第四章：Architecture 架構風格與 DDD 的協奏曲

目錄

開場

開場引言

"Architecture should speak of its time and place, but yearn for timelessness." - *Frank Gehry*

核心觀點

- DDD 是一套以 **Bounded Context** 劃界的思維框架，而非單一實作架構。
- 架構 = **品質屬性** + **功能需求**的平衡藝術。
- 架構應服務於 Domain 模型；勿讓技術層凌駕業務語言。
- 追求可演進，抵禦未知需求，才能保持系統韌性。

SaaSOvation 演進地圖

SaaSOvation 架構演進

階段	架構	驅動	收穫	風險
Startup	Monolith (Layered)	MVP 速度	快速迭代	技術債
Scale-up	Hexagonal + CQRS	可測試性	關注點分離	邊界模糊
Enterprise	多 Context + SOA	團隊協作	去耦	協調成本

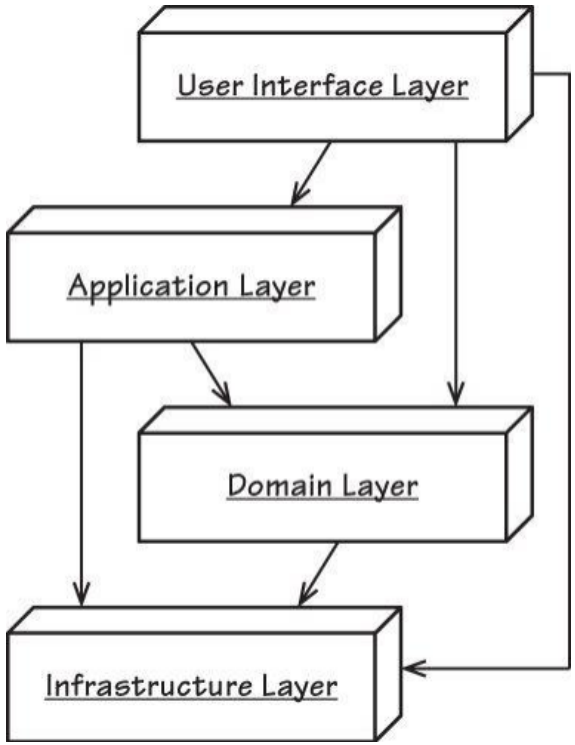
啟示

沒有一步到位的架構：**需求 × 風險 × 演進式重構 才是真正的長青之道。**

Layered Architecture

傳統分層架構 (Traditional Layers)

- 傳統分層架構：UI → Application → Domain → Infrastructure
- 每層只能依賴自己和下層
- 優點：結構清晰、好上手
- 問題：Repository 介面在 Domain，實作在 Infrastructure
- 缺陷：違反層級規則、難以測試、技術層凌駕業務層



依賴反轉原則 (DIP)

- 高階模組不應依賴低階模組，兩者都應依賴抽象
- 抽象不應依賴細節，細節應依賴抽象
- 解決方案：引入 DIP
- 邏輯上將 Infrastructure 層「移至上層」
- 實際上是讓 Infrastructure 實作 Domain 定義的介面

Layered Java (DIP)

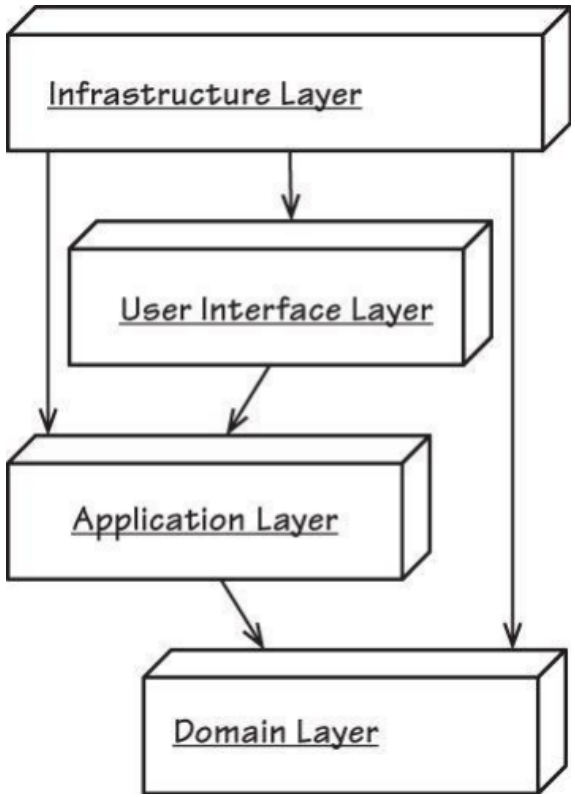
```
@Transactional
public void commitBacklogItemToSprint(
    String aTenantId, String aBacklogItemId, String aSprintId) {
    TenantId tenantId = new TenantId(aTenantId);
    BacklogItem backlogItem =
        backlogItemRepository.backlogItemOfId(
            tenantId, new BacklogItemId(aBacklogItemId));
    Sprint sprint = sprintRepository.sprintOfId(
        tenantId, new SprintId(aSprintId));
    backlogItem.commitTo(sprint);
}
```

