Make Your Domain Keep Its Promises!

演講者：Frank

2025/08/26

國立台北科技大學　資訊工程系
指導教授：鄭有進、謝金雲

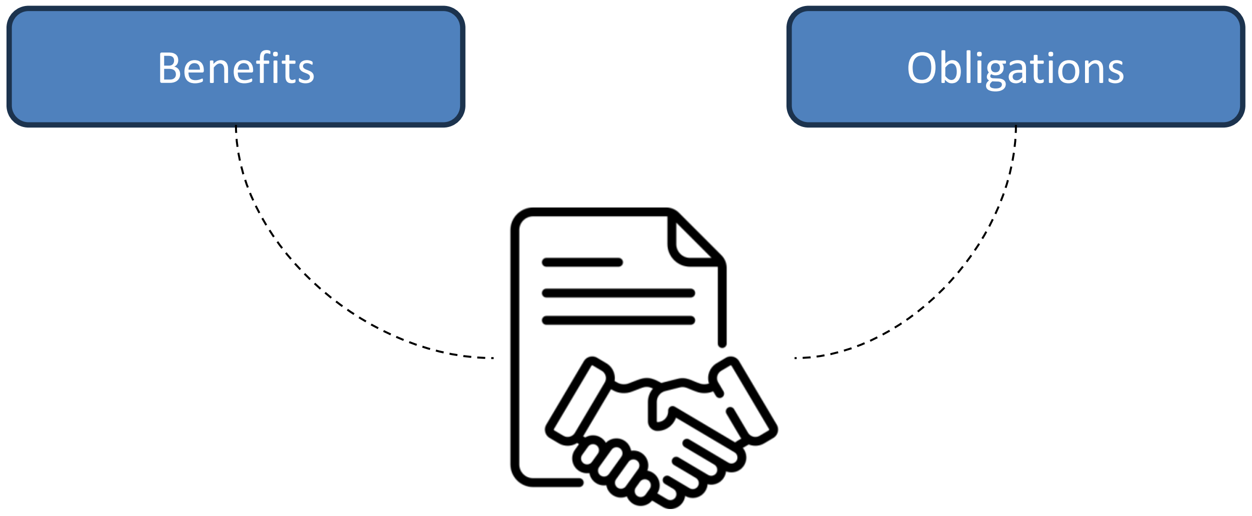
SOFTWARE SYSTEMS LAB
軟體系統實驗室 NTUT

Outline

- Design by contract
- The Three Pillars: Preconditions, Postconditions, Invariants
- Meet uContract: DBC for Java
- Case Study: ezKanban Example
- DBC and OOP
- Conclusion

Design by contract

Design by Contract, is a software correctness methodology in which software components interact through explicit agreements that define their responsibilities and guarantees.



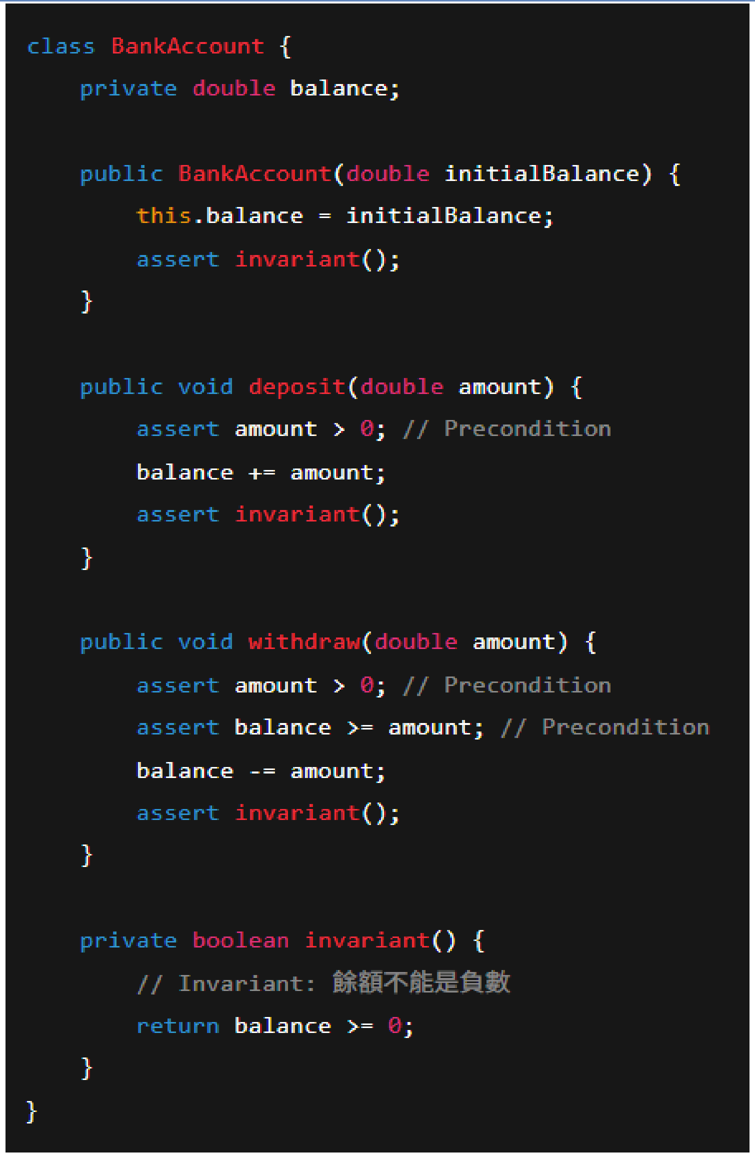
Bertrand Meyer

Design by contract

Party	Obligations	Benefits
Client	Provide letter or package of no more than 5 kgs. each dimension no more than 2 meters. Pay 200 NTD.	Get package delivered to recipient in 5 hours or less.
Supplier	Deliver package to recipient in 5 hours or less.	No need to deal with deliveries too big, too heavy, or unpaid.

