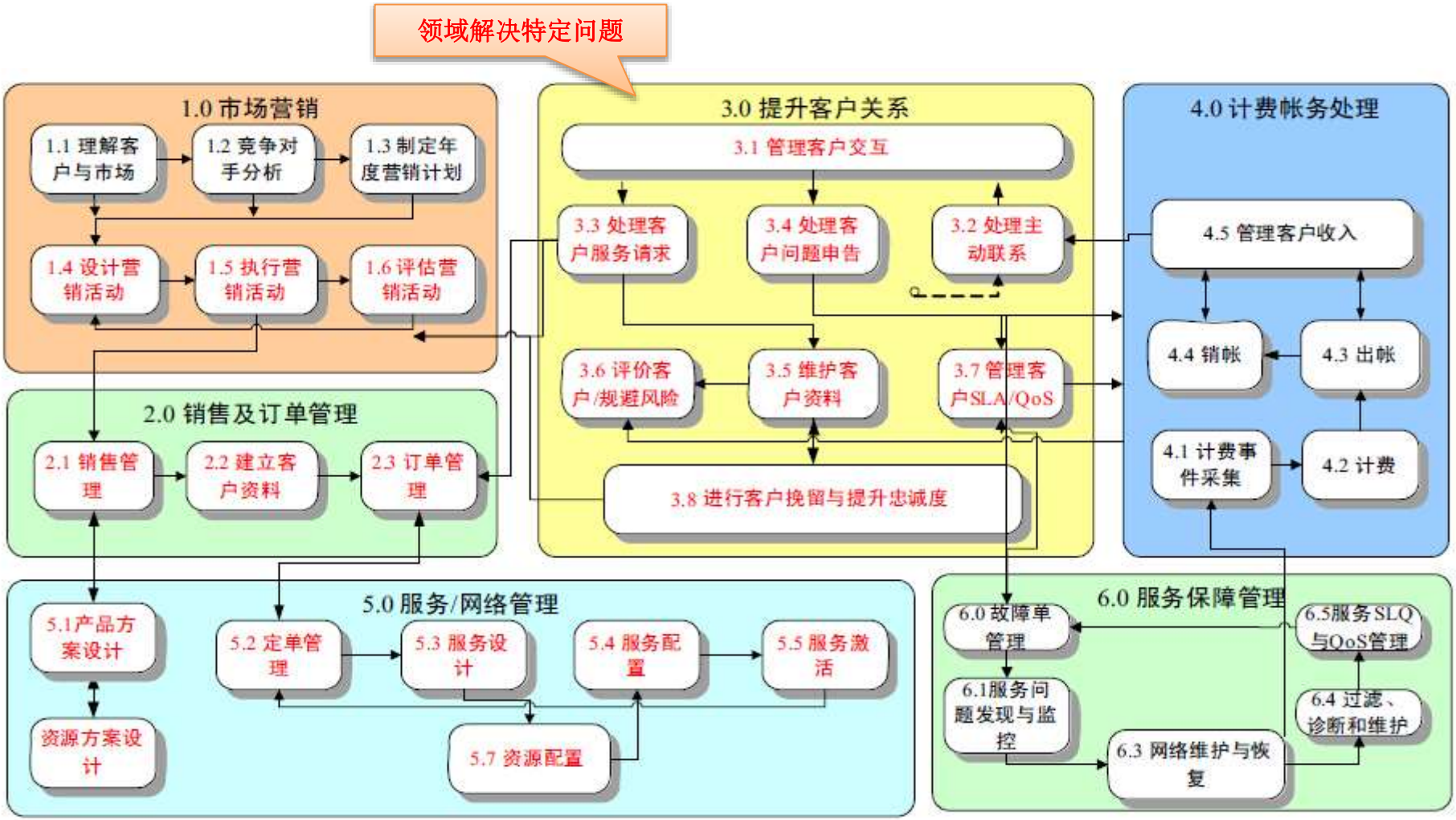


Level 2 Processes

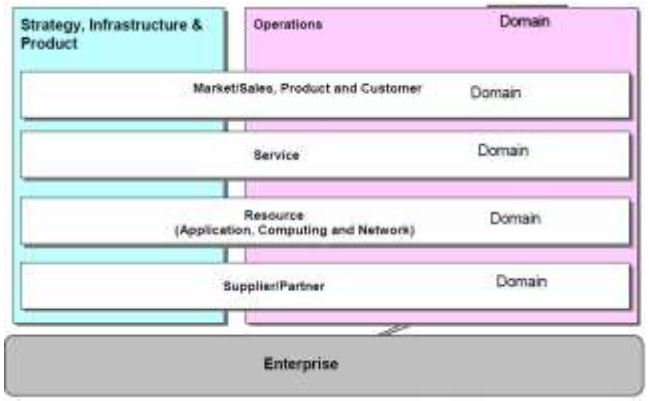


eTOM业务建模

BSS业务流程框架



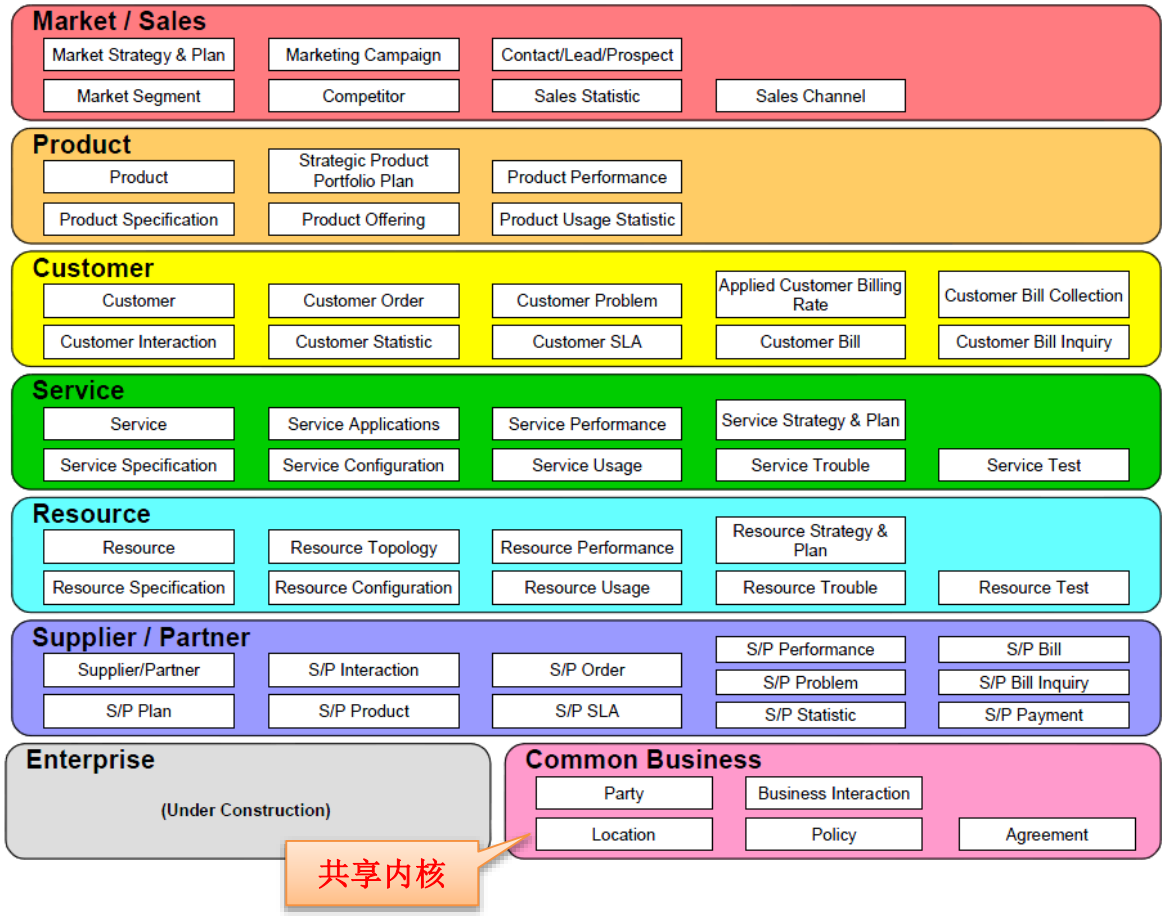
eTOM信息数据模型



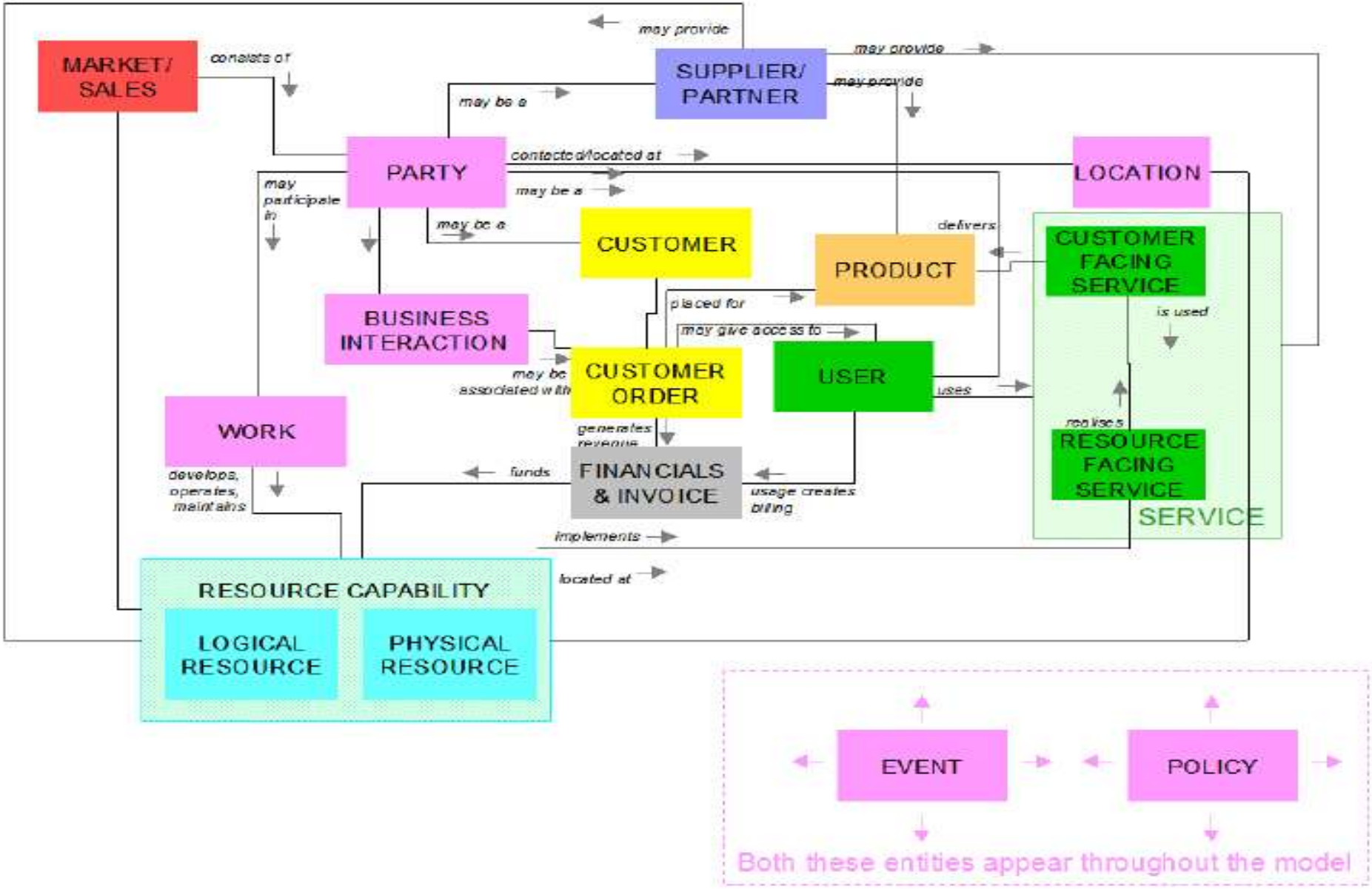
eTOM 0级视图

ABE：Aggregate Business Entity，ABE是SID中一组定义良好的实体，具有高内聚、低耦合的特征。

SID 1级视图



eTOM信息数据模型


may provide
MARKET/ SALES
consists of
SUPPLIER/ PARTNER
may provide
may provide
may be a
PARTY
contacts/located at
LOCATION
may participate in
may be a
may be a
CUSTOMER
delivers
PRODUCT
CUSTOMER FACING SERVICE
placed for
is used
BUSINESS INTERACTION
may give access to
may be associated with
CUSTOMER ORDER
USER
uses
WORK
generates revenue
realises
RESOURCE FACING SERVICE
funds
FINANCIALS & INVOICE
usage creates billing
develops, operates, maintains
SERVICE
implements
located at
RESOURCE CAPABILITY
LOGICAL RESOURCE
PHYSICAL RESOURCE
EVENT
POLICY
Both these entities appear throughout the model

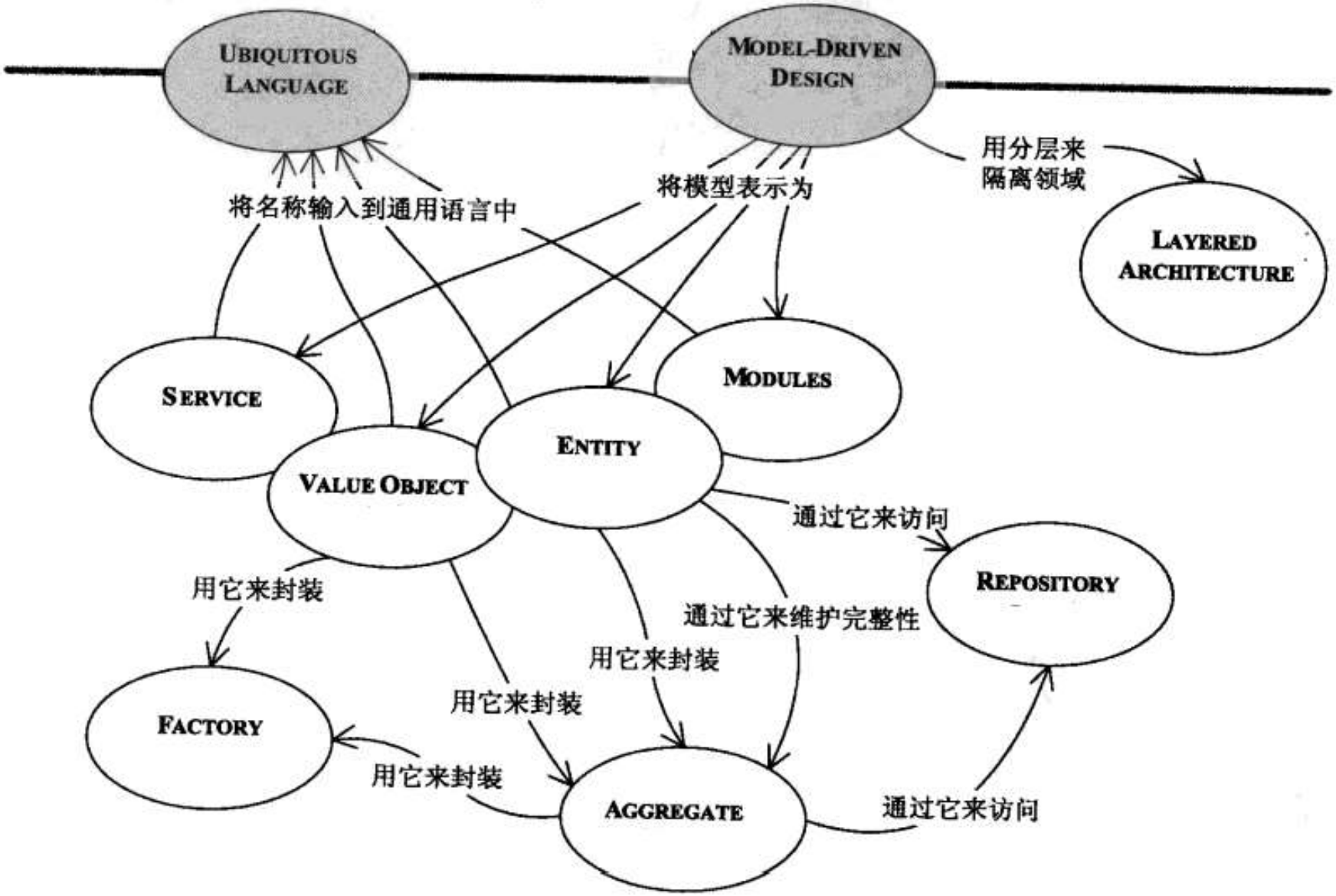
参考读物

- 《领域驱动设计—软件核心复杂性应对之道》及《实现领域驱动设计》中的相关章节
- 《软件方法-业务建模和需求》第三章“业务建模”中的相关内容
- 参考模型范例：
  - TMForum的eTOM模型：http://www.etom.com

培训内容

- 领域驱动设计简介
- 领域通用语言
- 领域驱动设计的构造块
- 领域驱动设计编程实践
- CQRS架构
- 模型驱动开发

领域驱动设计的构造块


UBIQUITOUS LANGUAGE
MODEL-DRIVEN DESIGN
用分层来隔离领域
LAYERED ARCHITECTURE
将名称输入到通用语言中
将模型表示为
SERVICE
MODULES
ENTITY
VALUE OBJECT
通过它来访问
REPOSITORY
用它来封装
通过它来维护完整性
用它来封装
用它来封装
FACTORY
用它来封装
AGGREGATE
通过它来访问

Entity（实体）

- 实体是一个具有唯一身份标识的对象，并且可以在相当长的一段时间内持续地变化。我们可以对实体做多次修改，故一个实体对象可能和它先前的对象大不相同，但是由于它们拥有相同的身份标识（identity），它们依然是同一个实体。
- 我们通过标识对对象进行区分，而不是属性，此时我们应该将标识作为主要的模型定义。同时我们需要保持简单的类定义，并且关注对象在其生命周期中的连续性和唯一标识性。
- 随着对象的改变，我们可能会跟踪这样的改变，比如什么时候发生了改变，发生了什么改变，是谁做出的改变等。我们应该慎重对待在对象整个生命周期中所发生的合法改变。
- **唯一的身份标识**和**可变性**（mutability）特征将实体对象和值对象（Value Objects）区分开来。
  - 很多时候，一个领域概念应该建模成值对象，而不是实体对象。
  - 实体和值对象是领域模型概念，而不是数据存储模型概念。

Value Objects（值对象）

- 值对象的特征
  - 它度量或者描述了领域中的一件东西。
  - 它可以作为不变量。
  - 它将不同的相关的属性组合成一个概念整体
  - 当度量和描述改变时，可以用另一个值对象予以替换
  - 它可以和其他值对象进行相等性比较
  - 它不会对协作对象造成副作用。
- 当我们只关心一个模型元素的属性时，应把它归类为值对象。我们应该使这个模型元素能够表示出其属性的意义，并为它提供相关功能。值对象应该是不可变的。不要为它分配任何标识，而且不要把它设计成Entity那么复杂。
- 应该尽量使用值对象来建模而不是实体对象，即便一个领域概念必须建模成实体，在设计时也应该更偏向于将其作为值对象容器，而不是子实体容器。
- 实体对象与值对象是领域概念，而不是数据存储模型概念
  - 值对象可以与其所在的实体对象保存在同一张表中，值对象的每一个属性保存为一列；值对象也可以独立于其所在的实体对象保存在另一张表中，值对象获得委派主键，该主键对客户端是不可见的。

Entity和Value Object示例

```
public class User {
  private String firstName;
  private String lastName;
  private String login;
  private String password;
  ...
}
```

```
public class User {
  private int id;
  private PersonName name;
  private UserId login;
  private Password password;
  ...
}
```

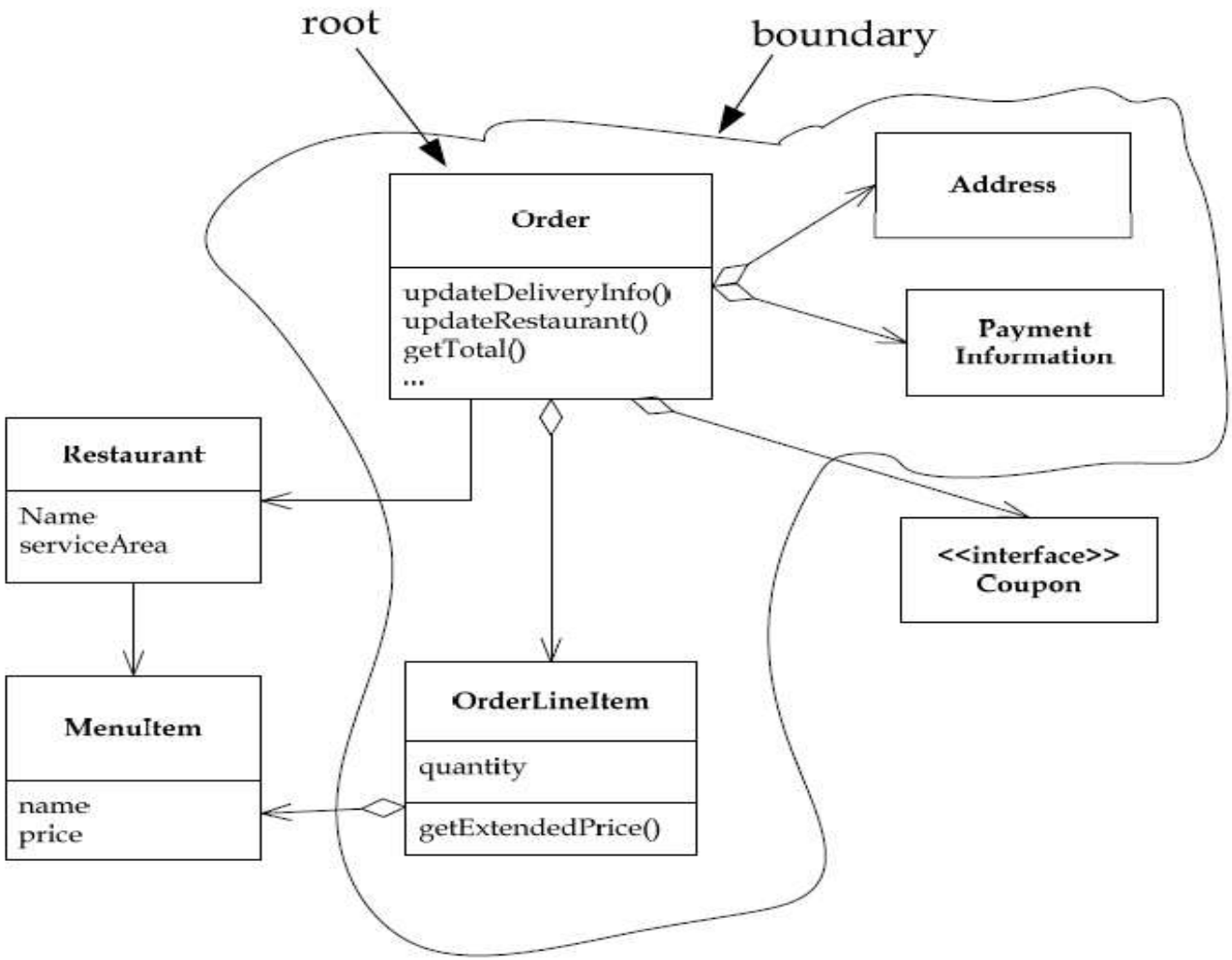
```
public class PersonName {
  private String firstName;
  private String lastName;

  PersonName() {
  }

  public PersonName(String firstName,
          String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
```

```
public class Password implements Serializable {

 private String passwordString;

public Password(String passwordString) {
   this.passwordString = passwordString == null ? null : passwordString.trim();
 }

 ...
@Override
 public String toString() {
  return new ToStringBuilder(this).append("password", "****").toString();
 }
```

Aggregates（聚合）

- 在具有复杂关联的模型中，要想保证对象更改的一致性是很困难的。不仅互不关联的对象需要遵守一些固定规则，而且紧密关联的各组对象也要遵守一些固定规则。然而，过于谨慎的锁定机制又会导致多个用户之间毫无意义地互相关绕，从而使系统不可用。在任何具有持久化数据存储的系统中，对数据进行修改的事务必须要有一个范围，而且要有一种保持数据一致性的方式。
- 聚合（Aggregate）是一组相关对象的集合，我们把它作为数据修改的单元。每个聚合都有一个根和一个边界，边界定义了聚合的内部都有什么，根则是聚合中所包含的一个特定实体。在聚合中，**根是唯一允许外部对象保持对它的引用的元素**，而边界内部的对象之间则可以互相引用。除根以外的其他Entity都有本地表示，但这些标识只有在聚合内部才需要加以区别，因为外部对象除了根Entity之外看不到其他对象。
- 聚合行为视为是一个整体，在每个事务完成时，必须要满足聚合内所应用的固定规则的要求，即保证数据变化的一致性。根实体最终检查固定规则；删除操作必须一次删除聚合边界之内的所有对象；当提交对聚合边界内部的任何对象的修改时，整个聚合中的所有固定规则都必须被满足。
- 原则：在一致性边界之内建模真正的不变条件；设计小聚合；通过唯一标识引用其他聚合；在边界之外使用最终一致性
- 尽量将根实体所包含的其他聚合建模成值对象，而不是实体。

