ARAF KARSH HAMID
Co-Founder / CTO
MetaMagic Global Inc., NJ, USA
@arafkarsh
arafkarsh

Domain Driven Design

Agenda

1

Hexagonal Architecture

- Hexagonal Architecture
- Shopping Portal Example
- RESTful Guidelines

2

Domain Driven Design

- Ubiquitous Language
- Bounded Context
- Context Maps
- Aggregate Roots
- Entities and Value Objects
- Factories
- Repositories
- Domain Events

3

Case Study

- Shopping Portal App

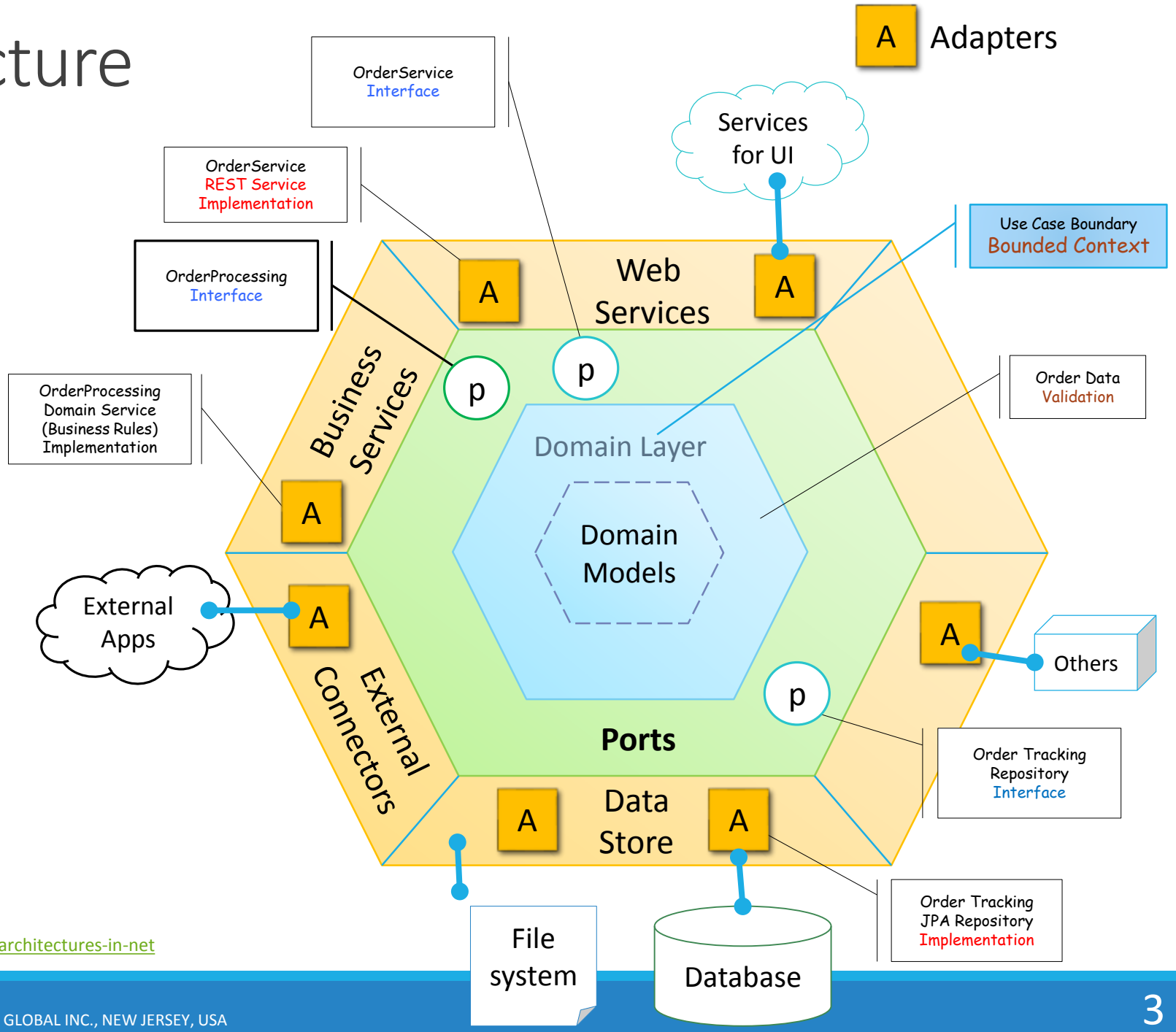
Hexagonal Architecture

Ports & Adapters

The layer between the **Adapter** and the **Domain** is identified as the **Ports** layer. The Domain is inside the port, adapters for external entities are on the outside of the port.

The notion of a "port" invokes the OS idea that any device that adheres to a known protocol can be plugged into a port. Similarly many adapters may use the Ports.

- Reduces Technical Debt
- Dependency Injection
- Auto Wiring



Source : http://alistair.cockburn.us/Hexagonal+architecture
https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net

Shopping Portal Modules – Code Packaging

Auth | Customer | Products | Cart | Order

Packaging Structure

Domain Layer ← Bounded Context → Domain Layer

Domain Layer

- Models
- Repo
- Services
- Factories

Domain Layer

- Models
- Repo
- Services
- Factories

Domain Models
(Entities, Value Objects, DTOs)
Interfaces (Ports)
(Repositories, Business Services, Web Services)
Entity Factories

Domain Layer

- Models
- Repo
- Services
- Factories

Adapters

- Repo
- Services
- Web Services

Adapters

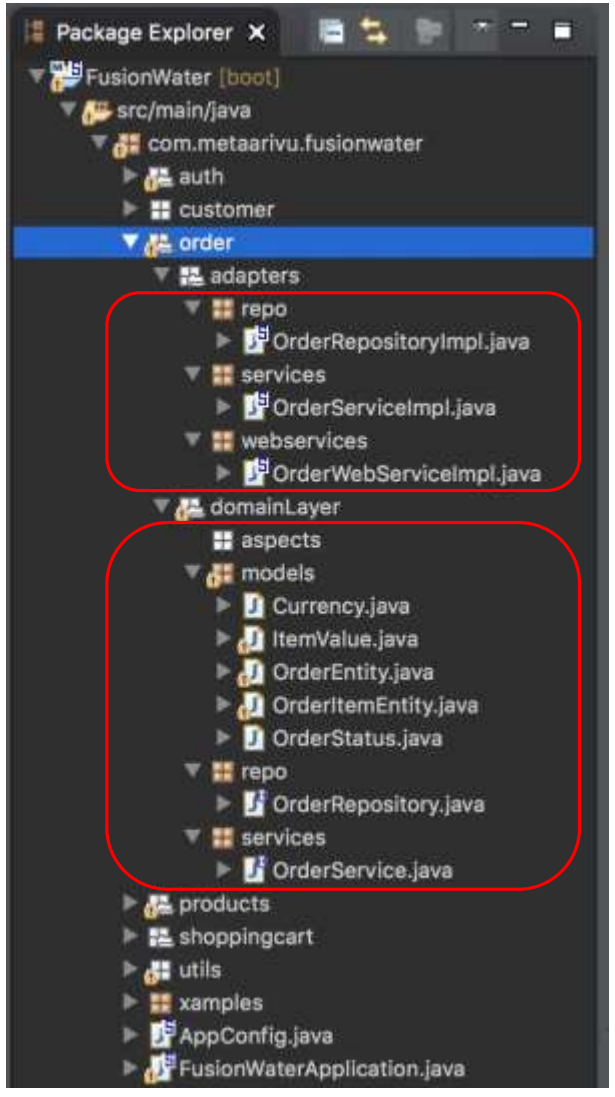
- Repo
- Services
- Web Services

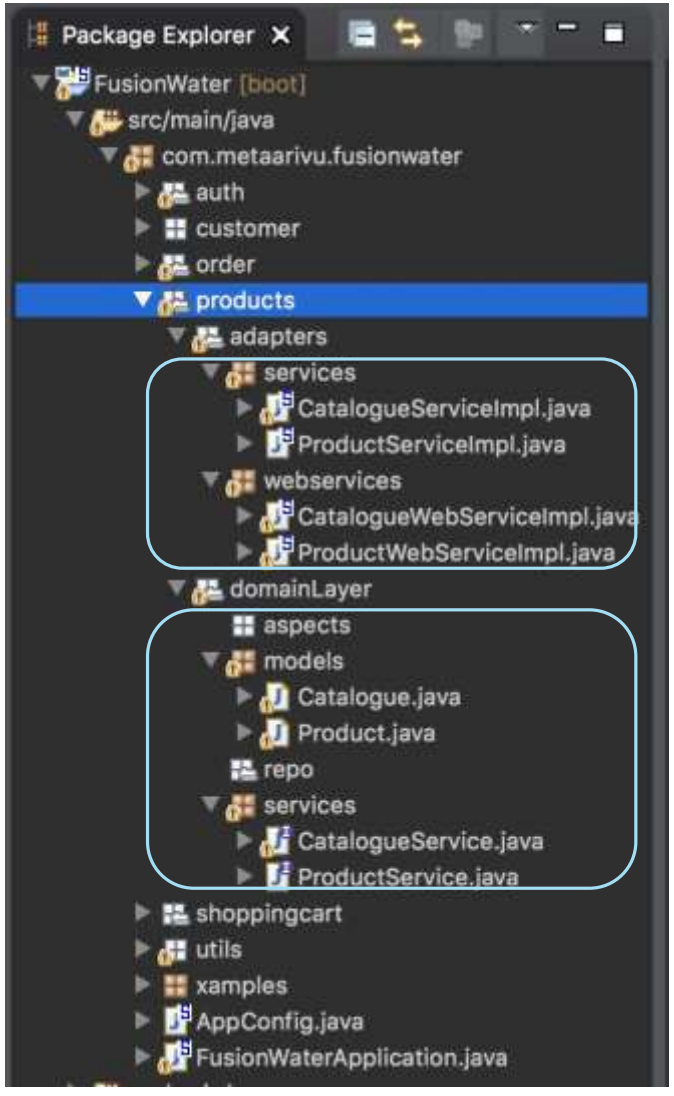
Implementation
(Repositories, Business Services, Web Services)