Layered Java (DIP)

```
1  package
2  com.saasovation.agilepm.infrastructure.persistence;
3  import com.saasovation.agilepm.domain.model.product.*;
4  public class HibernateBacklogItemRepository
5  implements BacklogItemRepository {
6  ...
7  @Override
8  @SuppressWarnings("unchecked")
9  public Collection<BacklogItem>
10 allBacklogItemsComittedTo(
11 Tenant aTenant, SprintId aSprintId) {
12 Query query =
13 this.session().createQuery(
14 "from -BacklogItem as _obj_ "
15 + "where _obj_.tenant = ? and
16 _obj_.sprintId = ?");
17 query.setParameter(0, aTenant);
18 query.setParameter(1, aSprintId);
19 return (Collection<BacklogItem>) query.list();
20 }
21 ...
22 }
```

DIP：分層的變革

- 傳統分層架構是「高階元件依賴低階元件」
- DIP 顛倒依賴方向：Domain 定義介面，Infrastructure 實作
- Infrastructure 被「邏輯上」放在最上層
- 它是「實作者」，不是「控制者」
- 符合 DDD 精神：Domain 是中心，技術在外圍服務它

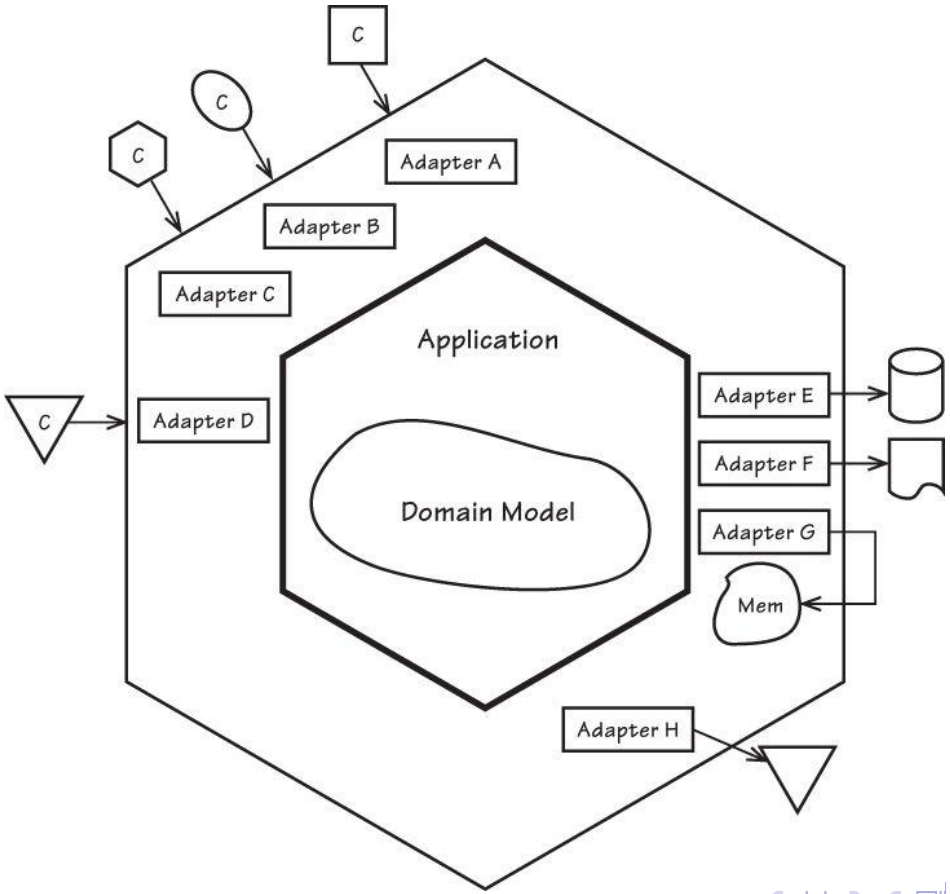


Hexagonal Architecture

Hexagonal 架構三大核心

1. 所有依賴都「向內」指向應用核心（Domain + Use Cases）
2. 對外協定由 Adapter 處理（如 HTTP、AMQP、JDBC）
3. Application Service 對 Port 發出請求，由 Adapter 實作