The Three Pillars

Preconditions
Conditions that must be true **before** a method is executed.
Postconditions
Conditions that are guaranteed to be true **after** the method finishes.
Invariants
Conditions that must **always hold true** during the lifetime of an object.



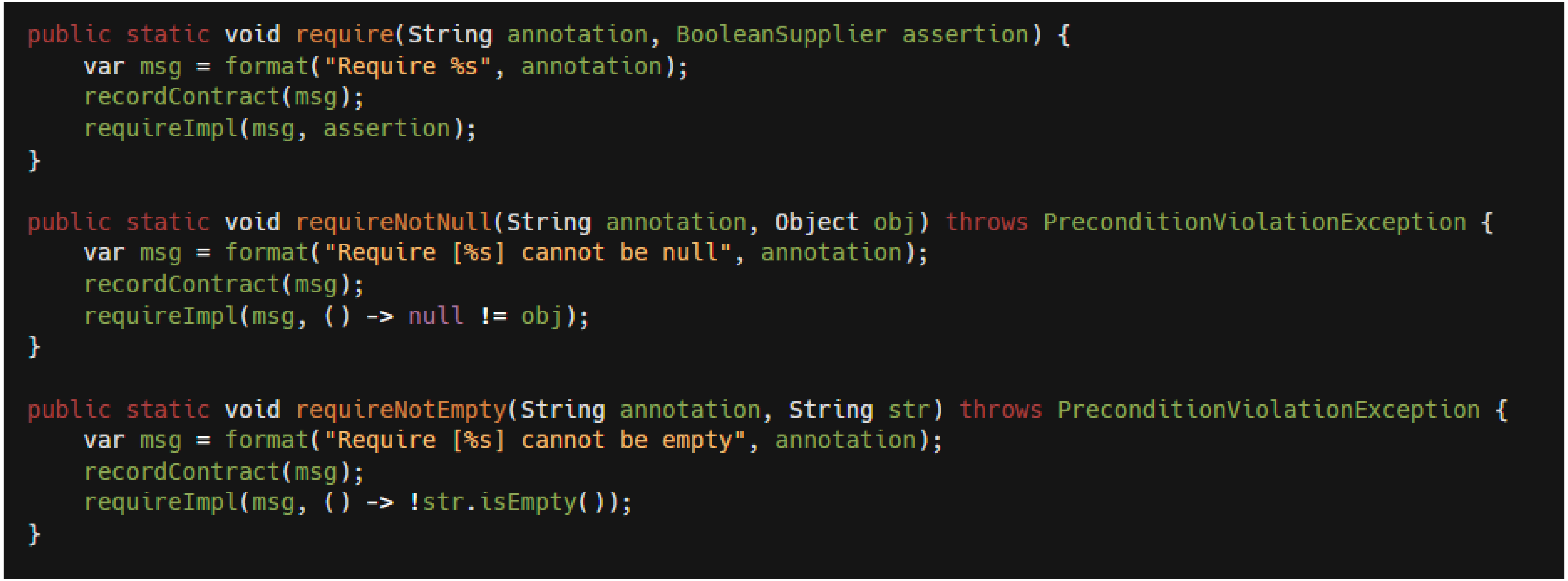
```java
class BankAccount {
    private double balance;

    public BankAccount(double initialBalance) {
        this.balance = initialBalance;
        assert invariant();
    }

    public void deposit(double amount) {
        assert amount > 0; // Precondition
        balance += amount;
        assert invariant();
    }

    public void withdraw(double amount) {
        assert amount > 0; // Precondition
        assert balance >= amount; // Precondition
        balance -= amount;
        assert invariant();
    }

    private boolean invariant() {
        // Invariant: 餘額不能是負數
        return balance >= 0;
    }
}
```

Meet uContract

uContract is a **Design By Contract (DBC)** tool implemented in Java. It is designed for writing contracts for event sourced aggregate roots in Domain-Driven Design (DDD). For implementing event sourced aggregate roots.

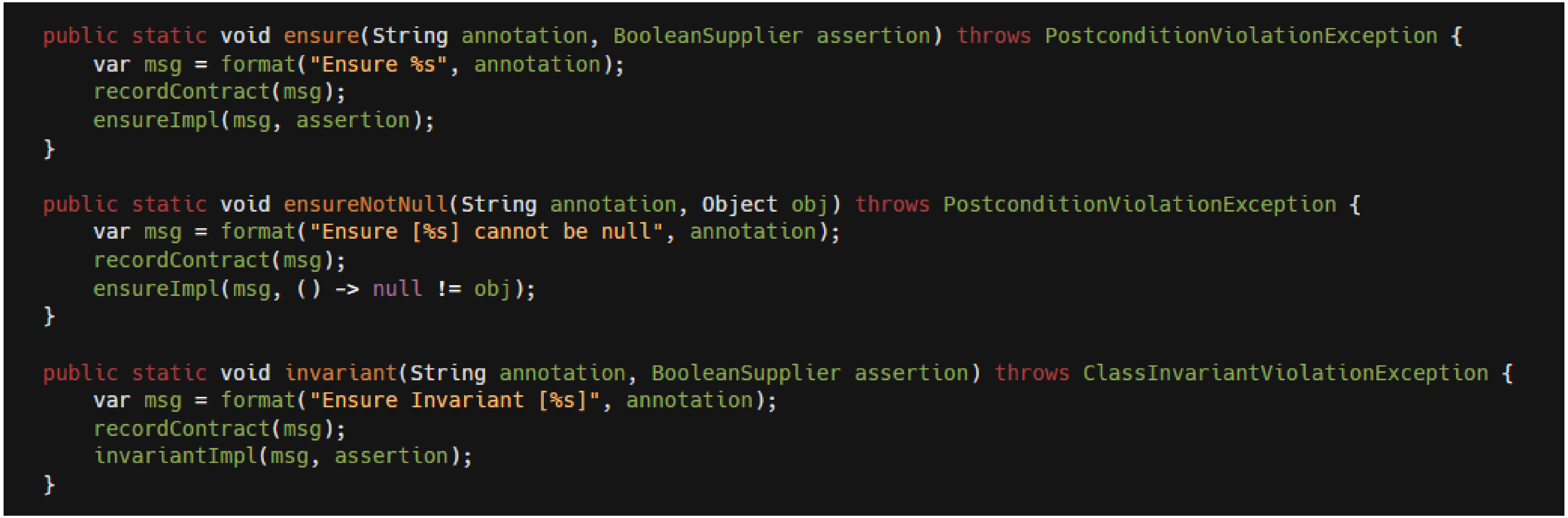
The preconditions are written before applying domain event, and the postconditions are written after applying domain event. The aggregate invariants are checked in the apply method of event sourced aggregate, as EsAggregateRoot.