Adapters

- Repo
- Services
- Web Services

Shopping Portal Design based on Hexagonal Architecture





RESTful Guidelines

1. **Endpoints as nouns, NOT verbs**

   Ex. /catalogues
   /orders
   /catalogues/products

   and NOT

   /getProducts/
   /updateProducts/

2. **Use plurals**

   Ex. /catalogues/{catalogueId}

   and NOT

   /catalogue/{catalogueId}

3. Documenting
4. Paging
5. Use SSL

6. **HTTP Methods**

   GET / POST / PUT / DELETE / OPTIONS / HEAD

7. HTTP Status Codes (Effective usage)

8. **Versioning**

   Use Media Type Version
   GET /account/5555 HTTP/1.1
   Accept: application/**vnd.catalogues.v1+json**

   Instead of URL path version
   https://domain/v1/catalogues/products

RESTful Guidelines – Query Examples

```java
113 @RestController
114 @RequestMapping("/catalogues/")
115 @Scope("request")
116 public class ProductWebServiceImpl {
117
118     @Autowired
119     private ProductService productService;
120
121     /**
122      * Get All Products
123      * @return
124      */
125     @GetMapping("/products")
126     public List<Product> findAll() {
127         return productService.findAll();
128     }
```

Search All Products

```java
130     /**
131      * Get All Products By Catalogue Id
132      * @param catalogueId
133      * @return
134      */
135     @GetMapping("/{catalogueId}/products")
136     public List<Product> findProductsBy(@PathVariable Integer catalogueId) {
137         return productService.findProductsBy(catalogueId);
138     }
```

Search Products By Catalogue ID

```java
140     /**
141      * Get All Products By Catalogue Id and Product Id
142      * @param catalogueId
143      * @param productId
144      * @return
145      */
146     @GetMapping("/{catalogueId}/products/{productId}")
147     public List<Product> findProductsBy(
148             @PathVariable Integer catalogueId,
149             @PathVariable Integer productId) {
150         return productService.findProductsBy(catalogueId, productId);
151     }
```

Search Products By Catalogue ID & Product ID

RESTful Guidelines – Query Examples

```java
153  /**
154   * Return Products By Catalogue ID within a Price Range
155   *                                Catalogue-ID      Price Range
156   * URL Ex. http://localhost:9000/catalogue/2/products/680/800
157   *
158   * @param catalogueId
159   * @param priceStart
160   * @param priceEnd
161   * @return
162   */
163  @GetMapping("/{catalogueId}/products/{priceStart}/{priceEnd}")
164  public List<Product> findProductsBy(
165          @PathVariable Integer catalogueId,
166          @PathVariable Double priceStart,
167          @PathVariable Double priceEnd) {
168      return productService.findProductsBy(catalogueId, priceStart, priceEnd);
169  }
```

Two different implementation of same query

```java
171  /**
172   * Return Products By Catalogue ID within a Price Range.
173   *                                catalogue-ID      Price Range
174   * URL Ex. http://localhost:9000/catalogue/products/2?priceStart=680&priceEnd=800
175   *
176   * @param catalogueId
177   * @param priceStart
178   * @param priceEnd
179   * @return
180   */
181  @GetMapping("/products/{catalogueId}")
182  public List<Product> findProductsByVars(
183          @PathVariable Integer catalogueId,
184          @RequestParam(value="priceStart", required=false) Double priceStart,
185          @RequestParam(value="priceEnd", required=false) Double priceEnd) {
186      return productService.findProductsBy(catalogueId, priceStart, priceEnd);
187  }
```

RESTful Guidelines – Get & Create Example

```
121     /**
122      * Get All Products
123      * @return
124      */
125     @GetMapping("/products")
126     public List<Product> findAll() {
127         return productService.findAll();
128     }
```

URL Remains the same.
HTTP Methods Get / Post
Defines the action

```
199     /**
200      * Add Product
201      * @param _product
202      */
203     @PostMapping(value="/products", produces = "application/json")
204     public @ResponseBody ServiceStatusBean   addProduct(@RequestBody Product _product) {
205         boolean status = productService.addProduct(_product);
206         if(status) {
207             return new ServiceStatusBean(status,"Product update Success!",HTTPStatus.HTTP_SUCCESS_200, "");
208         }
209         return new ServiceStatusBean(status,"Product updte failed",HTTPStatus.HTTP_CLIENT_400, "");
210     }
```

RESTful Guidelines – Update & Delete Example

```java
/**
 * Update the Product
 * @param _product
 * @param _productId
 */
@PutMapping(value="/products/{_productId}", produces = "application/json")
public @ResponseBody ServiceStatusBean  updateProduct(@RequestBody Product _product, @PathVariable Integer _productId) {
    boolean status = productService.updateProduct(_product, _productId);
    if(status) {
        return new ServiceStatusBean(status,"Product update Success!",HTTPStatus.HTTP_SUCCESS_200, "");
    }
    return new ServiceStatusBean(status,"Product updte failed",HTTPStatus.HTTP_CLIENT_400, "");
}
```

```java
/**
 * Delete the Product
 * @param _productId
 */
@DeleteMapping(value = "/products/{_productId}", produces = "application/json")
public @ResponseBody ServiceStatusBean  deleteProduct(@PathVariable Integer _productId) {
    boolean status = productService.deleteProduct(_productId);
    if(status) {
        return new ServiceStatusBean(status,"Product update Success!", HTTPStatus.HTTP_SUCCESS_200, "");
    }
    return new ServiceStatusBean(status,"Product updte failed", HTTPStatus.HTTP_CLIENT_400, "");
}
```

Domain Driven Design

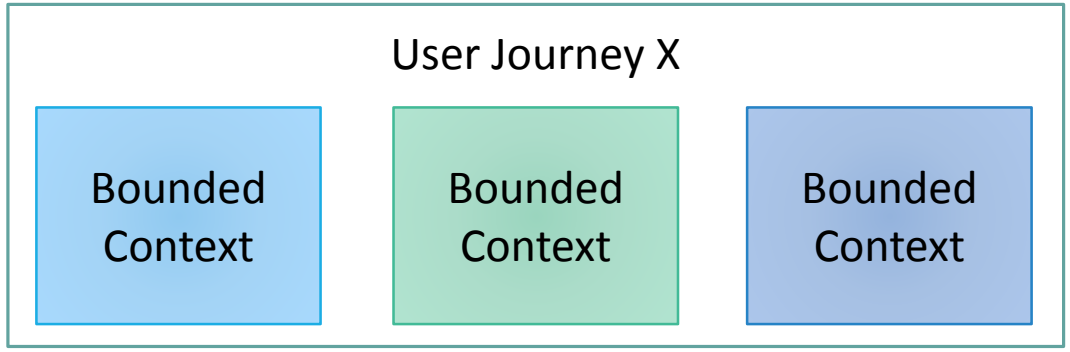
- Strategic Design
- Tactical Design

Bounded Context – Strategic Design

- Bounded Context is a Specific **Business Process / Concern**.
- Components / Modules inside the Bounded Context are context specific.
- Multiple Bounded Contexts are linked using **Context Mapping**.
- **One Team** assigned to a Bounded Context.
- Each Bounded Context will have it's own **Source Code Repository.**
- When the Bounded Context is being developed as a key strategic initiative of your organization, it's called the **Core Domain**.
- Within a Bounded Context the team must have same language called **Ubiquitous language** for Spoken and for Design / Code Implementation.

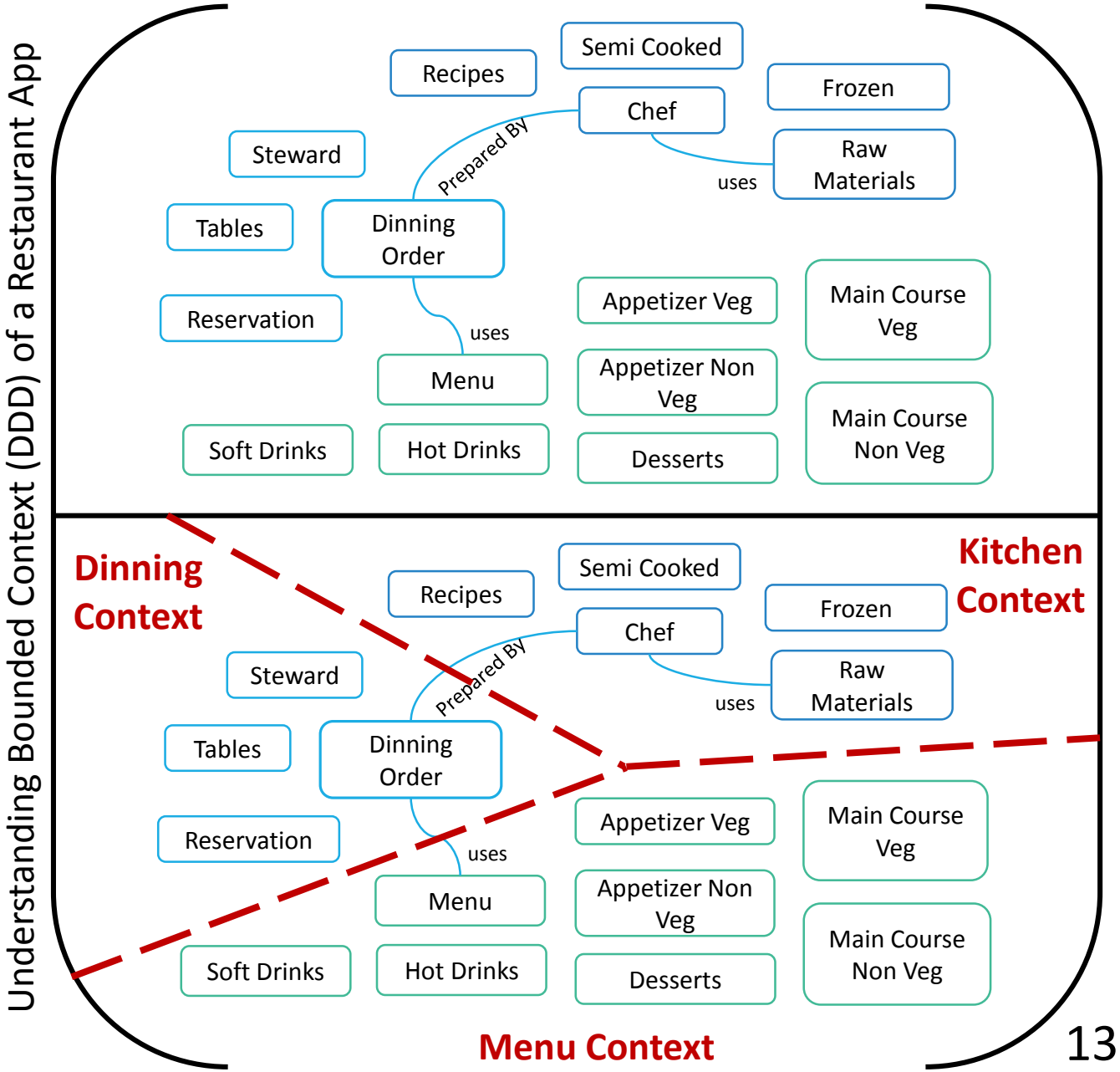
DDD: Bounded Context – Strategic Design

An App User's Journey can run across multiple Bounded Context / Micro Services.