→ **避免耦合框架，實現技術可替換性、測試友善性、邊界清晰性**

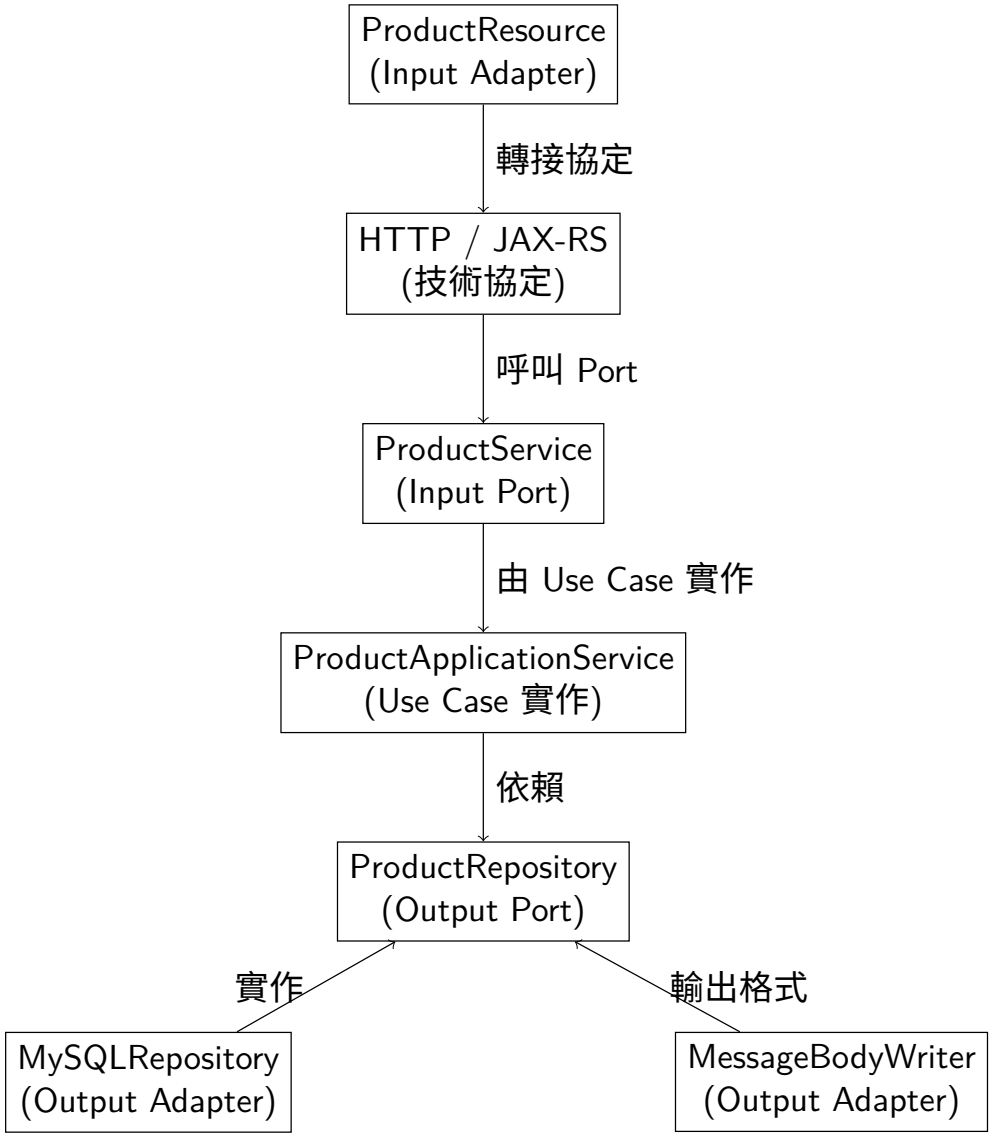


Hexagonal 架構 Input Port Java 範例

```java
// === Input Adapter ===
@Path("/tenants/{tenantId}/products") // 接收 HTTP 請求，是 Input Adapter
public class ProductResource extends Resource {
    // === Port ===
    private ProductService productService; // 這是 Input Port（介面）
    @GET
    @Path("{productId}")
    @Produces({"application/vnd.saasovation.projectovation+xml"})
    public Product getProduct(
        @PathParam("tenantId") String aTenantId,
        @PathParam("productId") String aProductId,
        @Context Request aRequest) {