Meet uContract



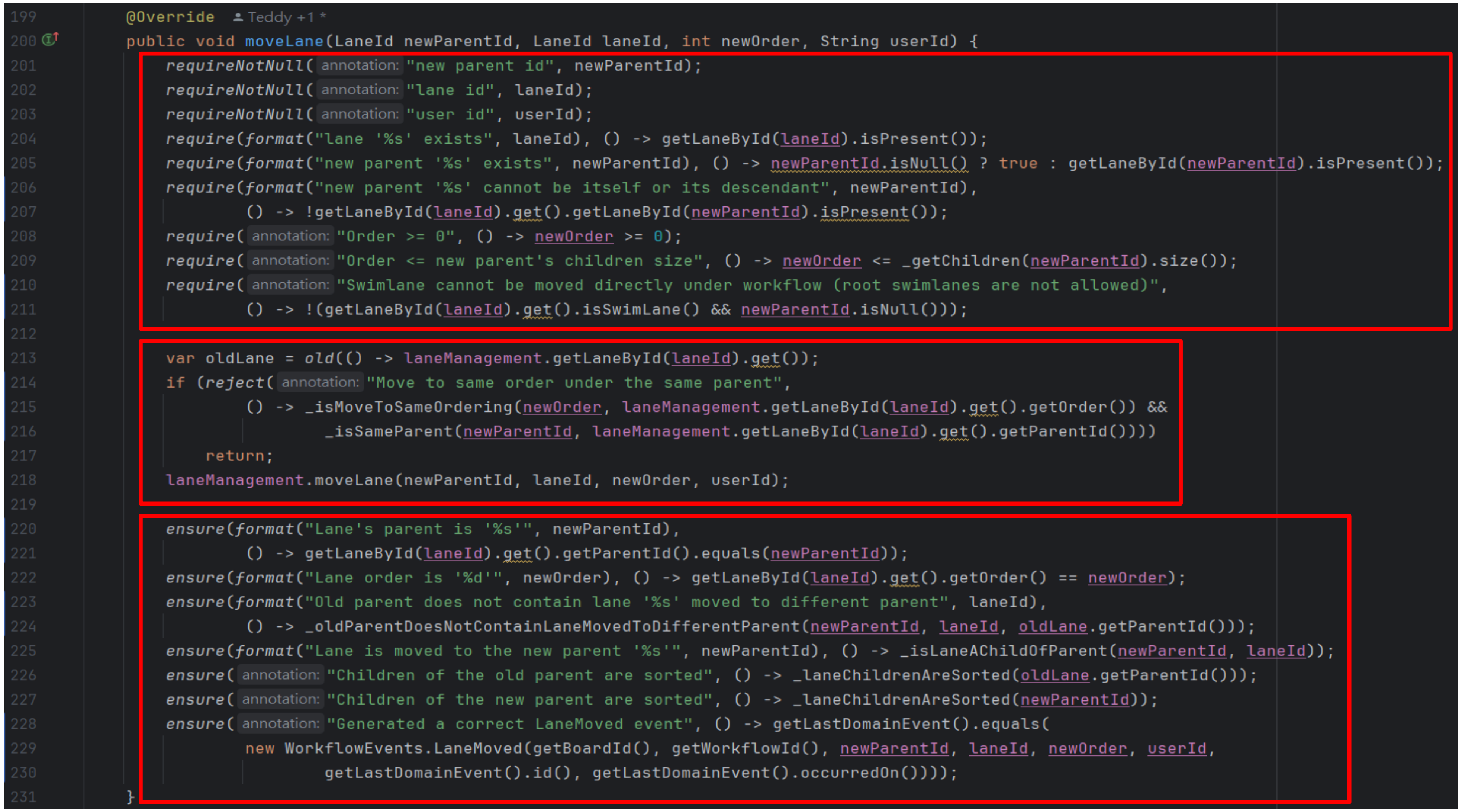
```
public static void require(String annotation, BooleanSupplier assertion) {
    var msg = format("Require %s", annotation);
    recordContract(msg);
    requireImpl(msg, assertion);
}

public static void requireNotNull(String annotation, Object obj) throws PreconditionViolationException {
    var msg = format("Require [%s] cannot be null", annotation);
    recordContract(msg);
    requireImpl(msg, () -> null != obj);
}

public static void requireNotEmpty(String annotation, String str) throws PreconditionViolationException {
    var msg = format("Require [%s] cannot be empty", annotation);
    recordContract(msg);
    requireImpl(msg, () -> !str.isEmpty());
}
```

Meet uContract



```java
public static void ensure(String annotation, BooleanSupplier assertion) throws PostconditionViolationException {
    var msg = format("Ensure %s", annotation);
    recordContract(msg);
    ensureImpl(msg, assertion);
}

public static void ensureNotNull(String annotation, Object obj) throws PostconditionViolationException {
    var msg = format("Ensure [%s] cannot be null", annotation);
    recordContract(msg);
    ensureImpl(msg, () -> null != obj);
}

public static void invariant(String annotation, BooleanSupplier assertion) throws ClassInvariantViolationException {
    var msg = format("Ensure Invariant [%s]", annotation);
    recordContract(msg);
    invariantImpl(msg, assertion);
}
```