Areas of the domain treated independently

Discovered as you assess requirements and build language



Source: Domain-Driven Design Reference by Eric Evans

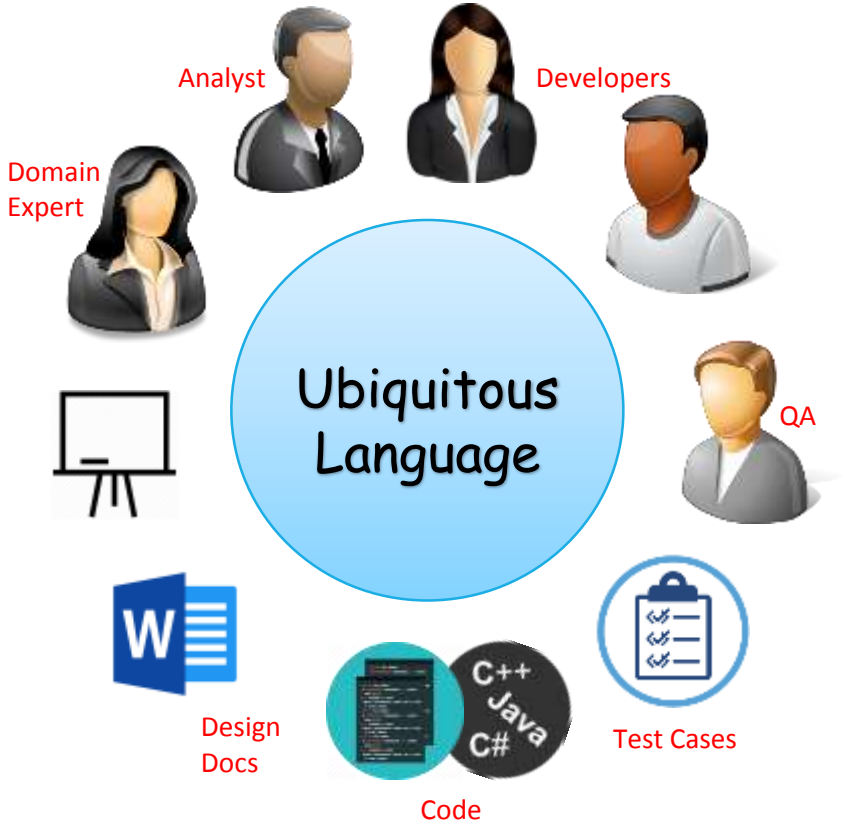
DDD: Ubiquitous Language: Strategic Design

Ubiquitous Language → Vocabulary shared by all involved parties → Used in all forms of spoken / written communication

Restaurant Context – Food Item :

Eg. Food Item (Navrathnakurma) can have different meaning or properties depends on the context.

- In the Menu Context it's a Veg Dish.
- In the Kitchen Context it's is recipe.
- And in the Dining Context it will have more info related to user feed back etc.



Role-Feature-Reason Matrix

As an Restaurant Owner
I want to know who my Customers are
So that I can serve them better

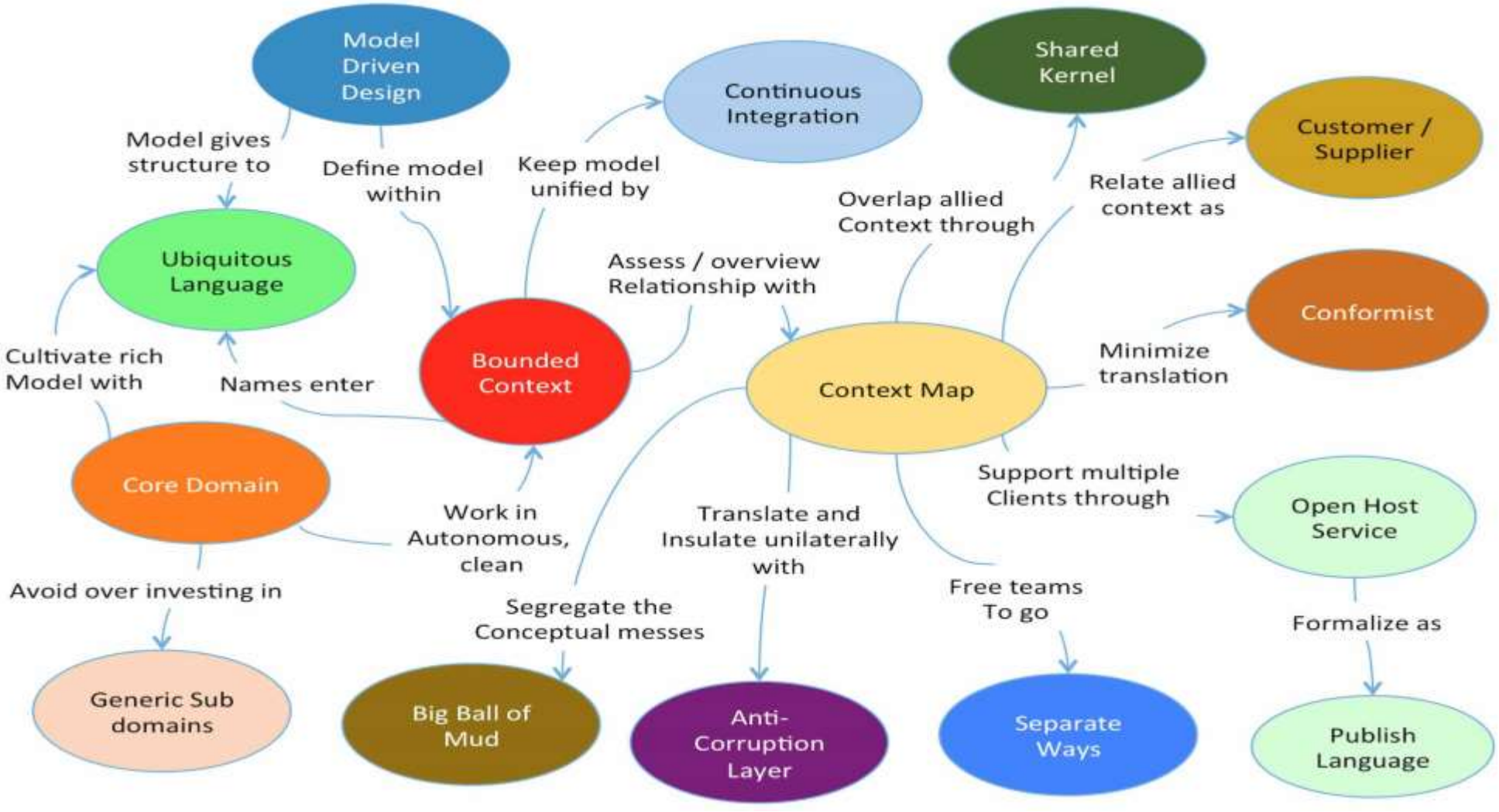
BDD Construct

Given	Customer John Doe exists
When	Customer orders food
Then	Assign customer preferences as Veg or Non Veg customer