Aggregates（聚合）示例

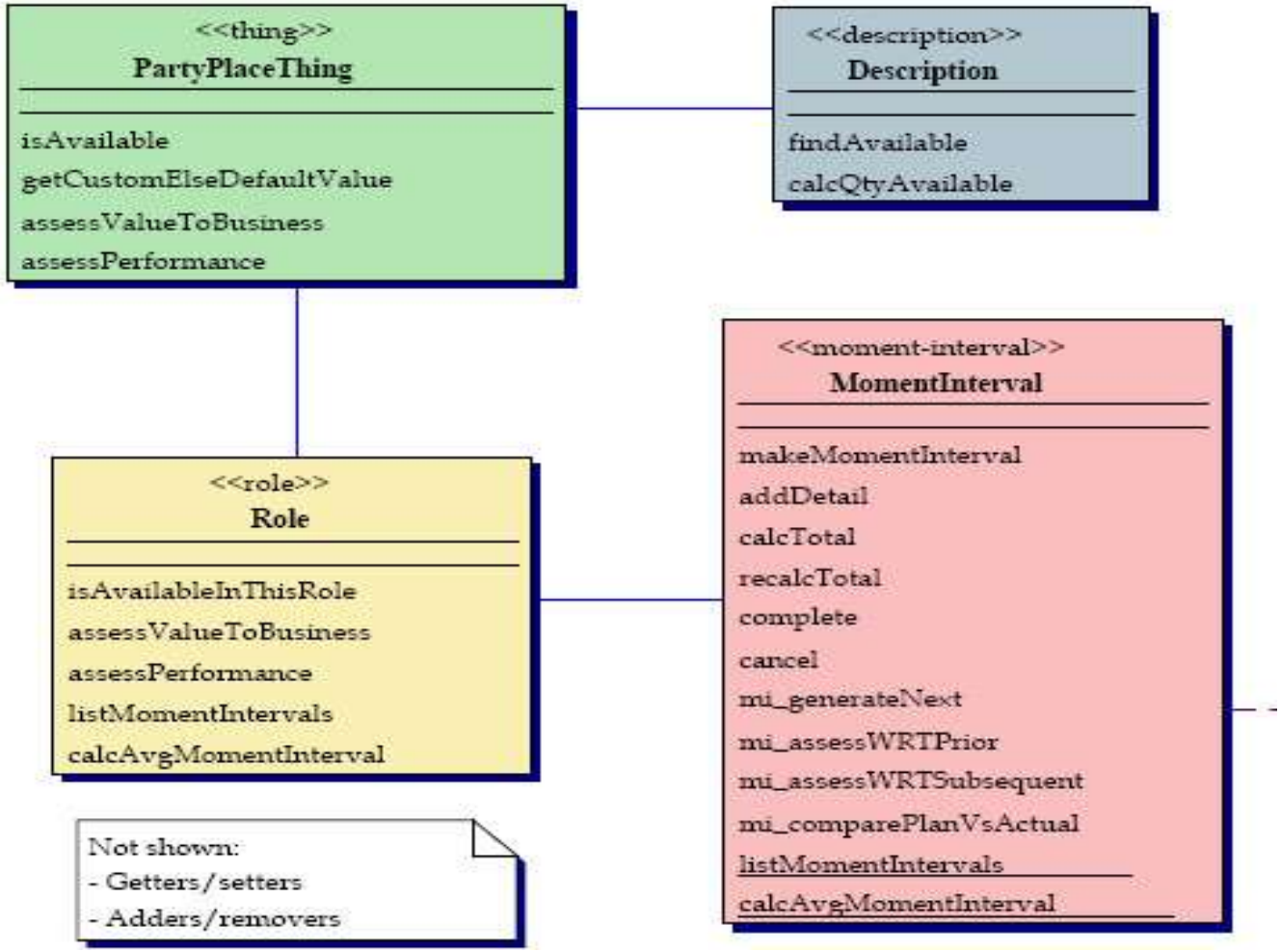


Domain Event（领域事件）

- **Domain Event(领域事件)**
  - 有时候应用需要记录跟踪事情的发生
  - 领域事件经常被建模为Value Object，但这些Value Object并不能被共享，因为领域事件本身是“唯一”的。
  - 一个领域事件是指一个在领域中“**有意义**”的事件
- Hints
  - UML四色原型中有一个相近概念，称为时刻-时段原型(Moment-interval)，即表示事物在某个时刻或某一段时间内发生。

参考：四色原型

- 四色原型是诞生于90年代，现在被广泛使用的一种系统分析方法，如Borland的Together架构师版，准确地说，是由Peter Coad 和 Mark Mayfield首先提出，然后由David North拓展。



Repositories（资源库/仓储）

- 客户需要以一种符合实际的方式来获取对以存在的领域对象的引用。为每种需要全局访问的对象类型创建一个对象，这个对象就相当于该类型的所有对象在内存中的一个集合的“替身”。通过一个众所周知的接口来提供访问。提供添加和删除对象的方法，用这些方法来封装在数据存储中实际插入或删除数据的操作。提供根据具体标准来挑选对象的方法，并返回属性值满足查询标准的对象或对象集合（所返回的对象是完全实例化的），从而将实际的存储和查询技术封装起来。只为那些确实需要直接访问的聚合提供Repository。让客户始终聚焦于模型，而将所有对象的存储和访问操作交给Repository来完成。
- Repository的接口应当采用**领域通用语言**。作为客户端，不应当知道数据库实现的细节。
- Repository和DAO的作用类似，二者的主要区别：
  - DAO是比Repository更低的一层，包含了如何从数据库中提取数据的代码。
  - Repository以“领域”为中心，所描述的是“领域语言”。Repository把ORM框架与领域模型隔离，对外隐藏封装了数据访问机制。

Repositories（资源库/仓储）示例

```
public interface AccountRepository {
   Account findAccount(String accountId);
   void addAccount(Account account);
}
```

```
public class HibernateAccountRepository implements AccountRepository {
   private HibernateTemplate hibernateTemplate;
   public HibernateAccountRepository(HibernateTemplate template) {
      hibernateTemplate = template;
   }

   public void addAccount(Account account) {
      hibernateTemplate.save(account);
   }

   public Account findAccount(final String accountId) {
      return (Account) DataAccessUtils.uniqueResult(hibernateTemplate.
         findByNamedQueryAndNamedParam(
            “Account.findAccountByAccountId”, “accountId”, accountId));
   }
}
```

Services（领域服务）

- 当领域中的某个操作过程或转换过程不是实体或值对象的职责时，我们便应该将该操作放在一个单独的接口中，即**领域服务**。如果勉强地把这些重要的领域功能归为Entity或ValueObject的职责，那么不是歪曲了基于模型的对象的定义，就是人为地增加了一些无意义的对象。应确保领域服务和通用语言是一致的，并且保证它是无状态的。
- 正确区分**领域服务**(Domain Service)和**应用服务**(Application Service)：
  - 我们不应把业务逻辑置于应用服务，但我们会把业务逻辑置于领域服务中。(应用)服务要做“薄”。
  - 领域服务职责：跨聚合实例业务逻辑；没办法合理放到实体中的其它业务逻辑。
  - 应用服务职责：跨限界上下文的业务逻辑；DTO转换；事务AOP、权限AOP、日志AOP、异常AOP；外部系统访问（邮件、消息队列）。
  - 领域服务设计原则：用来组织业务逻辑，面向业务逻辑；细粒度；内部视图看系统；一个请求对应多个服务的多个方法；服务之间会存在依赖；
  - 应用服务设计原则：用来封装业务逻辑；面向用例；粗粒度；外部视图看系统；一个请求对应一个方法；服务之间互不依赖。
  - 应用服务和领域服务区分非常敏感，有时候需要在快速性/方便性上做折衷。

Services（领域服务）示例

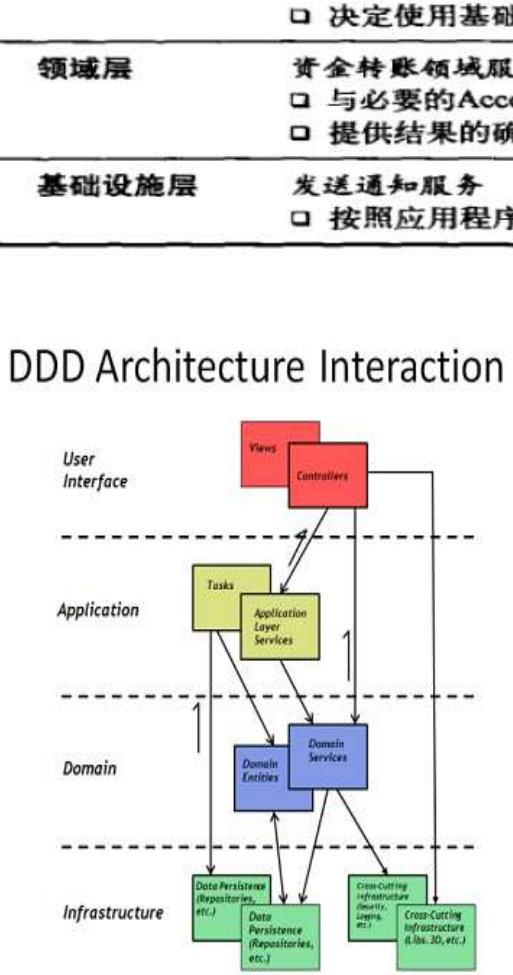
```
public interface MoneyTransferService {
   BankingTransaction transfer(String fromAccountId, String toAccountId, double amount);
}
```

```
public class MoneyTransferServiceImpl implements MoneyTransferService {
   private final AccountRepository accountRepository;
   private final BankingTransactionRepository bankingTransactionRepository;
   public MoneyTransferServiceImpl(AccountRepository accountRepository,
      BankingTransactionRepository bankingTransactionRepository)  {
      …
   }

   BankingTransaction transfer(String fromAccountId, String toAccountId,
      double amount) {
      …
   }
}
```

应用服务、领域服务和基础设施服务

将SERVICE划分到各个层中

应用层	资金转账应用服务 ❑ 获取输入（例如一个XML请求） ❑ 发送消息给领域层服务，要求其执行 ❑ 监听确认消息 ❑ 决定使用基础设施SERVICE来发送通知
领域层	资金转账领域服务 ❑ 与必要的Account（账户）和Ledger（总账）对象进行交互，执行相应的借入和贷出操作 ❑ 提供结果的确认（允许转账或拒绝转账，等等）
基础设施层	发送通知服务 ❑ 按照应用程序的指示发送电子邮件、信件和其他信息



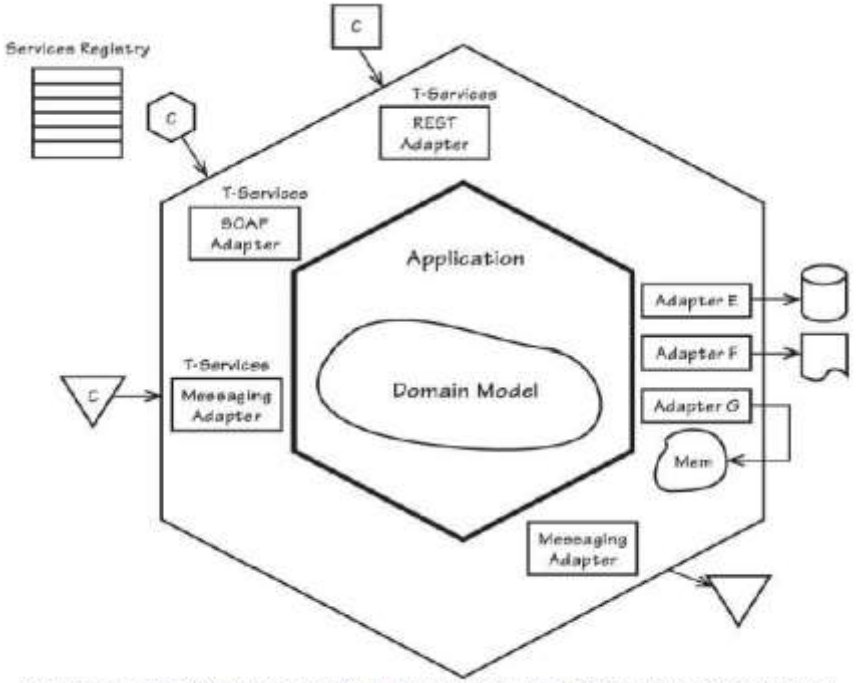


Figure 4.5. A Hexagonal Architecture supporting SOA, with REST, SOAP, and messaging services

Factories（工厂）

- 当创建一个对象或创建整个聚合时，如果创建工作很复杂，或者暴露了过多的内部结构，则可以使用Factory进行封装。应该将创建复杂对象的实例和聚合的职责转移到一个单独的对象，这个对象本身在领域模型中可能没有职责，但它仍是领域设计的一部分。
- 不同类型的工厂模式：
  - 工厂类
  - 工厂方法

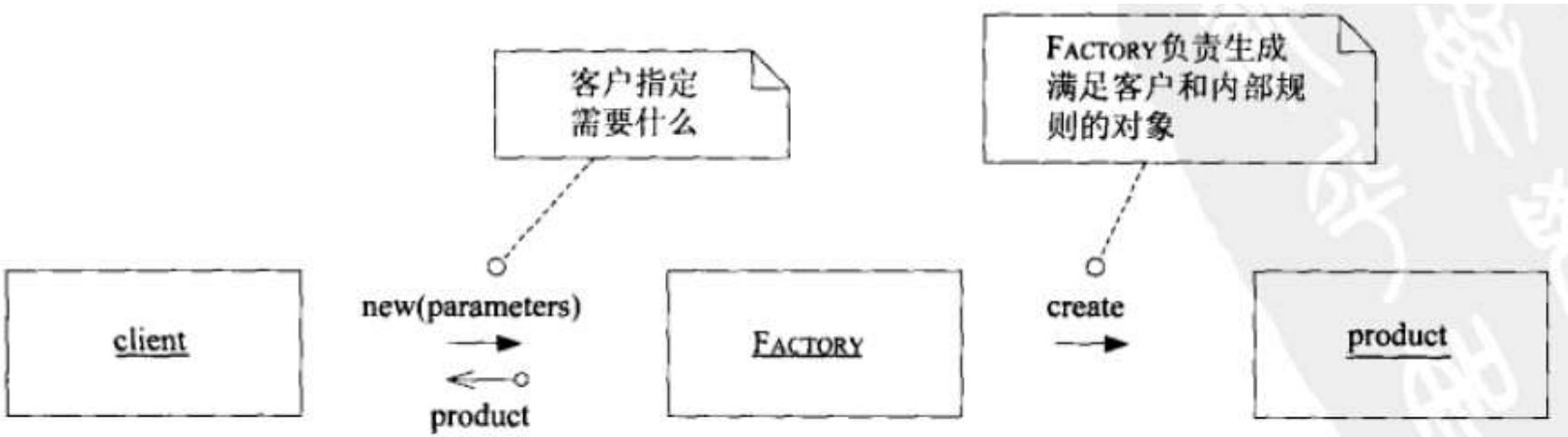


图6-12　与FACTORY的基本交互

Modules（模块）

- Module为人们提供了两种观察模型的方式，一是可以在Module中查看细节，而不会被整个模型淹没，二是观察Module之间的关系，而不考虑其内部细节。
- 模块之间应该是低耦合的，而在模块内部则是高内聚的。模块并不仅仅是代码的划分，而且也是概念的划分。一个人一次考虑的事情是有限的（因此才有低耦合）；不连贯的思想和“一锅粥”似的思想同样难于理解（因此才有高内聚）。
- 选择能够描述系统的Module，并使之包含一个内聚的概念集合。这通常会实现Module之间的低耦合，但如果效果不理想，则应寻找一种更改模型的方式来消除概念之间的耦合，或者找到一个可作为Module基础的概念，基于这个概念组织的模型可以以一种有意义的方式将元素集中到一起。找到一种低耦合的概念组织方式，从而可以相互独立地理解和分析这些概念。对模型进行精化，直到可以根据高层领域概念对模型进行划分，同时相应的代码也不会产生耦合。
- Module的名称应该是领域通用语言中的术语。模块及其名称应反映出领域的深层知识。

培训内容

- 领域驱动设计简介
- 领域通用语言
- 领域驱动设计的构造块
- 领域驱动设计编程实践
- CQRS架构
- 模型驱动开发

概念辨析-VO/DTO/DO/PO（一）

- View Object（视图对象）：视图对象，用于展示层，其作用是把某个指定页面（或组件）的所有数据封装起来。
- Data Transfer Object（数据传输对象）：这个概念来源于J2EE的设计模式，原来的目的是为了EJB的分布式应用提供粗粒度的数据实体，以减少分布式调用的次数，从而提高分布式调用的性能和降低网络负载，但在这里，我泛指用于展示层与服务层之间的数据传输对象。
- Domain Object（领域对象）：从现实世界中抽象出来的有形或无形的业务实体、值对象或领域服务。
- Persistent Object（持久化对象）：跟持久层（通常是关系型数据库）的数据结构形成一一对应的映射关系，如果持久层是关系型数据库，那么，数据表中的每个字段（或若干个）就对应PO的一个（或若干个）属性。

Ref: http://blog.csdn.net/zjrbiancheng/article/details/6253232

概念辨析-VO/DTO/DO/PO（二）

- VO与DTO：绝大多数应用场景下，VO与DTO的属性值基本一致，但对于设计层面来说，概念上还是存在VO和DTO的区别，DTO代表服务层需要接收的数据和返回的数据，而VO代表展示层需要显示的数据。
  - 示例：服务层有一个getUser的方法返回一个系统用户，其中有一个属性是gender(性别)，对于服务层来说，它只从语义上定义：1-男性，2-女性，0-未指定，而对于展示层来说，它可能需要用“帅哥”代表男性，用“美女”代表女性，用“秘密”代表未指定。说到这里，可能你还会反驳，在服务层直接就返回“帅哥美女”不就行了吗？对于大部分应用来说，这不是问题，但设想一下，如果需求允许客户可以定制风格，而不同风格对于“性别”的表现方式不一样，又或者这个服务同时供多个客户端使用（不同门户），而不同的客户端对于表现层的要求有所不同，那么，问题就来了。再者，回到设计层面上分析，从职责单一原则来看，服务层只负责业务，与具体的表现形式无关，因此，它返回的DTO，不应该出现与表现形式的耦合。
  - 实现层面是否需要区分二者概念？具体问题具体分析

概念辨析-VO/DTO/DO/PO（三）

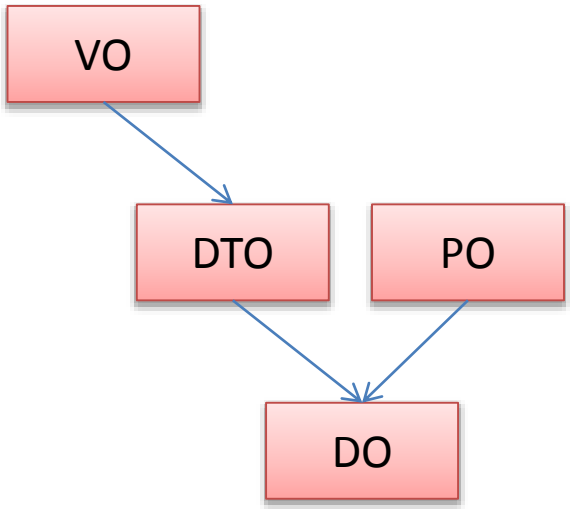
- DTO与DO：DTO是展示层和服务层之间的数据传输对象（可以认为是两者之间的协议），而DO是对现实世界各种业务角色的抽象，这就引出了两者在数据上的区别，例如UserInfo和User，对于一个getUser方法来说，本质上它永远不应该返回用户的密码，因此UserInfo至少比User少一个password的数据。而在领域驱动设计中，DO不是简单的POJO，它具有领域业务逻辑。
  - 在设计层面，展示层向服务层传递的DTO与服务层返回给展示层的DTO在概念上是不同的(如返回UserInfo应该不包含password，但创建User传入的参数需要包含password)，但在实现层面，我们通常很少会这样做（定义两个UserInfo，甚至更多），因为这样做并不见得很明智，我们完全可以设计一个完全兼容的DTO，在服务层接收数据的时候，不该由展示层设置的属性（如订单的总价应该由其单价、数量、折扣等决定），无论展示层是否设置，服务层都一概忽略，而在服务层返回数据时，不该返回的数据（如用户密码），就不设置对应的属性。
  - 为什么不在服务层中直接返回DO：DO具有一些不应该让展示层知道的数据；DO具有业务方法，如果直接把DO传递给展示层，展示层的代码就可以绕过服务层直接调用它不应该访问的操作，对于基于AOP拦截服务层来进行访问控制的机制来说，这问题尤为突出，而在展示层调用DO的业务方法也会因为事务的问题，让事务难以控制；ORM框架（如Hibernate）“延迟加载”技术，如果直接把DO暴露给展示层，对于大部分情况，展示层不在事务范围之内，如果其尝试在Session关闭的情况下获取一个未加载的关联对象，会出现运行时异常（对于Hibernate来说，就是LazyInitiliaztionException）；从设计层面来说，展示层依赖于服务层，服务层依赖于领域层，如果把DO暴露出去，就会导致展示层直接依赖于领域层，这虽然依然是单向依赖，但这种跨层依赖会导致不必要的耦合。
  - DTO应该是一个“扁平的二维对象”

概念辨析-VO/DTO/DO/PO（四）

- DO与PO：DO和PO在绝大部分情况下是一一对应的，PO是只含有get/set方法的POJO，但某些场景还是能反映出两者在概念上存在本质的区别。
  - DO在某些场景下不需要进行显式的持久化，例如利用策略模式设计的商品折扣策略，会衍生出折扣策略的接口和不同折扣策略实现类，这些折扣策略实现类可以算是DO，但它们只驻留在静态内存，不需要持久化到持久层，因此，这类DO是不存在对应的PO的。同样的道理，某些场景下，PO也没有对应的DO，例如老师Teacher和学生Student存在多对多的关系，在关系数据库中，这种关系需要表现为一个中间表，也就对应有一个TeacherAndStudentPO的PO，但这个PO在业务领域没有任何现实的意义，它完全不能与任何DO对应上。这里要特别声明，并不是所有多对多关系都没有业务含义，这跟具体业务场景有关，例如：两个PO之间的关系会影响具体业务，并且这种关系存在多种类型，那么这种多对多关系也应该表现为一个DO，又如：“角色”与“资源”之间存在多对多关系，而这种关系很明显会表现为一个DO——“权限”。
  - 某些情况下，为了某种持久化策略或者性能的考虑，一个PO可能对应多个DO，反之亦然。例如客户Customer有其联系信息Contacts，这里是两个一对一关系的DO，但可能出于性能的考虑（极端情况，权作举例），为了减少数据库的连接查询操作，把Customer和Contacts两个DO数据合并到一张数据表中。反过来，如果一本图书Book，有一个属性是封面cover，但该属性是一副图片的二进制数据，而某些查询操作不希望把cover一并加载，从而减轻磁盘IO开销，同时假设ORM框架不支持属性级别的延迟加载，那么就需要考虑把cover独立到一张数据表中去，这样就形成一个DO对应对个PO的情况。
  - PO的某些属性值对于DO没有任何意义，这些属性值可能是为了解决某些持久化策略而存在的数据，例如为了实现“乐观锁”，PO存在一个version的属性，这个version对于DO来说是没有任何业务意义的，它不应该在DO中存在。同理，DO中也可能存在不需要持久化的属性。
  - 现在的业务应用开发，基本上不需要区分DO与PO，PO完全可以通过JPA，Hibernate Annotations/hbm隐藏在DO之中。

概念辨析-VO/DTO/DO/PO（五）

- VO/DTO/DO/PO转换：
  - 工厂类或工厂方法；
  - 构造函数；
  - 工具类，如BeanUtils；
  - 转换实现中应注意类层次概念的依赖关系。



Entity（一）

- Entity的标识生成：
  - 用户提供
  - 应用程序生成
  - 持久化机制生成
  - 另一个限界上下文提供
- 在JPA中，有下面四种策略：
  - 容器自动生成 (GenerationType.AUTO)：由JPA自动生成
  - 使用数据库的自动增长字段生成(GenerationType.IDENTITY)：需要数据库支持
  - 根据数据库序列号(GenerationType.SEQUENCE)：Oracle支持对序列号的支持
  - 使用数据库表的字段生成(GenerationType.TABLE)：使用数据库中指定表的某个字段记录实体对象的标识，通过该字段的增长为新增加的实体对象赋唯一值。

Entity（二）

- 继承关系：因为关系数据库的表之间不存在继承关系，Entity提供三种基本的继承映射策略：
  - Single Table
  - Joined
  - Table per Class

Ref: http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html/ch05.html

Entity（三）

- 继承关系：Single Table

```
@SuppressWarnings("serial")
@Entity
@Table(name="Vehicle_Hierarchy")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="Discriminator", discriminatorType = DiscriminatorType.STRING,length=30)
@DiscriminatorValue("Vehicle")
public class Vehicle implements Serializable{
        //基类
         privateLong id;
         privateShort speed;//速度
         @Id
         @GeneratedValue
         @Column(columnDefinition="integer")//指定使用适配Integer长度的数据类型
         public Long getId() {
            returnid;
         }
         public void setId(Long id) {
            this.id = id;
         }
}
@SuppressWarnings("serial")
@Entity
@DiscriminatorValue("Car")
public class Car extends Vehicle{
         //Vehicle的子类
         privateString engine;//发动机
         @Column(nullable=true,length=30)
         publi cString getEngine() {
               returnengine;
         }
         public void setEngine(String engine) {
               this.engine = engine;
         }
 }
@SuppressWarnings("serial")
```

