SOFTWARE ENGINEERING

# Introduction to UML

**Dr. Rathachai Chawuthai**

Department of Computer Engineering
Faculty of Engineering
King Mongkut's Institute of Technology Ladkrabang

# Agenda

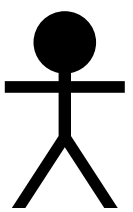
- Use Case Diagram
- Collaboration Diagram
- Sequence Diagram
- Class Diagram

# Overview

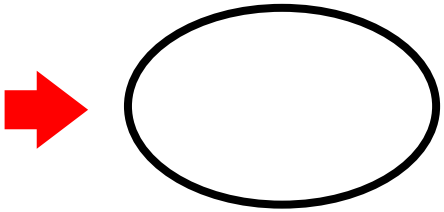
# Why?

**Ref:** • (image) http://weblog.infopraca.pl/2013/04/najlepsze-firmy-dla-inzynierow-w-polsce/

# All Diagrams

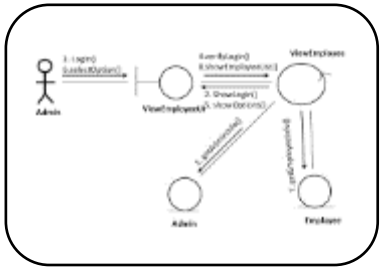


User

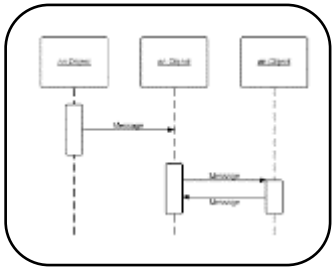


Use Case Diagram

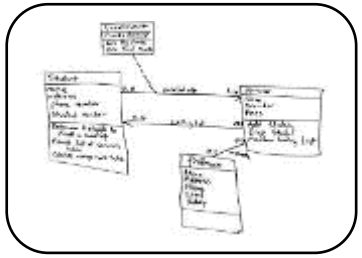
Use Case Specification



Collaboration Diagram



Sequence Diagram

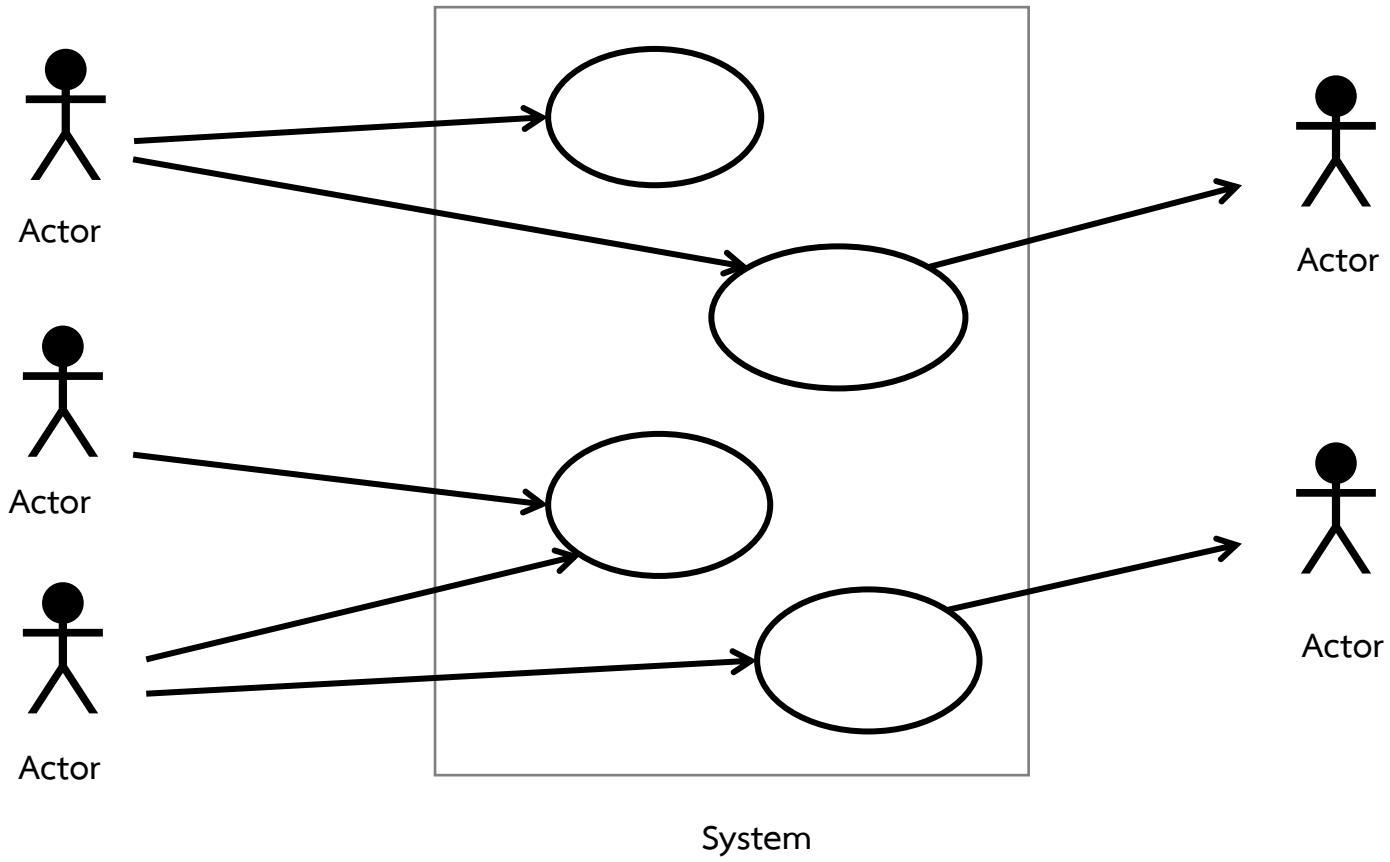


Class Diagram

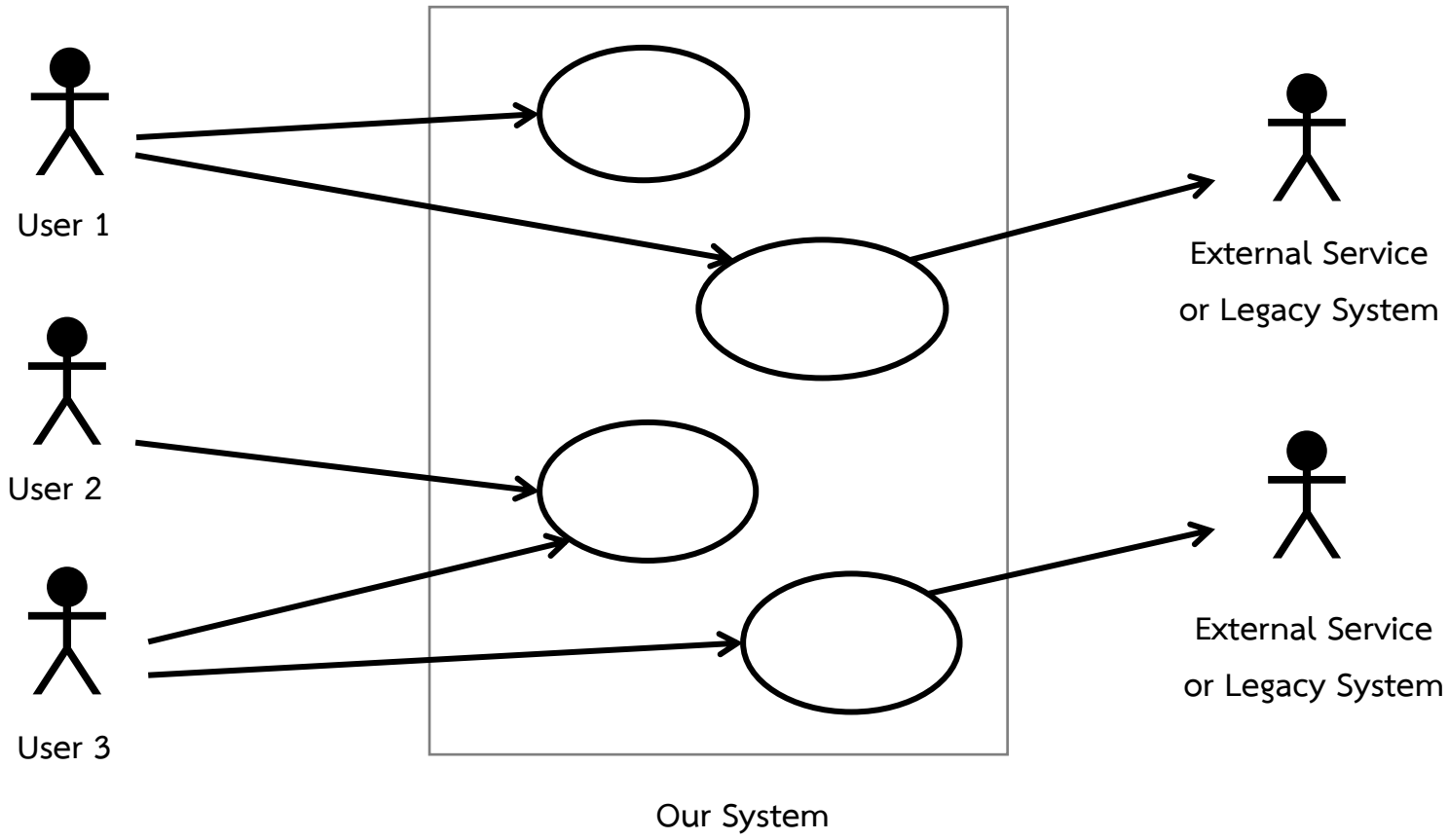
Use Case Realization

# Use Case Diagram

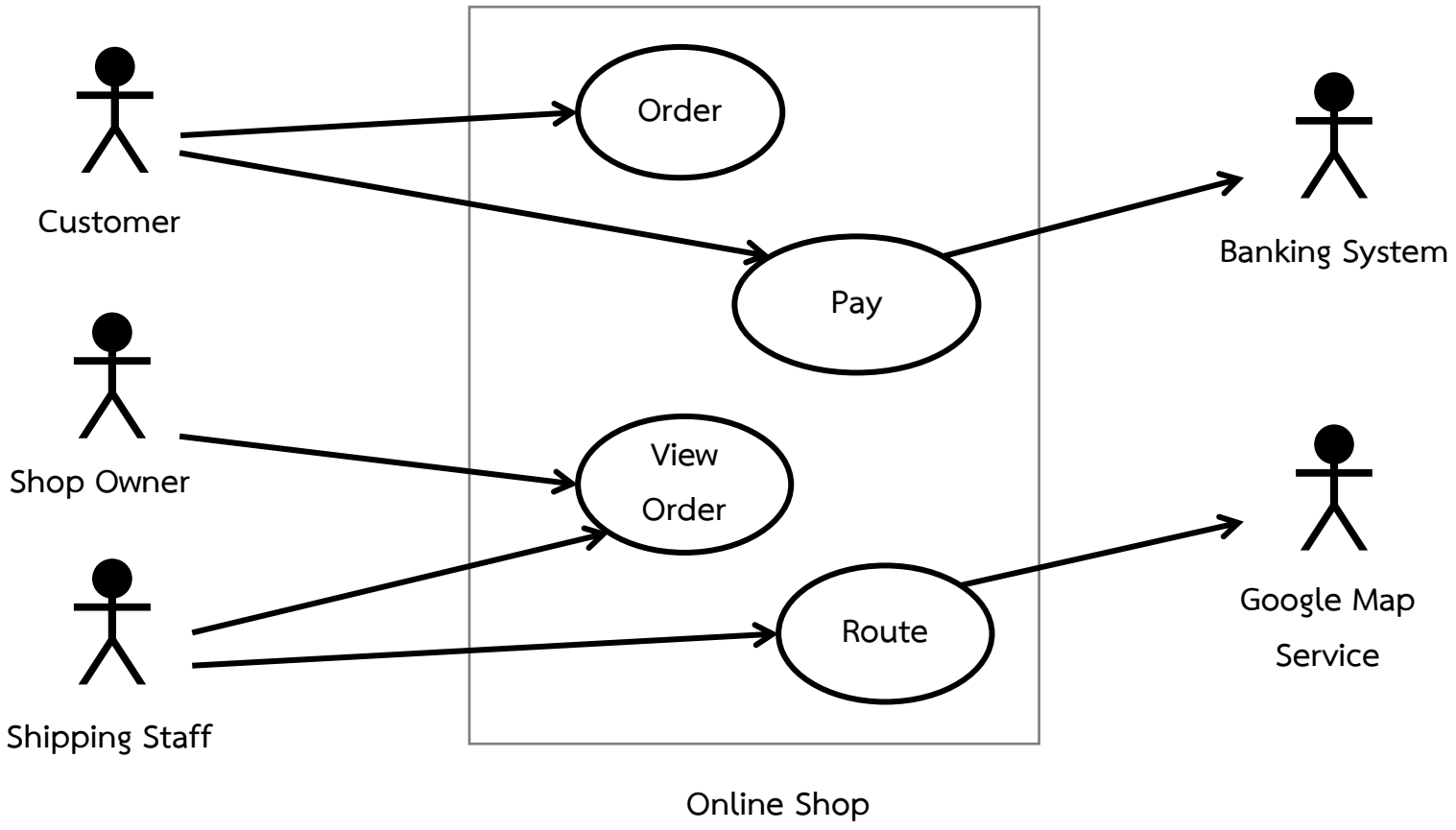
# Use Case Diagram



# Use Case Diagram



# Use Case Diagram : Online Shop

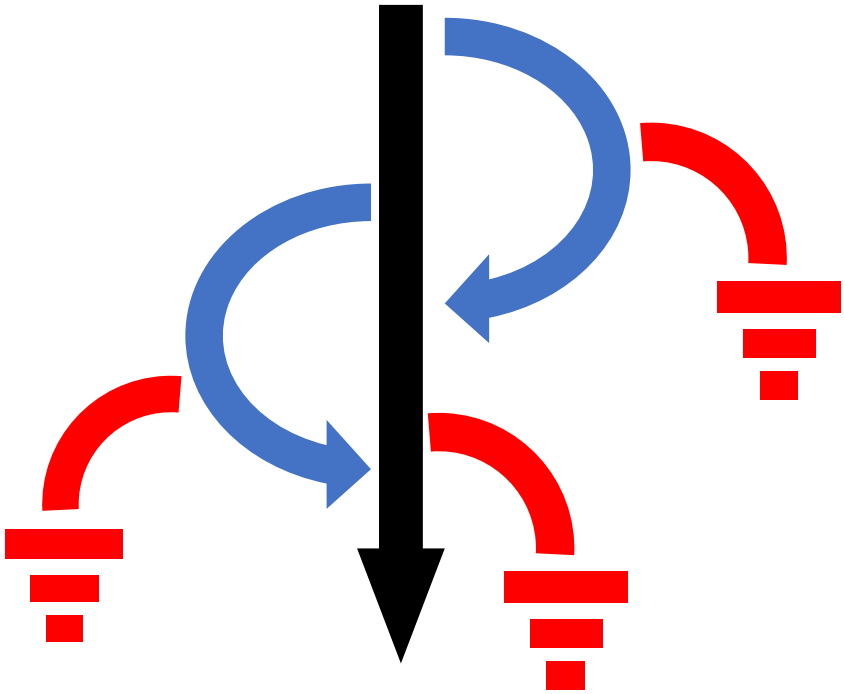


# Use-Case Flow of Events

- มี 1 Basic Flow