BDD – Behavior Driven Development

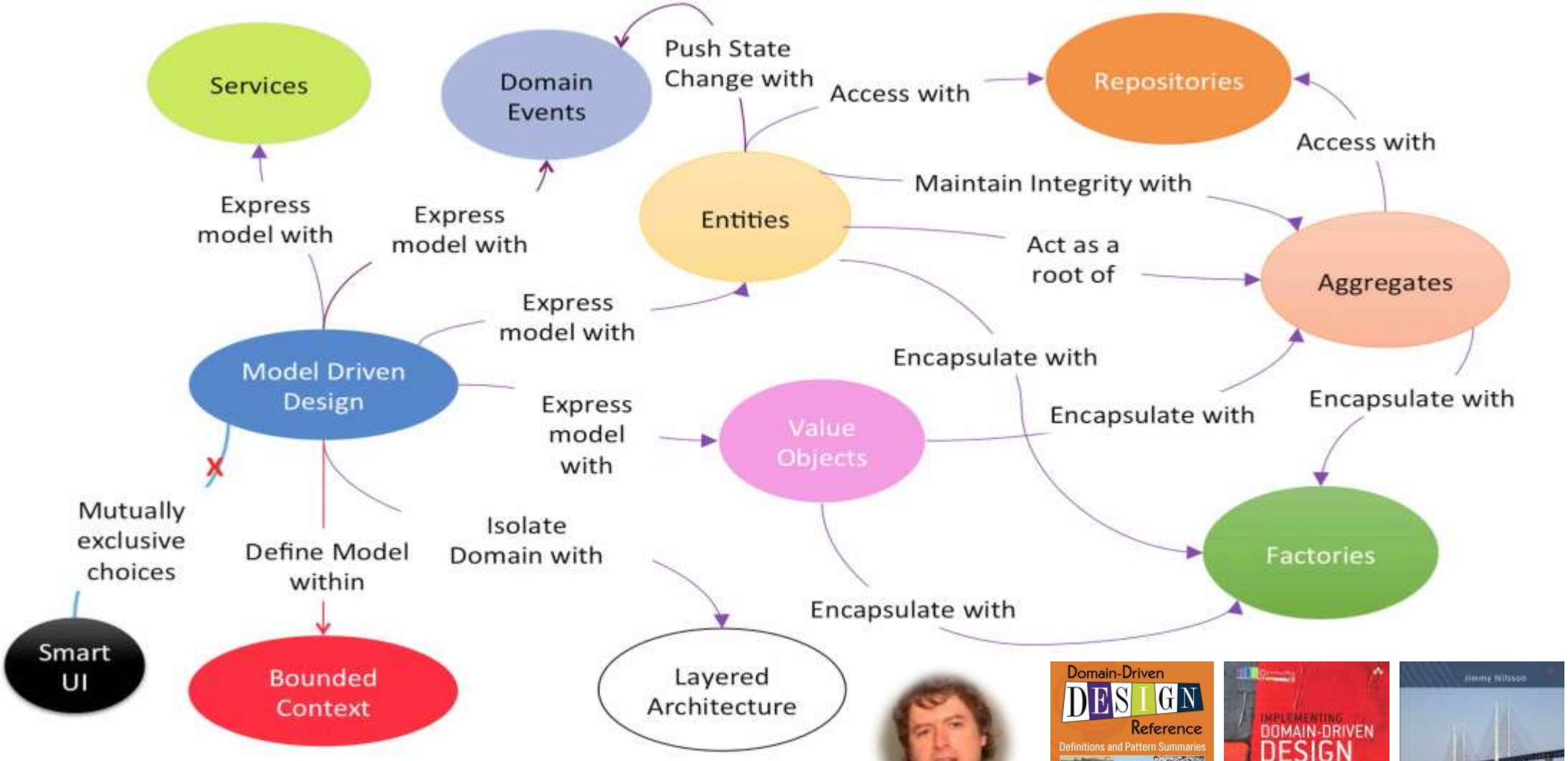
DDD : Context Map – Strategic Design

Context Map defines the relationship of Bounded Contexts



Source: Domain-Driven Design Reference by Eric Evans

Domain Driven Design – Tactical Design



Source: Domain-Driven Design Reference by Eric Evans

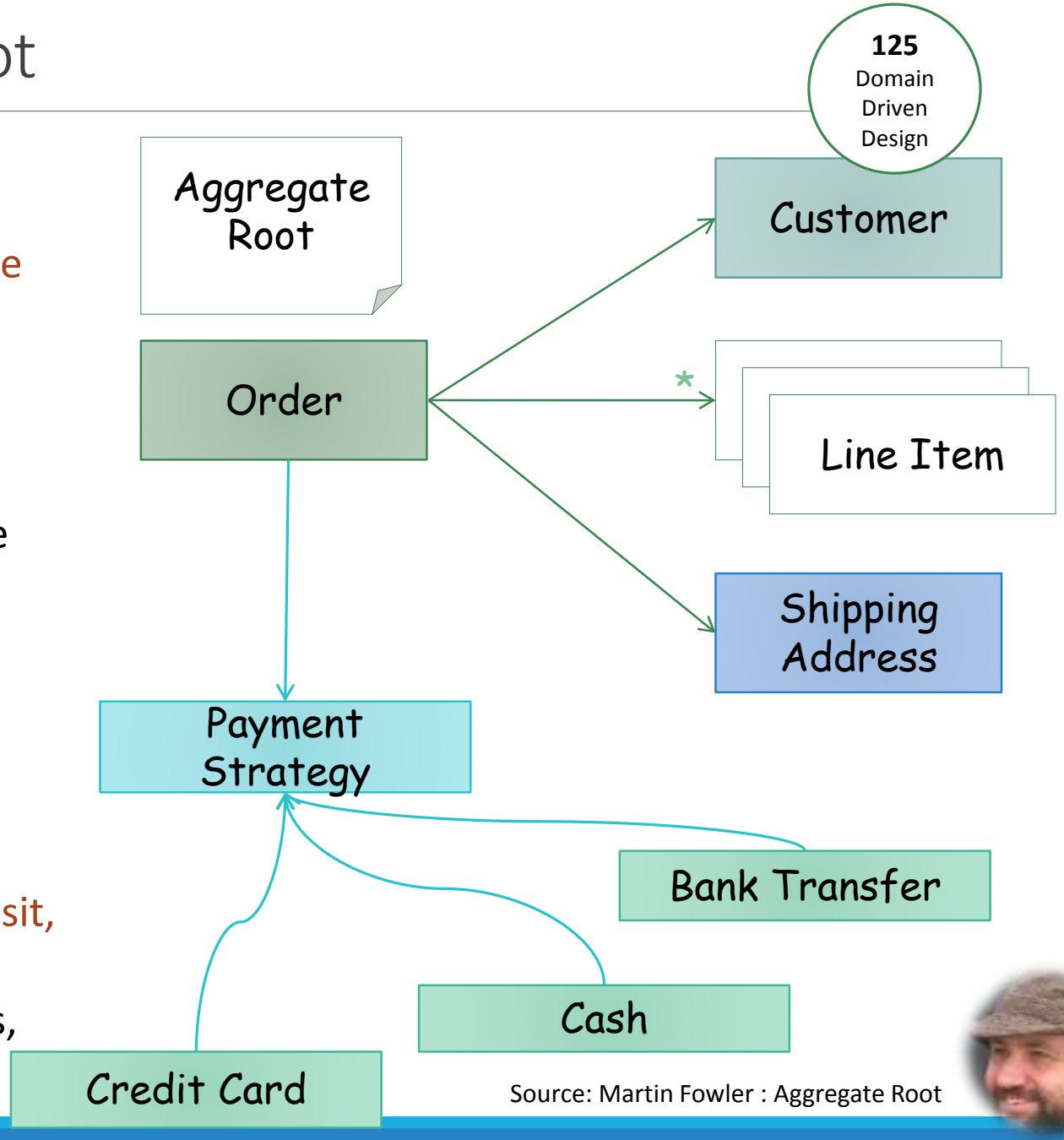
Understanding Aggregate Root

- An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.

- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Transactions should not cross aggregate boundaries.

- Aggregates are sometimes confused with collection classes (lists, maps, etc.).

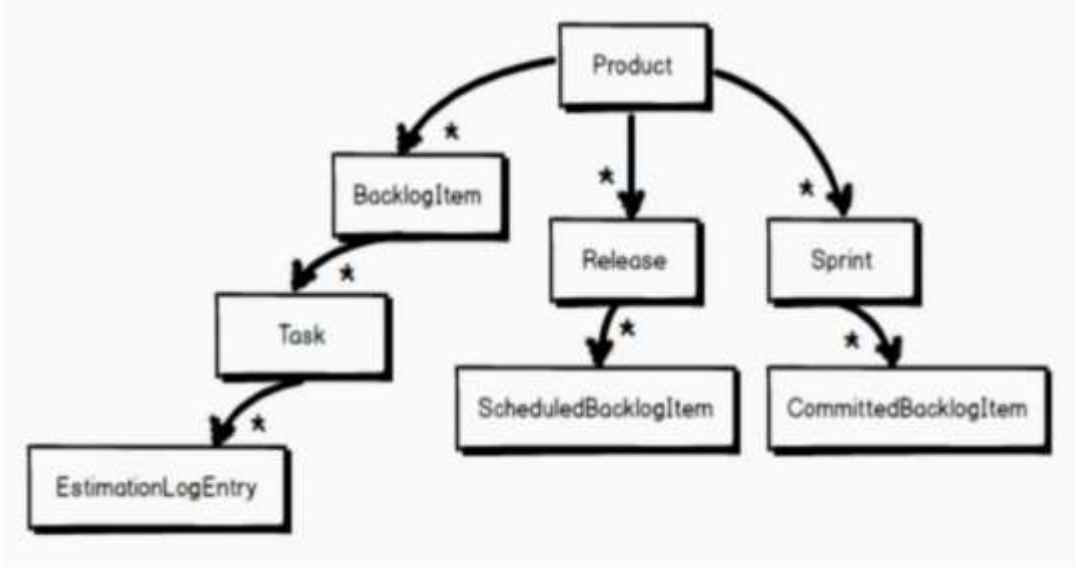
- Aggregates are domain concepts (order, clinic visit, playlist), while collections are generic. An aggregate will often contain multiple collections, together with simple fields.



Source: Martin Fowler : Aggregate Root

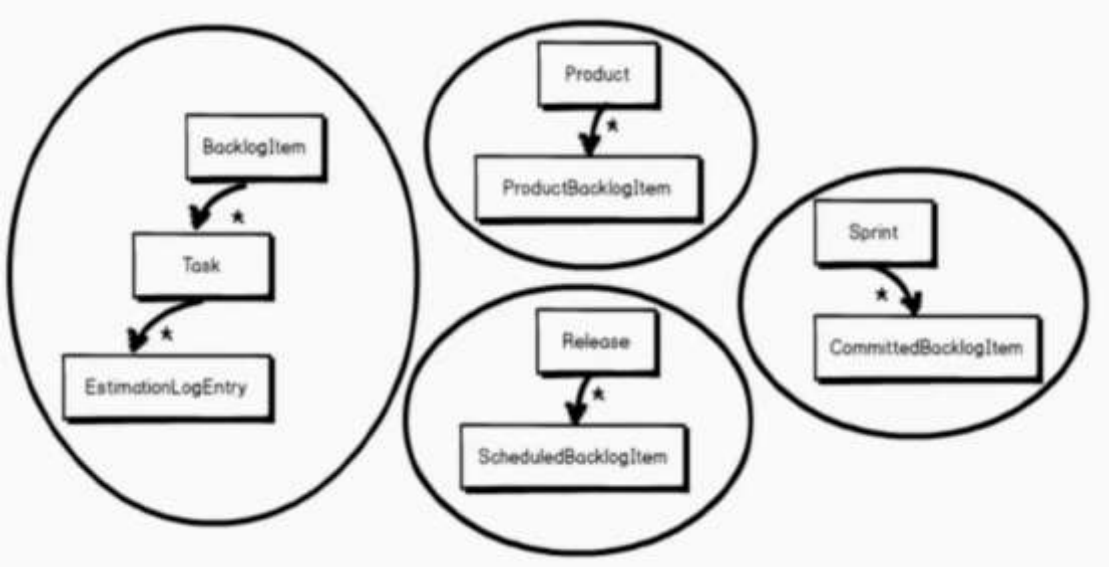
Designing and Fine Tuning Aggregate Root

Aggregate Root - #1



Super Dense Single Aggregate Root
Results in Transaction concurrency issues.

Aggregate Root - #2



Super Dense Aggregate Root is split into 4 different smaller Aggregate Root in the 2nd Iteration.

Working on different design models helps the developers to come up with best possible design.

Source : Effective Aggregate Design Part 1/2/3 : Vaughn Vernon
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf

Rules for Building Aggregate Roots

1. Protect True Invariants in Consistency Boundaries. This rule has the added implication that you should modify just one Aggregate instance in a single transaction. In other words, when you are designing an Aggregate composition, plan on that representing a transaction boundary.

2. **Design Small Aggregates.** The smallest Aggregate you can design is one with a single Entity, which will serve as the Aggregate Root.