        // 呼叫 Port 的方法，將外部資料導入應用層（Port 實作中會使用 Domain Model）
        Product product = productService.product(aTenantId, aProductId);
        if (product == null) {
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        // 回傳的是 Domain 物件，由 Output Adapter（MessageBodyWriter）序列化為 XML
        return product;
    }
}
```

Hexagonal 架構的 Port 與 Adapter

依賴方向皆由外往內，所有 *Adapter* 實作 *Port*，所有 *Port* 為內部定義



Hexagonal 架構的 DDD 價值

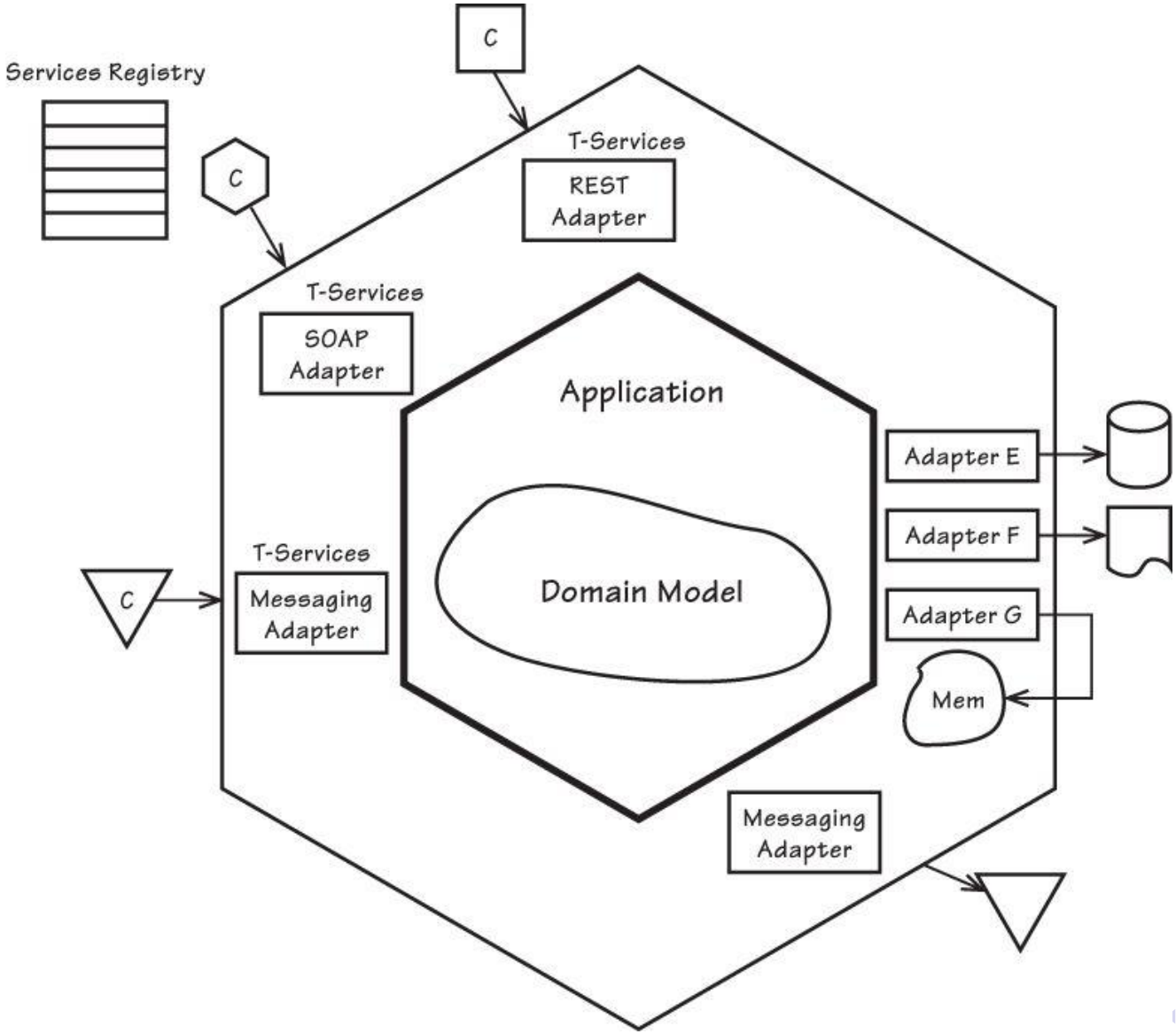
- 隔離技術細節，讓核心 Domain 不依賴外部框架
- 測試容易：可以在無 UI / 無 DB 下進行完整測試
- 高擴充性：新增一種協定只需新增對應 Adapter
- 是實踐 DDD 的最佳部署架構之一