- มี Alternative Flow ได้หลายอัน
  - a) Regular Variance
    เส้นทางอื่นที่ทำได้

  - a) Exceptional Flows
    สำหรับ Error

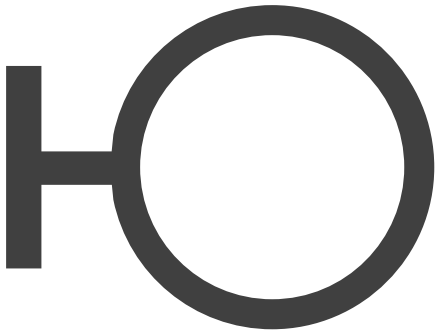


# Flow of Event : Pay

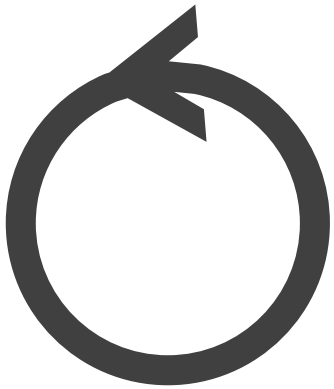
- Basic Flow
  - ลูกค้ายืนยันรายการสินค้าเพื่อจ่ายเงิน
  - Online Shop แจ้ง Banking System ว่าลูกค่า ID นี้ จะจ่ายเงินเท่านี้ สำหรับ Order ID นี้
  - ลูกค้าชำระเงินผ่าน Banking System
  - Banking System แจ้ง Online Shop ว่าการชำระเงินครบถ้วนถูกต้อง
  - ลูกค้าได้รับ Message ยืนยันว่าชำระเงินถูกต้อง
- Alternative Flows
  - ระบบ Banking ล่ม
  - ลูกค้าจ่ายเงินไม่ครบ
  - ลูกค้าปิดหน้าเว็ปการซื้อ

# Collaboration Diagram

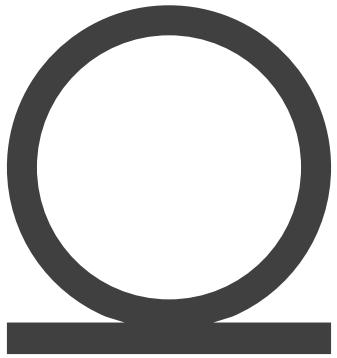
# Elements



Boundary



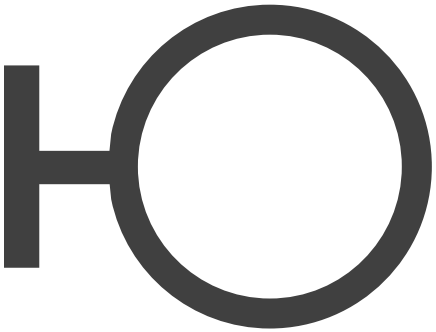
Control



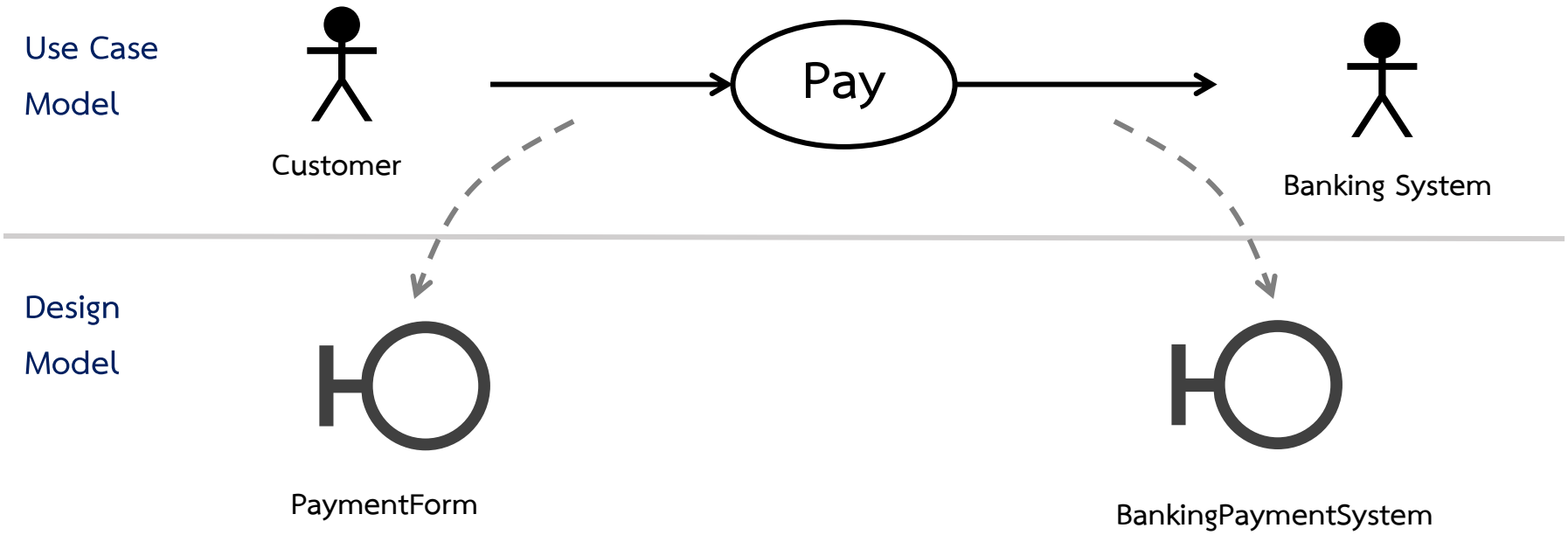
Entity

# Boundary

- เป็น Interface อยู่ระหว่างระบบที่จะพัฒนา กับ users หรือ external systems
- มีดังนี้
  - User interface classes
  - System interface classes
  - Device interface classes
- มี 1 boundary class ต่อ 1 คู่ของ actor กับ use-case