Entity（四）

- 继承关系：Joined

```
@SuppressWarnings(“serial”)
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="Vehicle")
public class Vehicle implements Serializable{    //基类
  private Long id;
  private Short speed;//速度
  @Id
  @GeneratedValue
  @Column(columnDefinition="integer")
  public Long getId() {
     returnid;
  }
  public void setId(Long id) {
     this.id = id;}
  public Short getSpeed() {
     returnspeed;}
  public void setSpeed(Short speed) {
     this.speed = speed;
  }
}
@SuppressWarnings("serial")
@Entity
@Table(name="Car")
@PrimaryKeyJoinColumn(name="CarID")   //把主键对应的列名更改为CarID</STRONG>
public class Car extends Vehicle{            //Vehicle的子类
private String engine;//发动机
@Column(nullable=true,length=30)
  public String getEngine() {
     returnengine;}
  public void setEngine(String engine) {
     this.engine = engine;}
}
@SuppressWarnings("serial")
@Entity
```

Entity（五）

- 继承关系：Table per Class
  - 一旦使用这种策略，意味着你不能使用AUTO generator 和IDENTITY generator，即主键值不能采用数据库自动生成。

```
@SuppressWarnings("serial")
@Entity     //或@MappedSuperclass
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name="Vehicle")  //当为MappedSuperclass时，不要@Table标注
public class Vehicle implementsSerializable{        //基类
   private Long id;
   private Short speed;//速度
   @Id
   @Column(columnDefinition="integer")
   public Long getId() {
      return id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public Short getSpeed() {
      return speed;
   }
   public void setSpeed(Short speed) {
      this.speed = speed;
   }
}
@SuppressWarnings("serial")
@Entity
@Table(name="Car")
public class Car extends Vehicle{          //Vehicle的子类
   private String engine;//发动机
   @Column(nullable=true,length=30)
   public String getEngine() {
      returnengine;
   }
```

Entity（六）

- 继承关系：多态查询
  - 实现JPA规范的查询语言时，需要重点处理的一个方面是多态查询，语句中的实体名有可能是abstract的或者supperclass，查询解析器能够处理这种多态查询。
  - 在某些应用场景中，我们可以希望结果集中只包含部分类型的子类，如假设有一个父类实体为Institution（机构），其有三个子类分别为Manufactures(制造商)、Suppliers(供应商)和Service Prividers(服务提供商)，我们希望在一次对机构的查询中只返回供应商和服务提供商，而不返回制造商商，怎么办呢？JPA规范提供了这种方法，代码示例如下：

```
List<Institution> institutions = em.createQuery(
      "SELECT i FROM Institution i WHERE TYPE(i) IN ( Supplier, ServiceProvider ) )
      .getResultList();
```

Entity（七）

- 审计(Audit)：最近一次修改时间，最近一次修改人。

```
@Entity
@EntityListeners({ JodaAuditListener.class })
public class Cargo extends AbstractDomainObject implements JodaAuditable, Identifiable {
   …
}

public interface JodaAuditable {

                        public void setCreatedBy(String createdBy);
                        public String getCreatedBy();
                        public void setCreatedDate(DateTime createdDate);
                        public DateTime getCreatedDate();

   public void setLastUpdatedBy(String updatedBy);
   public String getLastUpdatedBy();
   public void setLastUpdated(DateTime updateDate);
   public DateTime getLastUpdated();

}

public class JodaAuditListener  {
   @PreUpdate
   @PrePersist
   private void changeAuditInformation(JodaAuditable auditableEntity) {
                        DateTime lastUpdated = new DateTime();
      auditableEntity.setLastUpdated(lastUpdated);
      String lastUpdatedBy = ServiceContextStore.getCurrentUser();
      auditableEntity.setLastUpdatedBy(lastUpdatedBy);
      if (auditableEntity.getCreatedDate() == null)
         auditableEntity.setCreatedDate(lastUpdated);
      if (auditableEntity.getCreatedBy() == null)
         auditableEntity.setCreatedBy(lastUpdatedBy);
   }
}
```

Entity（八）

- 审计(Audit) ：使用事件记录实体状态变更事件，操作人以及状态改变内容等。

```
@Service("bettingService")
public class BettingServiceImpl implements BettingService {

        private static final Logger LOG = LoggerFactory.getLogger(BettingServiceImpl.class);
        …

        @Publish(eventType = BettingInstruction.class, topic = "bettingInstructionTopic", eventBus = "commandBus")
        public void placeBet(Bet bet) {
                LOG.info("### Placing bet: {}", bet);

                // do some initial validation...

                // new BettingInstruction will be published
        }

}
```

```
@Subscribe(topic = "bettingInstructionTopic", eventBus = "commandBus")
public  class BettingEngineImpl implements BettingEngine {

        …
        public void receive(Event event) {
                DynamicMethodDispatcher.dispatch(this, event, "handle");
        }

        public void handle(BettingInstruction betInstruction) {
                LOG.info("### Handling bet: {}", betInstruction);

                instructionRepository.save(betInstruction);

                …
        }
}
```

Entity（九）

- 并发冲突：乐观锁

```
import javax.persistence.Version;

@Entity
@EntityListeners({ JodaAuditListener.class })
public class Cargo extends AbstractDomainObject implements JodaAuditable, Identifiable {

                    …

                    @Version
                    @Column(name = "VERSION", nullable = false)
                    private Long version;

                    …
}
```

Value Object（一）

- 值对象可以与其所在的实体对象保存在同一张表中，值对象的每一个属性保存为一列；值对象也可以独立于其所在的实体对象保存在另一张表中，值对象获得委派主键，该主键对客户端是不可见的。

Value Object（二）

- 值对象可以与其所在的实体对象保存在同一张表中：

```
@Entity
@EntityListeners({ JodaAuditListener.class })
public class Cargo extends AbstractDomainObject implements JodaAuditable, Identifiable {
     ……
   @Embedded
   @AttributeOverrides({ @AttributeOverride(name = "identifier", column = @Column(name = "TRACKINGID", nullable = false, length = 100)) })
   @NotNull
   private TrackingId trackingId;

   public TrackingId getTrackingId() {
      return trackingId;
   }
}
```

```
@Embeddable
public class TrackingId extends AbstractDomainObject {
   private static final long serialVersionUID = 1L;
   @Column(name = "", nullable = false, length = 100, unique = true)
   @NotNull
   private String identifier;

   protected TrackingId() {}

   public TrackingId(String identifier) {
      super();
      Validate.notNull(identifier, "TrackingId.identifier must not be null");
      this.identifier = identifier;
   }
     ……
}
```

Value Object（三）

- 使用数据库实体保存值对象

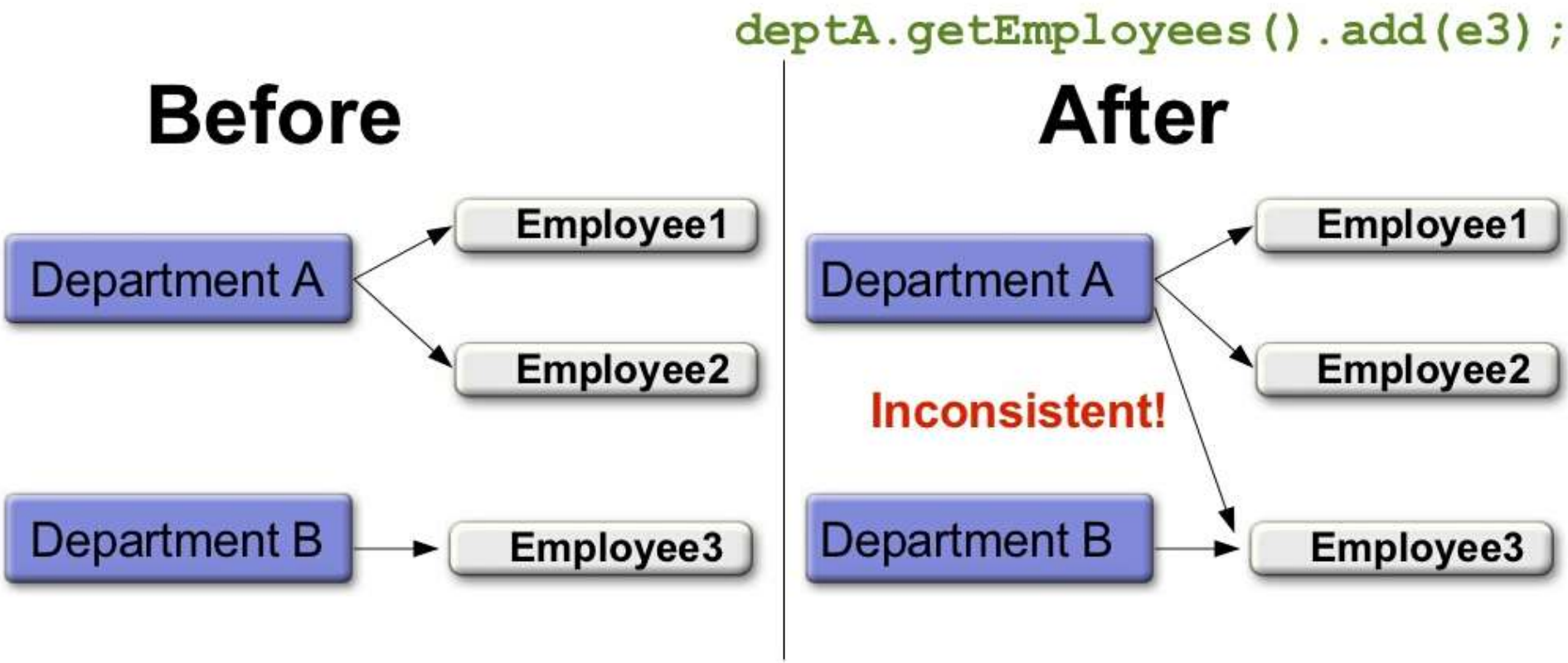
```
@Entity
@EntityListeners({ JodaAuditListener.class })
public class Cargo extends AbstractDomainObject implements JodaAuditable, Identifiable {
    ……
   @OneToOne(mappedBy = "cargo", cascade = CascadeType.ALL, fetch = FetchType.EAGER)
   private Itinerary itinerary;
    ……
}
```

注意区分数据库实体和领域实体两个概念的不同

```
@Entity(name = "Itinerary")
@Table(name = "ITINERARY")
public class Itinerary {
   private static final long serialVersionUID = 1L;
   static final Itinerary EMPTY_ITINERARY = new Itinerary();
   public Itinerary(final List<Leg> legs) {
      Validate.notEmpty(legs);
      Validate.noNullElements(legs);
      super.getLegs().addAll(legs);
   }
   Itinerary() { }
   @Override
   public List<Leg> getLegs() {
      return Collections.unmodifiableList(super.getLegs());
   }
  /**
    * Test if the given handling event is expected when executing this
    * itinerary.
    *
    * @param event
    *          Event to test.
    * @return <code>true</code> if the event is expected
    */
   public boolean isExpected(final HandlingEvent event) {
      ……..
   }
```

存储-对象关联关系（一）

- 对象关联关系：应用担负了维护对象关联关系的职责



存储-对象关联关系（二）

- 维护对象关联关系示例：

```
@Entity public class  Employee {
    …
  @ManyToOne(fetch = LAZY)
   private Department dept;
    …
}

@Entity public class Department {
   …
   @OneToMany(mappedBy = “dept”, fetch = LAZY)
   private Collection<Employee> emps = new …;
   …
}
```

```
public int addNewEmployee(…) {
   Employee e = new Employee(..);
    Department d = departmentRepository.loadDepartment(…);

   e.setDepartment(d);
   // Reverse relationship is not set
   em.persist(e);
   em.persist(d);

   return d.getEmployees().size();
}
```

INCORRECT!

```
public int addNewEmployee(…) {
   Employee e = new Employee(..);
    Department d = departmentRepository.loadDepartment(…);

   e.setDepartment(d);
   d.getEmployees().add(e);
   em.persist(e);
   em.persist(d);
```

存储-对象关联关系（三）

- 对象关联关系映射策略：
  - 单向一对一是关联关系映射中最简单的一种，简单地或就是可以从关联的一方去查询另一方，却不能反向查询。

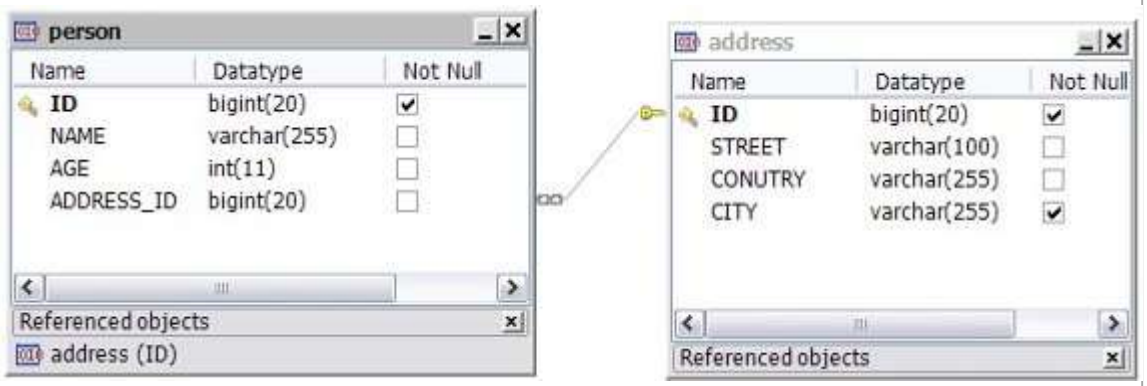
单向一对一关系的拥有端

```
@Entity
public class Person implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String name;
   private int age;

   @OneToOne
   private Address address;

   // Getters & Setters
}
```

单向一对一关系的反端

```
@Entity
public class Address implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String street;
   private String city;
   private String country;

    // Gettes& Setters
}
```

单向的一对一关系在数据库中是以外键的形式被映射的。其中关系的发出端存储一个指向关系的接收端的一个外键, 缺省情况下这个外键的字段名称，是以它指向的表的名称加下划线“_”加“ID”组成的, 当然我们也可以根据我们的喜好来修改这个字段，修改的办法就是使用 @JoinColumn 这个注解



存储-对象关联关系（四）

- 关联关系映射策略：
  - 双向一对一关系：

双向一对一关系中的接收端

```
@Entity
 public class  Address implements Serializable {
   @Id
   @GeneratedValue(strategy= GenerationType.AUTO)
   private Long id;
   private String street;
   private String city;
   private String country;

   @OneToOne(mappedBy="address")
   private Person person;

   //Gettes& Setters
}
```

双向关系有一方为关系的拥有端，另一方是关系的反端，也就是“Inverse”端。在这里例子中 Person 拥有这个关系，而 Address 就是关系的“Inverse”端。Address 中我们定义了一个 person 属性，在这个属性上我们使用了 @OneToOne 注解并且定义了他的“mappedBy”属性，这个在双向关系的“Inverse”端是必需的。

存储-对象关联关系（五）

- 对象关系维护的控制：
  - 单向OneToMany关系：

单向一对多关系的发出 端

```
@Entity
public class Person implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String name;
   private int age;
   @OneToMany
   private List<CellPhone> cellPhones;
   // Getters and Setters
}
```

单向一对多关系的接收端

```
@Entity
public class CellPhone implements Serializable {
   private static final long serialVersionUID = 1L;
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   private String manufacture;
   private String color;
   private Long  phoneNo;
   // Getters and Setters
}
```

在一对多关联关系映射中，默认是以中间表的方式来映射这种关系的。如在本例中，中间表为 person_cellphone，表的名称为关系的拥有端和 Inverse 端中间用下划线连接。中间表的字两个字段分别为两张表的得表名加下划线“_”加 ID 组成。当然我们也可以改表这种默认的中间表的映射方式，我们可以在关系的拥有端使用 @JoinClolum 来使用外键的方式映射这个关系。

存储-对象关联关系（六）

- 对象关系维护的控制：
  - 双向OneToMany关系：

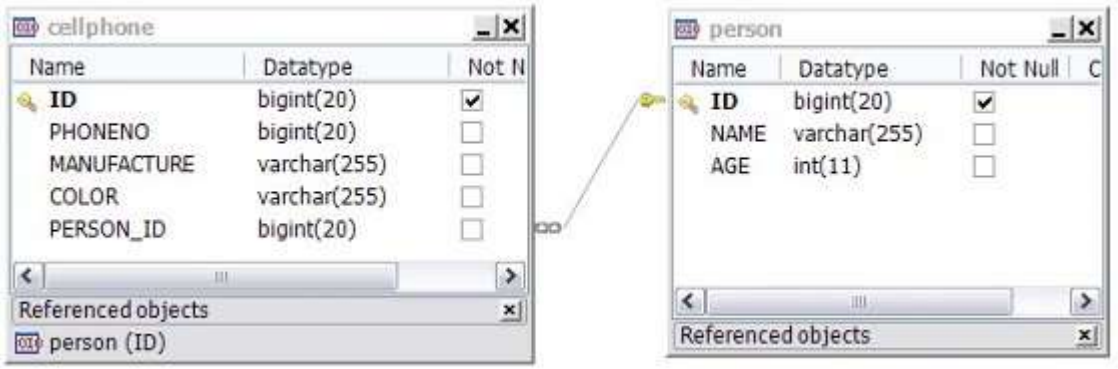
双向一对多关系的接受端

```
@Entity
public class Person implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    privateLong id;
    privateString name;
    privateint age;

    @OneToMany(mappedBy= "person")
    privateList<CellPhone> cellPhones;

    //Getters and Setters
}
```

双向一对多关系的发出端

```
@Entity
public class CellPhone implements Serializable {
    @Id
    @GeneratedValue(strategy= GenerationType.AUTO)
    privateLong id;
    privateString manufacture;
    privateString color;
    privateLong  phoneNo;

    @ManyToOne
    privatePerson person;

    //Getters and Setters
}
```

在@OneToMany里加入mappedBy属性避免生成中间表。在当前例子中，cellPhones这一端是关系的拥有者，CellPhone一方的表中生成到关联类的外键。



存储-对象关联关系（七）

- 对象关系维护的控制：
  - 单向ManyToMany关系：

单向多对多关系的发出端

```
@Entity
public class Teacher implements Serializable {
    @Id
    @GeneratedValue(strategy =GenerationType.AUTO)
    private Long id;
    private String name;
    private Boolean gender;
    private int age;
    private int height;

    @ManyToMany
    private List<Student> students;

    // Getters  and Setters
}
```

单向多对多关系的反端

```
@Entity
public class Student implements Serializable {
    @Id
    @GeneratedValue(strategy =GenerationType.AUTO)
    private Long id;
    private String name;
    private Boolean gender;
    private int age;
    private int height;

    //Getters and  Setters
}
```

多对多关联关系中只能通过中间表的方式进行映射。我们使用了 ManyToMany 这个注解来对 Teacher 中的 Students 进行注释，其中 Teacher 就是关系的发出端。而在 Student 中我们并没有作任何定义，这是单向多对多的所要求的。

存储-对象关联关系（八）

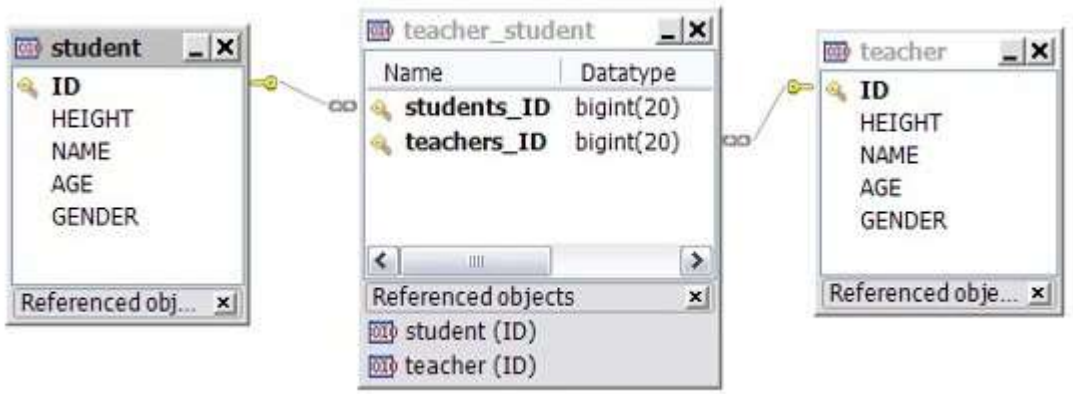
- 对象关系维护的控制：
  - 双向ManyToMany关系：

双向多对多关系的拥有端

```
@Entity
public class Teacher implements Serializable {
    @Id
    @GeneratedValue(strategy =GenerationType.AUTO)
    private Long id;
    private String name;
    private Boolean gender;
    private int age;
    private int height;

    @ManyToMany
    private List<Student> students;

    // Getters  and Setters
}
```

双向多对多关系的反端

```
@Entity
public class Student implements Serializable {
    @Id
    @GeneratedValue(strategy =GenerationType.AUTO)
    private Long id;
    private String name;
    private Boolean gender;
    private int age;
    private int height;

    @ManyToMany(mappedBy = "students")
    private List<Teacher> teachers;

    //Getters and  Setters
}
```



存储-对象关联关系（九）

- 关联对象加载策略：
  - EAGER – immediate
  - LAZY – loaded only when needed
  - LAZY适合对象数量多，或者嵌套层次深的关联对象
  - JPA中，1:m或m:n关系的默认加载策略是LAZY；
  - 实际应用中，对于大数据库(eg BLOB)的属性，以及不经常使用的关联对象，采用LAZY
  - 对于Detached对象，使用LAZY load会报异常（参考DO/DTO区别）

```
@Entity public class Department {
   @Id private int id;
   @OneToMany(mappedBy = “dept”)
   private Collectiion<Employee> emps;
      …
}
```

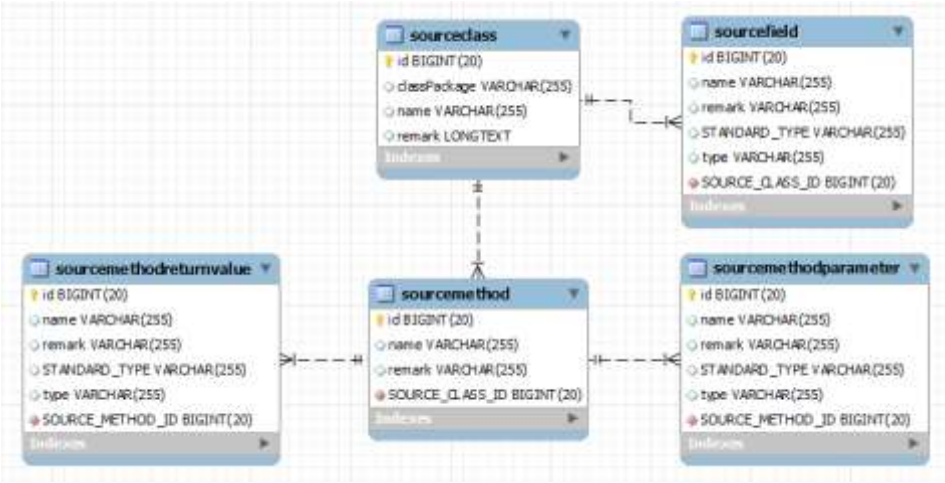
存储-对象关联关系（十）

- Cascade(级联)：
  - CascadeType.PERSIST(级联保存)：持久保存拥有方实体时，也会持久保存该实体的所有相关数据
  - CascadeType.REMOVE(级联删除)：删除一个实体时，也会删除该实体的所有相关数据
  - CascadeType.MERGE(级联更新)：将Detached(游离)的实体重新合并到活动的持久性上下文时，也会合并该实体的所有相关数据
  - CascadeType.REFRESH(级联刷新)：假如有一条数据(就有name[值为B]和sex[值为male]两个字段)，A用户取出来在进行修改操作(修改name为A)，正在A修改的过程中(未提交表单)，B用户也对这条数据进行修改操作(修改sex为female)，B先将性别修改后提交数据库...接着A用户也提交表单，但是，此时在entityManager中的持久化实体的性别为male，没有更新为B用户修改成的female，所以此时执行一次Refresh操作，就会将该实体更新为数据库中的最新记录，然后再进行提交..做级联的时候就会将关联的实体的也获取最新的然后再更新，前提是要执行Refresh操作，CasCadeType.Refresh才会生效
  - CascadeType.ALL(包含以上全部级联操作)
  - 是否应该使用级联应该是个业务问题而不是技术问题
    - 在一对多的级联中，如果对象之间是组合关系(UML术语)，使用级联；
    - 如果是聚合关系(UML术语)，不用级联。
    - 避免在深层嵌套关系中使用MERGE和ALL级联

存储-对象关联关系（十一）

- Cascade(级联)代码示例：

```
@Entity
@Table( uniqueConstraints = { @UniqueConstraint(columnNames = { "CLASSPACKAGE",  "NAME" }) })
public class SourceClass {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   // 类的包路径
   private String classPackage = "";
   // 类名称
   private String name = "";
   // 类描述
   @Column(name="remark", length=1000)
   private String remark = "";
   // 类的字段
   @OneToMany(mappedBy = "sourceClass",cascade=CascadeType.ALL)
   private Collection<SourceField> fields = new HashSet<SourceField>();
   // 类的方法集合
   @OneToMany(mappedBy = "sourceClass",cascade=CascadeType.ALL)
   private Collection<SourceMethod> methods = new HashSet<SourceMethod>();
   // …
}
```



存储-对象关联关系（十二）

- Cascade(级联)代码示例：

```
@Entity
public class Employee {
   // 类的字段
   @ManyToOne(cascade=CascadeType.PERSIST)
   private Address address;
   // …
}
```

值对象

如果不用Cascade, 则提交一个有address的Employee对象需要这样…

```
Address address = new Address();
em.persist(address);
employee.setAddress(address);
Em.persist(employee);
```

另举一个聚合根的例子（Planet是聚合根，Moon是非聚合根）：

```
@Entity  public class Planet extends AbstractDomainObject implements Auditable, Identifiable {
   // …
   @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, mappedBy = "planet", fetch = FetchType.EAGER)
   @ForeignKey(name = "FK_MOON_PLANET_PLANET", inverseName = "FK_MOON_PLANET_MOON")
   @NotNull
   private Set<Moon> moons = new HashSet<Moon>();
   // …
}
```

```
@Entity
@Table(name = "MOON")
@EntityListeners({ AuditListener.class })
public class Moon extends AbstractDomainObject implements Auditable, Identifiable {
   // …
}
```

Aggregate（一）

- 超大聚合的问题：
  - 并发访问时的数据一致性(consistency)以及事务失败问题(transactional failures)；
  - 性能和扩展性问题，如一次加载过多数据或内存不足等；

```
public class Product extends ConcurrencySafeEntity {
    private Set<BacklogItem> backlogItems;
    private String description;
    private String name;
    private ProductId productId;
    private Set<Release> releases;
    private Set<Sprint> sprints;
    private TenantId tenantId;
    ...
}
```

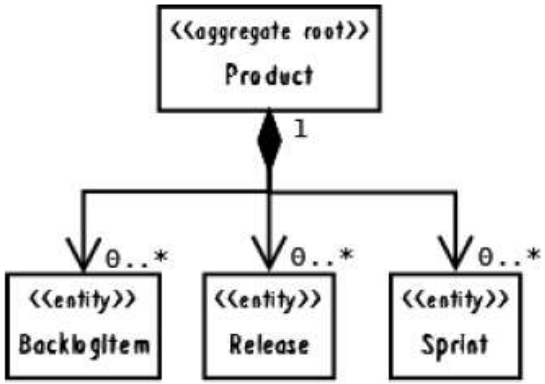


Figure 1: `Product` modeled as a very large **aggregate**.