3. Reference Other Aggregates **Only By Identity.**

4. Use **Eventual Consistency** Outside the Consistency Boundary. This means that ONLY ONE Aggregate instance will be required to be updated in a single transaction. All other Aggregate instances that must be updated as a result of any one Aggregate instance update can be updated within some time frame (using a Domain Event). The business should determine the allowable time delay.

5. Build Unidirectional Relationship from the Aggregate Root.

Data Transfer Object vs. Value Object

A small simple object, like money or a date range, whose equality isn't based on identity.

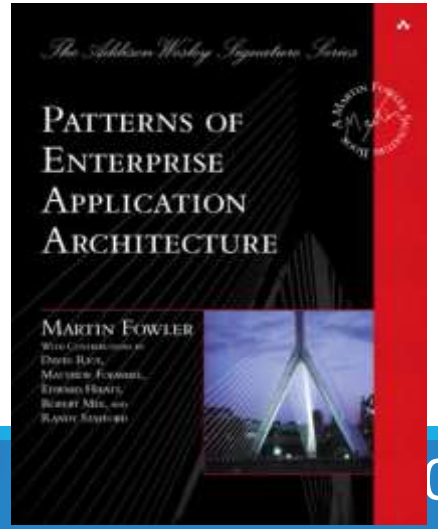
Data Transfer Object	Value Object
A DTO is just a data container which is used to transport data between layers and tiers.	A Value Object represents itself a fix set of data and is similar to a Java enum.
It mainly contains of attributes and it's a serializable object.	A Value Object doesn't have any identity, it is entirely identified by its value and is immutable.
DTOs are anemic in general and do not contain any business logic.	A real world example would be Color.RED, Color.BLUE, Currency.USD

486
P of EAA

Java EE 7 Retired the DTO

In Java EE the RS spec became the de-facto standard for remoting, so the implementation of serializable interface is no more required. To transfer data between tiers in Java EE 7 you get the following for FREE!

1. JAXB : Offer JSON / XML serialization for Free.
2. Java API for JSON Processing – Directly serialize part of the Objects into JSON



DTO – Data Transfer Object

401
P of EAA

An object that carries data between processes in order to reduce the number of method calls.

Problem: How do you preserve the simple semantics of a procedure call interface without being subject to the latency issues inherent in remote communication?

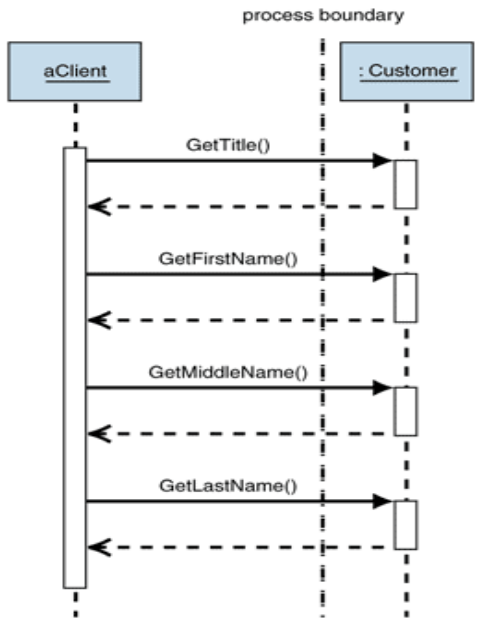
Benefits

1. Reduced Number of Calls
2. Improved Performance
3. Hidden Internals
4. Discovery of Business objects

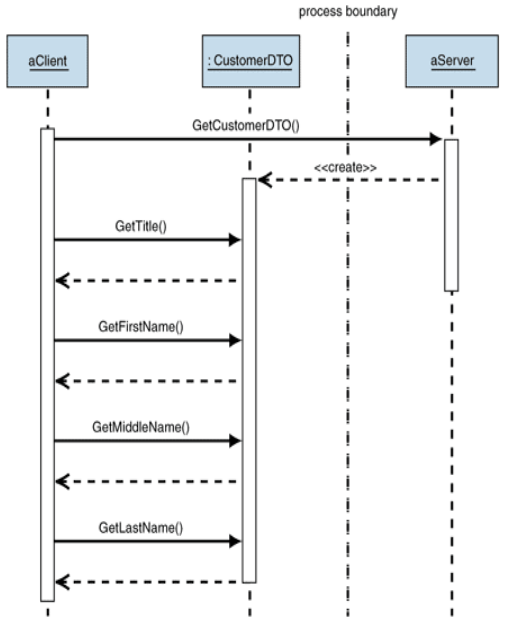
Liabilities

1. Class Explosion
2. Additional Computation
3. Additional Coding Effort

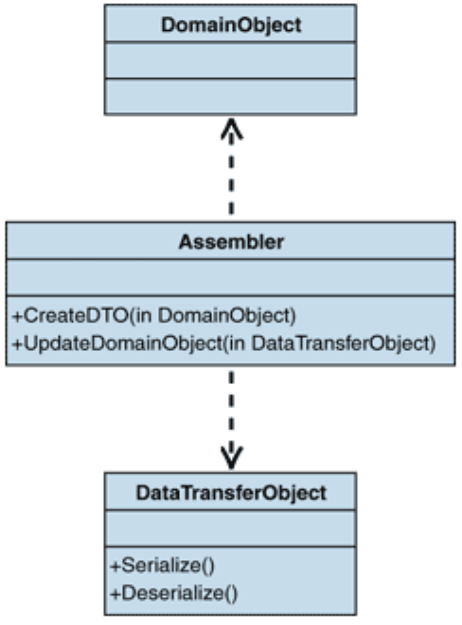
The Problem



The Solution



Assembler Pattern



Security Considerations

Data obtained from untrusted sources, such as user input from a Web page, should be cleansed and validated before being placed into a DTO. Doing so enables you to consider the data in the DTO relatively safe, which simplifies future interactions with the DTO.

DTO – Data Transfer Object

401
P of EAA

An object that carries data between processes in order to reduce the number of method calls.

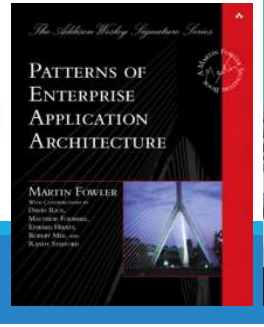
Don't underestimate the cost of [using DTOs].... It's significant, and it's painful - perhaps second only to the cost and pain of object-relational mapping.

Another argument I've heard is using them in case you want to distribute later. This kind of speculative distribution boundary is what I rail against. Adding remote boundaries adds complexity.

One case where it is useful to use something like a DTO is when you have a significant mismatch between the model in your presentation layer and the underlying domain model.

In this case it makes sense to make presentation specific facade/gateway that maps from the domain model and presents an interface that's convenient for the presentation.

Patterns of Enterprise Application Architecture : Martin Fowler
http://martinfowler.com/books/eaa.html

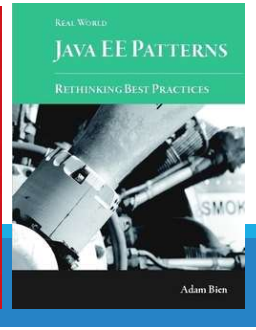


The most misused pattern in the Java Enterprise community is the DTO.

DTO was clearly defined as a solution for a distribution problem.

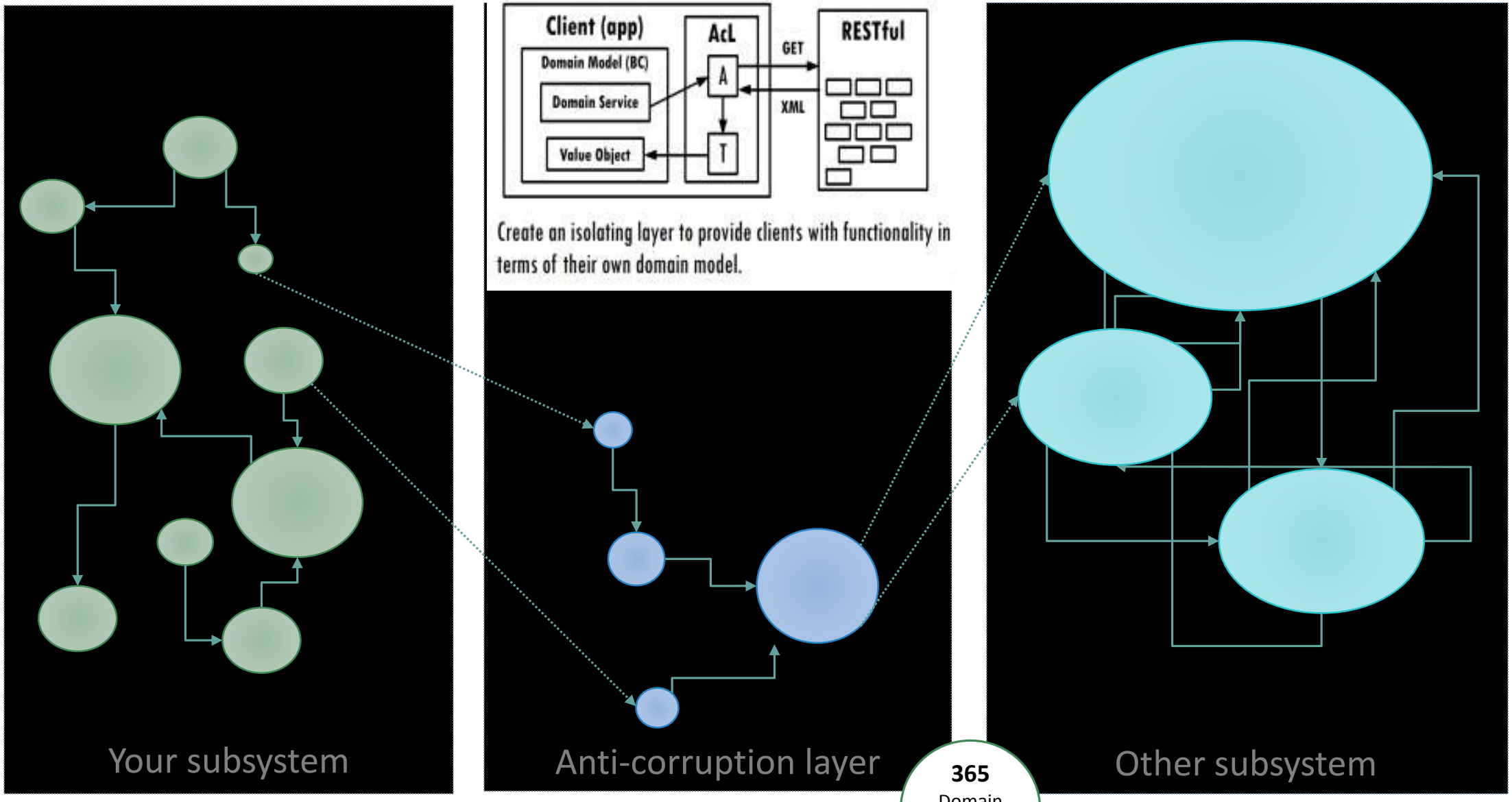
DTO was meant to be a coarse-grained data container which efficiently transports data between processes (tiers).