SOA Architecture

SOA 架構精要

- 每個 Bounded Context $\rightarrow$ 獨立服務 (Service)
- 技術介面可為 REST / SOAP / Messaging
- 不同技術服務可共享同一語意邊界（如一組 REST + Queue 為一 BC 的對外服務）
- 一個業務服務 = 多個技術服務 + 多個 Bounded Context
- 治理與韌性：Registry、Contract-First、Versioning、Circuit Breaker、Retry 等

Hexagonal Architecture supporting SOA, with REST, SOAP, and messaging services



REST architectural style

Contract-First Design

- 先編寫明確的介面定義（「合約」），再實作服務程式碼
- 合約成為單一事實來源：
  - **Provider**：依照合約生成或手寫服務端程式碼
  - **Consumer**：生成存根/SDK 或用於驗證與模擬
  - **Governance**：版本控制、相容性檢查、策略執行、文件

協定	合約格式	程式碼生成工具
REST/HTTP	OpenAPI 3 / AsyncAPI	Swagger Codegen, openapi-generator
gRPC	Protocol Buffers (.proto)	protoc, Buf, Grpc-Tools
SOAP	WSDL + XSD	wsimport, wsdl2java
Events	Avro Schema, AsyncAPI	avro-tools, Karate, Springs Cloud Stream

為何選擇 Contract-First？

Code-First（實作優先）	**Contract-First（合約優先）**
API 常模仿內部類別結構；後續重構會破壞客戶端	API 專為外部需求設計；與內部模型解耦
客戶端團隊需等待服務端存根或模擬服務	合約早期發布 → 客戶端和服務端可並行工作
更難追蹤破壞性變更	可比較合約差異，執行語義版本檢查
較少保證；文件容易過時	程式碼、測試、文件、模擬都從單一來源生成