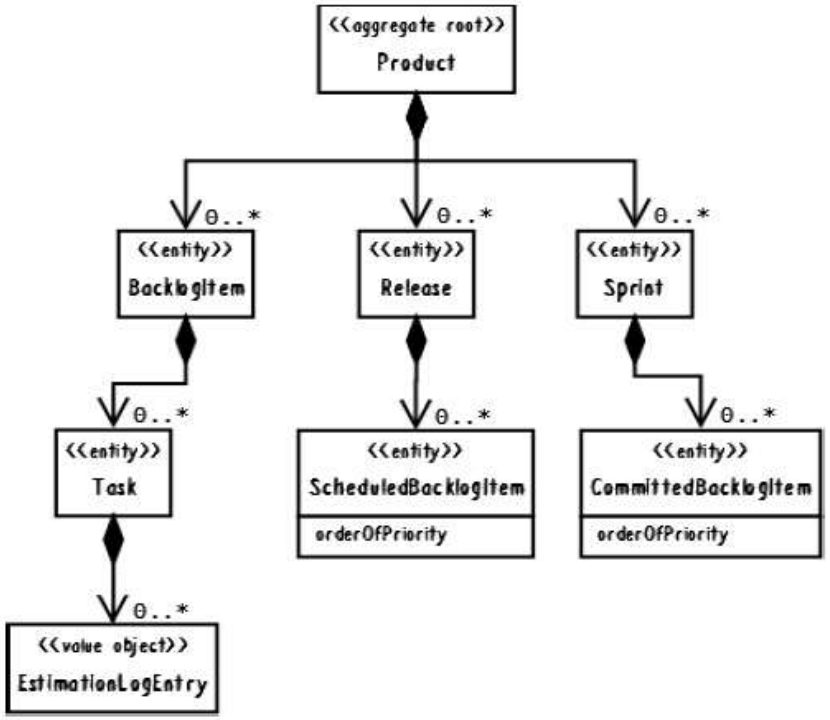


Figure 3: With this `Product` model, multiple large collections load during many basic operations.

Aggregate（二）

- 设计小聚合：
  - 探索业务中真正必须一致（强一致）的规则，依据一致性边界发现聚合；
  - 不要轻易相信所有用例，有时候数据的最终一致是可以接受的；

下例中，我们重新划分了四个聚合，每个聚合都依赖一个公共的值对象ProductId(Product领域实体的标识)。
采用领域服务协调

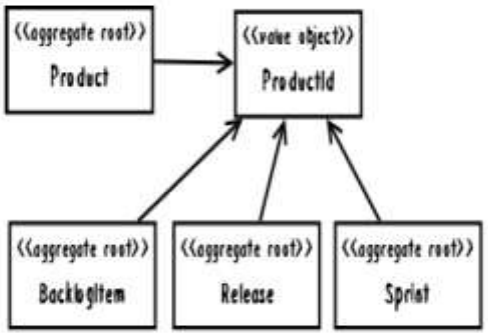


Figure 2: Product and related concepts are modeled as separate **aggregate** types.

```
public class Product ... {
   ...
   public BacklogItem planBacklogItem(
      String aSummary, String aCategory,
      BacklogItemType aType,
      StoryPoints aStoryPoints) {
       ...
   }

   public Release scheduleRelease(
      String aName, String aDescription,
      Date aBegins, Date anEnds) {
      ...
   }

   public Sprint scheduleSprint(
      String aName, String aGoals,
      Date aBegins, Date anEnds) {
      ...
   }
   ...
}
```

```
public class ProductBacklogItemService ... {
   ...
   @Transactional
   public void planProductBacklogItem(
      String aTenantId, String aProductId,
      String aSummary, String aCategory,
      String aBacklogItemType, String aStoryPoints) {

      Product product =
         productRepository.productOfId(
            new TenantId(aTenantId),
            new ProductId(aProductId));

      BacklogItem plannedBacklogItem =
         product.planBacklogItem(
            aSummary,
            aCategory,
            BacklogItemType.valueOf(aBacklogItemType),
            StoryPoints.valueOf(aStoryPoints));

      backlogItemRepository.add(plannedBacklogItem);
   }
   ...
}
```

Aggregate（三）

- 设计小聚合：如何解决聚合间相互引用问题
  - 在聚合内引用其他聚合对象
    - 一致性边界控制问题：人们可能在一个聚合内，修改其它聚合对象；
    - 模型加载与遍历的性能问题

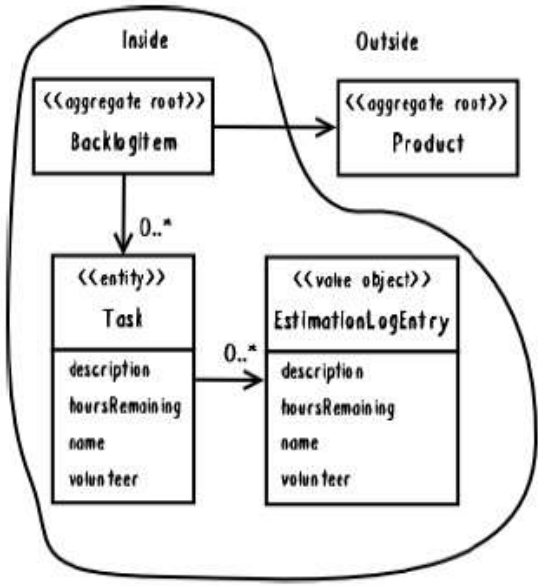


Figure 5: There are two **aggregates**, not one.

```
public class BacklogItem extends ConcurrencySafeEntity {
   ...
   private Product product;
   …
}
```

Aggregate（四）

- 设计小聚合：如何解决聚合间相互引用问题
  - 在聚合内使用标识(Value Object)引用其他聚合；

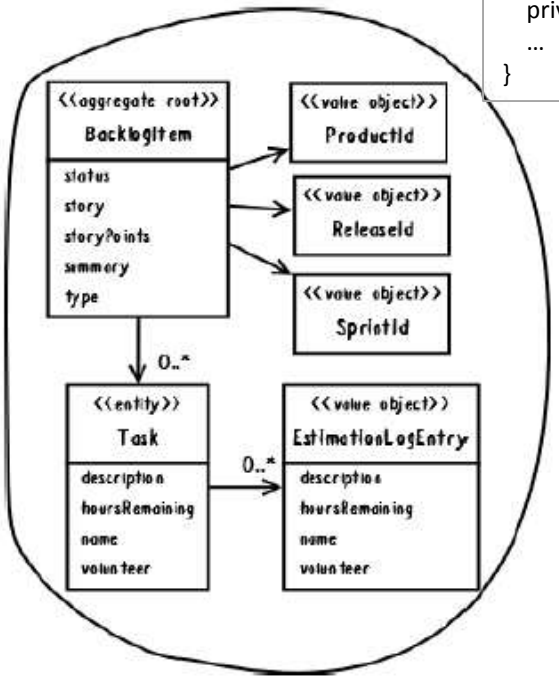


Figure 6: The `BacklogItem` **aggregate**, inferring associations outside its boundary with identities.

```
public class BacklogItem extends ConcurrencySafeEntity
{
   ...
   private ProductId productId;
   ...
}
```

```
public class ProductBacklogItemService ... {
   ...
   @Transactional
   public void assignTeamMemberToTask (
      String aTenantId,
      String aBacklogItemId,
      String aTaskId,
      String aTeamMemberId) {
      BacklogItem backlogItem =
         backlogItemRepository.backlogItemOfId(
            new TenantId(aTenantId) , new BacklogItemId(aBacklogItemId));
      Team ofTeam = teamRepository.teamOfId(
         backlogItem.tenantId(), backlogItem.teamId());
      backlogItem.assignTeamMemberToTask(
         new TeamMemberId(aTeamMemberId),
         ofTeam,
         new TaskId(aTaskId));
   }
   ...
}
```

领域服务协调聚合间访问

```
public class BacklogItem extends ConcurrencySafeEntity {
   ...
   public void  commitTo(Sprint aSprint) {
      ...
      DomainEventPublisher
         .instance()
         .publish(new BacklogItemCommitted(
            this.tenantId(),  this.backlogItemId(), this.sprintId()));
      ...
}
```

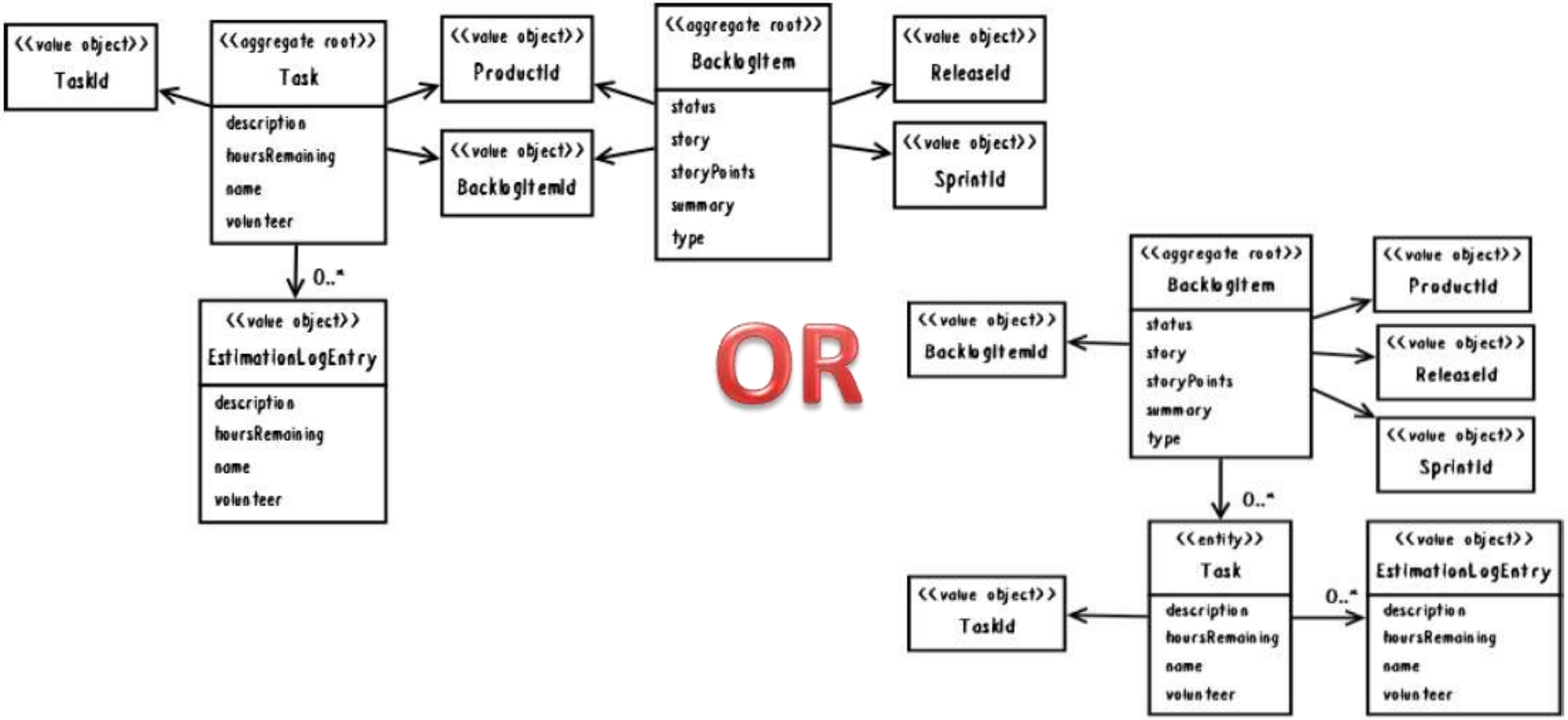
采用最终一致方式保证聚合间数据一致性

Aggregate（五）

- 应遵循把聚合作为数据一致性边界的原则，允许有以下例外（谨慎使用）：
  - 用户交互的便捷性：如批量导入/更新等用户操作行为；
  - 全局事务，两阶段提交事务；
  - 因为查询性能，需要直接在聚合中引用其他聚合对象；
  - 技术框架不支持。

Aggregate（六）

- 是否需要进一步划分聚合应衡量哪些因素？
  - “通用语言”，即业务的一致性规则要求；
  - 衡量聚合的代价：一次加载对象数，内存消耗，以及通用的使用场景(出现频率)



Aggregate（七）

- **Aggregate Object**
  - Sometimes the responsibility of the whole aggregate doesn’t really belong to the Aggregate Root entity.
  - Sometimes preserving integrity of the aggregate as a whole is a responsibility that deserves a role of its own
  - An Aggregate Object might coordinate invariants checking and state management between the different entities of the aggregate

存储-事务控制（一）

- 事务控制的边界位于服务层(Service Layer)

```
public class ProductBacklogItemService … {
   …
   @Transactional
   public void assignTeamMemberToTask (
      String aTenantId,
      String aBacklogItemId,
      String aTaskId,
      String aTeamMemberId) {
      BacklogItem backlogItem =
         backlogItemRepository.backlogItemOfId(
            new TenantId(aTenantId) , new BacklogItemId(aBacklogItemId));
      Team ofTeam = teamRepository.teamOfId(
         backlogItem.tenantId(), backlogItem.teamId());
      backlogItem.assignTeamMemberToTask(
         new TeamMemberId(aTeamMemberId),
         ofTeam,
         new TaskId(aTaskId));
   }
   …
}
```

存储-事务控制（二）

- 事务管理方式：
  - JTA：全局事务
  - RESOURCE_LOCAL：本地事务

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
       version="2.0">
  <persistence-unit name="DDDSampleEntityManagerFactory" transaction-type="RESOURCE_LOCAL">
     <description>JPA configuration for DDDSample </description>
     <provider>org.hibernate.ejb.HibernatePersistence</provider>
     <non-jta-data-source>java:comp/env/jdbc/applicationDS</non-jta-data-source>
     <!-- annotated classes -->
     <class>org.sculptor.dddsample.cargo.domain.Cargo</class>
     <class>org.sculptor.dddsample.carrier.domain.CarrierMovement</class>
     <class>org.sculptor.dddsample.carrier.domain.CarrierMovementId</class>
     <class>org.sculptor.dddsample.cargo.domain.HandlingEvent</class>
     <class>org.sculptor.dddsample.cargo.domain.Itinerary</class>
     <class>org.sculptor.dddsample.cargo.domain.Leg</class>
     <class>org.sculptor.dddsample.location.domain.Location</class>
     <class>org.sculptor.dddsample.routing.domain.RtCarrierMovement</class>
     <class>org.sculptor.dddsample.routing.domain.RtLocation</class>
     <class>org.sculptor.dddsample.cargo.domain.TrackingId</class>
     <class>org.sculptor.dddsample.routing.domain.TransitEdge</class>
     <class>org.sculptor.dddsample.routing.domain.TransitPath</class>
     <class>org.sculptor.dddsample.location.domain.UnLocode</class>
     <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
     <validation-mode>AUTO</validation-mode>
     <!-- properties -->
     <properties>
        <property name="hibernate.dialect" value="org.sculptor.framework.persistence.CustomHSQLDialect" />
        <property name="query.substitutions" value="true 1, false 0" />
```

存储-事务控制（三）

- Transaction Propagation属性值
  - Required：在有transaction状态下执行；如果当前没有transaction，则创建新的transaction。这是最常用的属性，也是默认属性。
  - Mandatory：必须在有transaction状态下执行，如果当前没有transaction，则抛出异常IllegalTransactionStateException。通常在“Client Orchestration transaction strategy”中使用。
  - RequiresNew：创建新的transaction并执行，如果当前已有transaction，则将当前transaction挂起。通常在Auditing或Logging等数据操作情形下使用，因为这些操作与整个基础事务的成败没有关系。
  - Supports：如当前有transaction，则在transaction状态下执行，如果当前没有transaction，在无transaction状态下执行。主要用于数据库的只读操作，与NotSupported不同，Supports属性在有当前事务时，读取数据可以保持与当前事务的数据保持一致。
  - NotSupported：在无transactioin状态下执行；如果当前已有transaction，则将当前transaction挂起。通常在有数据库存储过程情形下使用（数据库不支持嵌套事务）。
  - Never：在无transaction状态下执行；如果当前已有transaction，则抛出异常IllegalTransactionStateException。唯一的使用场景可能就是测试了…

Ref: http://www.ibm.com/developerworks/java/library/j-ts2/index.html

存储-事务控制（四）

- 事务控制需要注意的事项：
  - 在事务中，不要运行昂贵而不必要的、与事务无关的操作指令，如日志记录，其磁盘读写代价高，非常消耗资源
  - 不要在浏览数据的时候打开事务(设置值@TransactionAttribute(NOT_SUPPORTED))

存储-事务控制（五）

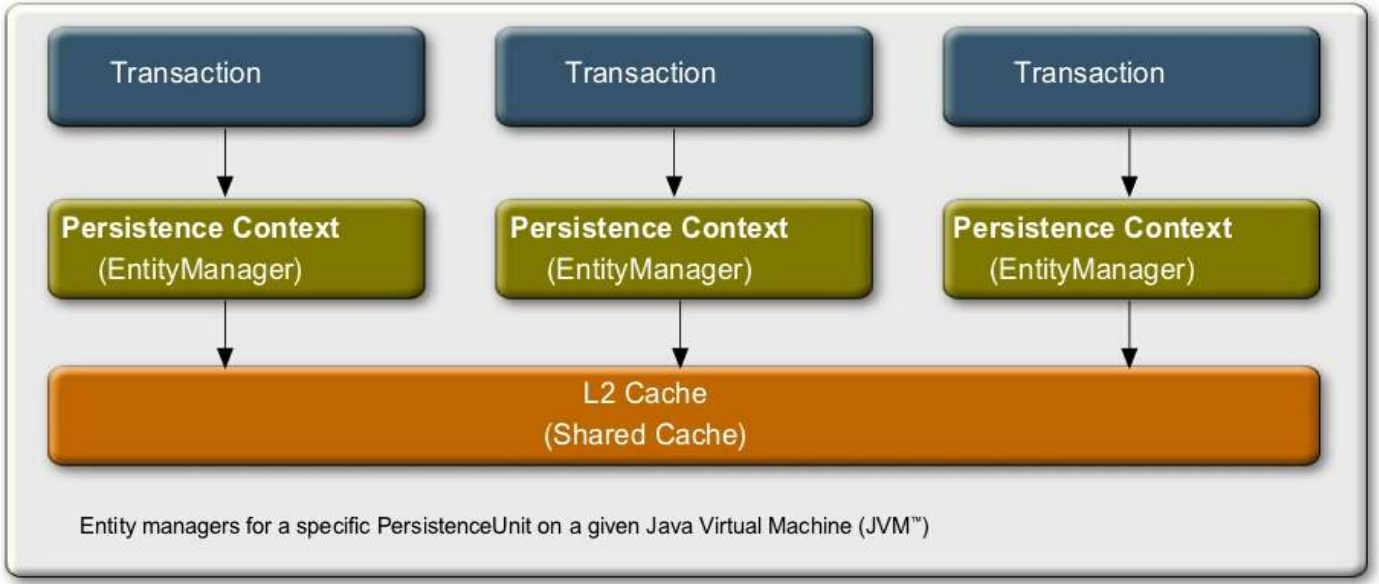
- 事务策略：
  - Client Orchestration transaction strategy：http://www.ibm.com/developerworks/java/library/j-ts4/index.html
  - API Layer transaction strategy：http://www.ibm.com/developerworks/java/library/j-ts3/index.html
  - High Concurrency transaction strategy：http://www.ibm.com/developerworks/java/library/j-ts5/index.html
  - High-Speed Processing transaction strategy：http://www.ibm.com/developerworks/java/library/j-ts6/index.html

存储-缓存（一）

- 经过领域建模以后，整个模型对象都是完全面向业务的，并且具有丰富的行为。而对象和数据库之间又存在天然的矛盾，这种矛盾主要体现在领域对象保存到数据库时，需要把其一层层打开，然后放入数据库，而从数据库里生成领域对象又需要将裸体的数据穿上衣服，最终形成领域对象，这个过程非常的麻烦。因此我们需要把领域对象用完以后放到某一种地方，而这种地方可以方便快捷地取出对象和放入对象，这个对象其实就是缓存。*缓存降低了应用程序对持久性数据源的访问，从而使得应用程序具有更好的性能*。
- 缓存是领域对象在内存中的生存场所，是一种面向业务的存储方式，而同时我们的领域模型也是一种面向业务的模型，有了面向业务的存储以后，我们就可以进行面向业务的运算，而正是这种面向业务的运算使得我们的系统具有更好的伸缩性和扩展性。因为此时的领域对象通过缓存都是跑在中间件中，而在负载增多的时候，通过水平的增加中间件服务器来进行水平伸缩。
- 缓存+领域模型是面向业务的对象模型，面向业务的存储，面向业务的运算结合的基础，而数据库则是一种完全面向数据的存储方式，因此数据库思维和对象模型思维是不匹配的。
- 领域模型不应当关注缓存的技术实现细节，Repository是恰当的隐藏缓存技术实现的地方。

存储-缓存（二）

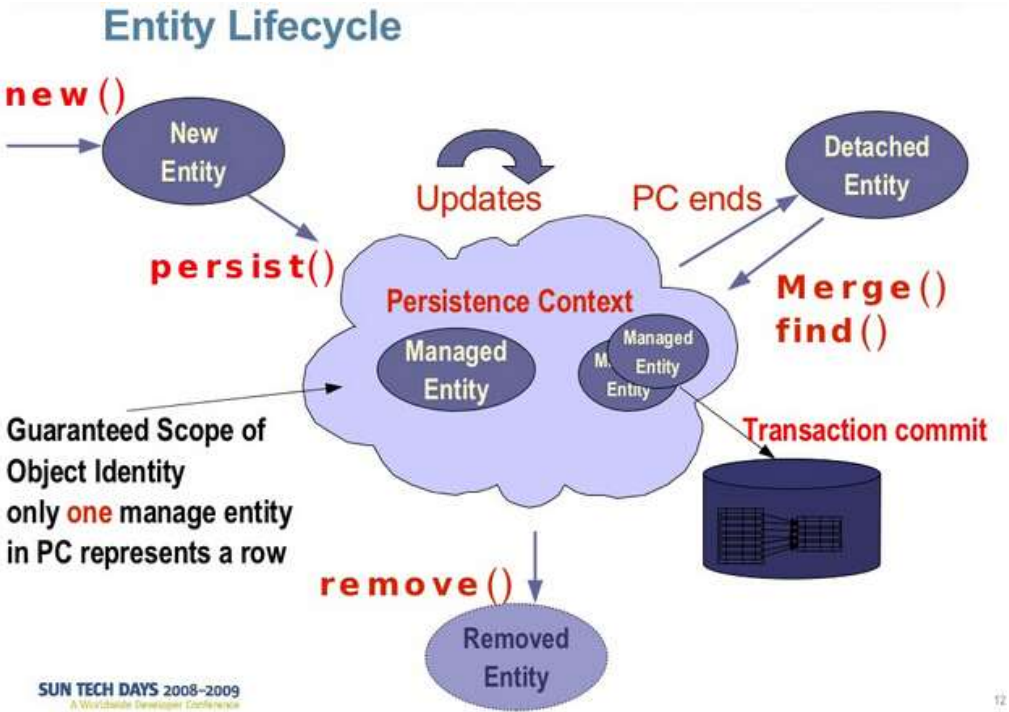
- 二级缓存机制与查询缓存
  - 二级缓存机制
    - 一级缓存：生命周期和session的生命周期一致，当前session一旦关闭，一级缓存就会消失，因此一级缓存也叫session缓存或者事务级缓存。
    - 二级缓存：也称为进程缓存或者sessionFactory级的缓存，它可以被所有的session共享，二级缓存的生命周期和sessionFactory的生命周期一致。二级缓存一般针对经常被读、很少被修改、过期也不产生重大影响的Entity对象。
  - 查询缓存：二级缓存策略是针对于ID查询的缓存机制，对于条件查询则毫无作用。针对条件查询可以启用QueryCache，查询缓存中以键值对的方式存储，key键为查询的条件语句，value为查询之后等到的结果集的ID列表。当数据表发生数据变动的话，hibernate就会自动清除查询缓存中对应的QueryKey

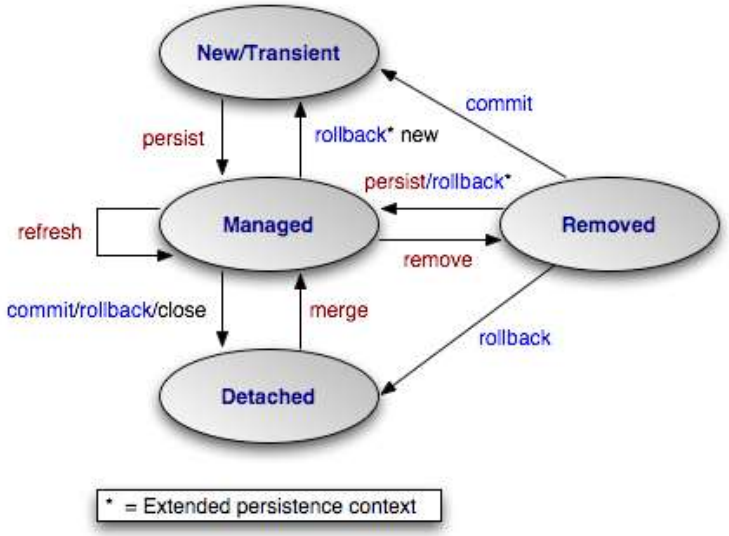


存储-缓存（三）

- JPA一级缓存(Persistence Context)工作机制，以及JPA Entity的生命周期
  - New / Transient：新创建的实体对象，没有主键值
  - Managed：对象处于Persistent Context（持久化上下文）中，被EntityManager管理
  - Detached：对象已经游离到Persistent Context之外，进入Application Domain
  - Removed：实体对象被删除

***注意JPA Context中的Entity和DDD Context中的Entity的概念区别！**





存储-缓存（四）

- Hibernate JPA中配置Ehcache二级缓存示例
  - (1)JPA的persistence.xml配置