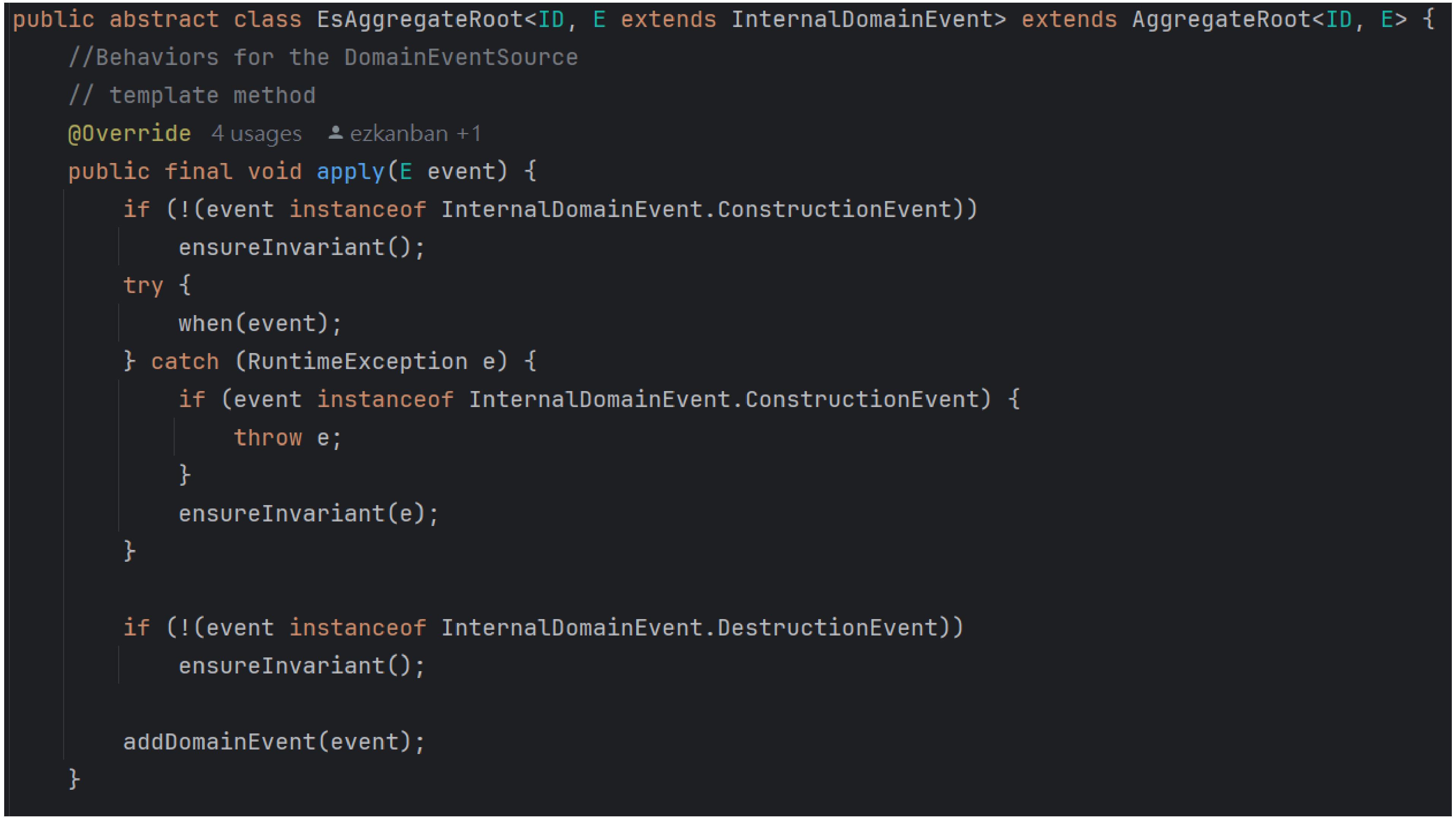
Case Study



```
@Override  Teddy +1 *
public void moveLane(LaneId newParentId, LaneId laneId, int newOrder, String userId) {
    requireNotNull( annotation: "new parent id", newParentId);
    requireNotNull( annotation: "lane id", laneId);
    requireNotNull( annotation: "user id", userId);
    require(format("lane '%s' exists", laneId), () -> getLaneById(laneId).isPresent());
    require(format("new parent '%s' exists", newParentId), () -> newParentId.isNull() ? true : getLaneById(newParentId).isPresent());
    require(format("new parent '%s' cannot be itself or its descendant", newParentId),
            () -> !getLaneById(laneId).get().getLaneById(newParentId).isPresent());
    require( annotation: "Order >= 0", () -> newOrder >= 0);
    require( annotation: "Order <= new parent's children size", () -> newOrder <= _getChildren(newParentId).size());
    require( annotation: "Swimlane cannot be moved directly under workflow (root swimlanes are not allowed)",
            () -> !(getLaneById(laneId).get().isSwimLane() && newParentId.isNull()));

    var oldLane = old(() -> laneManagement.getLaneById(laneId).get());
    if (reject( annotation: "Move to same order under the same parent",
            () -> _isMoveToSameOrdering(newOrder, laneManagement.getLaneById(laneId).get().getOrder()) &&
                    _isSameParent(newParentId, laneManagement.getLaneById(laneId).get().getParentId())))
        return;
    laneManagement.moveLane(newParentId, laneId, newOrder, userId);

    ensure(format("Lane's parent is '%s'", newParentId),
            () -> getLaneById(laneId).get().getParentId().equals(newParentId));
    ensure(format("Lane order is '%d'", newOrder), () -> getLaneById(laneId).get().getOrder() == newOrder);
    ensure(format("Old parent does not contain lane '%s' moved to different parent", laneId),
            () -> _oldParentDoesNotContainLaneMovedToDifferentParent(newParentId, laneId, oldLane.getParentId()));
    ensure(format("Lane is moved to the new parent '%s'", newParentId), () -> _isLaneAChildOfParent(newParentId, laneId));
    ensure( annotation: "Children of the old parent are sorted", () -> _laneChildrenAreSorted(oldLane.getParentId()));
    ensure( annotation: "Children of the new parent are sorted", () -> _laneChildrenAreSorted(newParentId));
    ensure( annotation: "Generated a correct LaneMoved event", () -> getLastDomainEvent().equals(
            new WorkflowEvents.LaneMoved(getBoardId(), getWorkflowId(), newParentId, laneId, newOrder, userId,
                    getLastDomainEvent().id(), getLastDomainEvent().occurredOn())));
}
```