REST Java 實踐範例

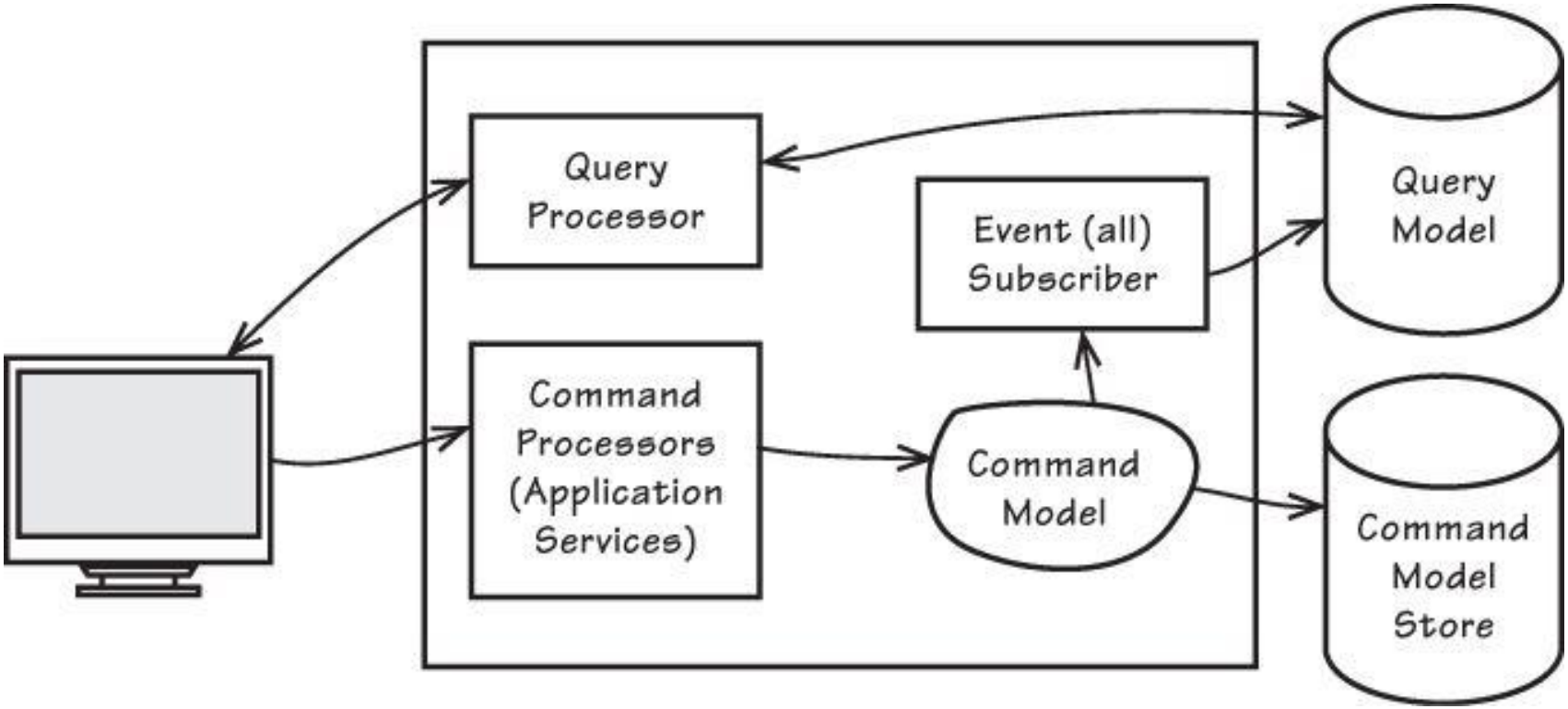
```java
@RestController
@RequestMapping("/orders")
class OrderController {
    private final OrderService svc;
    OrderController(OrderService svc) { this.svc = svc; }

    @GetMapping("/{id}")
    RepresentationModel<OrderDTO> get(@PathVariable String id) {
        Order o = svc.findById(id);
        OrderDTO dto = new OrderDTO(o.status().name());
        dto.add(linkTo(methodOn(OrderController.class).get(id)).withSelfRel());
        return dto;
    }
}
```

CQRS Pattern

CQRS 核心概念

- Command　Query 完全分離；寫入模型 (Write Model) 不直接用於查詢
- 寫入優先強一致；讀取可為最終一致並為效能/報表優化
- 常配 Event Sourcing；需要處理同步/異步投影更新



CQRS Java —Command Side

```java
public record CreateUserCommand(String id, String name) {}

public class UserCommandHandler {
    private final UserRepository repo;
    public UserCommandHandler(UserRepository repo) { this.repo = repo; }
    public void handle(CreateUserCommand cmd) {
        repo.save(new User(cmd.id(), cmd.name()));
    }
}
```