```
<property name="hibernate.cache.provider_class"
          value="org.hibernate.cache.SingletonEhCacheProvider" />
<property name="hibernate.cache.provider_configuration" value="/ehcache.xml" />
<property name="hibernate.cache.use_second_level_cache" value="true" />
<property name="hibernate.cache.use_query_cache" value="true" />
```

存储-缓存（五）

- Hibernate JPA中配置Ehcache二级缓存示例
  - (2)对ehcache进行简单的配置(ehcache.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
    <ehcache>
<!--

maxElementsInMemory:缓存中最大允许创建的对象数

eternal:缓存中对象是否为永久的，如果是，超时设置将被忽略，对象从不过期

timeToIdleSeconds:缓存数据钝化时间(设置对象在它过期之前的空闲时间)

timeToLiveSeconds:缓存数据的生存时间(设置对象在它过期之前的生存时间)

overflowToDisk:内存不足时，是否启用磁盘缓存

clearOnFlush:内存数量最大时是否清除


-->
    <defaultCache maxElementsInMemory="1000" eternal="false"
        timeToIdleSeconds="1200" timeToLiveSeconds="1200" overflowToDisk="false"
        clearOnFlush="true">
    </defaultCache>
     <!-- 单独对某个entity的缓存策略设置-->
    <cache name="com.payment.entity.PromotionEntity" maxElementsInMemory="100"
        eternal="false"
        timeToIdleSeconds="1200" timeToLiveSeconds="1200" overflowToDisk="false"
        clearOnFlush="true">
    </cache>
   </ehcache>
```

示例中启用了Ecache非分布式缓存，也可以结合Terracotta启用分布式缓存。

存储-缓存（六）

- Hibernate JPA中配置Ehcache二级缓存示例
  - (3)JPA的Entity类中生命缓存的隔离机制

```
import org.hibernate.annotations.Cache;
    import org.hibernate.annotations.CacheConcurrencyStrategy;

    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
    @Entity
    @Table(name = "catagory")
    public class CatagoryEntity extends BaseEntity { ... }
```

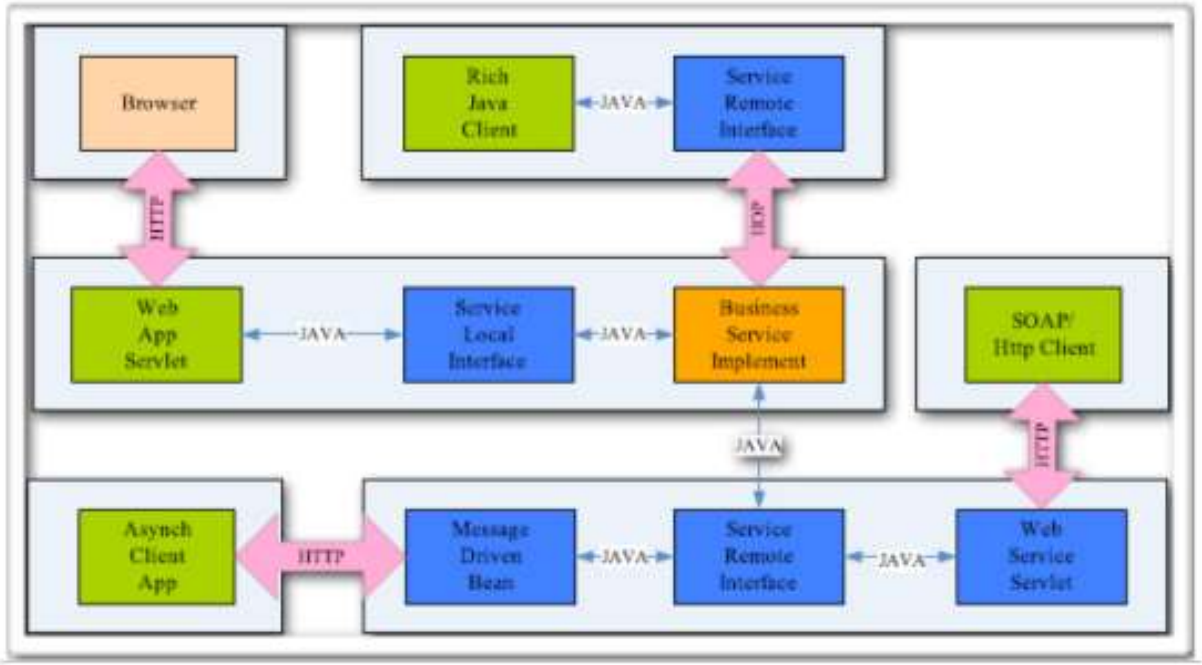
Event

- 事件的两大类别：Command和Event
- 两类事件的命名方式：
  - Command：主动式，如PlaceOrder / PlaceOrderCmd
  - Event：被动式，如OrderPlaced / OrderPlacedEvent

Service

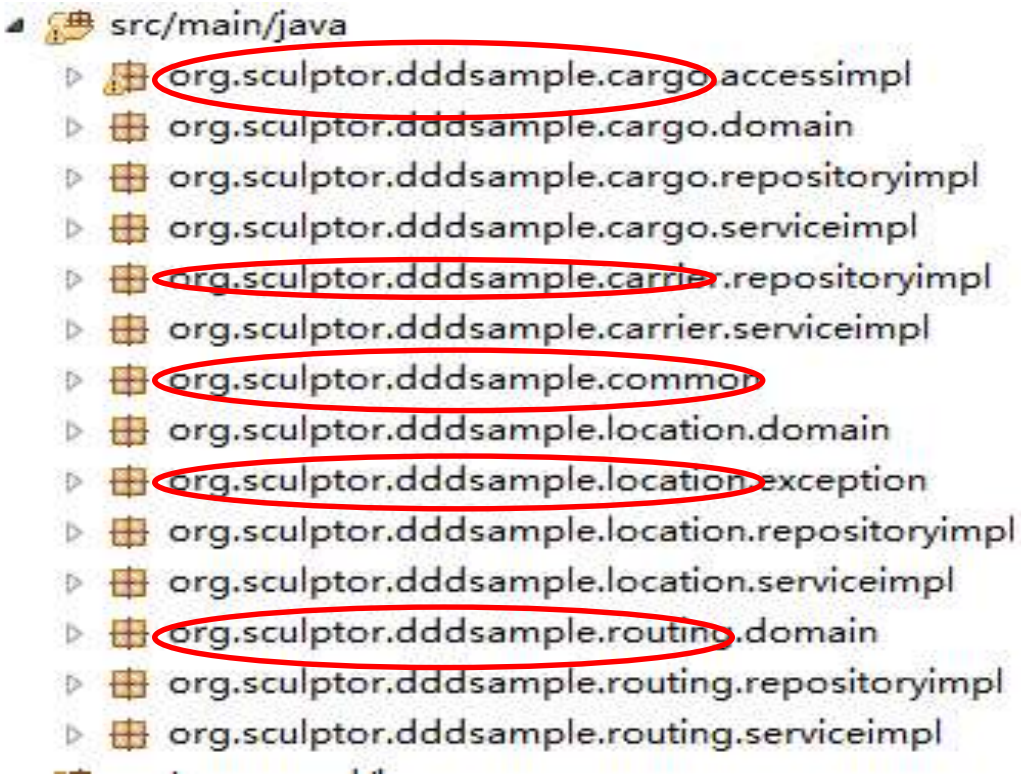
- Service间通信方式
  - 本地服务通信
  - 远程服务通信
    - Synchronous HTTP ⇔ asynchronous AMQP
    - Formats: JSON, XML, Protocol Buffers, Thrift, ...
    - Even via the database

*Asynchronous is preferred
*JSON is fashionable but binary format is more efficient



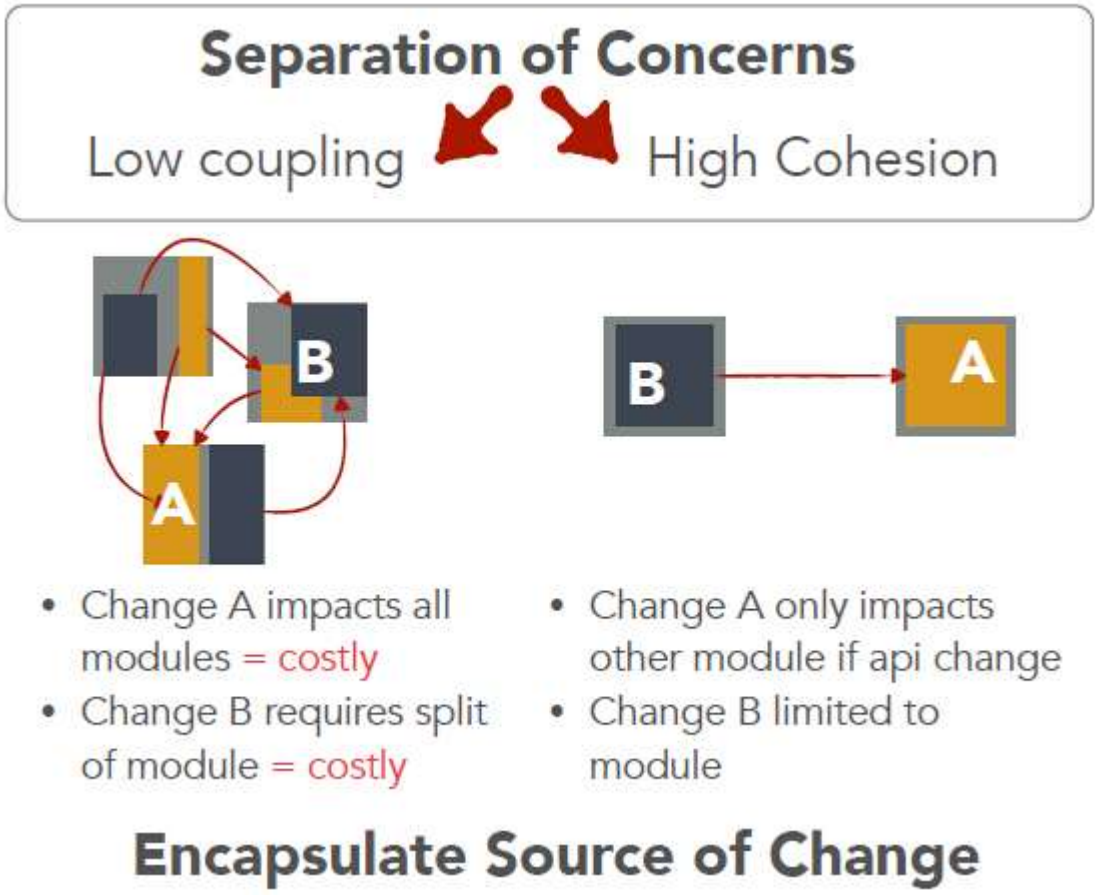
Module (一)

- 模块在实际语言中的映射
  - 模块在Java语言实现中用包（Package）表示
  - 模块和模块之间不允许存在相互依赖。
  - 一个模块中的服务(Service)不允许直接访问另一个模块的仓储(Repository)，它们之间必须通过服务接口进行访问。



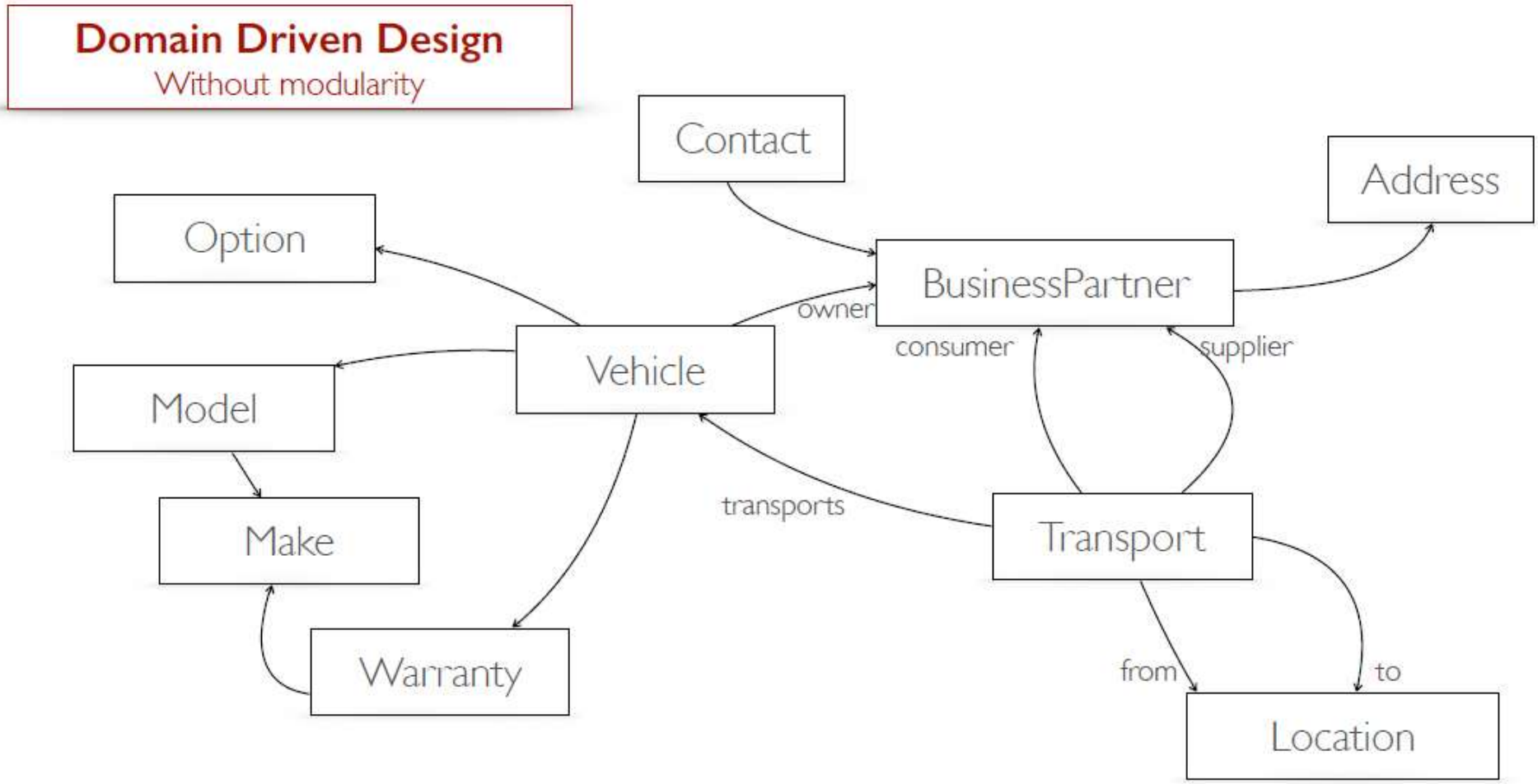
Module（二）

- 模块设计的原则：低耦合，高内聚



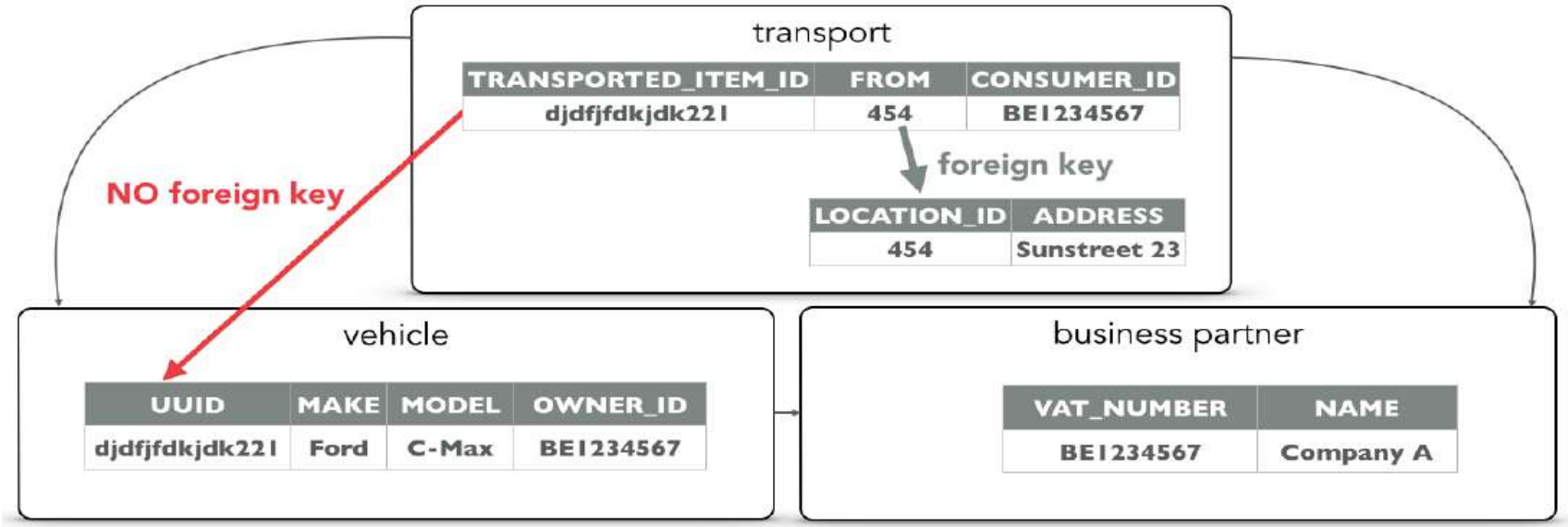
Module (三)

- 示例：DDD without modularity



Module（四）

- 模块化数据库Schema设计要求
  - 数据库结构不允许跨领域(模块)
  - 查询不允许跨领域边界



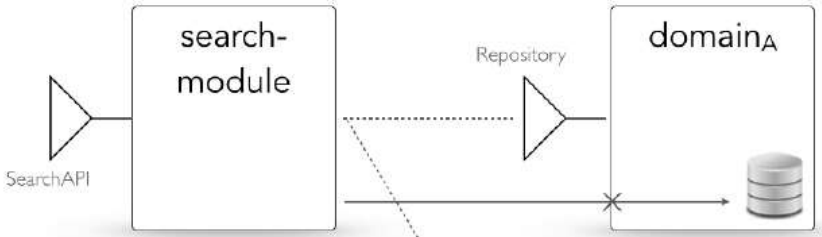
好处：

- 模块可以独立迁移到下一版本
- 在一个系统中，不同模块可以采用不同存储/数据库技术

Module（五）

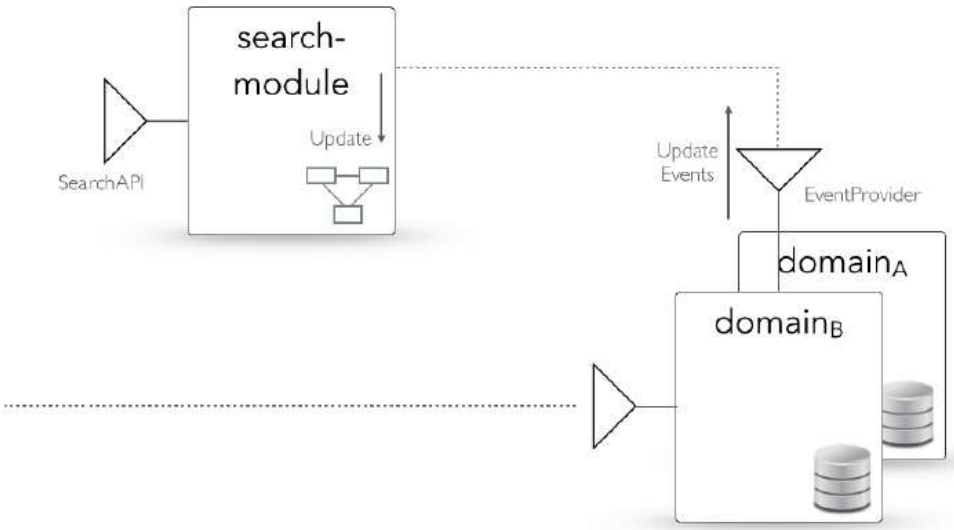
- 跨边界查询两种方案对比：

A. Search encapsulated in a separate module leveraging the functionality of other domain modules



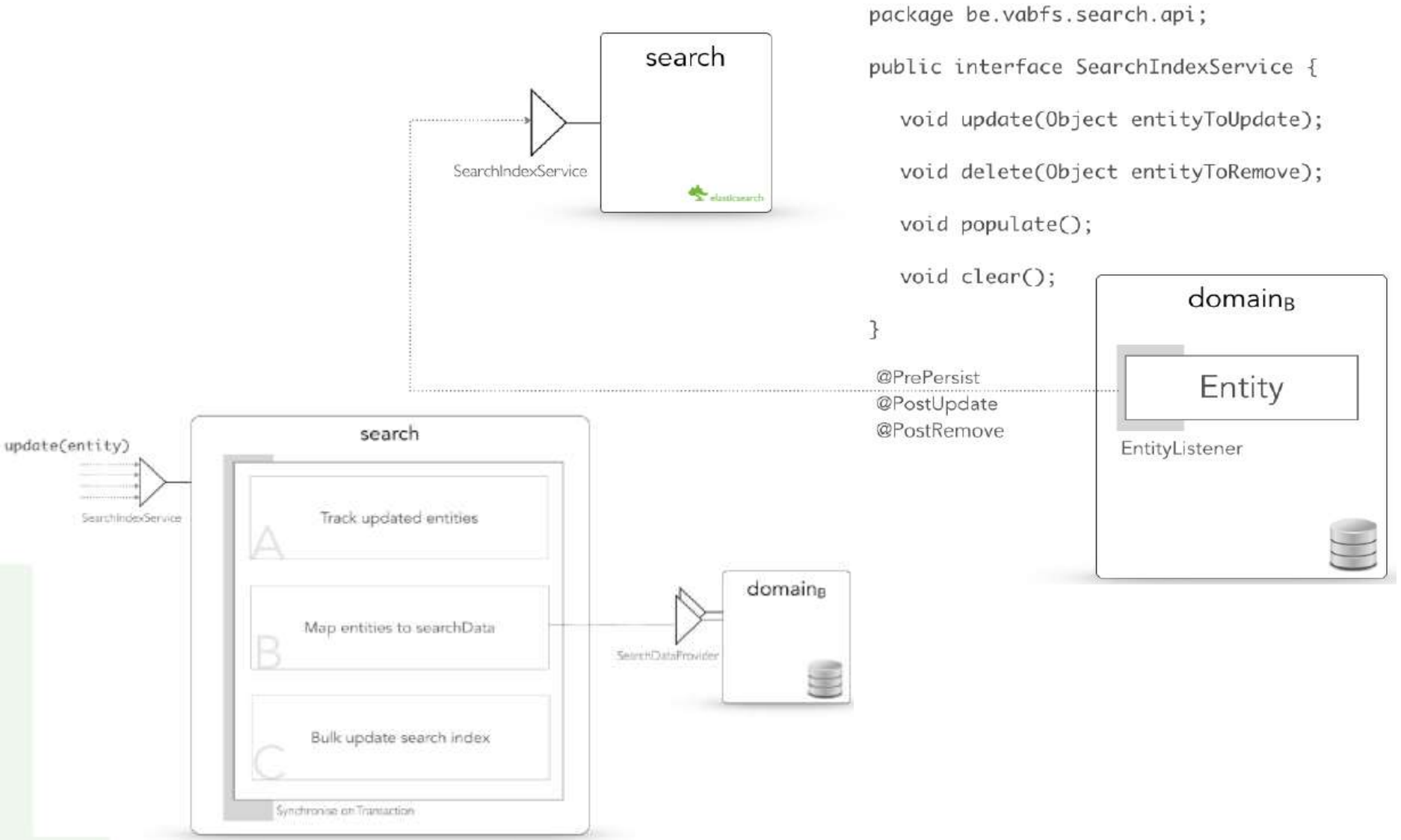
- requires specific implementation for every search
- complexity increases with number of modules
- inability to leverage power of the datastore