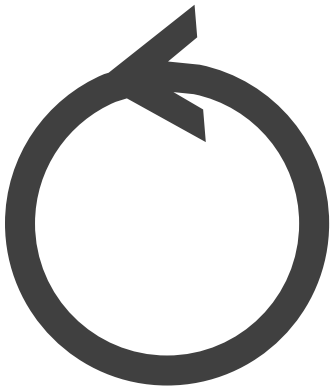


# Boundary

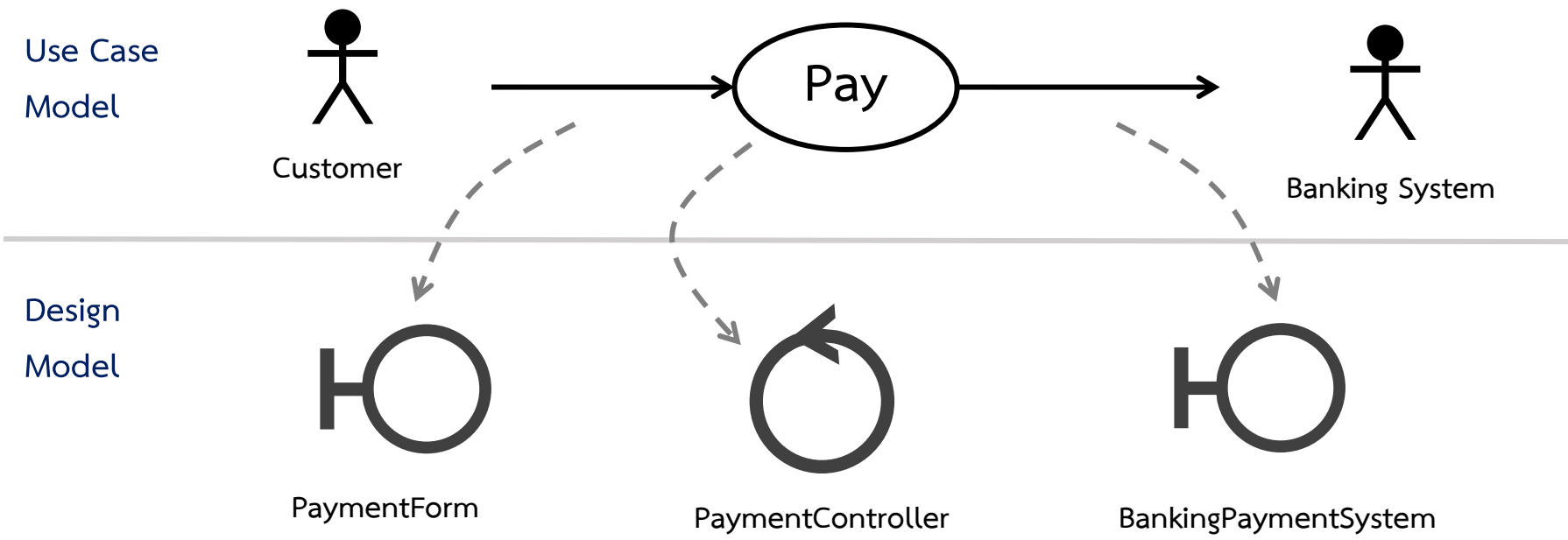

Use Case
Model
Customer
Pay
Banking System
Design
Model
PaymentForm
BankingPaymentSystem

# Control

- เป็นส่วนที่ไว้ใช้ควบคุมกระบวนการทั้งหมดของ Use Case นั้น
- ปกติจะมี 1 Control ต่อ 1 Use Case (แต่ถ้า Use Case ซับซ้อนมากจะมีมากว่า 1 Control ก็ได้)

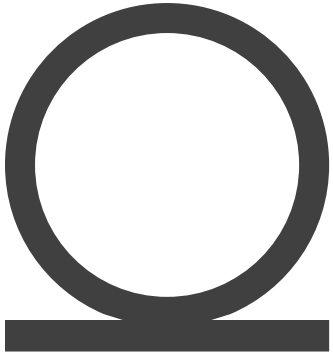


# Control


Use Case
Model
Customer
Pay
Banking System
Design
Model
PaymentForm
PaymentController
BankingPaymentSystem

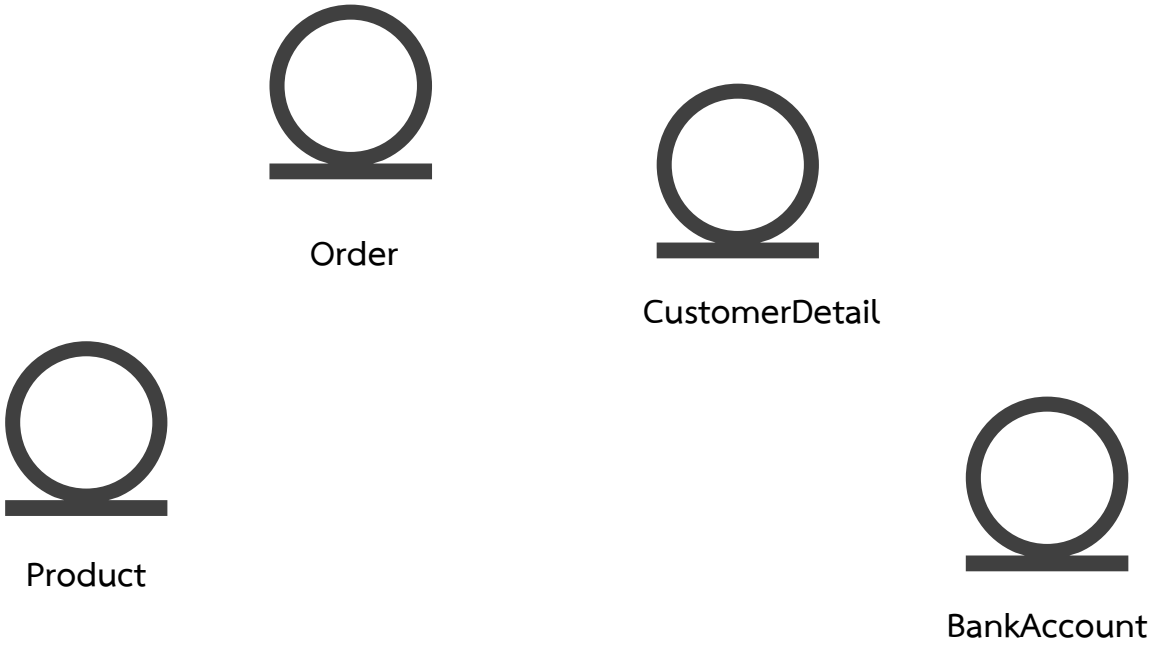
# Entity

- เป็น Object แสดงหน่วยข้อมูลต่างๆ ที่ใช้ใน Use Case นี้
- เป็น object ที่ใช้เป็นทั้ง Input, Output, และ ส่วนประกอบอื่นๆ ที่จำเป็น
- ข้อสังเกต
  - เป็นคำนาม
  - ต้องไม่ซ้ำซ้อน
  - ต้องไม่กำกวม
  - ไม่รวม actor
  - เป็น object ไม่ใช่ attributes