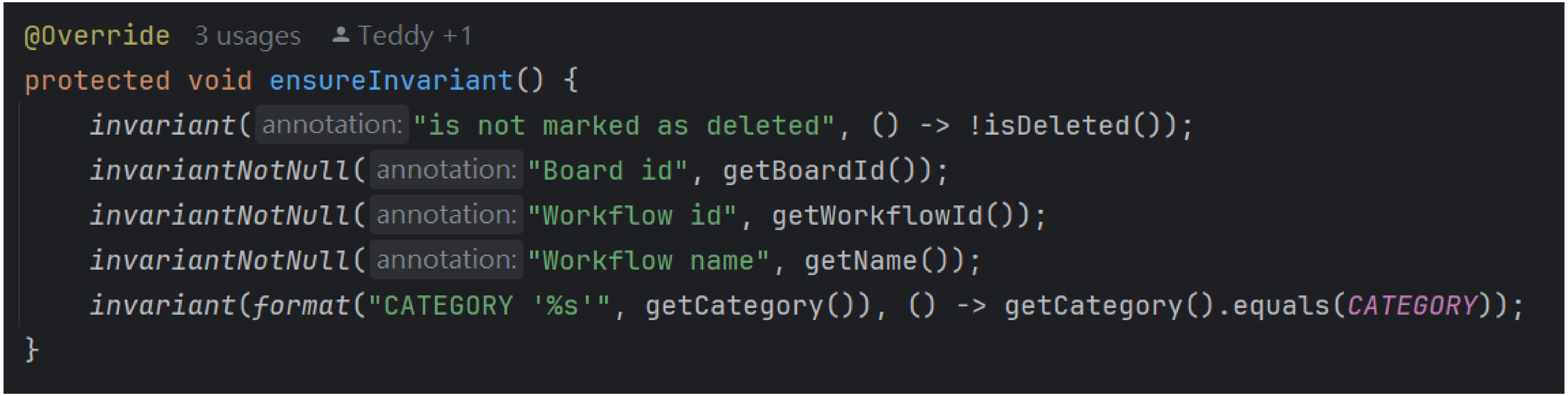
2025/8/29

Case Study



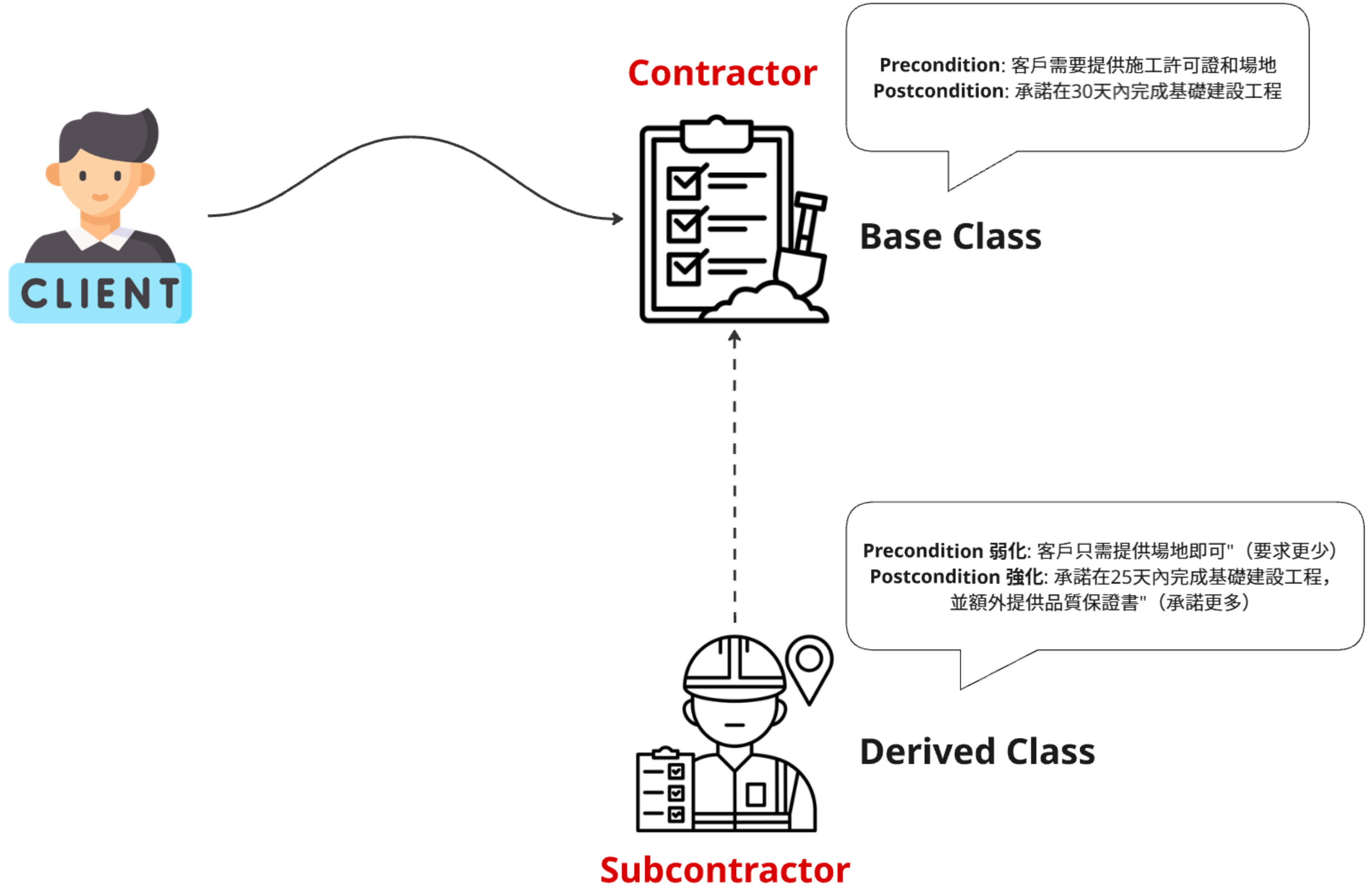
```
public abstract class EsAggregateRoot<ID, E extends InternalDomainEvent> extends AggregateRoot<ID, E> {
    //Behaviors for the DomainEventSource
    // template method
    @Override  4 usages   ezkanban +1
    public final void apply(E event) {
        if (!(event instanceof InternalDomainEvent.ConstructionEvent))
            ensureInvariant();
        try {
            when(event);
        } catch (RuntimeException e) {
            if (event instanceof InternalDomainEvent.ConstructionEvent) {
                throw e;
            }
            ensureInvariant(e);
        }

        if (!(event instanceof InternalDomainEvent.DestructionEvent))
            ensureInvariant();

        addDomainEvent(event);
    }
```