B. Search is using a separate query model



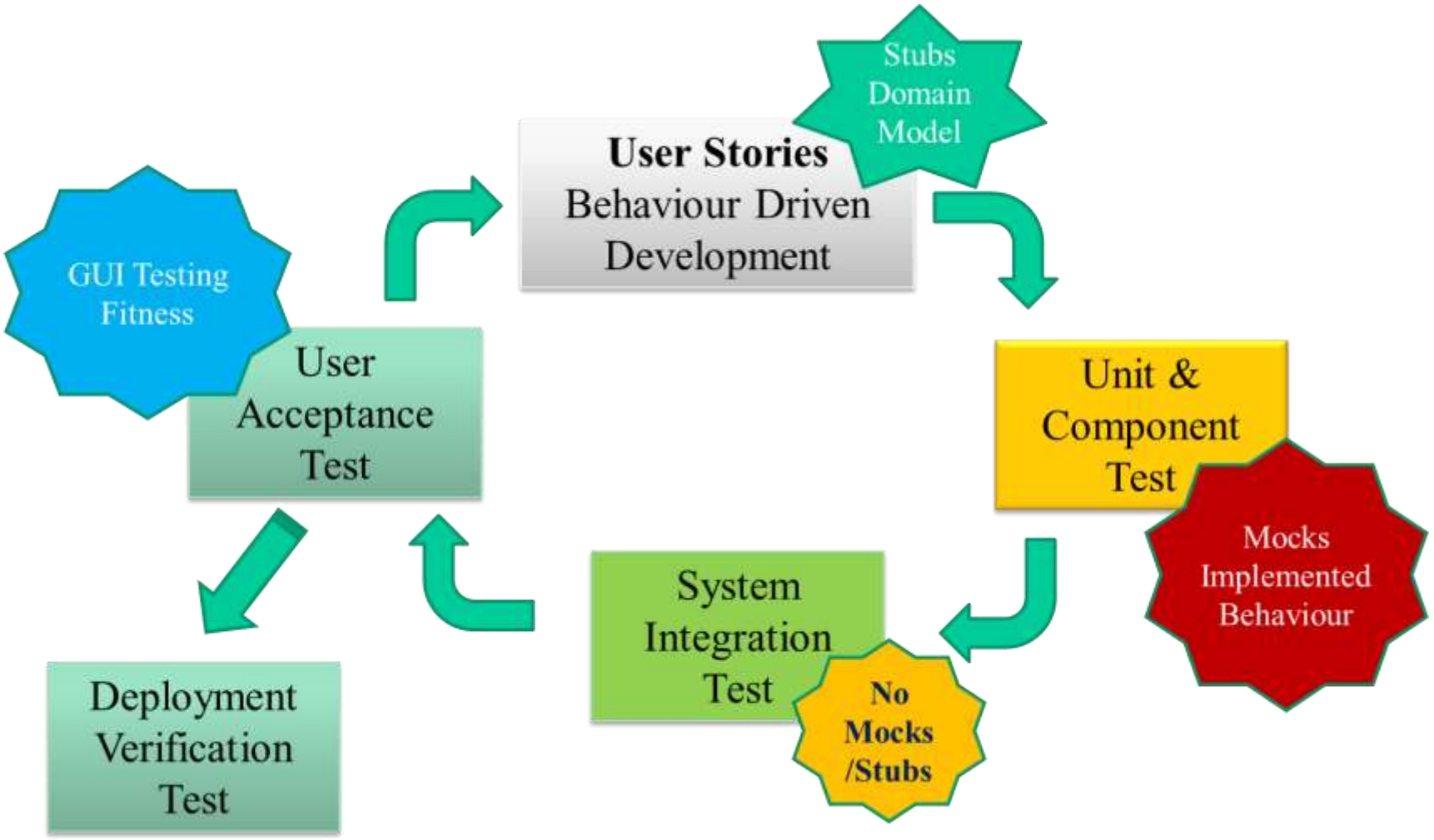
Module（六）

- 使用ElasticSearch实现跨模块查询视图：



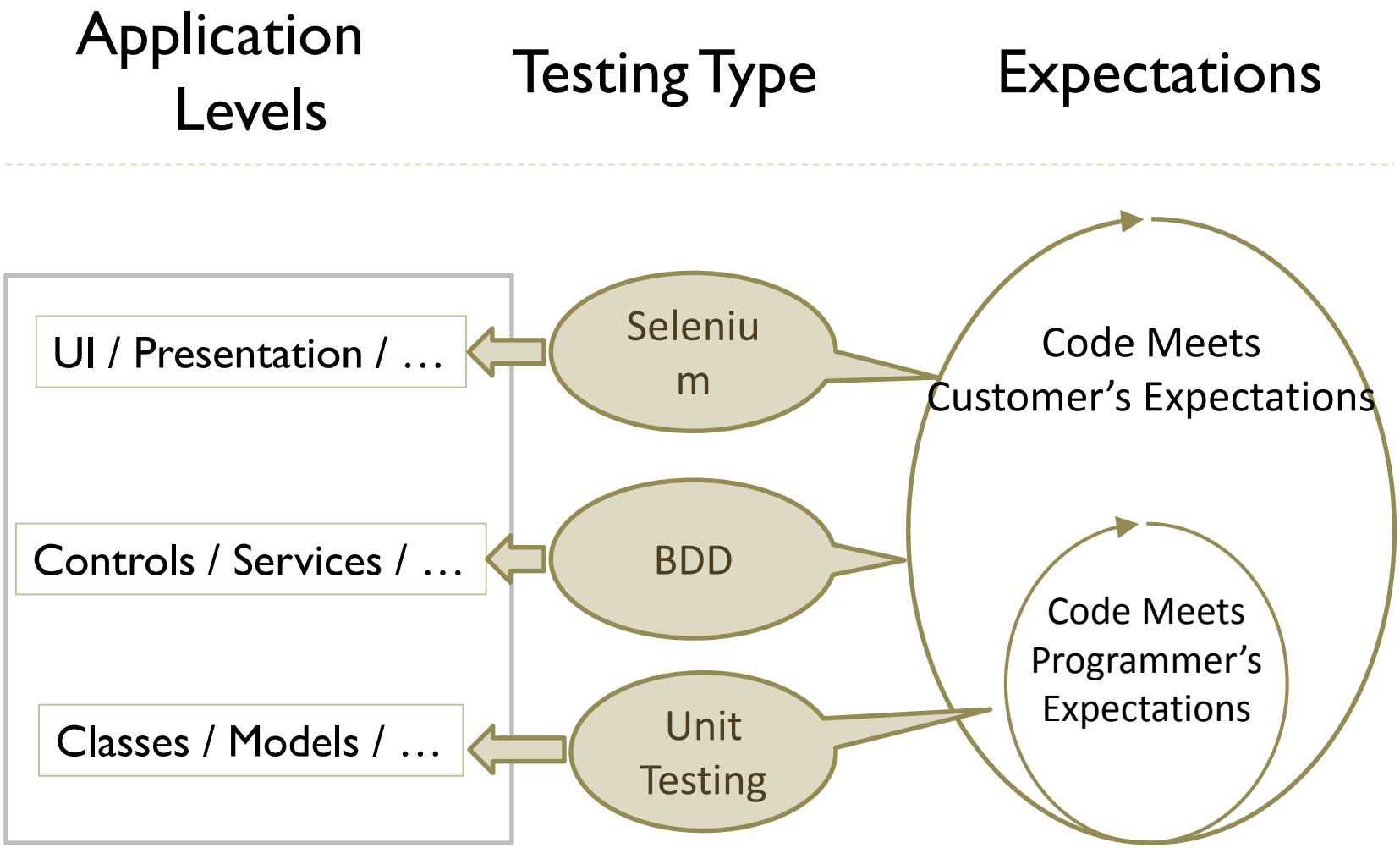
测试（一）

- 领域驱动设计软件开发/测试各阶段



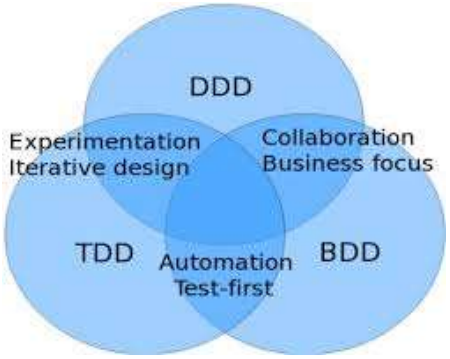
测试（二）

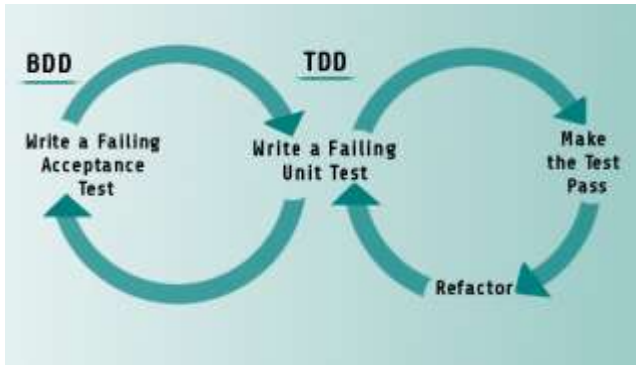
- 测试的目的与相应方法

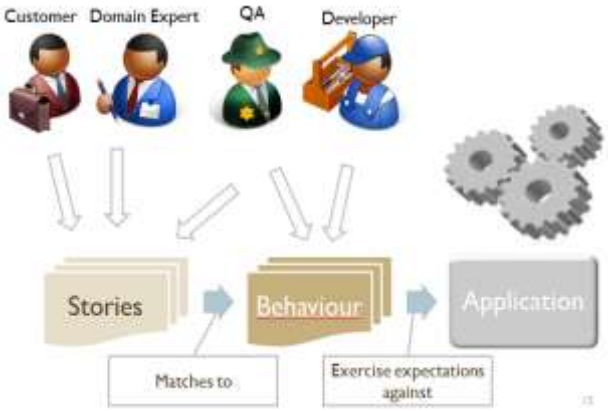


测试（三）

- 行为驱动开发(Behavior Driven Development)：BDD = TDD + DDD
  - 测试驱动开发(TDD)，是一种新型的开发方法，要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简介可用和高质量的代码，并加速开发过程。(Ref: 《Test-Driven Development By Example》, Kent Beck)
  - 行为驱动开发(BDD)：BDD的重点是通过与利益相关者的讨论取得对预期的软件行为的清醒认识。它通过用自然语言书写非程序员可读的测试用例扩展了测试驱动开发方法。行为驱动开发人员使用混合了领域中统一的语言的母语语言来描述他们的代码的目的，使得开发者得以把精力集中在代码如何实现**商业价值**上。
  - TDD和BDD都是Test-First Development，都借助自动化测试工具。BDD是对TDD的拓展，把关注点从“测试”本身，转移到“商业价值”上来。





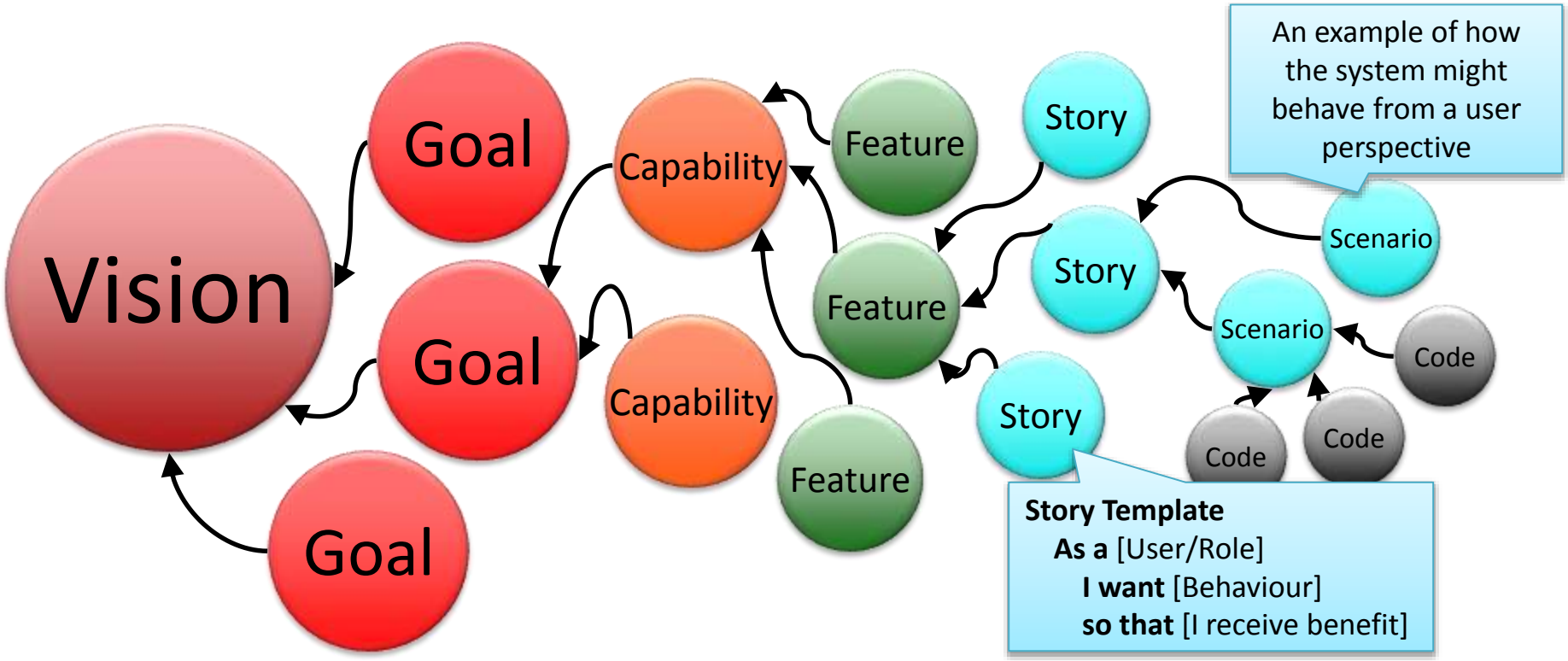


测试（四）

- 行为驱动开发：用举例来阐述行为

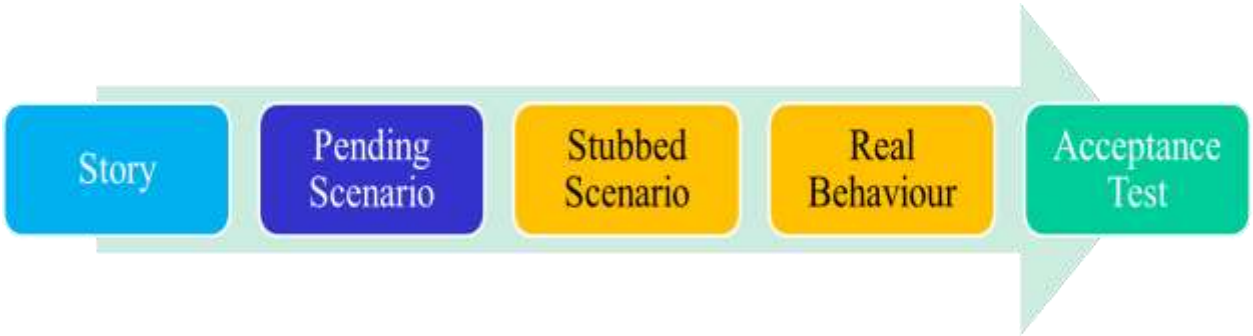
Given Fred has bought a microwave
And the microwave cost £100
When we refund the microwave
Then Fred *should* be refunded £100.

Given a context
When an event happens
Then an outcome *should* occur



测试（五）

- 在项目中实施行为驱动开发的步骤
  1. For each scenario describing a feature
  2. Run the scenario – it fails (go red)
  3. Define the first step – go red
  4. Write down the application code getting the step to pass – go green
  5. Refactor the code and repeat steps 4 & 5 for each step until 6
  6. The scenario passes – go green
  7. Refactor the application code



* 测试驱动开发或者行为驱动开发在做法上的要求是一致的，即在编写代码的过程中，不要急于探索如何进行内部实现的细节或机制，而是首先考虑客户（调用方/使用者）使用软件（应用/模块/类/...）的场景。满足客户的需求，不多也不少。

测试（六）

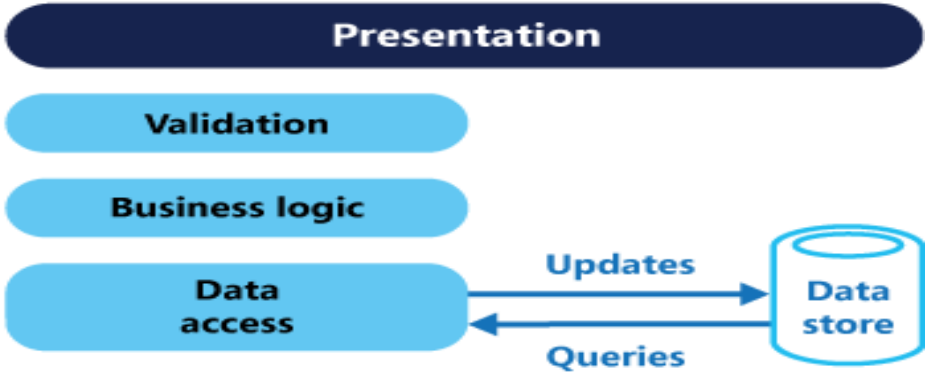
- 自动化测试常用工具
  - 单元测试框架：Junit (http://junit.org/)
  - 行为驱动测试框架：Jbehave (Ref: http://jbehave.org/)
  - Mock框架: EasyMock(http://www.**easymock**.org/)、JMockit(http://jmockit.github.io/)等
  - 内存数据库：HSQLDB(http://**hsqldb**.org/)

培训内容

- 领域驱动设计简介
- 领域通用语言
- 领域驱动设计的构造块
- 领域驱动设计编程实践
- CQRS架构
- 模型驱动开发

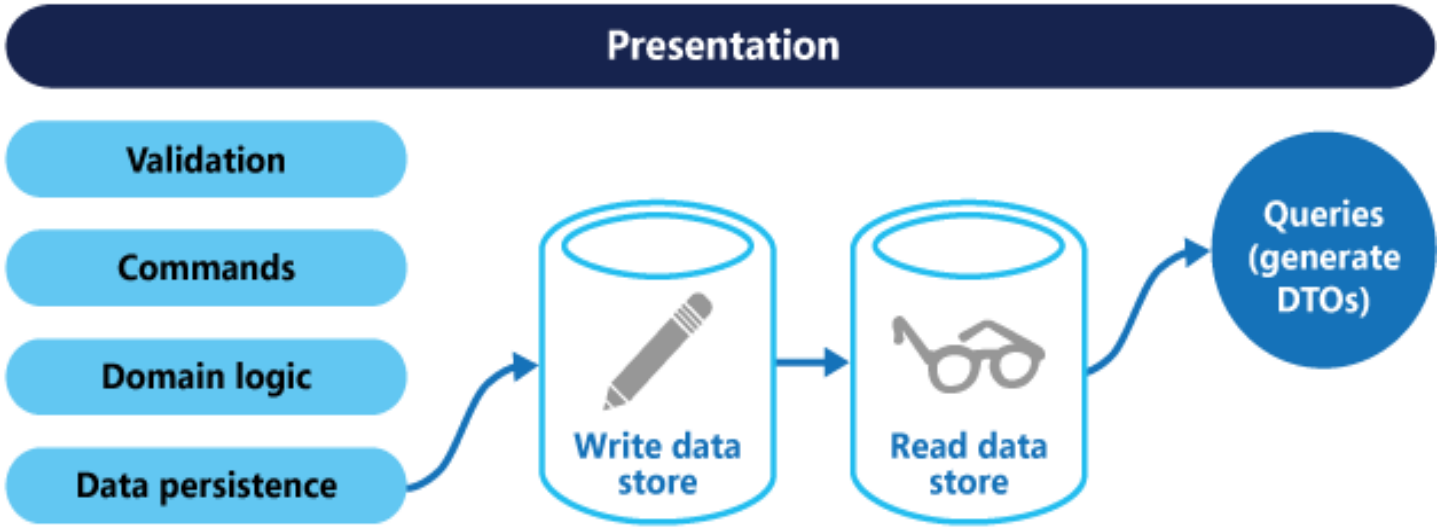
传统的CRUD方法

- 传统的CRUD方法的问题：
  - 使用同一个对象实体来进行数据库读写可能会太粗糙，大多数情况下，比如编辑的时候可能只需要更新个别字段，但是却需要将整个对象都穿进去，有些字段其实是不需要更新的。在查询的时候在表现层可能只需要个别字段，但是需要查询和返回整个实体对象。
  - 使用同一实体对象对同一数据进行读写操作的时候，可能会遇到资源竞争的情况，经常要处理的锁的问题，在写入数据的时候，需要加锁。读取数据的时候需要判断是否允许脏读。这样使得系统的逻辑性和复杂性增加，并且会对系统吞吐量的增长会产生影响。
  - 同步的，直接与数据库进行交互在大数据量同时访问的情况下可能会影响性能和响应性，并且可能会产生性能瓶颈。
  - 由于同一实体对象都会在读写操作中用到，所以对于安全和权限的管理会变得比较复杂。
  - 这里面很重要的一个问题是，**系统中的读写频率比**，是偏向读，还是偏向写，就如同一般的**数据结构在查找和修改上时间复杂度**不一样，在设计系统的结构时也需要考虑这样的问题。解决方法就是我们经常用到的对数据库进行读写分离。



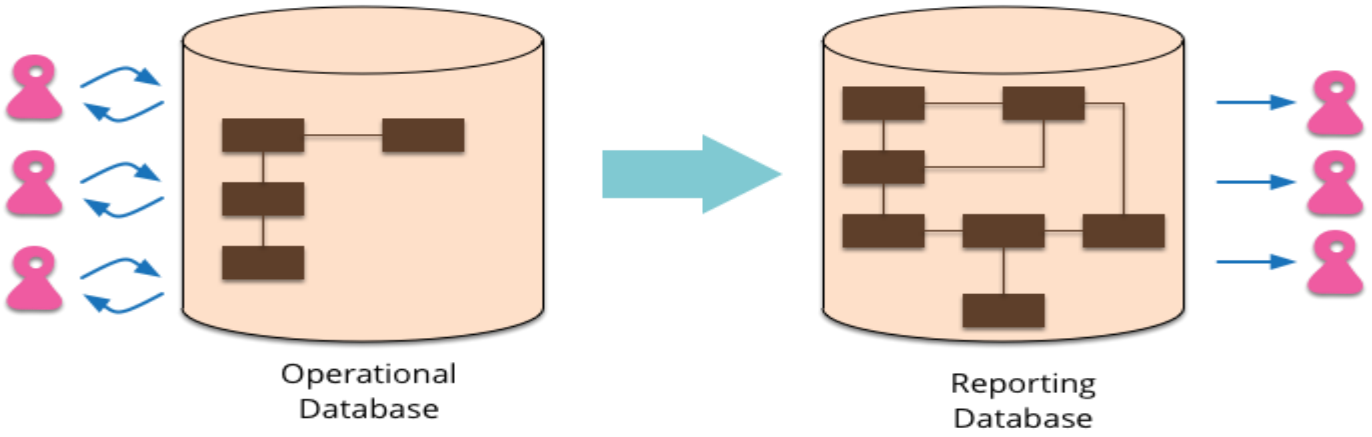
CQRS简介（一）

- CQRS由Creg Yound在CQRS, Task Based UIs, Event Sourcing agh! 这篇文章中提出。“CQRS只是简单的将之前只需要创建一个对象拆分成了两个对象，这种分离是基于方法是执行命令还是执行查询这一原则来定的”。
  - CQRS使用分离的接口将数据查询操作(Queries)和数据修改操作(Commands)分离开，这也意味着在查询和更新过程中使用的数据模型也是不一样的。这样读和写逻辑就隔离开了。
  - 主数据库处理CUD，从库处理R，从库的的结构可以和主库的结构完全一样，也可以不一样，从库主要用来进行只读的查询操作。在数量上从库的个数也可以根据查询的规模进行扩展，在业务逻辑上，也可以根据专题从主库中划分出不同的从库。