CQRS Java —Query Side

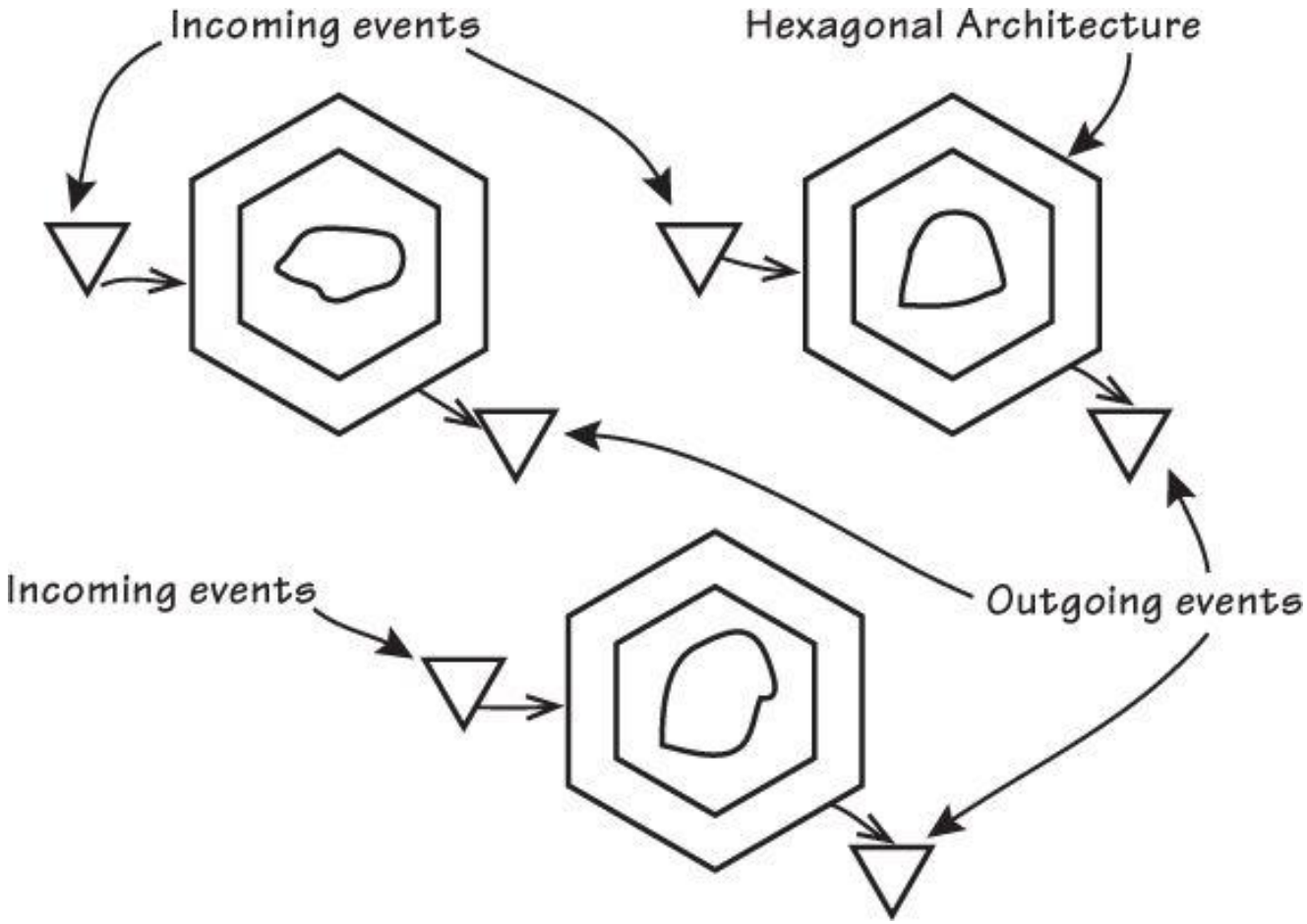
```java
public class UserProjection {
    @EventListener
    public void on(UserCreated e) {
        // 更新 Read Model (denormalized DB)
    }
}

public class UserQueryService {
    private final UserReadRepo repo;
    public UserQueryService(UserReadRepo r) { this.repo = r; }
    public UserDTO fetch(String id) { return repo.find(id); }
}
```