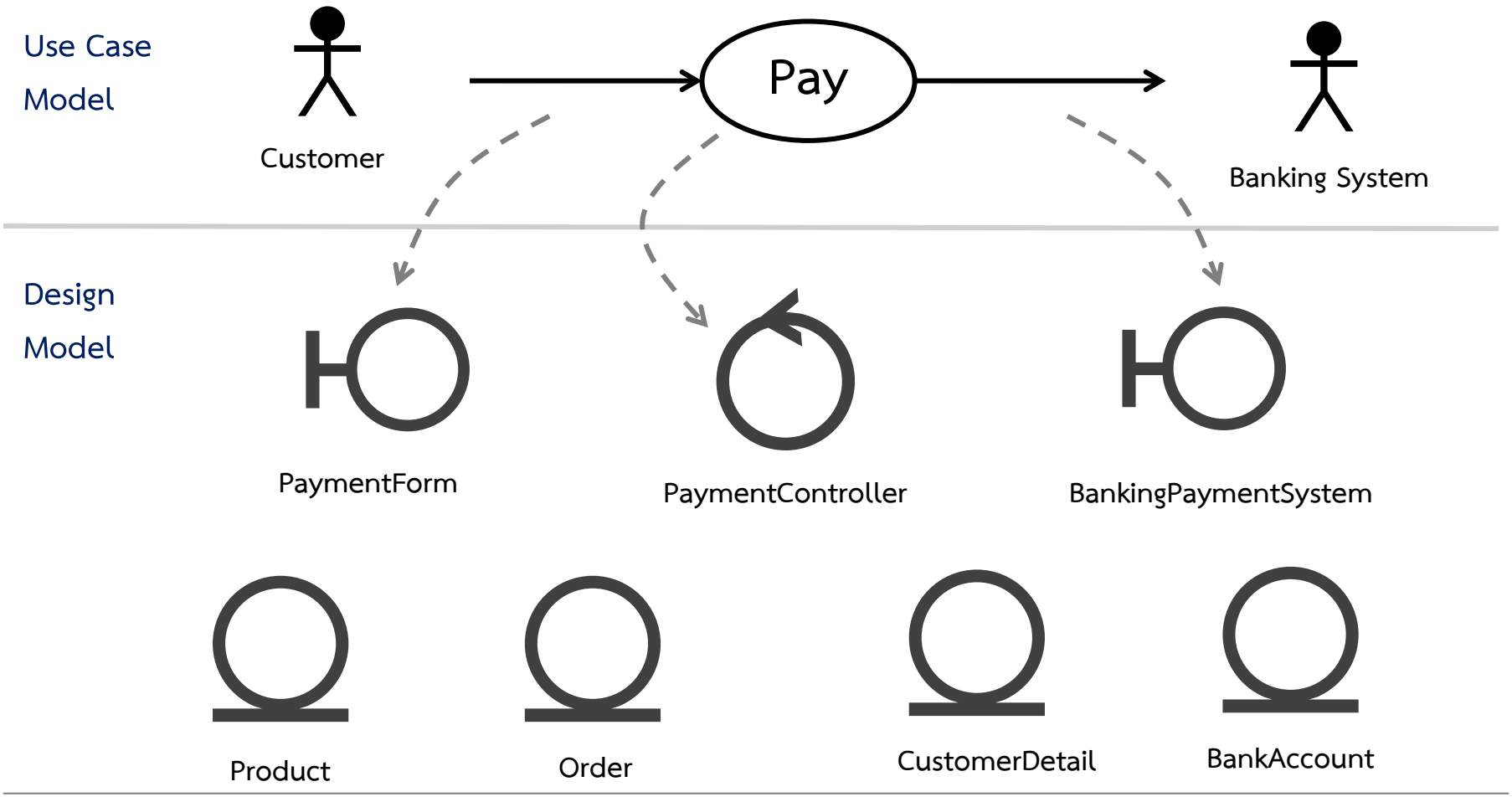


# Entity

- ตัวอย่างของ Payment



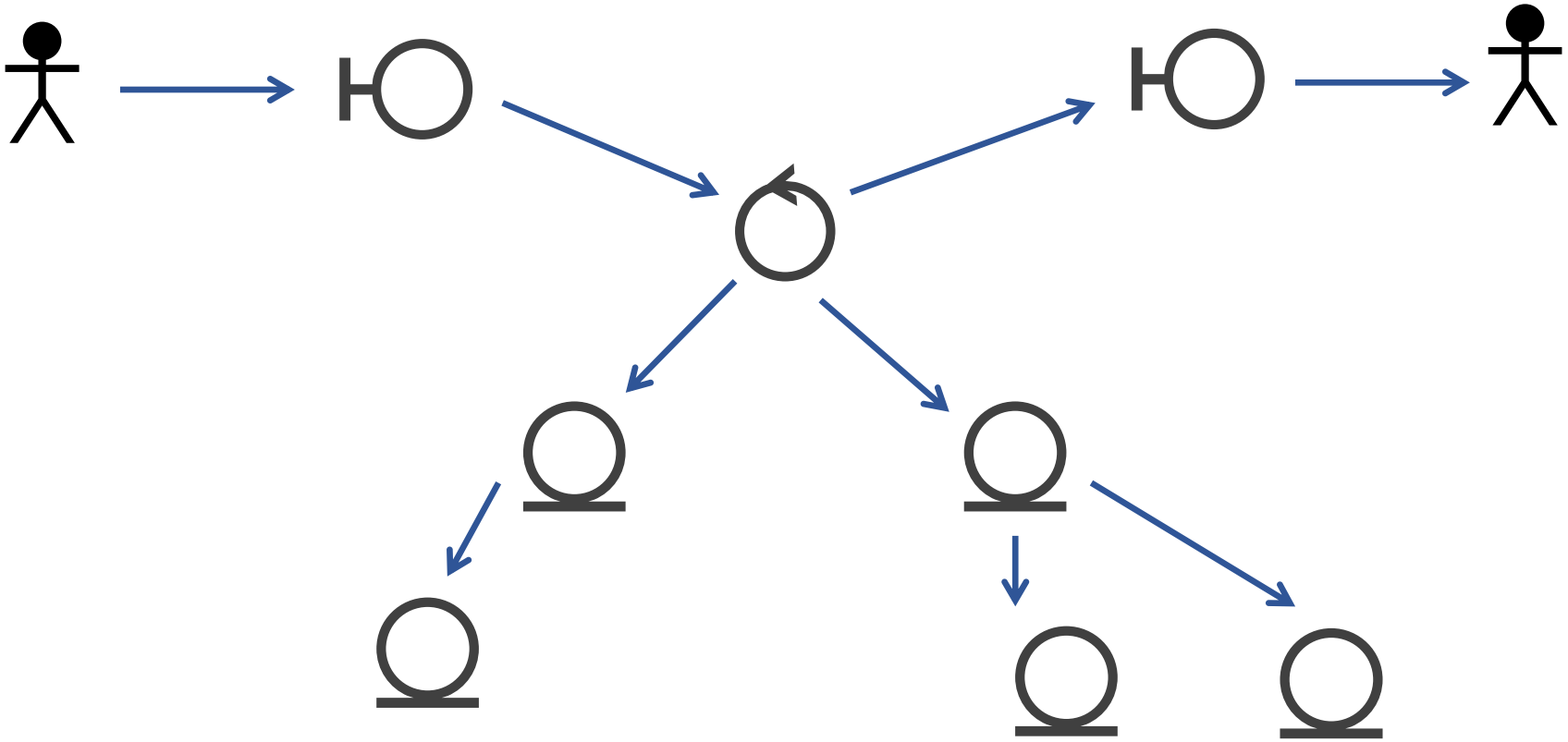
# Entity


Use Case
Model
Customer
Pay
Banking System
Design
Model
PaymentForm
PaymentController
BankingPaymentSystem
Product
Order
CustomerDetail
BankAccount

# Collaboration Diagram

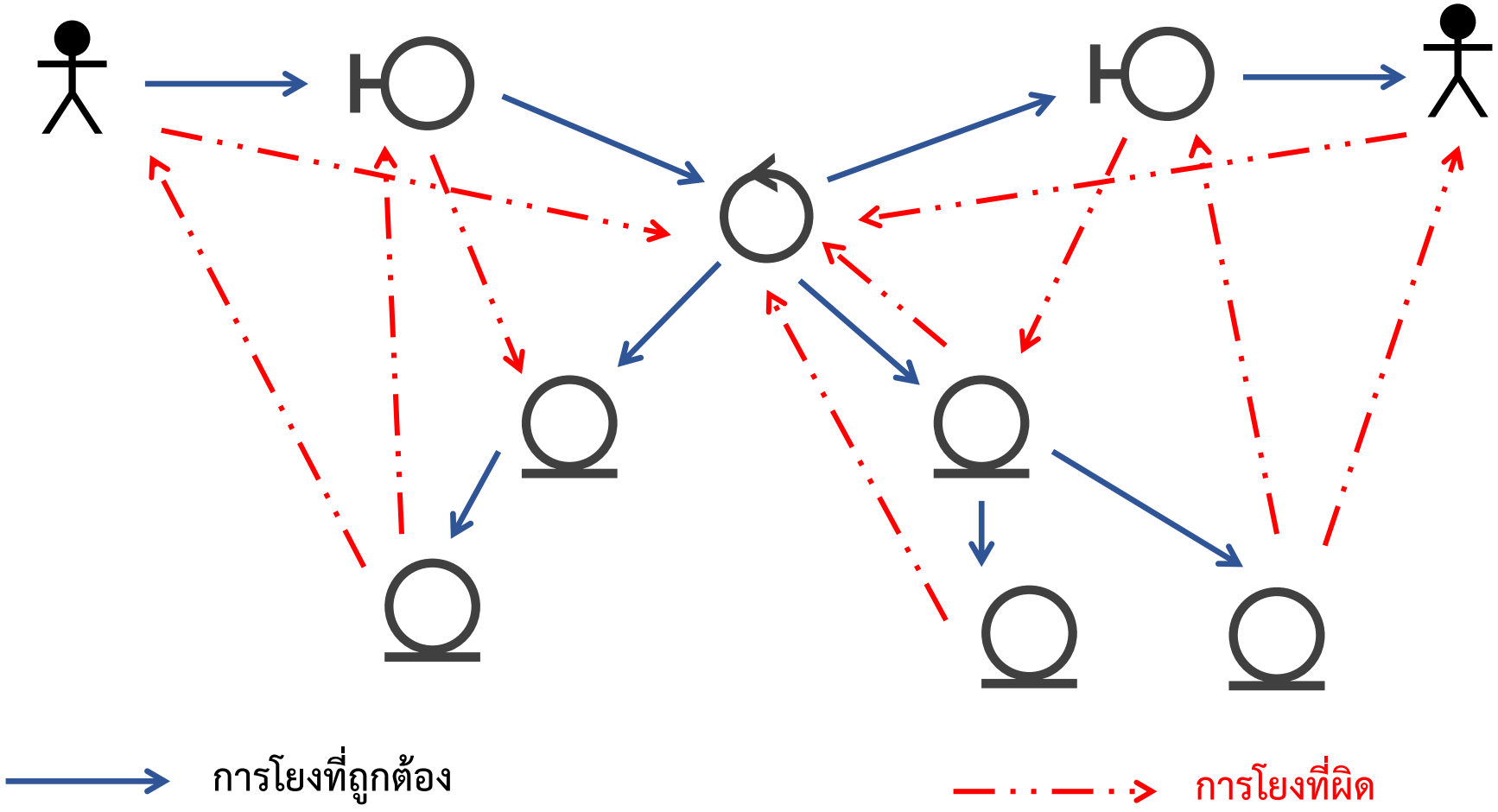
- แต่ละ element จะมีลูกศรโยงหาความสัมพันธ์กันดังนี้
  - โยงลูกศรระหว่าง actor กับ boundary เท่านั้น
  - โยงลูกศรระหว่าง boundary กับ control เท่านั้น
  - โยงลูกศระหว่าง control ไปยัง entity อย่างน้อย 1 การเชื่อมโยง
  - โยงลูกศรระหว่าง entity กับ entity ได้
  - ห้ามโยงลูกศรจาก control ไปยัง actor เด็ดขาด
  - ห้ามโยงลูกศรจาก entity ไป control หรือ boundary หรือ actor เด็ดขาด
- การเขียน action กำกับ
  - เขียนหมายเลขลำดับของ action บน (หรือบริเวณ) เส้นที่โยง
- ข้อเสนอแนะ
  - การโยงลูกศรระหว่าง entity ควรทำให้เรียบง่ายที่สุด ไม่ควรให้เกิดการโยงซับซ้อนหรือเกิด loop