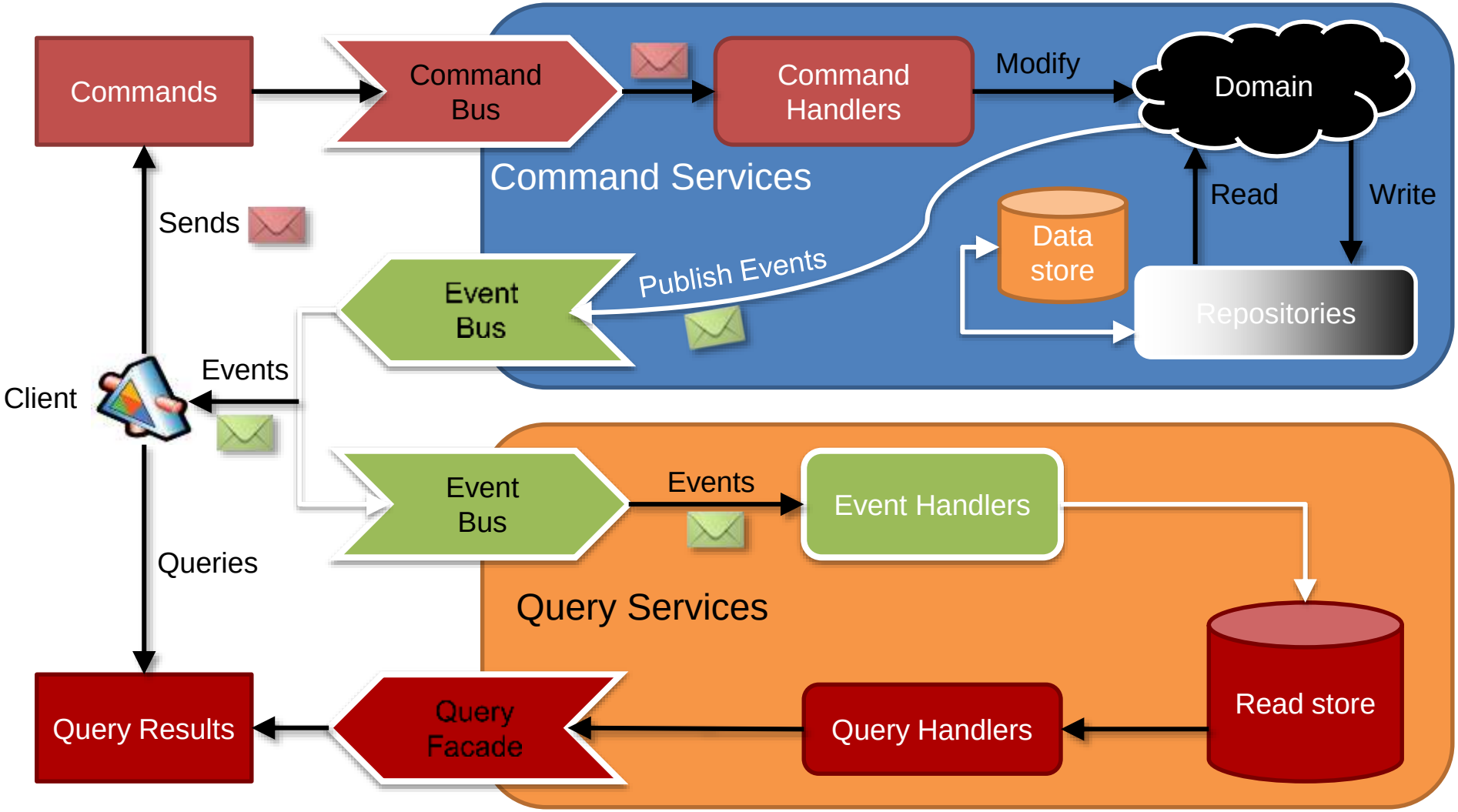
CQRS简介（二）

- 从库也可以实现成ReportingDatabase，根据查询的业务需求，从主库中抽取一些必要的数据生成一系列查询报表来存储。使用ReportingDatabase的一些优点通常可以使得查询变得更加简单高效：
  - ReportingDatabase的结构和数据表会针对常用的查询请求进行设计。
  - ReportingDatabase数据库通常会去正规化，存储一些冗余而减少必要的Join等联合查询操作，使得查询简化和高效，一些在主数据库中用不到的数据信息，在ReportingDatabase可以不用存储。
  - 可以对ReportingDatabase重构优化，而不用去改变操作数据库。
  - 对ReportingDatabase数据库的查询不会给操作数据库带来任何压力。
  - 可以针对不同的查询请求建立不同的ReportingDatabase库。



CQRS架构

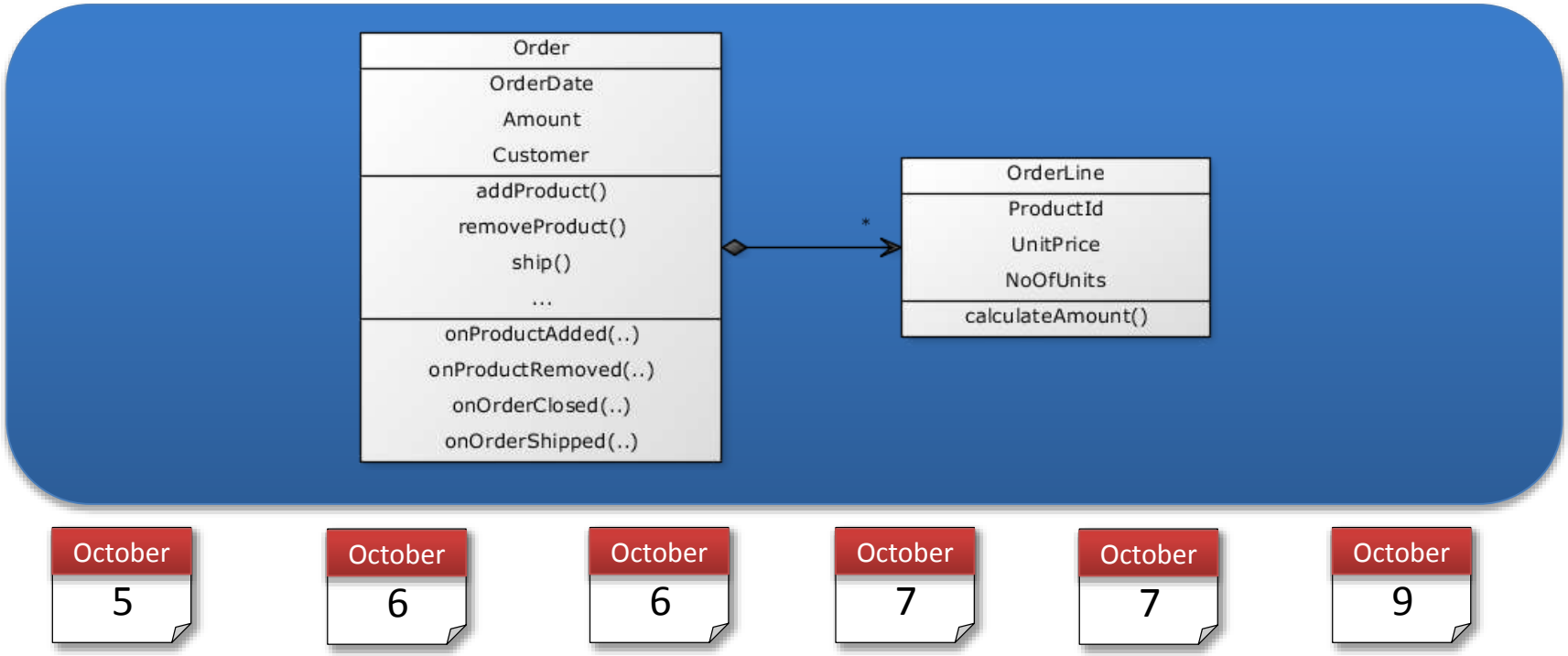
- CQRS架构示意图：



事件源

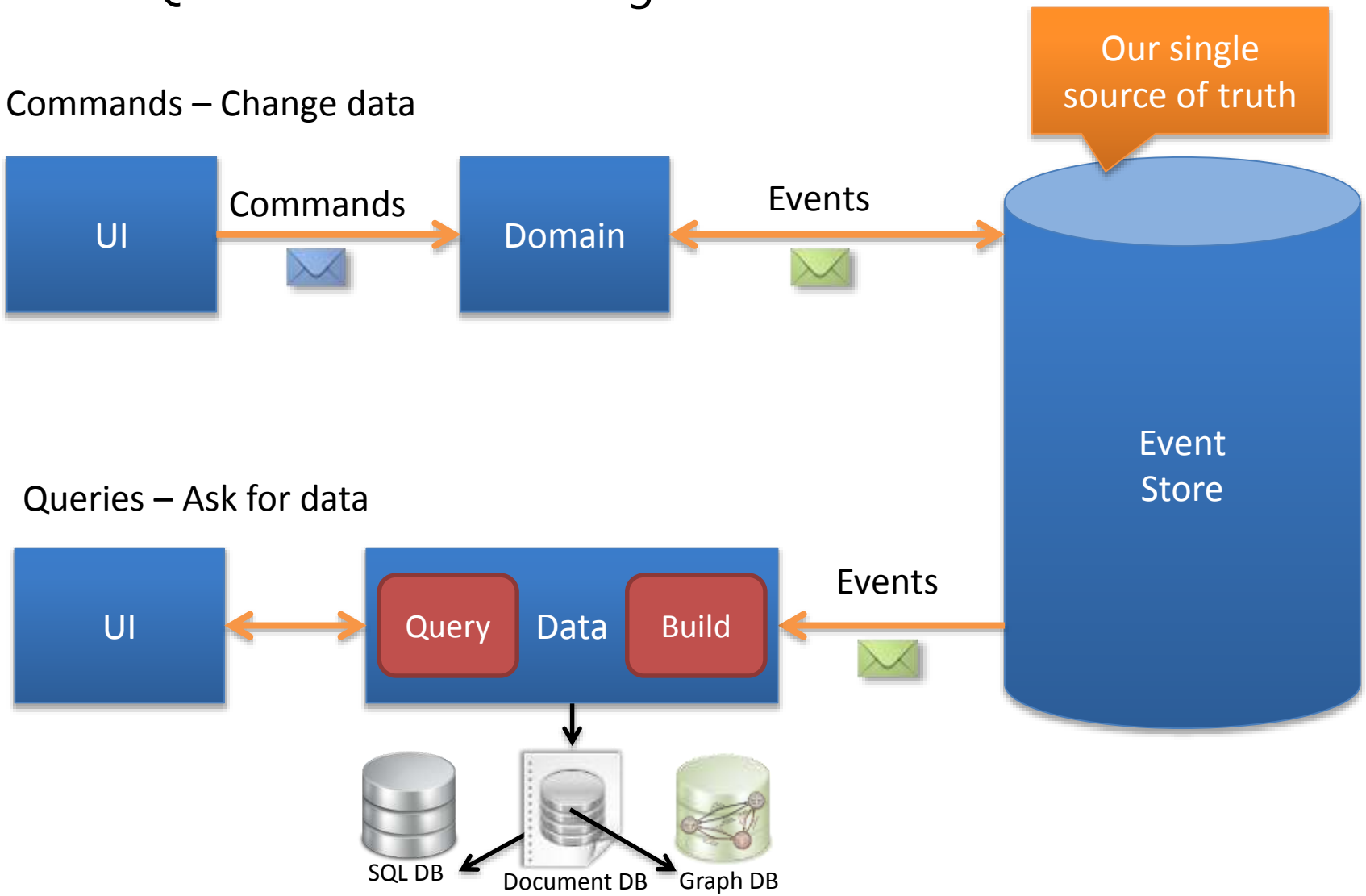
- 事件源(Event Sourcing):

Aggregates track their own Domain Events
and derive state from them



事件源实现CQRS(一)

- FULL CQRS with Event Sourcing:



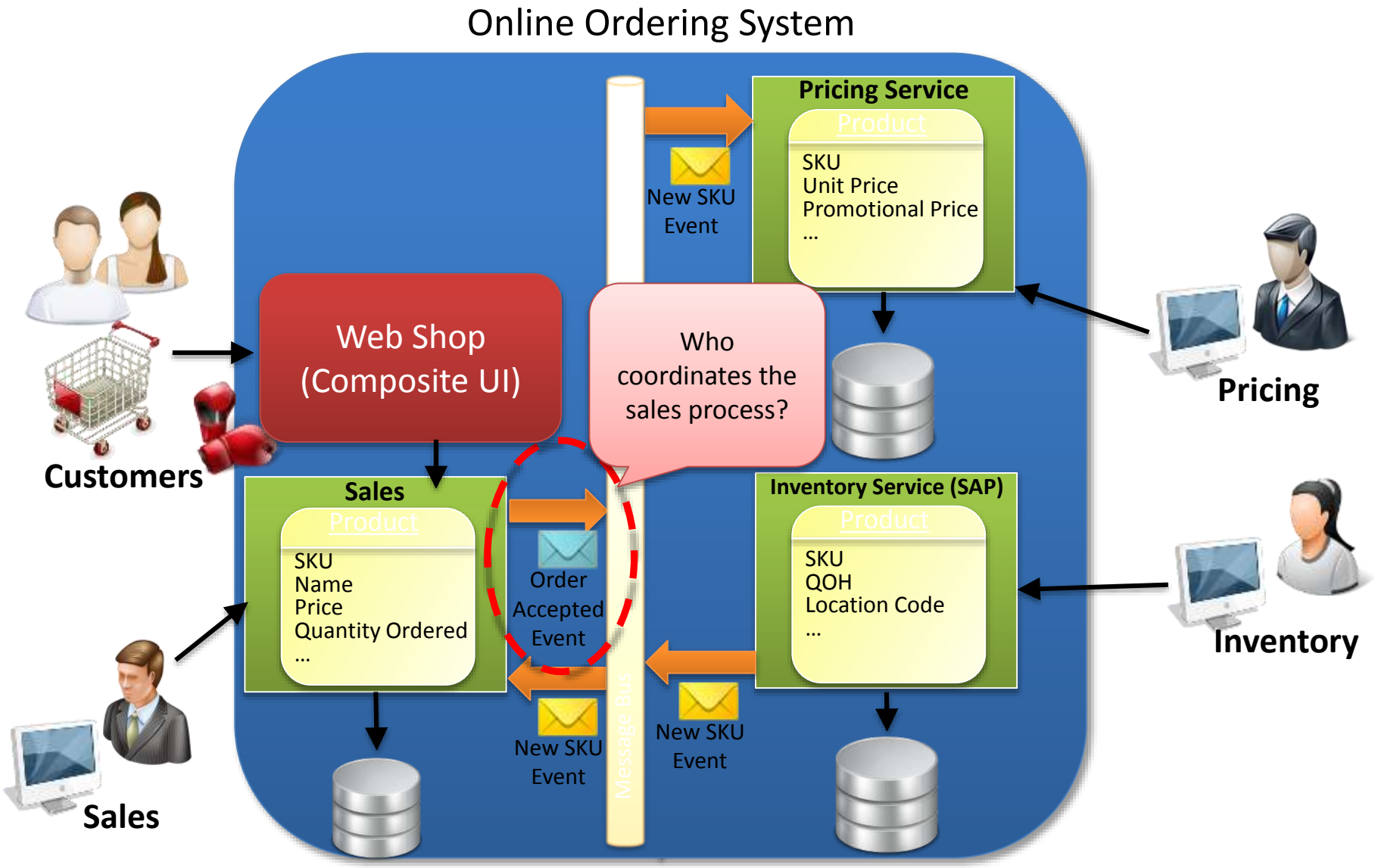
事件源实现CQRS(二)

- FULL CQRS with Event Sourcing代码风格示例:

```
public class CustomerCommandHandler {
  private Repository<Customer> customerRepository;

  public CustomerCommandHandler(Repository<Customer> customerRepository) {
    this.customerRepository = customerRepository;
  }

  @CommandHandler
  public void handle(UnsignCustomer cmd) {
    Customer customer = repository.load(cmd.getCustomerId());
    customer.unsign();
  }
}
```

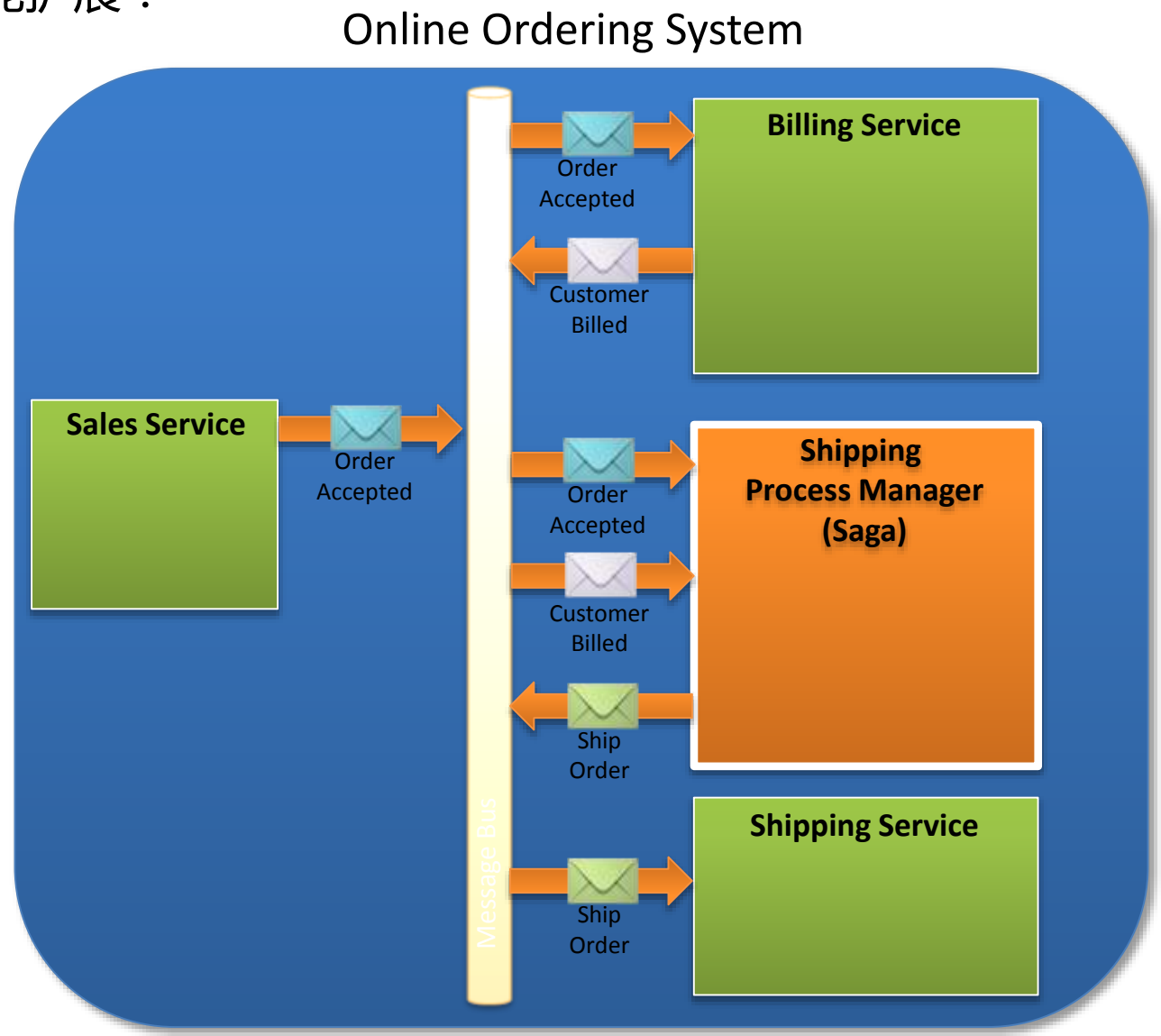
```
public class Customer {
  private boolean signedUp;

  public void unsign() {
    if (signedUp) {
      apply(new CustomerUnsignedEvent());
    }
  }

  @EventHandler
  private void handle(CustomerUnsignedEvent event) {
    signedUp = false;
  }
}
```

CQRS应用示例（一）


Online Ordering System
Pricing Service
Product
SKU
Unit Price
Promotional Price
...
New SKU Event
Web Shop (Composite UI)
Who coordinates the sales process?
Customers
Pricing
Sales
Product
SKU
Name
Price
Quantity Ordered
...
Order Accepted Event
New SKU Event
Message Bus
Inventory Service (SAP)
Product
SKU
QOH
Location Code
...
New SKU Event
Inventory
Sales

CQRS应用示例（二）

- 应用功能扩展：



CQRS模式的优点

- 分工明确，可以负责不同的部分
- 将业务上的命令和查询的职责分离能够提高系统的性能、可扩展性和安全性。并且在系统的演化中能够保持高度的灵活性，能够防止出现CRUD模式中，对查询或者修改中的某一方进行改动，导致另一方出现问题的情况。
- 逻辑清晰，能够看到系统中的那些行为或者操作导致了系统的状态变化。
- 可以从数据驱动(Data-Driven) 转到任务驱动(Task-Driven)以及事件驱动(Event-Driven)。

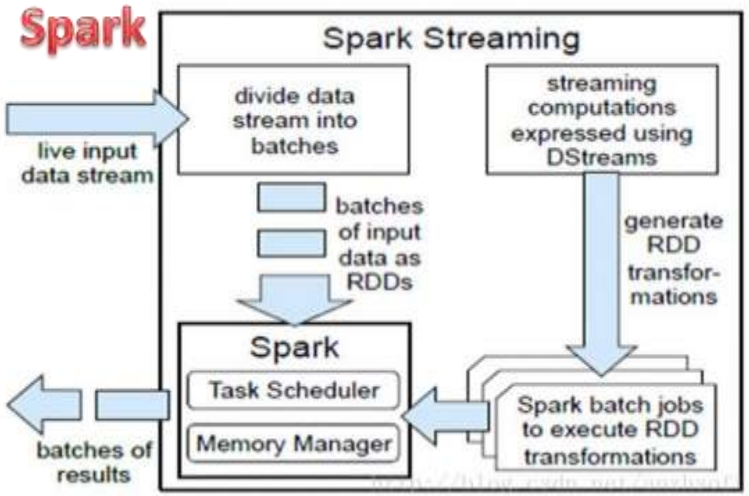
CQRS的应用场景

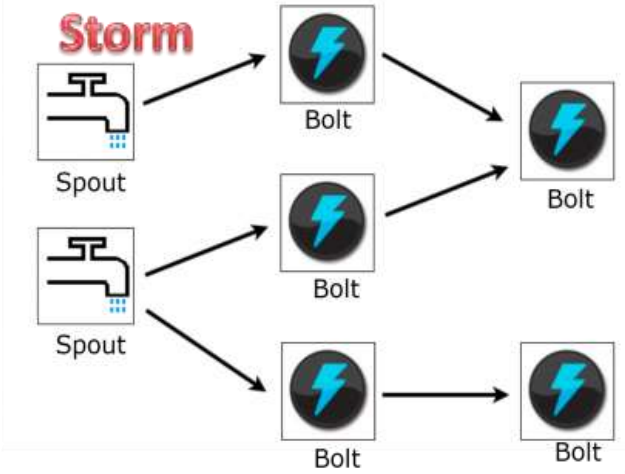
- 当在业务逻辑层有很多操作需要相同的实体或者对象进行操作的时候。CQRS使得我们可以对读和写定义不同的实体和方法，从而可以减少或者避免对某一方面的更改造成冲突
- 对于一些基于任务的用户交互系统，通常这类系统会引导用户通过一系列复杂的步骤和操作，通常会需要一些复杂的领域模型，并且整个团队已经熟悉领域驱动设计技术。写模型有很多和业务逻辑相关的命令操作的堆，输入验证，业务逻辑验证来保证数据的一致性。读模型没有业务逻辑以及验证堆，仅仅是返回DTO对象为视图模型提供数据。读模型最终和写模型相一致。
- 适用于一些需要对查询性能和写入性能分开进行优化的系统，尤其是读/写比非常高的系统，横向扩展是必须的。比如，在很多系统中读操作的请求时远大于写操作。为适应这种场景，可以考虑将写模型抽离出来单独扩展，而将写模型运行在一个或者少数几个实例上。少量的写模型实例能够减少合并冲突发生的情况
- 适用于一些团队中，一些有经验的开发者可以关注复杂的领域模型，这些用到写操作，而另一些经验较少的开发者可以关注用户界面上的读模型。
- 对于系统在将来会随着时间不段演化，有可能会包含不同版本的模型，或者业务规则经常变化的系统
- 需要和其他系统整合，特别是需要和事件溯源Event Sourcing进行整合的系统，这样子系统的临时异常不会影响整个系统的其他部分。

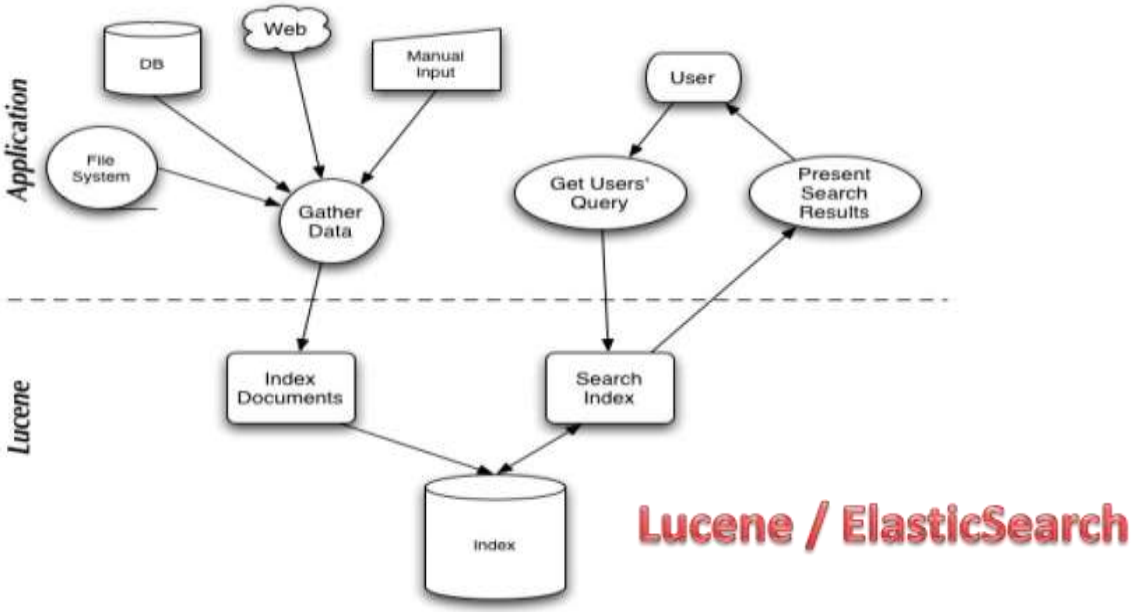
CQRS不适用的场景

- 领域模型或者业务逻辑比较简单，这种情况下使用CQRS会把系统搞复杂。
- 对于简单的，CRUD模式的用户界面以及与之相关的数据访问操作已经足够的话，没必要使用CQRS，这些都是一个简单的对数据进行增删改查。
- 不适合在整个系统中到处使用该模式。在整个数据管理场景中的特定模块中CQRS可能比较有用。但是在有些地方使用CQRS会增加系统不必要的复杂性。