Case Study



```
@Override  3 usages  Teddy +1
protected void ensureInvariant() {
    invariant( annotation: "is not marked as deleted", () -> !isDeleted());
    invariantNotNull( annotation: "Board id", getBoardId());
    invariantNotNull( annotation: "Workflow id", getWorkflowId());
    invariantNotNull( annotation: "Workflow name", getName());
    invariant(format("CATEGORY '%s'", getCategory()), () -> getCategory().equals(CATEGORY));
}
```

DBC and OOP



Precondition: 客戶需要提供施工許可證和場地
Postcondition: 承諾在30天內完成基礎建設工程

Precondition 弱化: 客戶只需提供場地即可"（要求更少）
Postcondition 強化: 承諾在25天內完成基礎建設工程，並額外提供品質保證書"（承諾更多）

Conclusion

核心理念：軟體元件之間以「契約」互動，明確規範雙方責任與權利。

三大元素：

1.Preconditions（前置條件）：呼叫者必須滿足的條件（Caller’s obligation）。

2.Postconditions（後置條件）：被呼叫者必須保證的結果（Callee’s obligation）。

3.Invariants（不變項）：類別在穩定狀態下必須始終成立的條件。

特性：

- Reliability：避免模糊責任歸屬，快速定位錯誤。
- Maintainability：減少不必要的防禦性程式碼。
- 支援OOP一致性：保證繼承、多型與動態綁定下的行為相容性。