# Collaboration Diagram



การโยงที่ถูกต้อง

# Collaboration Diagram



# Collaboration Diagram: Payment

Customer

PaymentForm

Payment Controller

BankingPayment System

Banking System

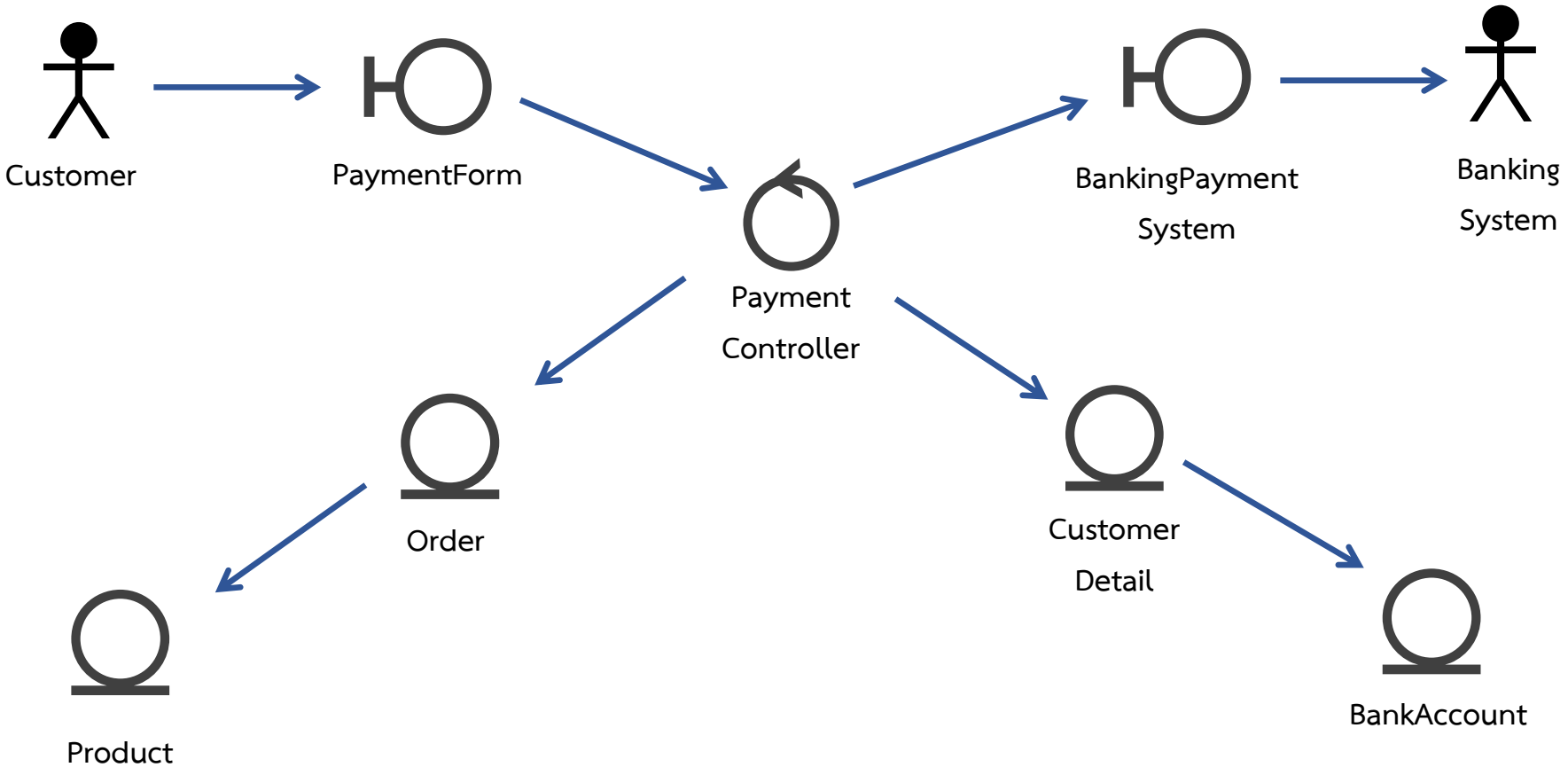
Order

Customer Detail

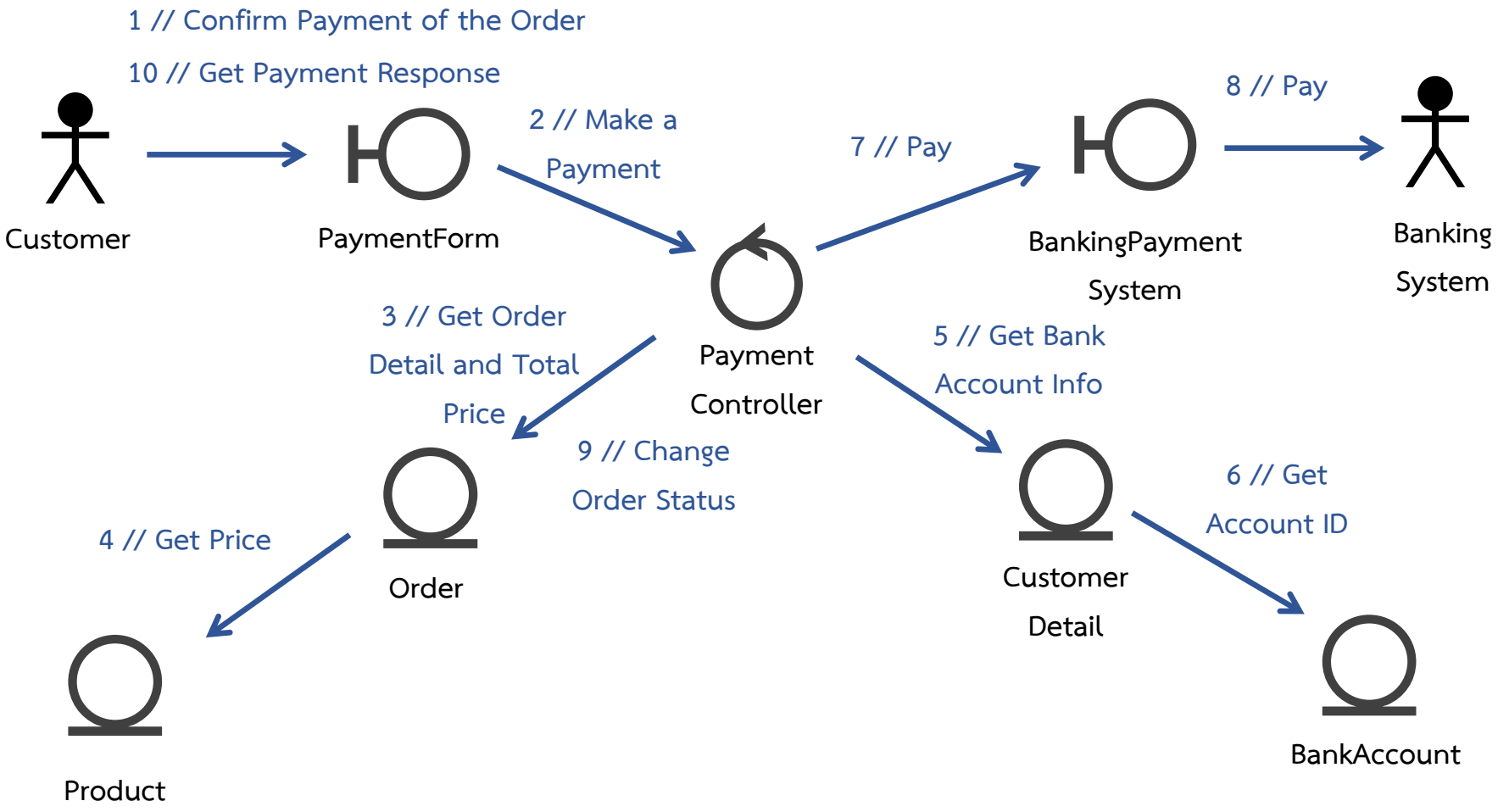
Product

BankAccount

# Collaboration Diagram: Payment



# Collaboration Diagram: Payment

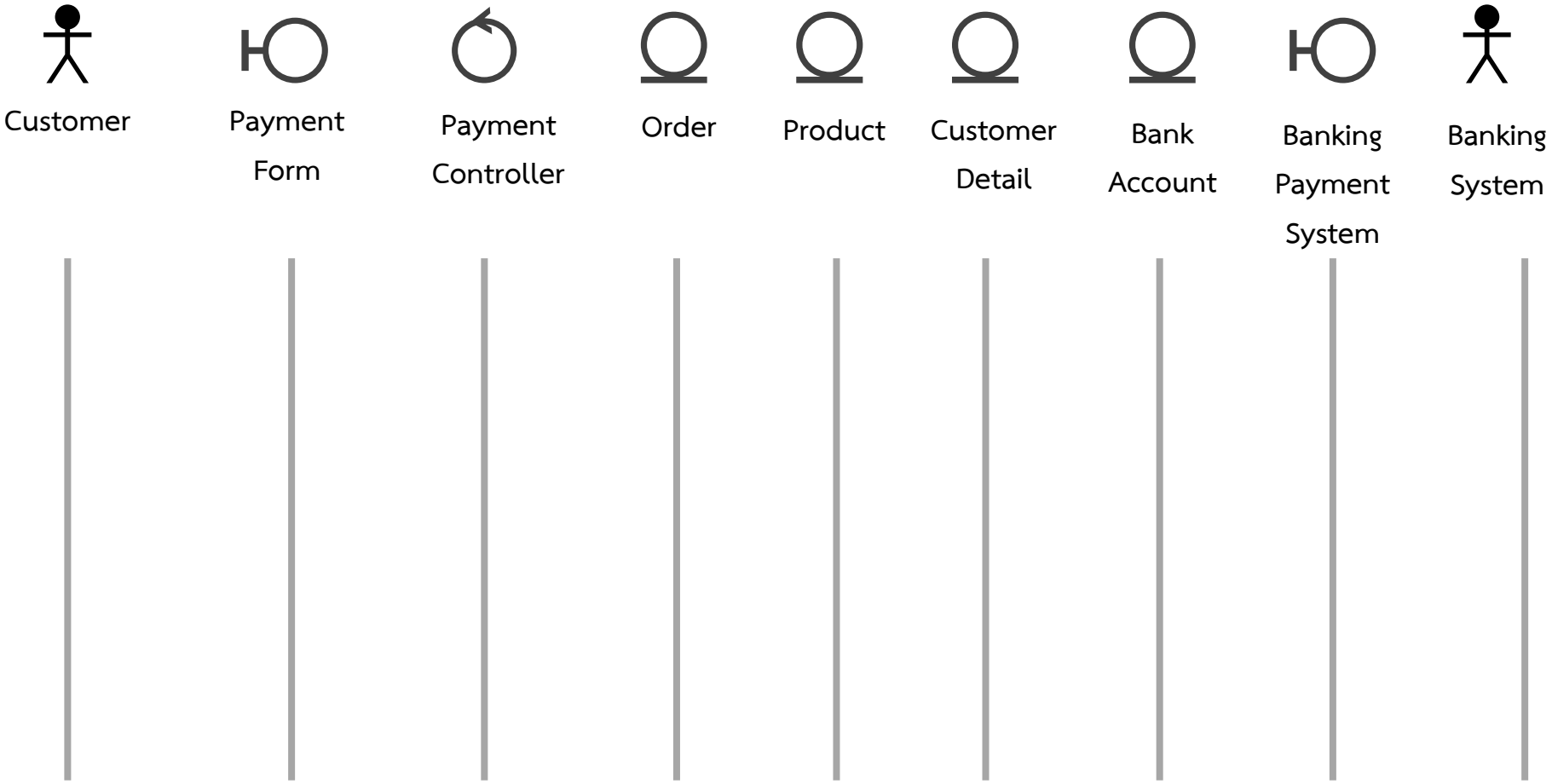


# Sequence Diagram

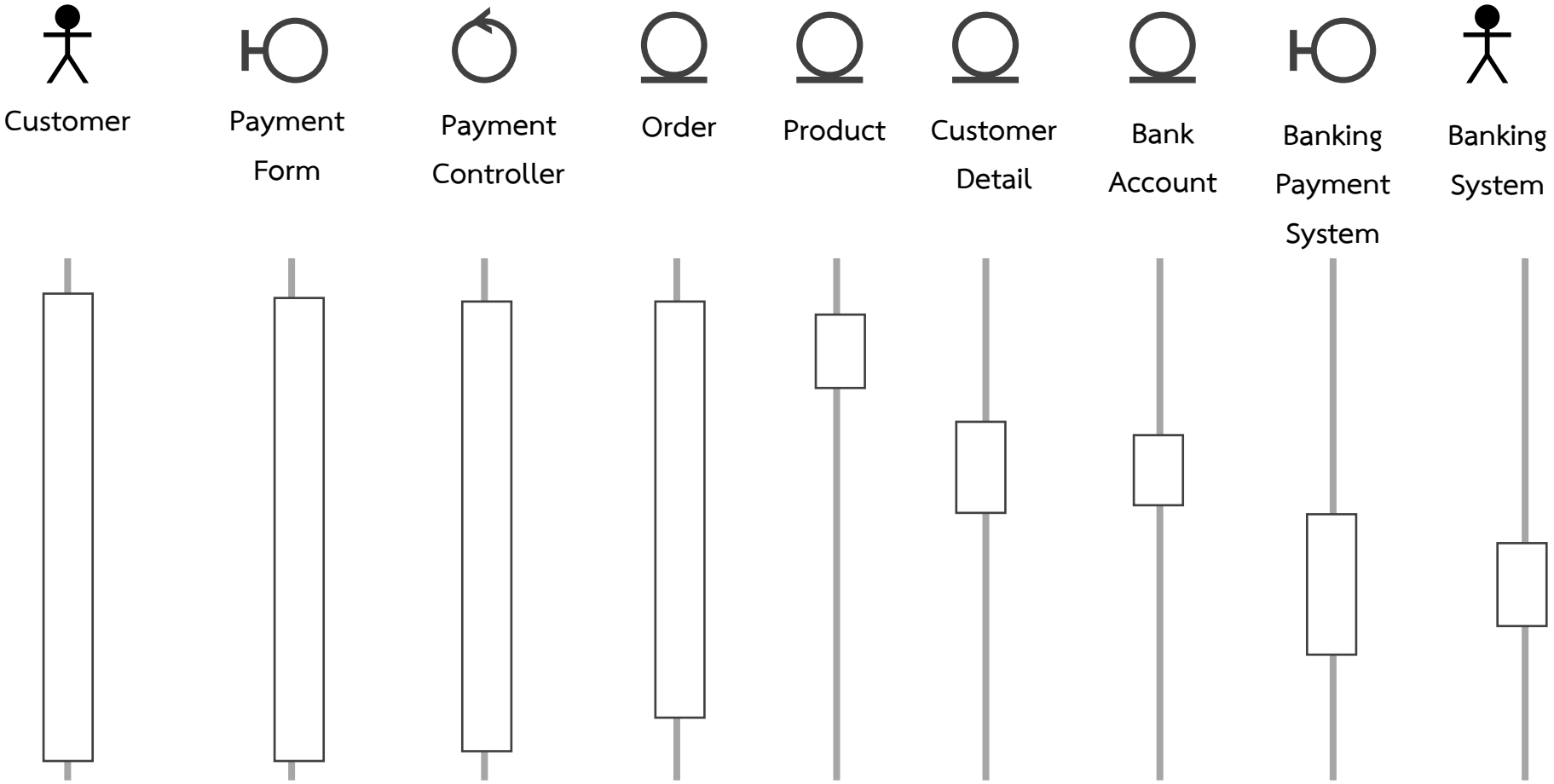
# Sequence Diagram

- มี  1 Sequence Diagram ต่อ 1 Use Case
- เขียนจาก Basic Flow เป็นหลัก
- มี alternative flow ใส่ในกล่องเงื่อนไข
- มีทุก elements จากจาก Collaboration Diagram ของ Use Case นั้น
- มี life time ของแต่ละ element
- มีลูกศรเส้นทึบและข้อความของทุก action ของของ collaboration diagram นั้น (สามารถเพิ่มเติมได้ตามความจำเป็น)
- มีลูกศรเส้นประย้อนกลับ เพื่อเป็นค่า return ของ action ได้
- ใส่ค่าให้กับ parameter ของแต่ละ action
  เช่น makePayment เป็น makePayment(OrderID, CustomerID) เป็นต้น