On the other hand considering a dedicated DTO layer as an investment, rarely pays off and often lead to over engineered bloated architecture.



Real World Java
EE Patterns
Adam Bien
http://realworldpatterns.com

Anti Corruption Layer – ACL



365
Domain
Driven
Design

Repository Pattern

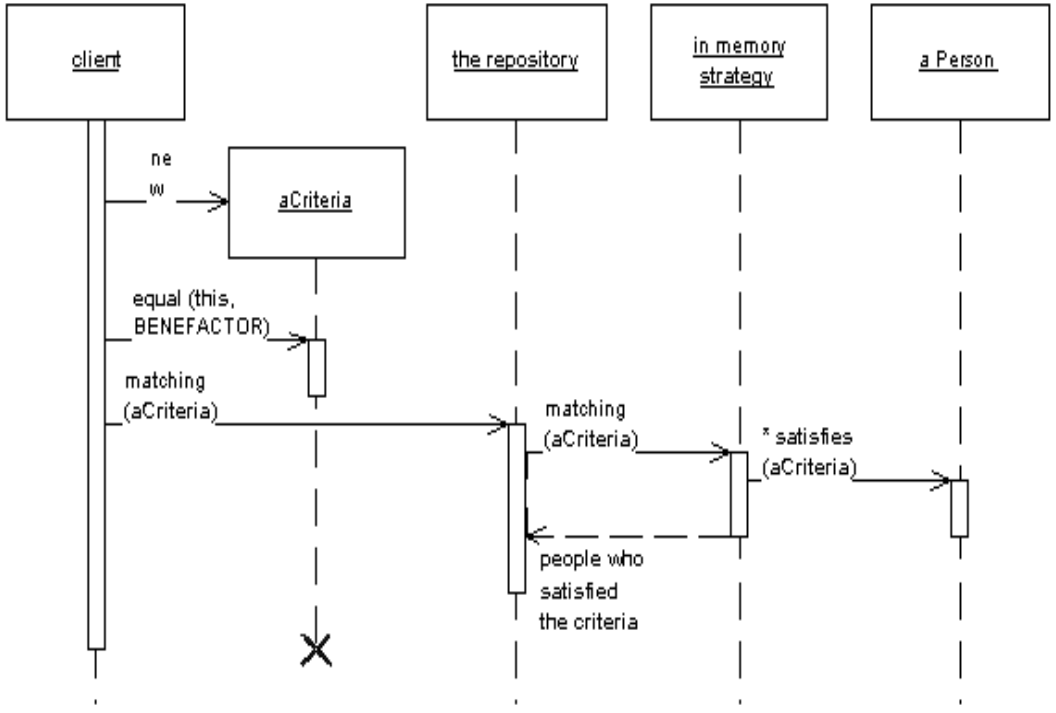
322
P of EAA

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

Objectives

Use the Repository pattern to achieve one or more of the following objectives:

- You want to maximize the amount of code that can be tested with automation and to isolate the data layer to support unit testing.
- You access the data source from many locations and want to apply centrally managed, consistent access rules and logic.
- You want to implement and centralize a caching strategy for the data source.
- You want to improve the code's maintainability and readability by separating business logic from data or service access logic.
- You want to use business entities that are strongly typed so that you can identify problems at compile time instead of at run time.
- You want to associate a behavior with the related data. For example, you want to calculate fields or enforce complex relationships or business rules between the data elements within an entity.
- You want to apply a domain model to simplify complex business logic.



Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

Repository Pattern Source:
Martin Fowler : http://martinfowler.com/eaaCatalog/repository.html | Microsoft : https://msdn.microsoft.com/en-us/library/ff649690.aspx

Anemic Domain Model : Anti Pattern

- There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have.
- The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.
- The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented design; which is to combine data and process together.
- The anemic domain model is really just a procedural style design, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk.

```java
package com.fusionfire.examples.commons.utils;

import java.util.ArrayList;

public class AnemicUser {

    private String name;

    private boolean isUserLocked;

    private ArrayList<String> addresses;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public boolean isUserLocked() {
        return isUserLocked;
    }

    public void setUserLocked(boolean isUserLocked) {
        this.isUserLocked = isUserLocked;
    }

    public ArrayList<String> getAddresses() {
        return addresses;
    }

    public void setAddresses(ArrayList<String> addresses) {
        this.addresses = addresses;
    }

}
```

- lockUser()
- unlockUser()
- addAddress(String address)
- removeAddress(String address)

Source: Anemic Domain Model By Martin Fowler : http://martinfowler.com/bliki/AnemicDomainModel.html

Procedural Design Vs. Domain Driven Design

```java
@Stateless
public class ShipmentService {
  public final static int BASIC_COST = 5;

  @PersistenceContext
  private EntityManager em;

    public int getShippingCosts(int loadId) {
        Load load = em.find(Load.class, loadId);
        return computeShippingCost(load);
    }

    int computeShippingCost(Load load){
        int shippingCosts = 0;
        int weight = 0;
        int defaultCost = 0;
        for (OrderItem orderItem : load.getOrderItems()) {
            LoadType loadType = orderItem.getLoadType();
            weight = orderItem.getWeight();
            defaultCost = weight * 5;
            switch (loadType) {
                case BULKY:
                    shippingCosts += (defaultCost + 5);
                    break;
                case LIGHTWEIGHT:
                    shippingCosts += (defaultCost - 1);
                    break;
                case STANDARD:
                    shippingCosts += (defaultCost);
                    break;
                default:
               throw new IllegalStateException("Unknown type: " + loadType);
            }
        }
        return shippingCosts;
    }
}
```

1. Anemic Entity Structure
2. Massive IF Statements
3. Entire Logic resides in Service Layer
4. Type Dependent calculations are done based on conditional checks in Service Layer

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

Polymorphic Business Logic inside a Domain object

```java
@Entity
public class Load {

    @OneToMany(cascade = CascadeType.ALL)
    private List<OrderItem> orderItems;
    @Id
    private Long id;

    protected Load() {
        this.orderItems = new ArrayList<OrderItem>();
    }

    public int getShippingCosts() {
        int shippingCosts = 0;
        for (OrderItem orderItem : orderItems) {
            shippingCosts += orderItem.getShippingCost();
        }
        return shippingCosts;
    }
//...
}
```

Computation of the total cost realized inside a rich Persistent Domain Object (PDO) and not inside a service.

This simplifies creating very complex business rules.

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

Type Specific Computation in a Sub Class

```java
@Entity
public class BulkyItem extends OrderItem{

    public BulkyItem() {
    }

    public BulkyItem(int weight) {
        super(weight);
    }

    @Override
    public int getShippingCost() {
        return super.getShippingCost() + 5;
    }
}
```

of

We can change the computation of the shipping cost of a Bulky Item without touching the remaining classes.

Its easy to introduce a new Sub Class without affecting the computation of the total cost in the Load Class.

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

Object Construction : Procedural Way Vs. Builder Pattern

Procedural Way

```
Load load = new Load();
OrderItem standard = new OrderItem();
standard.setLoadType(LoadType.STANDARD);
standard.setWeight(5);
load.getOrderItems().add(standard);
OrderItem light = new OrderItem();
light.setLoadType(LoadType.LIGHTWEIGHT);
light.setWeight(1);
load.getOrderItems().add(light);
OrderItem bulky = new OrderItem();
bulky.setLoadType(LoadType.BULKY);
bulky.setWeight(1);
load.getOrderItems().add(bulky);
```

Builder Pattern

```
Load build = new Load.Builder().
    withStandardItem(5).
    withLightweightItem(1).
    withBulkyItem(1).
    build();
```

Domain Driven Design with Java EE 6
By Adam Bien | Javaworld

Source: http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html

DDD – Summary

Strategic Design

1. Bounded Context
2. Ubiquitous Language
3. Context Map

Tactical Design

1. Aggregate Root
2. Value Object
3. Domain Events
4. Repository Pattern
5. Factory

More on this in Event Sourcing & CQRS Section.

Shopping Portal

Order Module

Domain Layer

Models

Value Object

- Currency
- Item Value
- Order Status
- Payment Type
- Record State
- Audit Log

Entity

- **Order** (Aggregate Root)
- Order Item
- Shipping Address
- Payment

DTO

- Order
- Order Item
- Shipping Address
- Payment

Services / Ports

- Order Repository
- Order Service
- Order Web Service
- Order Query Web Service
- Shipping Address Web Service
- Payment Web Service

Utils

- Order Factory
- Order Status Converter
- Record State Converter

Adapters

- Order Repository
- Order Service
- Order Web Service
- Order Query Web Service
- Shipping Address Web Service
- Payment Web Service

Adapters Consists of Actual Implementation of the Ports like Database Access, Web Services API etc.

Converters are used to convert an Enum value to a proper Integer value in the Database. For Example Order Status Complete is mapped to integer value 100 in the database.