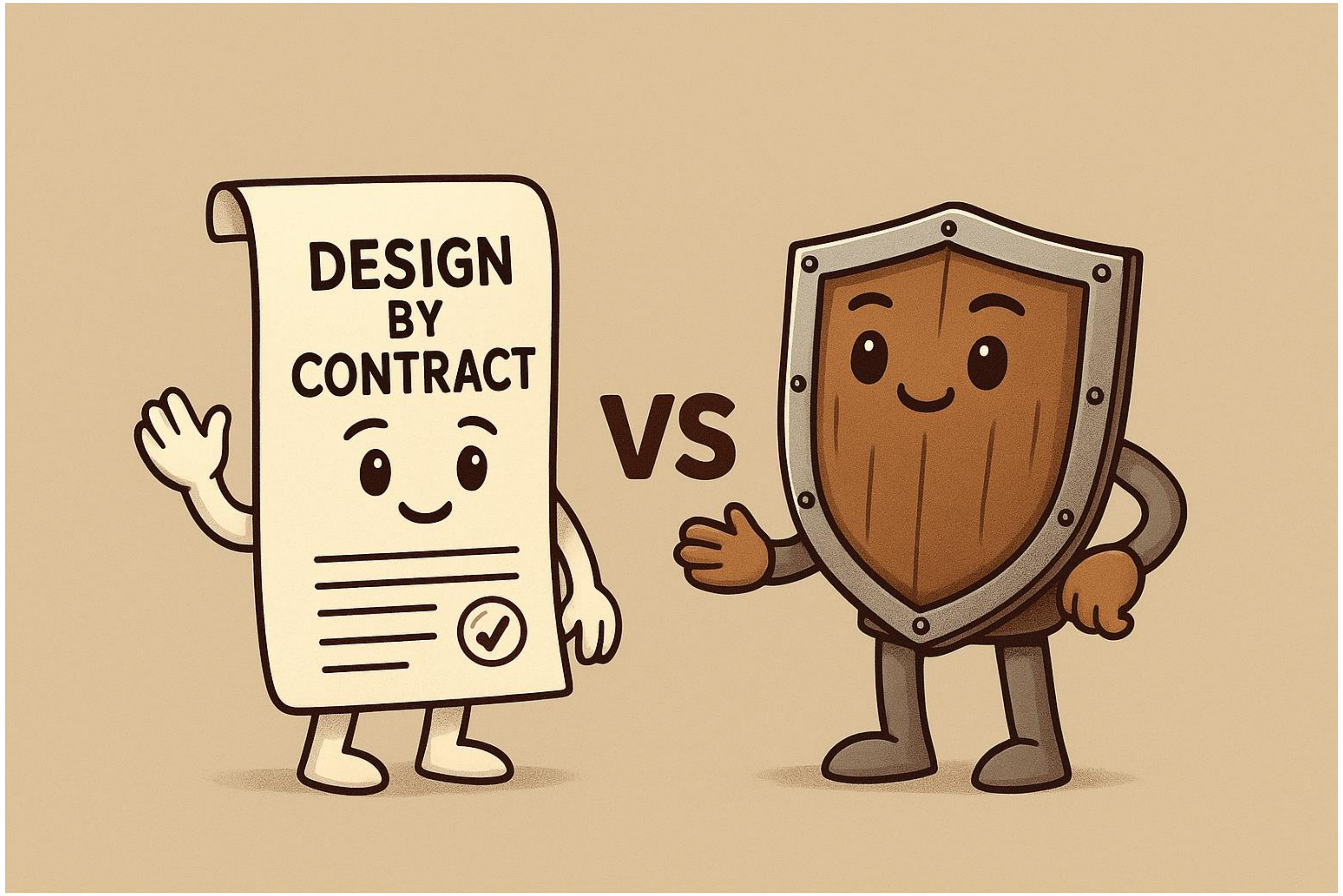
Reference

- Maven Central: tw.teddysoft.ucontract:uContract:1.0.2
- 搞笑談軟工:用合約彌補測試案例的不足（上）：撰寫合約
- 搞笑談軟工:用合約彌補測試案例的不足（下）：透過合約發現系統行為缺陷
- 事件溯源（8）：什麼是CQRS？
- 契約式設計 (DBC Design By Contract) vs 防禦式程式設計(Defensive Programming)
- Meyer, B. (1992). *Applying "Design by Contract"*. **Computer**, 25(10), 40–51. IEEE.