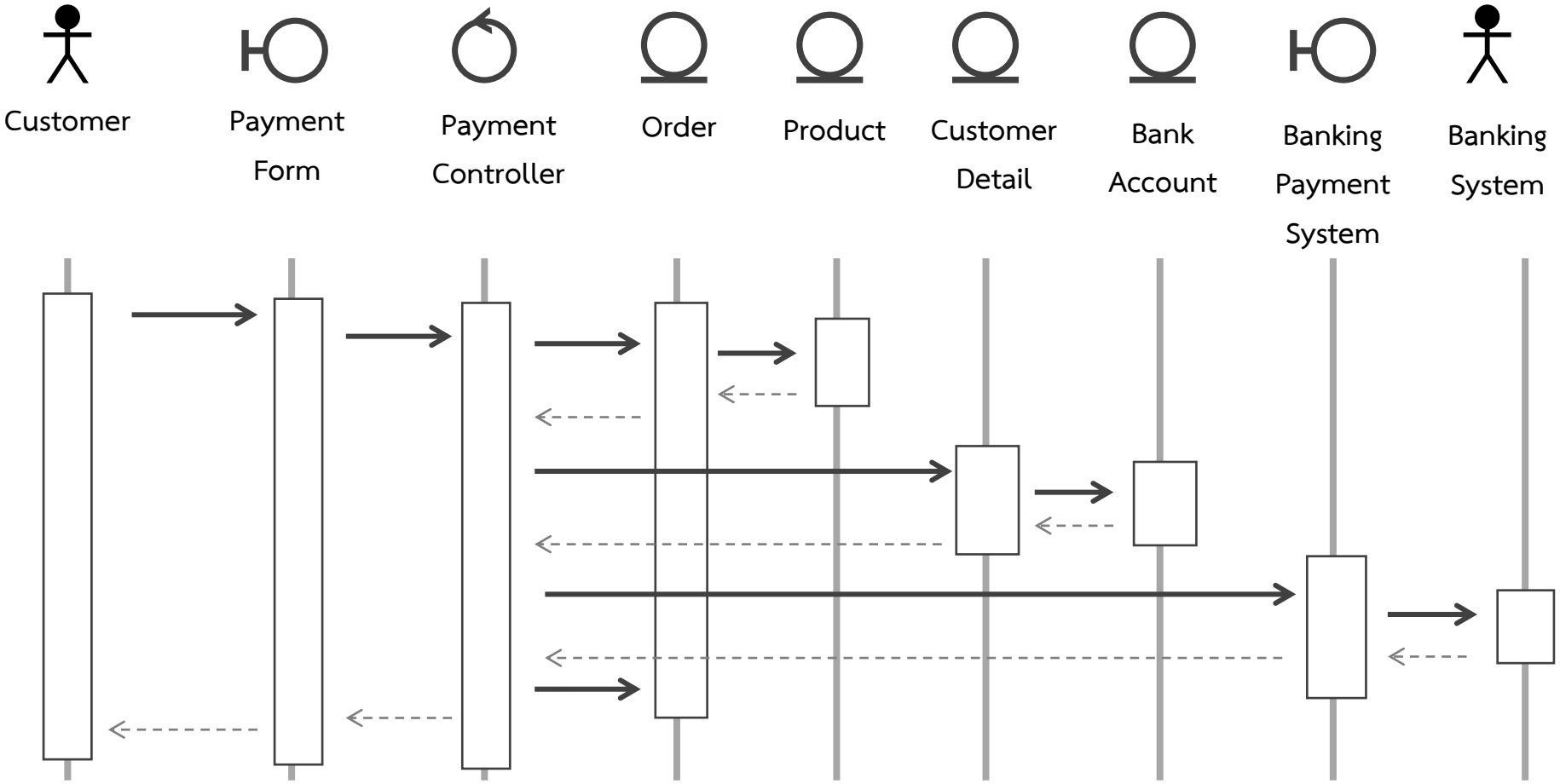
# Sequence Diagram: Payment



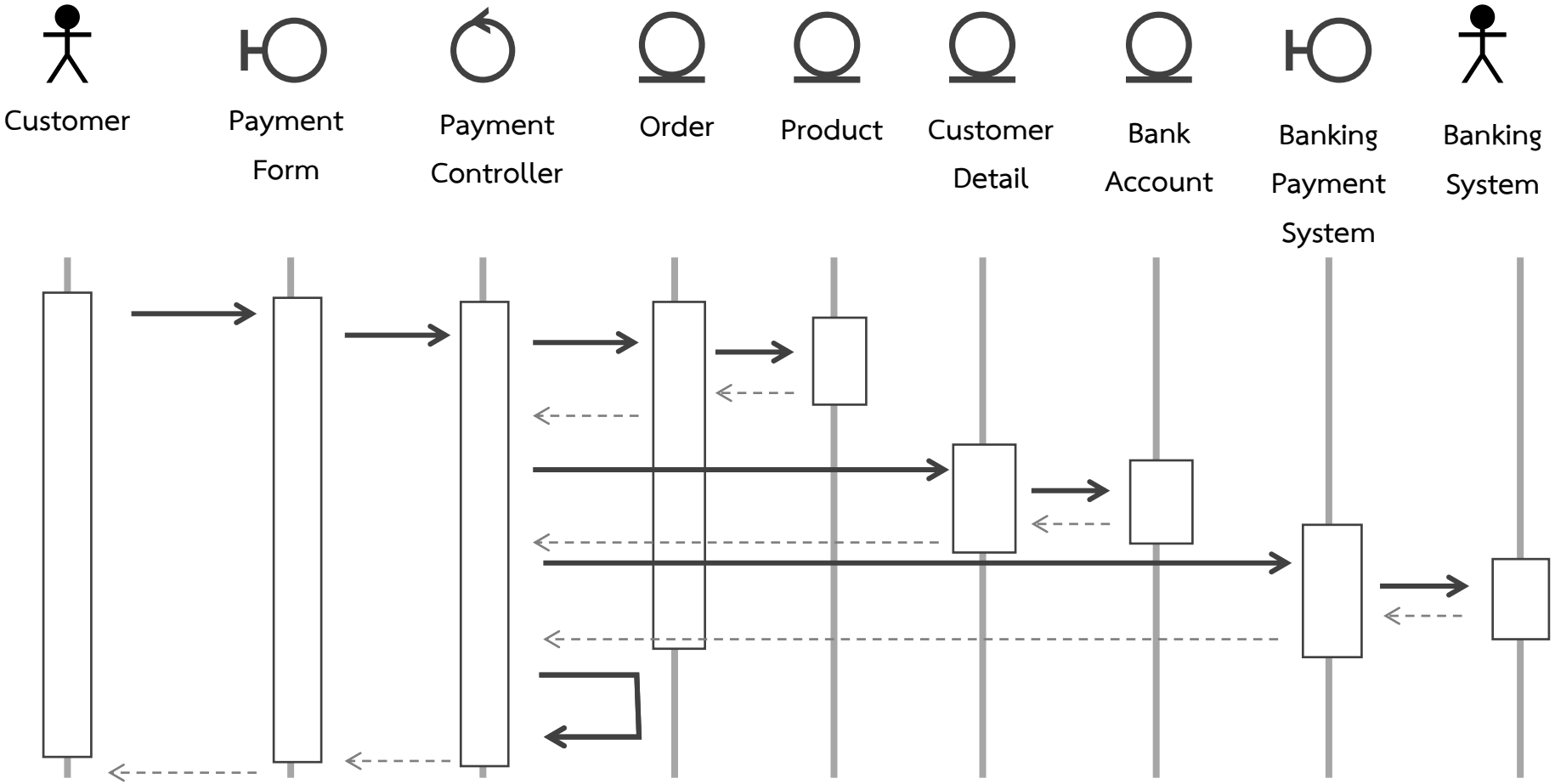
# Sequence Diagram: Payment



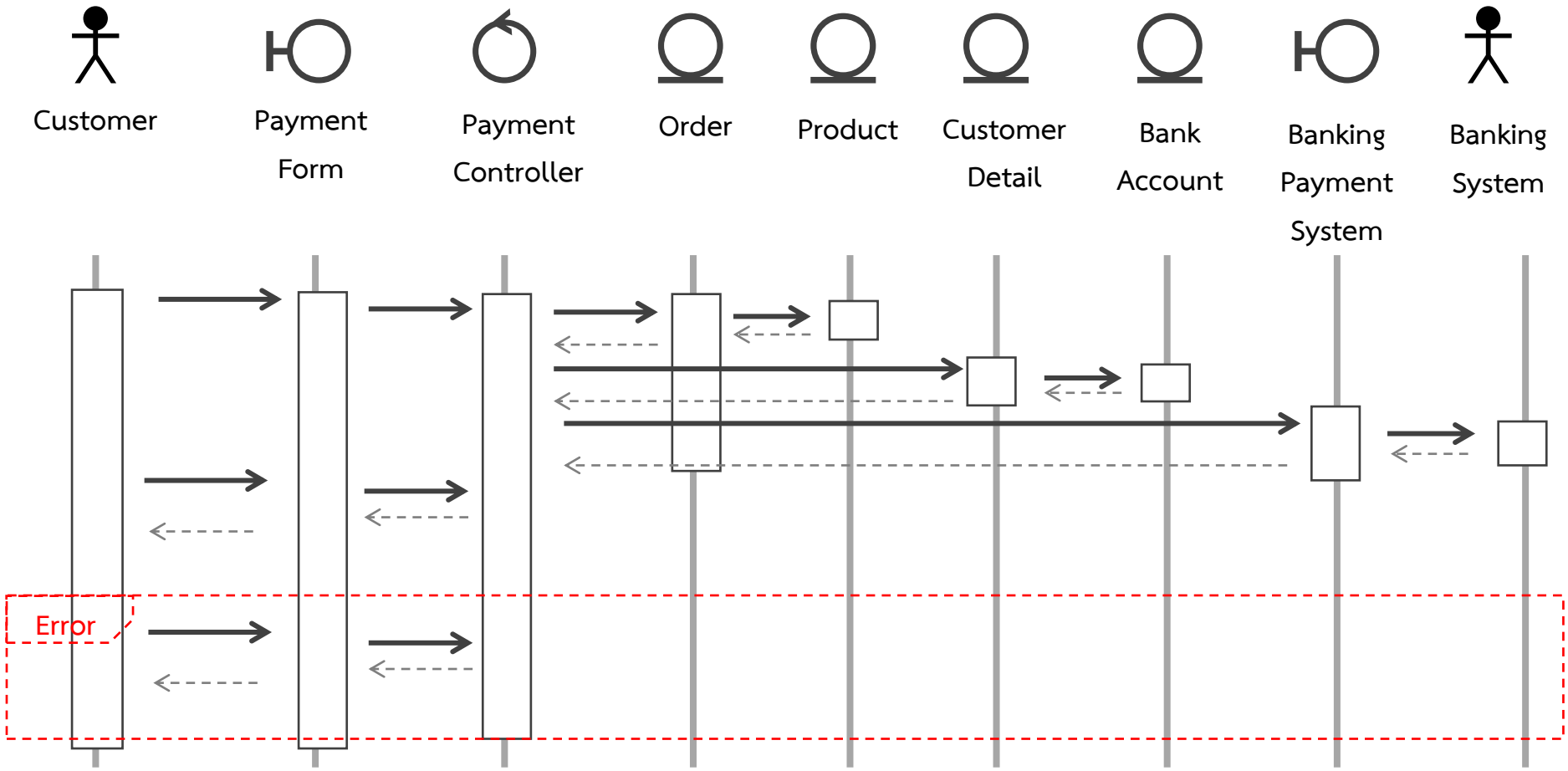
# Sequence Diagram: Payment



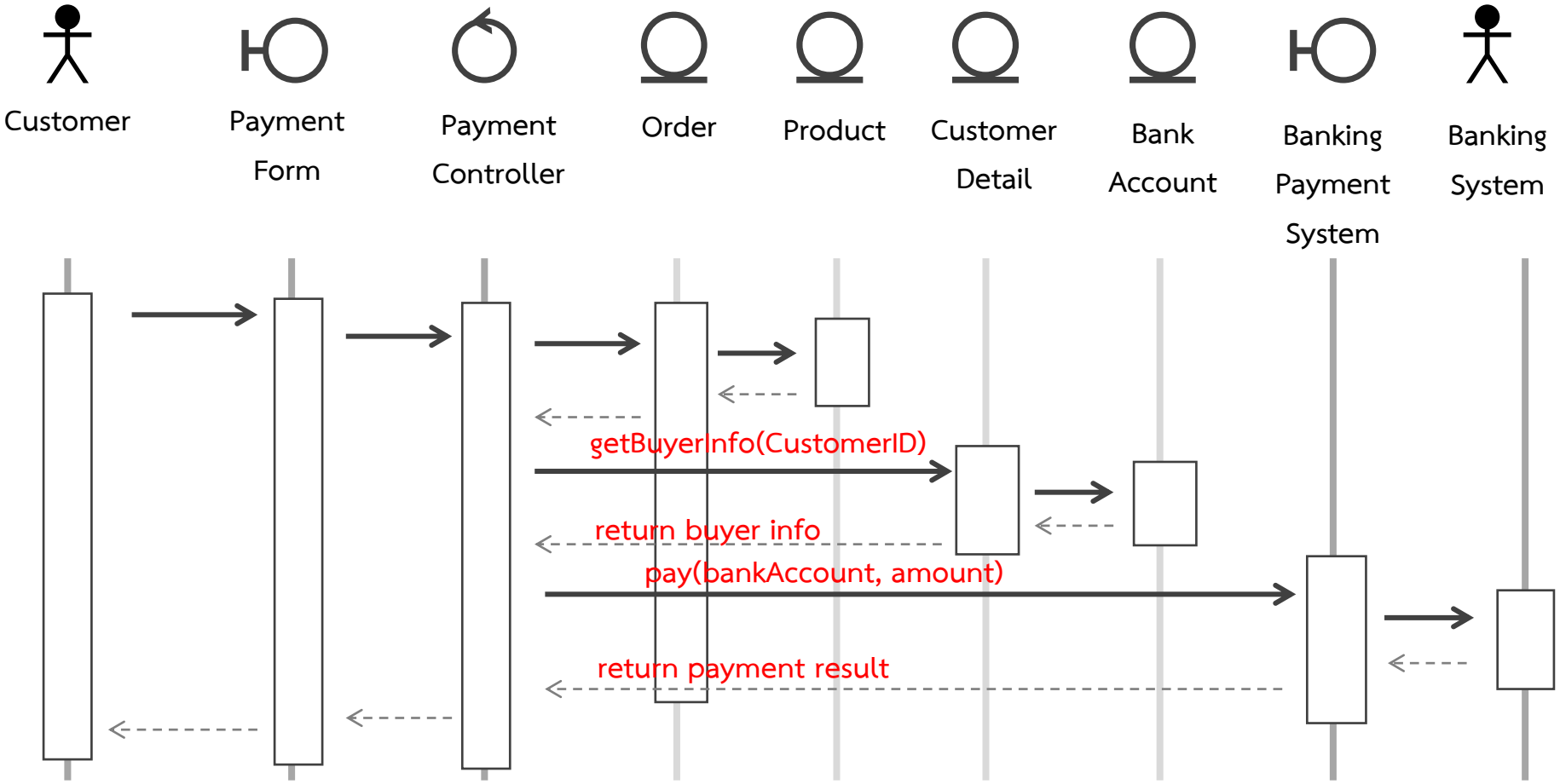
# Sequence Diagram: Payment



# Sequence Diagram: Payment



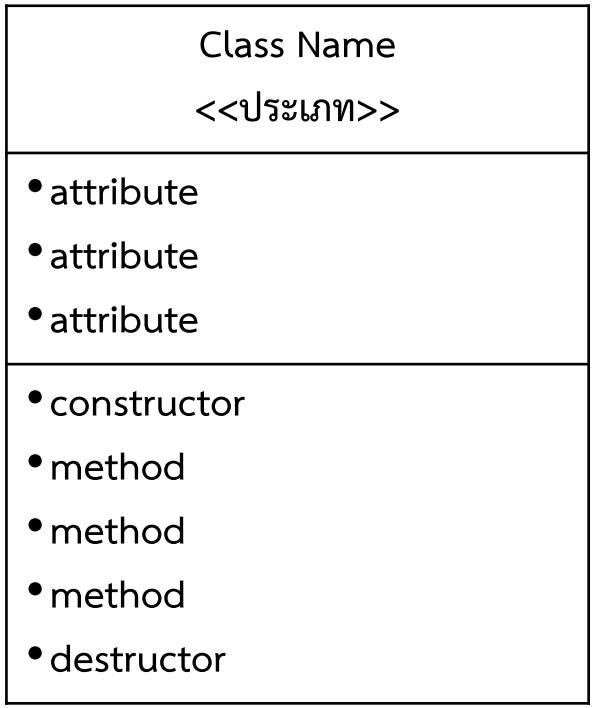
# Sequence Diagram: Payment

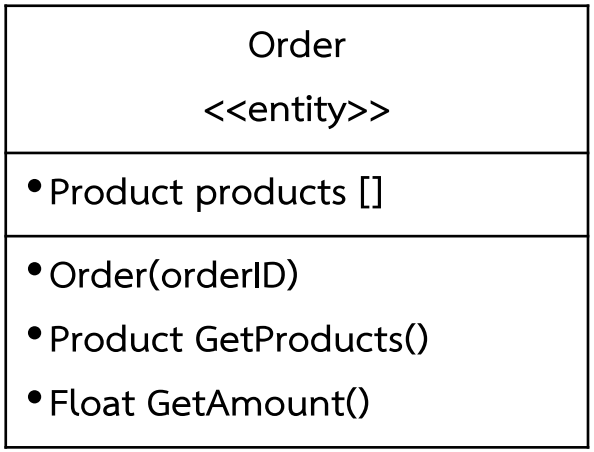


# Class Diagram

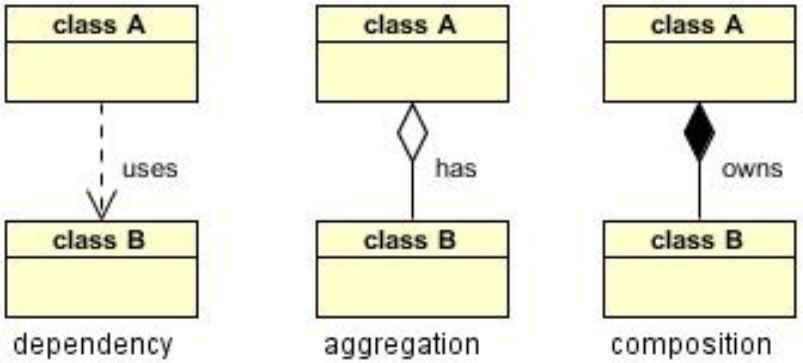
# Class

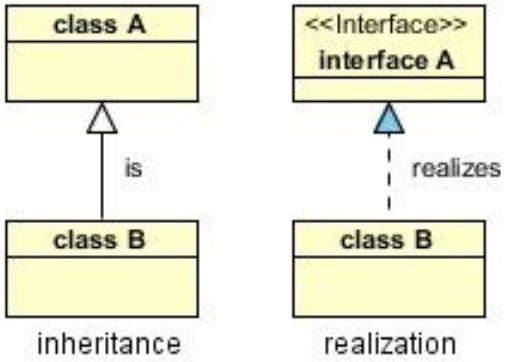
- แต่ละ element ต้องเป็น class



# Relationships

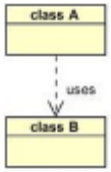