CQRS与大数据技术

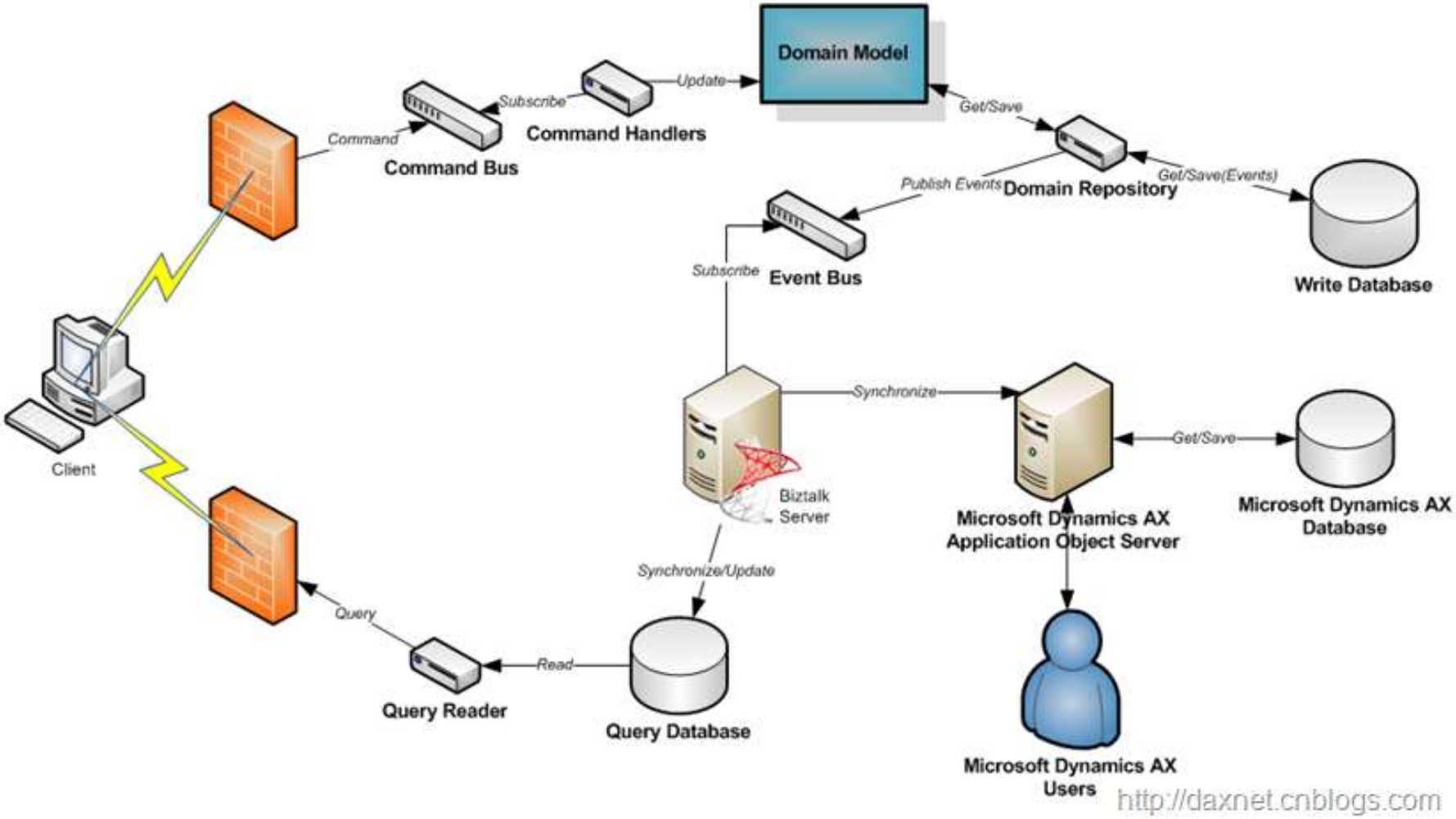

Spark
Spark Streaming
live input data stream
divide data stream into batches
streaming computations expressed using DStreams
batches of input data as RDDs
generate RDD transfor-mations
Spark
Task Scheduler
Memory Manager
Spark batch jobs to execute RDD transformations
batches of results


Storm
Spout
Spout
Bolt
Bolt
Bolt
Bolt
Bolt


Application
DB
Web
Manual Input
File System
Gather Data
User
Get Users' Query
Present Search Results
Lucene
Index Documents
Search Index
Index
Lucene / ElasticSearch

CQRS

- 示例：微软.NET框架组件提供的CQRS架构



附：CQS

- Betrand Meyer(Eiffel语言之父，开-闭原则OCP提出者)在Object Oriented Software Construction一书中提到一种命令查询分离(Command Query Separation, CQS)的概念。

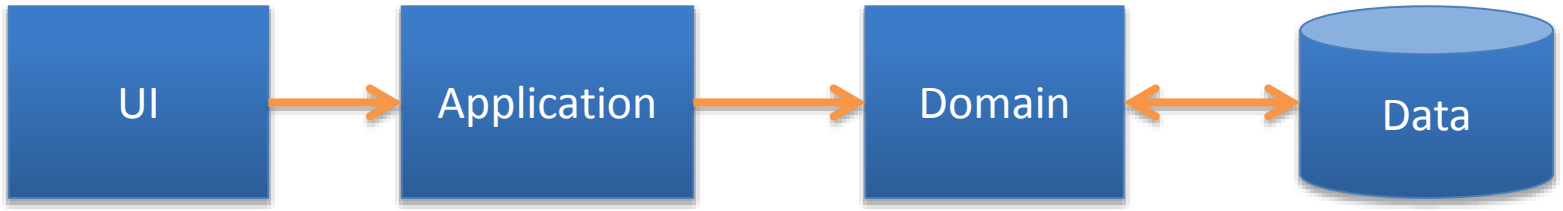
Separation of
functions that **write**
&
functions that **read**

Functions that write are called **Command** methods and must **not return a value**

Functions that read are called **Query** methods and must have **no side effects**

CQS

Commands – Change data

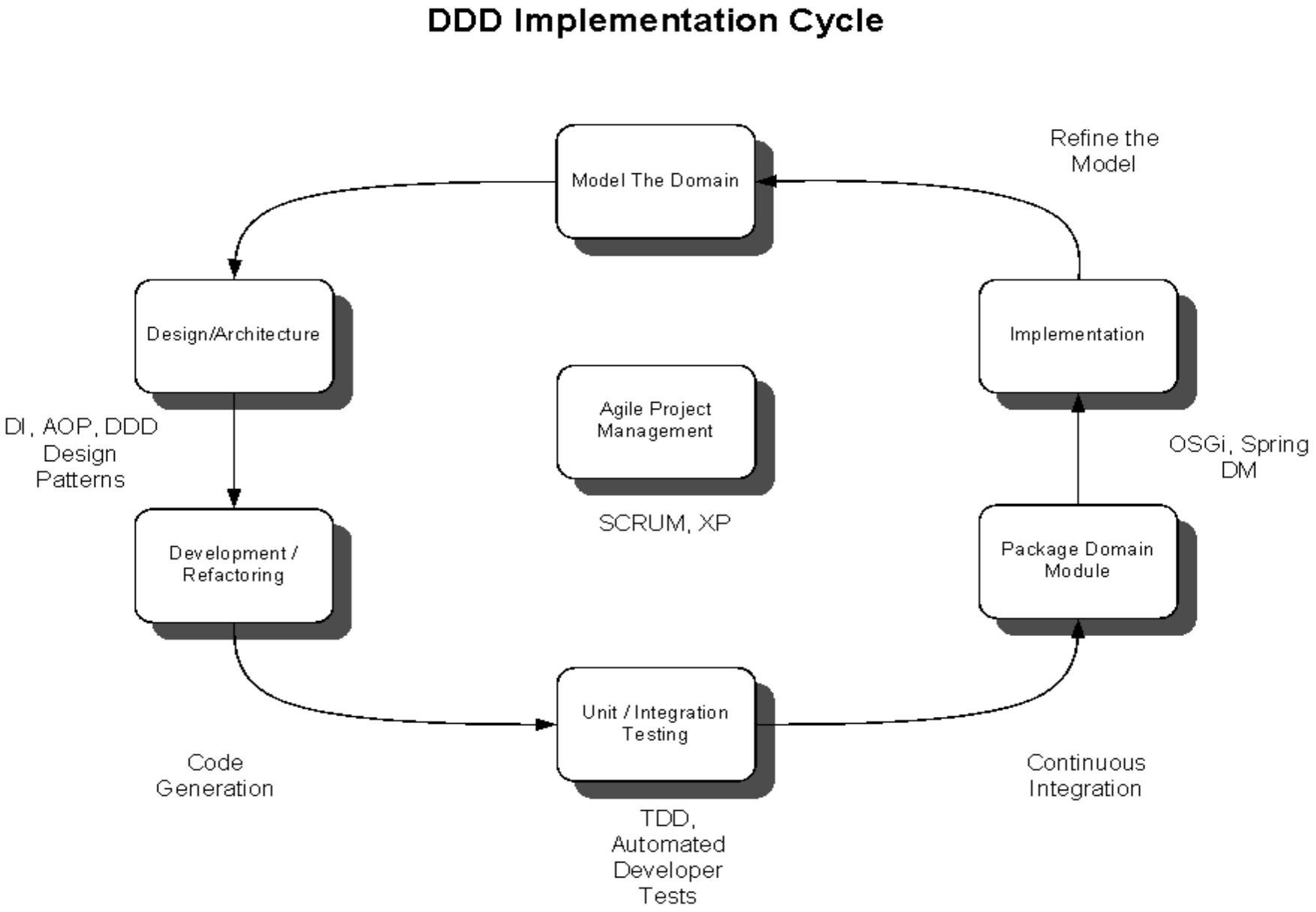


Queries – Ask for data (no side effects)

培训内容

- 领域驱动设计简介
- 领域通用语言
- 领域驱动设计的构造块
- CQRS架构
- 领域驱动设计编程实践
- 模型驱动开发

领域驱动设计研发生命周期

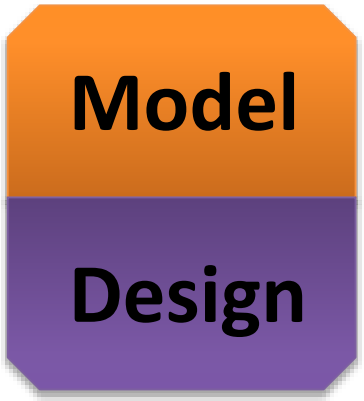


模型

A model is a representation of the domain *serving a specific purpose*.
There is no “perfect” model for a given domain.
A good model is tailored on the given purpose.

模型的表达

Model and the underlying design must evolve in sync.

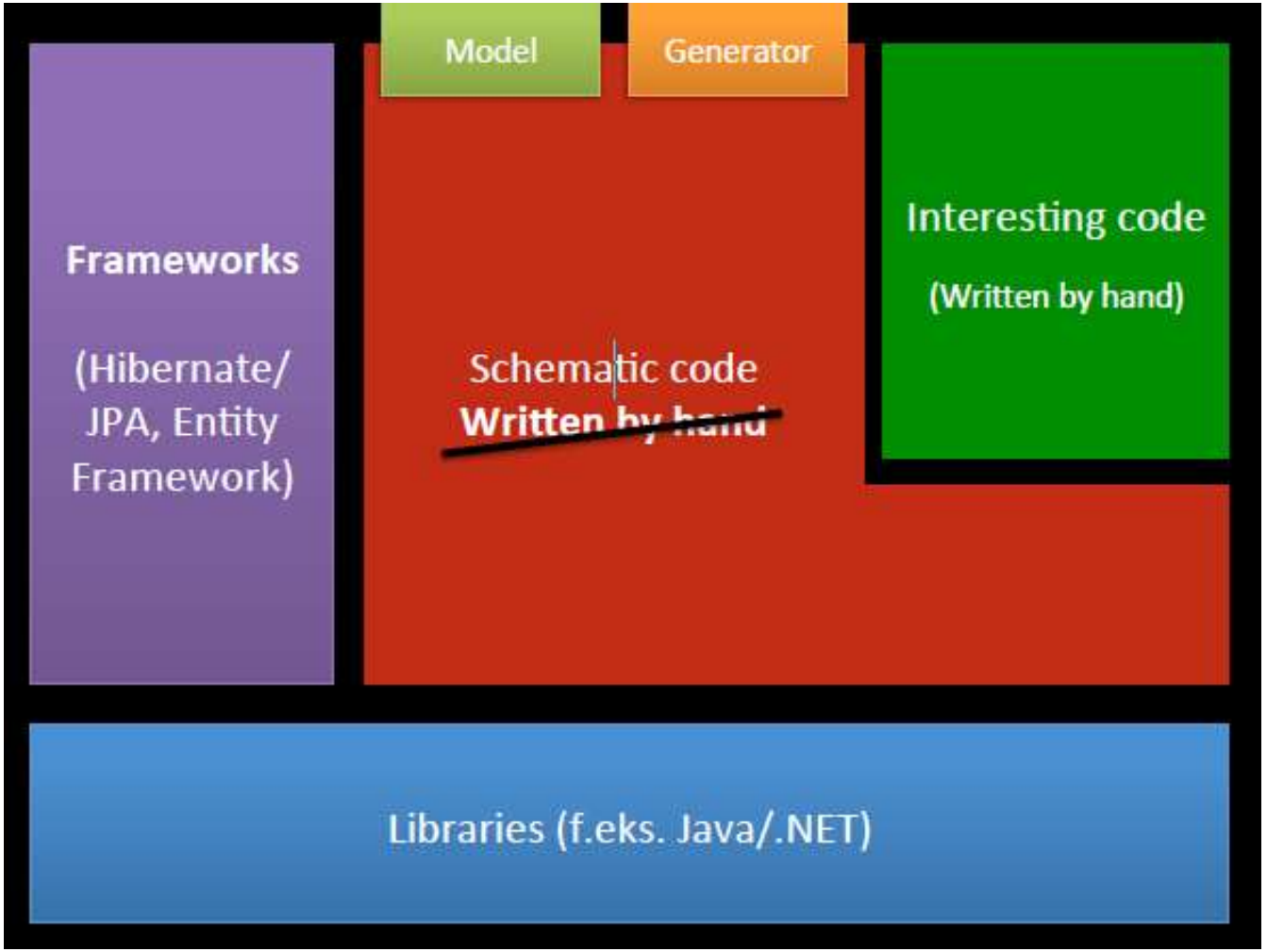


Code is the ultimate way to express the model.

Intermediate artifacts, diagrams and docs serve a temporary goal.

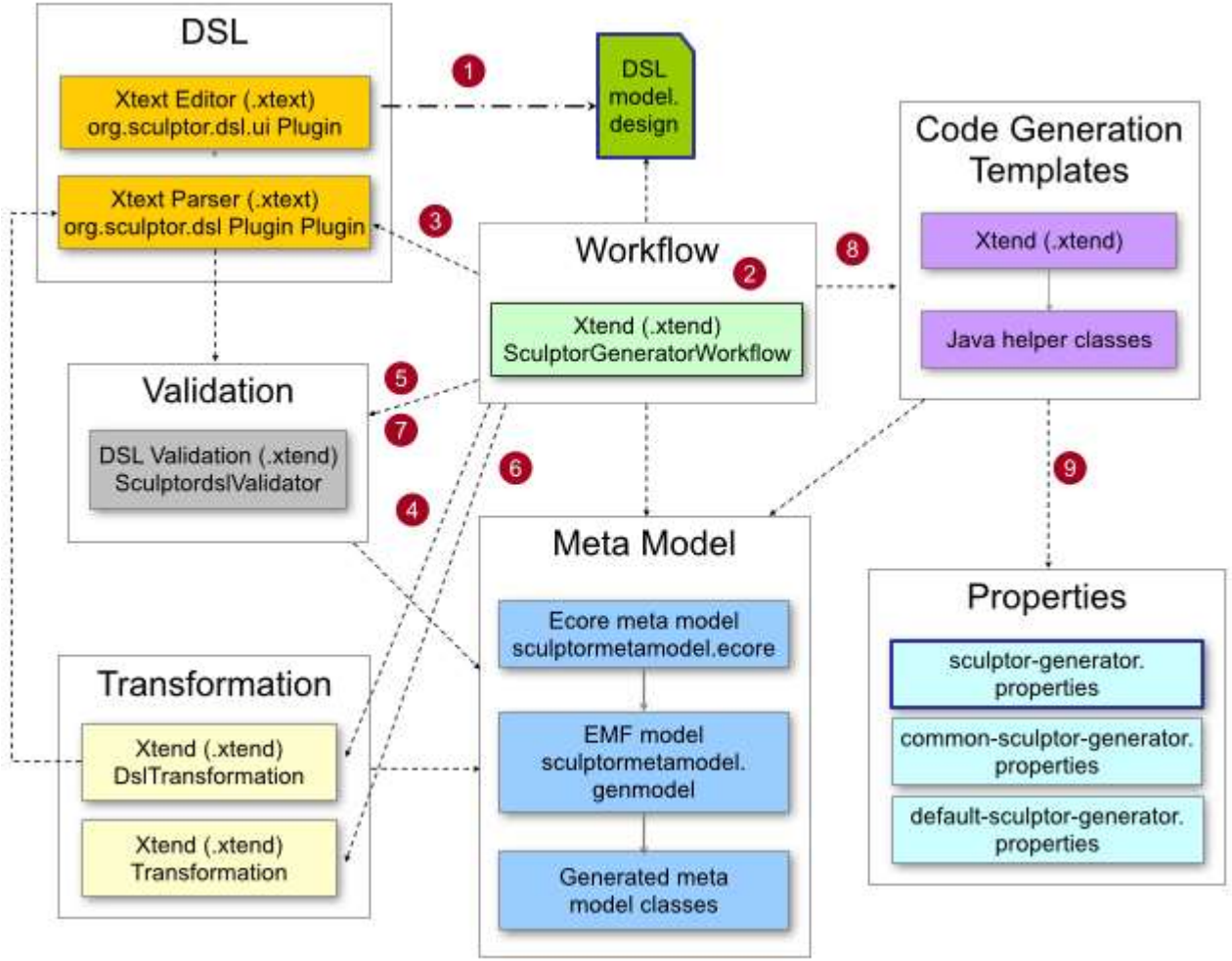
自动代码生成（一）

自动代码生成工具可以大量减少手工代码量，并且可以增强代码的规范性。



自动代码生成（二）

Java领域驱动设计自动化代码生成工具：SculptorGenerator介绍



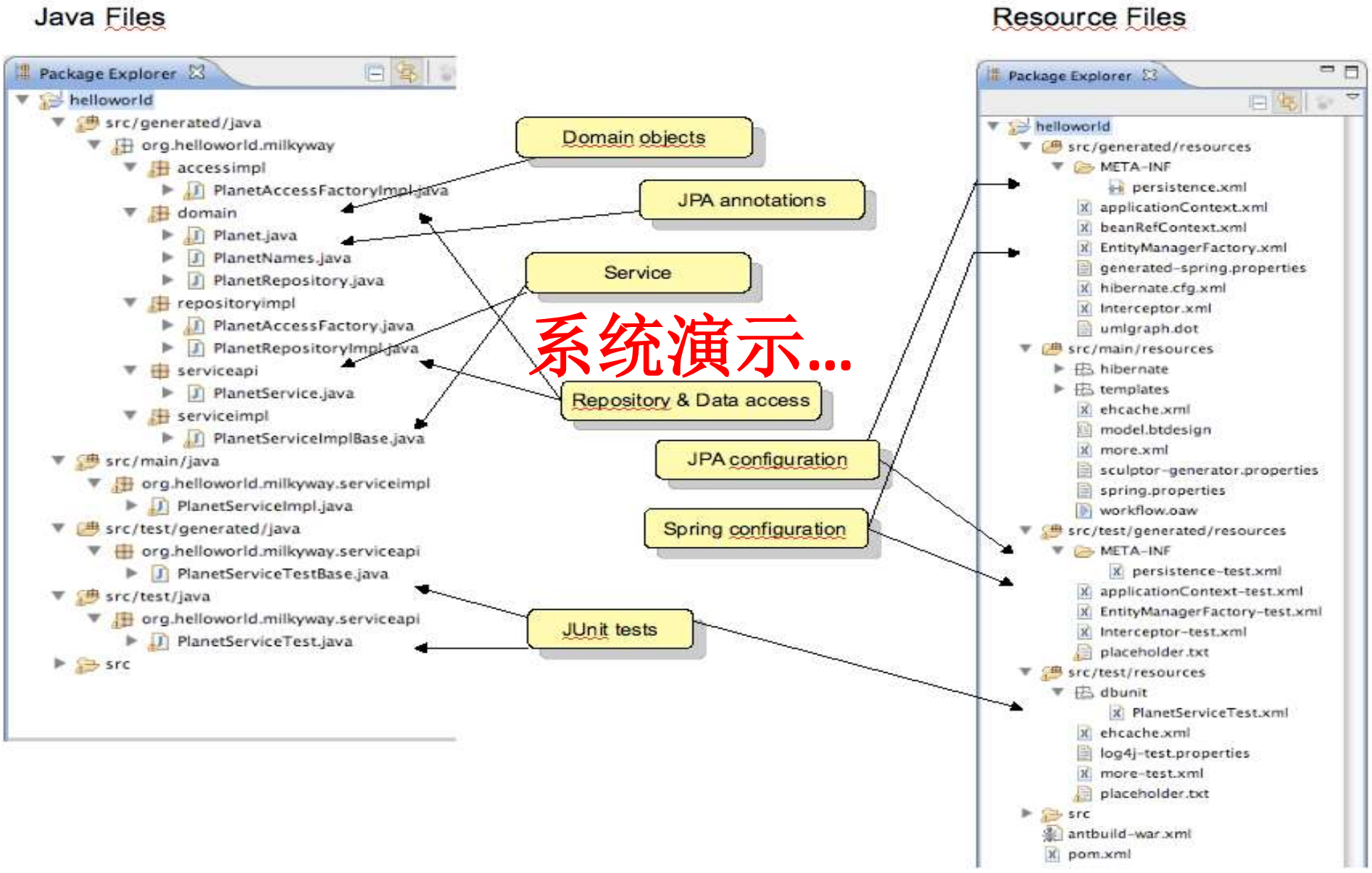
http://sculptorgenerator.org

自动代码生成（三）

DSL建模示例：

```
Application Universe {
  basePackage=org.sculptor.example.helloworld

  Module milkyway {

    Service PlanetService {
      String sayHello(String planetName) throws PlanetNotFoundException;
      protected findByKey => PlanetRepository.findByKey;
      @Planet getPlanet(String planetName) throws PlanetNotFoundException;
    }

    Entity Planet {
      gap
      scaffold
      String name key;
      String message;
      Integer diameter nullable min="1";
      Integer population nullable min="0";
      - Set<@Moon> moons opposite planet;

      Repository PlanetRepository {
        findByKeys;
        findByKey;
        save;
        findAll(PagingParameter pagingParameter);
        findAll;
      }
    }

    Entity Moon {
      not aggregateRoot // belongs to Planet Aggregate
      String name key;
      Integer diameter nullable;
      - @Planet planet opposite moons;
    }
  }
}
```

自动代码生成（四）


Java Files
Resource Files
Package Explorer
helloworld
src/generated/java
org.helloworld.milkyway
accessimpl
PlanetAccessFactoryImpl.java
domain
Planet.java
PlanetNames.java
PlanetRepository.java
repositoryimpl
PlanetAccessFactory.java
PlanetRepositoryImpl.java
serviceapi
PlanetService.java
serviceimpl
PlanetServiceImplBase.java
src/main/java
org.helloworld.milkyway.serviceimpl
PlanetServiceImpl.java
src/test/generated/java
org.helloworld.milkyway.serviceapi
PlanetServiceTestBase.java
src/test/java
org.helloworld.milkyway.serviceapi
PlanetServiceTest.java
src
Domain objects
JPA annotations
Service
系统演示...
Repository & Data access
JPA configuration
Spring configuration
JUnit tests
Package Explorer
helloworld
src/generated/resources
META-INF
persistence.xml
applicationContext.xml
beanRefContext.xml
EntityManagerFactory.xml
generated-spring.properties
hibernate.cfg.xml
Interceptor.xml
umlgraph.dot
src/main/resources
hibernate
templates
ehcache.xml
model.btdesign
more.xml
sculptor-generator.properties
spring.properties
workflow.oaw
src/test/generated/resources
META-INF
persistence-test.xml
applicationContext-test.xml
EntityManagerFactory-test.xml
Interceptor-test.xml
placeholder.txt
src/test/resources
dbunit
PlanetServiceTest.xml
ehcache.xml
log4j-test.properties
more-test.xml
placeholder.txt
src
antbuild-war.xml
pom.xml

THANKS