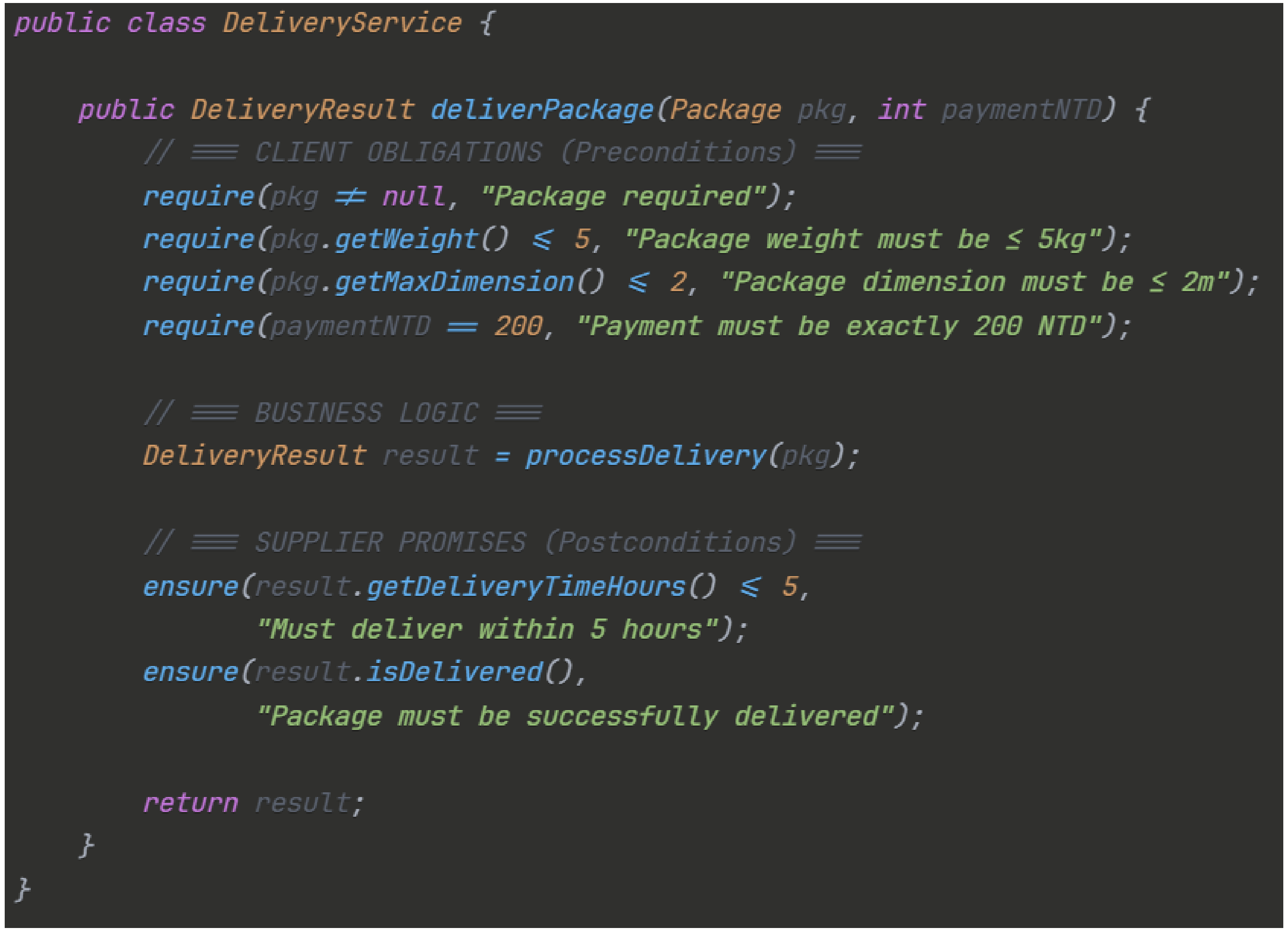
Thanks for listening

QA

Design by Contract v.s. Defensive Programming

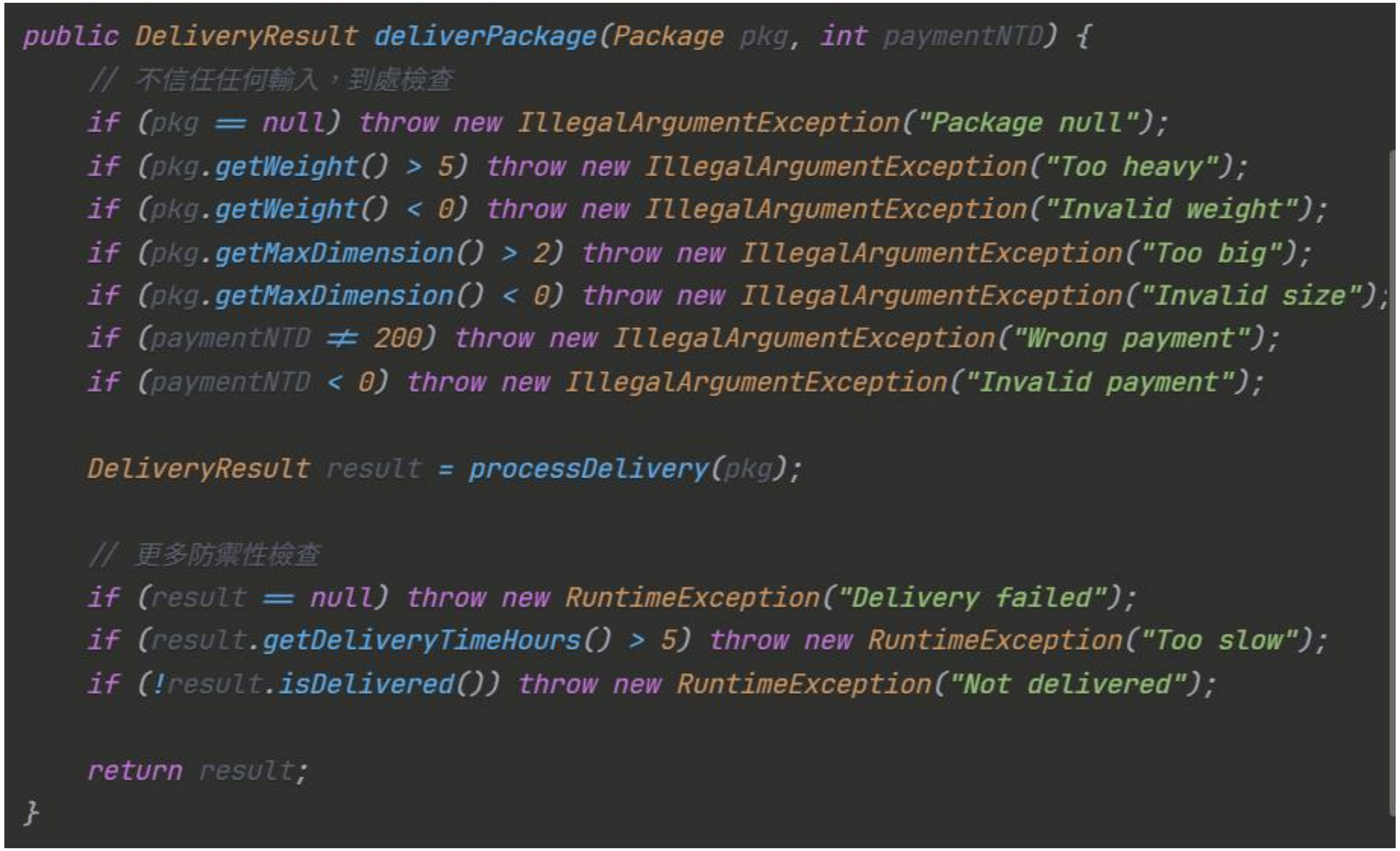


Design by Contract v.s. Defensive Programming



```java
public class DeliveryService {

    public DeliveryResult deliverPackage(Package pkg, int paymentNTD) {
        // === CLIENT OBLIGATIONS (Preconditions) ===
        require(pkg != null, "Package required");
        require(pkg.getWeight() <= 5, "Package weight must be ≤ 5kg");
        require(pkg.getMaxDimension() <= 2, "Package dimension must be ≤ 2m");
        require(paymentNTD == 200, "Payment must be exactly 200 NTD");

        // === BUSINESS LOGIC ===
        DeliveryResult result = processDelivery(pkg);

        // === SUPPLIER PROMISES (Postconditions) ===
        ensure(result.getDeliveryTimeHours() <= 5,
               "Must deliver within 5 hours");
        ensure(result.isDelivered(),
               "Package must be successfully delivered");

        return result;
    }
}
```

Design by Contract v.s. Defensive Programming



```java
public DeliveryResult deliverPackage(Package pkg, int paymentNTD) {
    // 不信任任何輸入，到處檢查
    if (pkg == null) throw new IllegalArgumentException("Package null");
    if (pkg.getWeight() > 5) throw new IllegalArgumentException("Too heavy");
    if (pkg.getWeight() < 0) throw new IllegalArgumentException("Invalid weight");
    if (pkg.getMaxDimension() > 2) throw new IllegalArgumentException("Too big");
    if (pkg.getMaxDimension() < 0) throw new IllegalArgumentException("Invalid size");
    if (paymentNTD != 200) throw new IllegalArgumentException("Wrong payment");
    if (paymentNTD < 0) throw new IllegalArgumentException("Invalid payment");

    DeliveryResult result = processDelivery(pkg);

    // 更多防禦性檢查
    if (result == null) throw new RuntimeException("Delivery failed");
    if (result.getDeliveryTimeHours() > 5) throw new RuntimeException("Too slow");
    if (!result.isDelivered()) throw new RuntimeException("Not delivered");

    return result;
}
```

Design by Contract v.s. Defensive Programming

面向	Design by Contract	Defensive Programming
核心差異	明確責任分工，建立軟體元件間的契約	不信任任何輸入，到處保護
責任與錯誤處理	明確責任分工，違反契約精確定責	假設輸入不可信，統一防禦性例外
檢查策略	介面邊界自動檢查，與業務邏輯分離	內部到處檢查，與業務邏輯混合
開發特徵	契約優先設計，文件同步，需理解責任分工	實作時防護，邏輯散落，概念直觀簡單