Event-Driven Architecture

事件驅動架構

- 服務間透過事件進行非同步通訊
- 鬆耦合、高擴展性、最終一致性
- 可結合事件溯源（Event Sourcing）模式

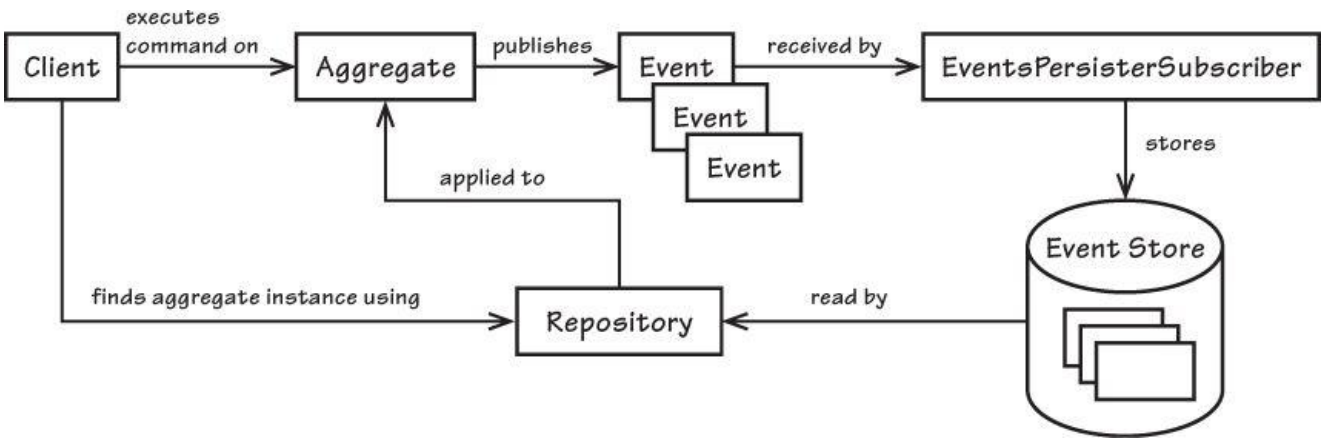


Saga 編排範例

```
class ShippingSaga {
    @SagaEventHandler
    void on(OrderCreated e) {
        send(new ReserveInventory(e.id()));
    }
    @SagaEventHandler
    void on(InventoryReserved e) {
        send(new ArrangeShipment(e.orderId()));
    }
    @SagaEventHandler
    void on(ShipmentArranged e) {
        send(new MarkOrderShipped(e.orderId()));
        end();
    }
}
```

事件溯源模式

- 將 Aggregate 的每次狀態變更以事件持久化 (Append-Only)
- 可重播事件重建任何時間點狀態，支援審計與時間旅行
- 常與 CQRS / Saga 搭配使用



Event Sourcing Java 範例

```java
public interface DomainEvent { Instant occurredAt(); }

public record OrderCreated(String id, Instant occurredAt) implements DomainEvent
      {}

public class OrderAggregate {
    private String id;
    private OrderStatus status;

    public static OrderAggregate reconstitute(List<DomainEvent> history) {
        OrderAggregate agg = new OrderAggregate();
        history.forEach(agg::apply);
        return agg;
    }
    private void apply(DomainEvent e) {
        if (e instanceof OrderCreated oc) {
            this.id = oc.id();
            this.status = OrderStatus.CREATED;
        }
    }
}
```

Data Fabric

Data Fabric 架構要點

- 統一資料平面：整合 OLTP / OLAP / Streams / Cache
- Smart Cache、Federated Query、Consistency Policy、Observability
- 與 DDD 聚合分片結合，確保資料局部性與效能

Hazelcast 快取範例

```
Config cfg = new Config();
HazelcastInstance hz = Hazelcast.newHazelcastInstance(cfg);
IMap<String, OrderSummary> orders = hz.getMap("orders");

OrderSummary summary = new OrderSummary("ID-123", 1023, Instant.now());
orders.set(summary.id(), summary, 30, TimeUnit.MINUTES);

Collection<OrderSummary> highValue = orders.values(
    Predicates.greaterThan("total", 1000));
```

Apache Ignite 分散運算範例

```
Ignition.start();
Ignite ignite = Ignition.ignite();
IgniteCache<Integer, Tick> cache = ignite.getOrCreateCache("ticks");

IgniteCallable<Double> task = () -> {
    List<List<?>> rows = cache.query(
      new SqlFieldsQuery("SELECT price FROM Tick WHERE pid = ?")
        .setArgs(portfolioId)).getAll();
    List<Double> prices = rows.stream()
        .map(r -> (Double) r.get(0))
        .sorted()
        .collect(Collectors.toList());
    return prices.get((int)(prices.size() * 0.05)); // Value-at-Risk
};

Double var = ignite.compute().call(task);
```

Data Fabric 風險與治理建議

- **記憶體壓力**：熱資料量估錯 $\rightarrow$ OOM，建議 TTL + LRU Eviction
- **Schema 演進**：需有 Registry + 相容性驗證
- **Split-Brain**：多區部署需啟用 CP 模式或資料同步機制
- **安全**：敏感資料須加密（靜態/傳輸/使用中）與審計紀錄

結語

結語：架構即演進

- 沒有銀彈架構，DDD 幫助你因應變化、協作清晰
- 每種架構風格皆服務於 Domain 模型的演進
- 避免技術導向的「錯配式設計」；堅持語言一致性與 Context 純度
- 架構設計的重點不是「選哪一種」，而是「如何隨著需求演進」

謝謝收看！

Slides by