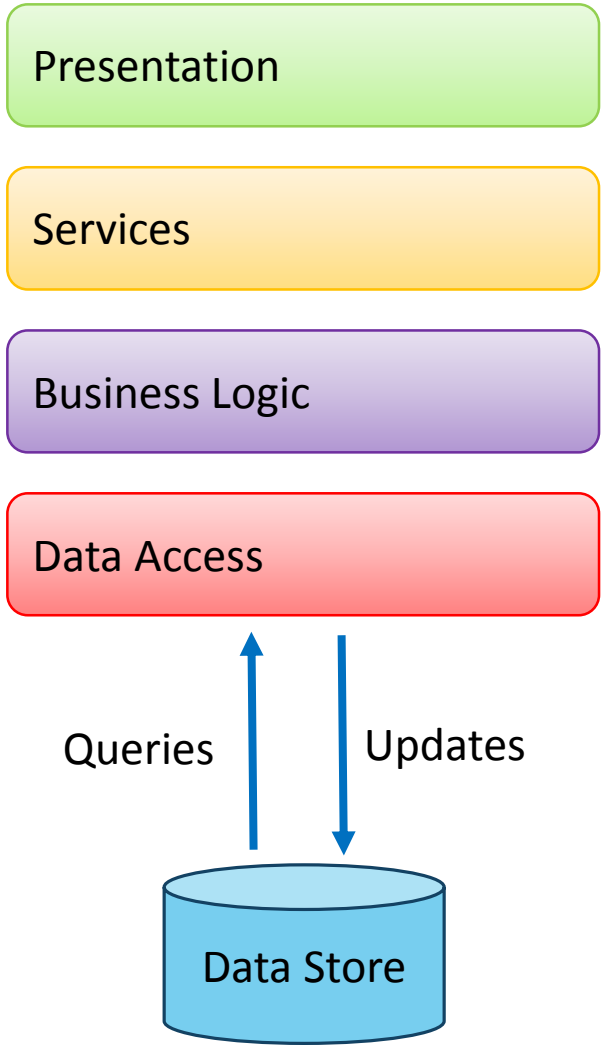
CRUD / CQRS & Event Sourcing

A brief introduction, more in Part 2 of the Series
Event Storming and SAGA

CRUD and CQRS

Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
https://blogs.msdn.microsoft.com/maarten_mullender/2004/07/23/crud-only-when-you-can-afford-it-revisite

Traditional CRUD Architecture

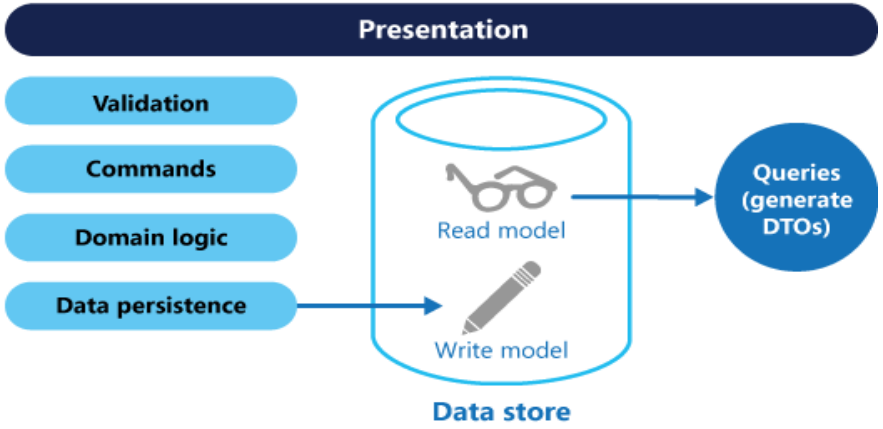


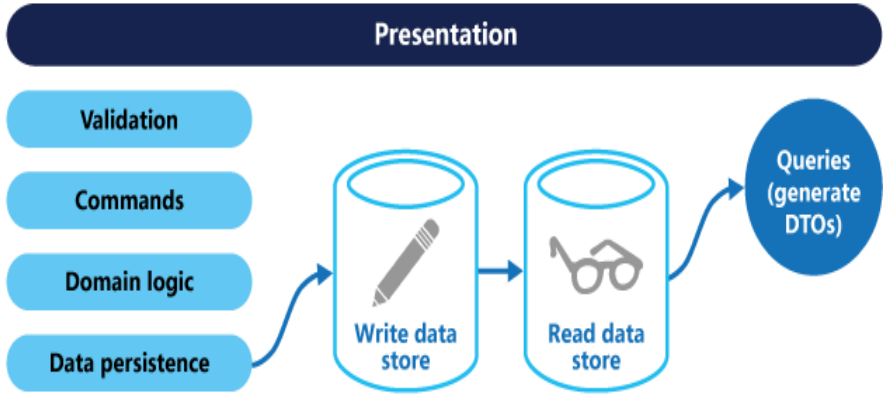
CRUD Disadvantages

- **A mismatch between the read and write** representations of the data.
- It **risks data contention when records are locked in the data store** in a collaborative domain, where multiple actors operate in parallel on the same set of data. These risks increase as the complexity and throughput of the system grows.
- It can make **managing security and permissions more complex** because each entity is subject to both read and write operations, which might expose data in the wrong context.

Event Sourcing & CQRS (Command and Query Responsibility Segregation)

- In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository.
- CQRS is a pattern that segregates the operations that read data (Queries) from the operations that update data (Commands) by using separate interfaces.
- CQRS should only be used on specific portions of a system in Bounded Context (in DDD).
- CQRS should be used along with Event Sourcing.





Java Axon Framework Resource : http://www.axonframework.org

MSDN – Microsoft https://msdn.microsoft.com/en-us/library/dn568103.aspx | Martin Fowler : CQRS – http://martinfowler.com/bliki/CQRS.html

Axon Framework For Java

CQS : Bertrand Meyer

Greg Young

Event Sourcing Intro

Standard CRUD Operations – Customer Profile – Aggregate Root

Profile Created	Title Updated	New Address added	Notes Removed
Time T1	T2	T3	T4

Event Sourcing and Derived Aggregate Root

Commands

1. Create Profile
2. Update Title
3. Add Address
4. Delete Notes

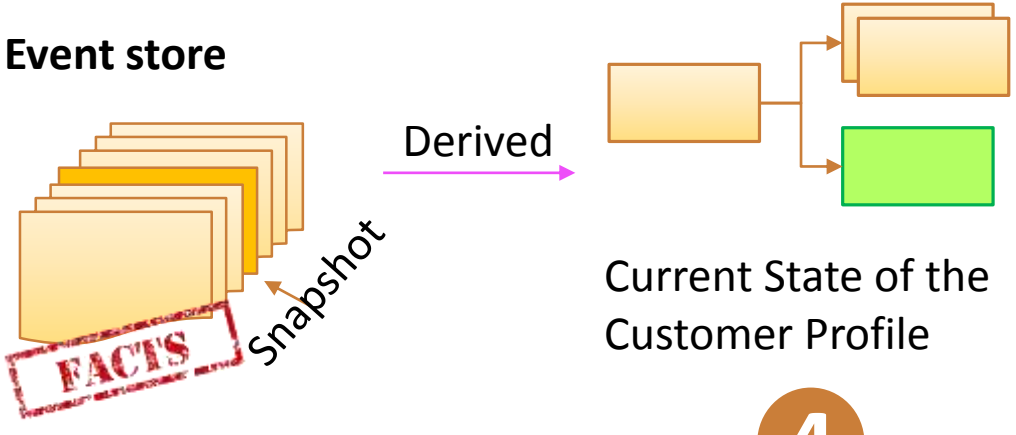
2

Events

1. Profile Created Event
2. Title Updated Event
3. Address Added Event
4. Notes Deleted Event

3

Event store



4

Greg Young

Restaurant Dining – Event Sourcing and CQRS

Processes 1

When people arrive at the Restaurant and take a table, a Table is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served. Table is closed then the bill is prepared.

Customer Journey thru Dinning Processes

Commands 2

- Add Drinks
- Add Food
- Update Food

- **Open Table**
- Add Juice
- Add Soda
- Add Appetizer 1
- Add Appetizer 2

- Remove Soda
- Add Food 1
- Add Food 2
- Place Order
- **Close Table**

- Serve Drinks
- Prepare Food
- Serve Food

- **Prepare Bill**
- Process Payment

ES Aggregate 4

- Dinning Order
- Billable Order

Microservices

Food Menu | Dining | Kitchen | Order | Payment

Events 3

- Drinks Added
- Food Added
- Food Updated
- Food Discontinued

- Table Opened
- Juice Added
- Soda Added
- Appetizer 1 Added
- Appetizer 2 Added

- Remove Soda
- Food 1 Added
- Food 2 Added
- Order Placed
- Table Closed

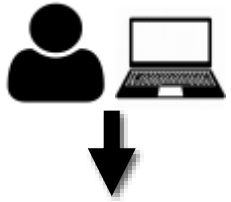
- Juice Served
- Soda Served
- Appetizer Served
- Food Prepared
- Food Served

- Bill Prepared
- Payment Processed

- Payment Approved
- Payment Declined
- Cash Paid

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Use Case : Shopping Portal – Event Sourcing / CQRS



Commands are End-User interaction with the App and based on the commands (Actions) Events are created. These Events includes both Domain Events and Integration Events. Event Sourced Aggregates will be derived using Domain Events. Each Micro Service will have its own separate Database. Depends on the scalability requirement each of the Micro Service can be scaled separately. For Example. Catalogue can be on a 50 node cluster compared to Customer Micro Service.

Customer Journey thru Shopping Process

Commands 2

- Search Products
- Add Products
- Update Products

- Add to Cart
- Remove Item
- Update Quantity

- Process Order
- Select Address
- Select Delivery Mode

- Proceed for Payment
- Confirm Order for Payment
- Cancel Order

ES Aggregate 4

- Customer
- Shop.. Cart
- Order

Catalogue

Shopping Cart

Customer

Order

Payment

Microservices

Events 3

- Product Added
- Product Updated
- Product Discontinued

- Item Added
- Item Removed / Discontinued
- Item Updated
- Order Initiated

- Address Selected
- Delivery Mode Selected
- Order Created

- Payment Initiated
- Order Cancelled
- Order Confirmed

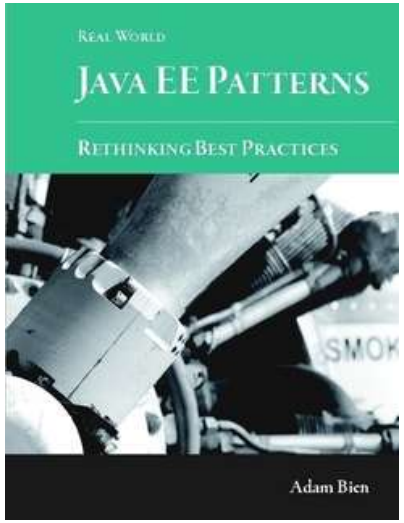
- OTP Send
- Payment Approved
- Payment Declined

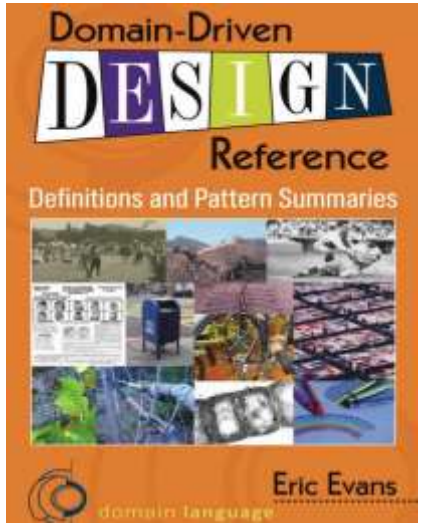
The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Summary – Event Sourcing and CQRS

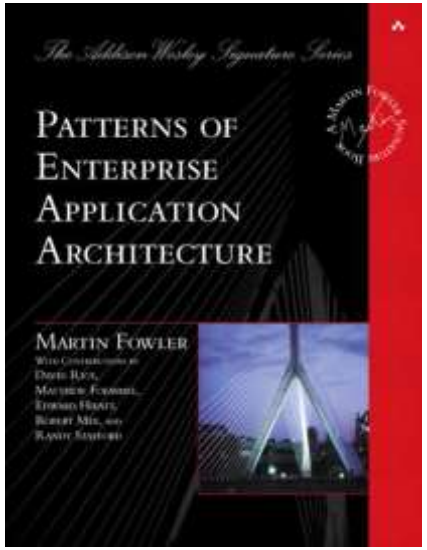
1. Immutable Events
2. Events represents the state change in Aggregate Root
3. Aggregates are Derived from a Collection of Events.
4. Separate Read and Write Models
5. Commands (originated from user or systems) creates Events.
6. Commands and Queries are always separated and possibly reads and writes using different data models.

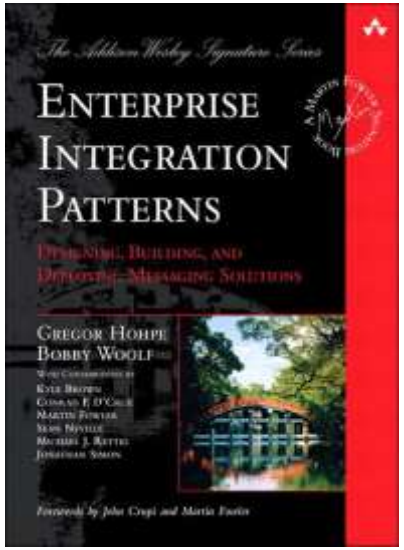
Design Patterns – Holy Grail of Developers

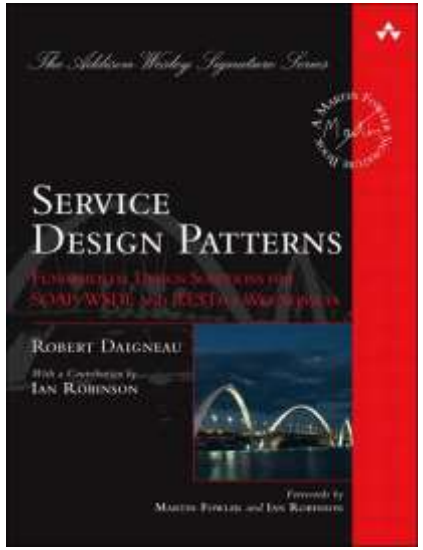


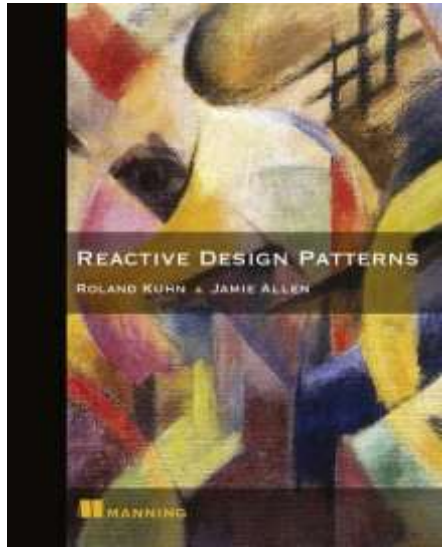


Design Patterns are solutions to general problems that software developers faced during software development.









References

1. Lewis, James, and Martin Fowler. "Microservices: A Definition of This New Architectural Term", March 25, 2014.
2. Miller, Matt. "Innovate or Die: The Rise of Microservices". *e Wall Street Journal*, October 5, 2015.
3. Newman, Sam. *Building Microservices*. O'Reilly Media, 2015.
4. Alagarasan, Vijay. "Seven Microservices Anti-patterns", August 24, 2015.
5. Cockcroft, Adrian. "State of the Art in Microservices", December 4, 2014.
6. Fowler, Martin. "Microservice Prerequisites", August 28, 2014.
7. Fowler, Martin. "Microservice Tradeoffs", July 1, 2015.
8. Humble, Jez. "Four Principles of Low-Risk Software Release", February 16, 2012.
9. Zuul Edge Server, Ketan Gote, May 22, 2017
10. Ribbon, Hysterix using Spring Feign, Ketan Gote, May 22, 2017
11. Eureka Server with Spring Cloud, Ketan Gote, May 22, 2017
12. Apache Kafka, A Distributed Streaming Platform, Ketan Gote, May 20, 2017
13. Functional Reactive Programming, Araf Karsh Hamid, August 7, 2016
14. Enterprise Software Architectures, Araf Karsh Hamid, July 30, 2016
15. Docker and Linux Containers, Araf Karsh Hamid, April 28, 2015

References

Domain Driven Design

16. Oct 27, 2012 What I have learned about DDD Since the book. By Eric Evans
17. Mar 19, 2013 Domain Driven Design By Eric Evans
18. May 16, 2015 Microsoft Ignite: Domain Driven Design for the Database Driven Mind
19. Jun 02, 2015 Applied DDD in Java EE 7 and Open Source World
20. Aug 23, 2016 Domain Driven Design the Good Parts By Jimmy Bogard
21. Sep 22, 2016 GOTO 2015 – DDD & REST Domain Driven API's for the Web. By Oliver Gierke
22. Jan 24, 2017 Spring Developer – Developing Micro Services with Aggregates. By Chris Richardson
23. May 17. 2017 DEVOXX – The Art of Discovering Bounded Contexts. By Nick Tune

Event Sourcing and CQRS

23. Nov 13, 2014 GOTO 2014 – Event Sourcing. By Greg Young
24. Mar 22, 2016 Spring Developer – Building Micro Services with Event Sourcing and CQRS
25. Apr 15, 2016 YOW! Nights – Event Sourcing. By Martin Fowler
26. May 08, 2017 When Micro Services Meet Event Sourcing. By Vinicius Gomes

References

27. MSDN – Microsoft https://msdn.microsoft.com/en-us/library/dn568103.aspx
28. Martin Fowler : CQRS – http://martinfowler.com/bliki/CQRS.html
29. Udi Dahan : CQRS – http://www.udidahan.com/2009/12/09/clarified-cqrs/
30. Greg Young : CQRS - https://www.youtube.com/watch?v=JHGkaShoyNs
31. Bertrand Meyer – CQS - http://en.wikipedia.org/wiki/Bertrand_Meyer
32. CQS : http://en.wikipedia.org/wiki/Command–query_separation
33. CAP Theorem : http://en.wikipedia.org/wiki/CAP_theorem
34. CAP Theorem : http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
35. CAP 12 years how the rules have changed
36. EBay Scalability Best Practices : http://www.infoq.com/articles/ebay-scalability-best-practices
37. Pat Helland (Amazon) : Life beyond distributed transactions
38. Stanford University: Rx https://www.youtube.com/watch?v=y9xudo3C1Cw
39. Princeton University: SAGAS (1987) Hector Garcia Molina / Kenneth Salem
40. Rx Observable : https://dzone.com/articles/using-rx-java-observable

References – Micro Services – Videos

41. Martin Fowler – Micro Services : https://www.youtube.com/watch?v=2yko4TbC8cI&feature=youtu.be&t=15m53s
42. GOTO 2016 – Microservices at NetFlix Scale: Principles, Tradeoffs & Lessons Learned. By R Meshenberg
43. Mastering Chaos – A NetFlix Guide to Microservices. By Josh Evans
44. GOTO 2015 – Challenges Implementing Micro Services By Fred George
45. GOTO 2016 – From Monolith to Microservices at Zalando. By Rodrigue Scaefer
46. GOTO 2015 – Microservices @ Spotify. By Kevin Goldsmith
47. Modelling Microservices @ Spotify : https://www.youtube.com/watch?v=7XDA044tI8k
48. GOTO 2015 – DDD & Microservices: At last, Some Boundaries By Eric Evans
49. GOTO 2016 – What I wish I had known before Scaling Uber to 1000 Services. By Matt Ranney
50. DDD Europe – Tackling Complexity in the Heart of Software By Eric Evans, April 11, 2016
51. AWS re:Invent 2016 – From Monolithic to Microservices: Evolving Architecture Patterns. By Emerson L, Gilt D. Chiles
52. AWS 2017 – An overview of designing Microservices based Applications on AWS. By Peter Dalbhanjan
53. GOTO Jun, 2017 – Effective Microservices in a Data Centric World. By Randy Shoup.
54. GOTO July, 2017 – The Seven (more) Deadly Sins of Microservices. By Daniel Bryant
55. Sept, 2017 – Airbnb, From Monolith to Microservices: How to scale your Architecture. By Melanie Cubula
56. GOTO Sept, 2017 – Rethinking Microservices with Stateful Streams. By Ben Stopford.
57. GOTO 2017 – Microservices without Servers. By Glynn Bird.

Thank you

Araf Karsh Hamid : Co-Founder / CTO
araf.karsh@metamagic.in
USA: +1 (973) 969-2921
India: +91.999.545.8627
Skype / LinkedIn / Twitter / Slideshare : arafkarsh
http://www.slideshare.net/arafkarsh
https://www.linkedin.com/in/arafkarsh/

http://www.slideshare.net/arafkarsh/software-architecture-styles-64537120

http://www.slideshare.net/arafkarsh/functional-reactive-programming-64780160

http://www.slideshare.net/arafkarsh/function-point-analysis-65711721

JPA 2.2 and JDBC

Java Type	JDBC Type
java.time.LocalDate	DATE
java.time.LocalTime	TIME
java.time.LocalDateTime	TIMESTAMP
java.time.OffsetTime	TIME_WITH_TIMEZONE
java.time.OffsetDateTime	TIMESTAMP_WITH_TIMEZONE