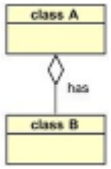
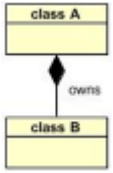
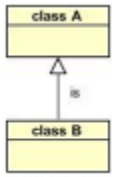
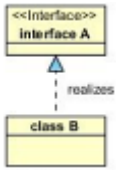



1. **Dependency** : class A uses class B
2. **Aggregation** : class A has a class B
3. **Composition** : class A owns a class B
4. **Inheritance** : class B is a Class A (or class A is extended by class B)
5. **Realization** : class B realizes Class A (or class A is realized by class B)

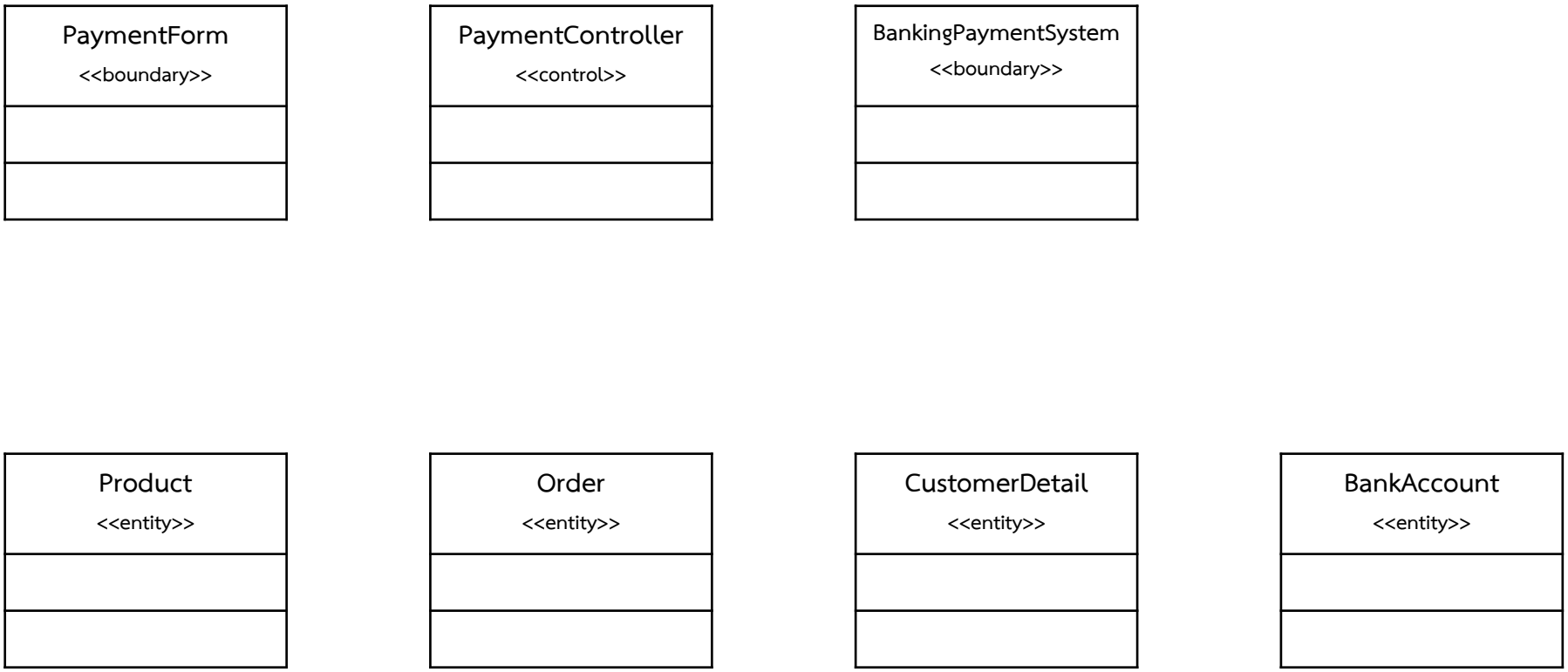
**Ref:** • (image) http://usna86-techbits.blogspot.com/2012/11/uml-class-diagram-relationships.html

# Relationships

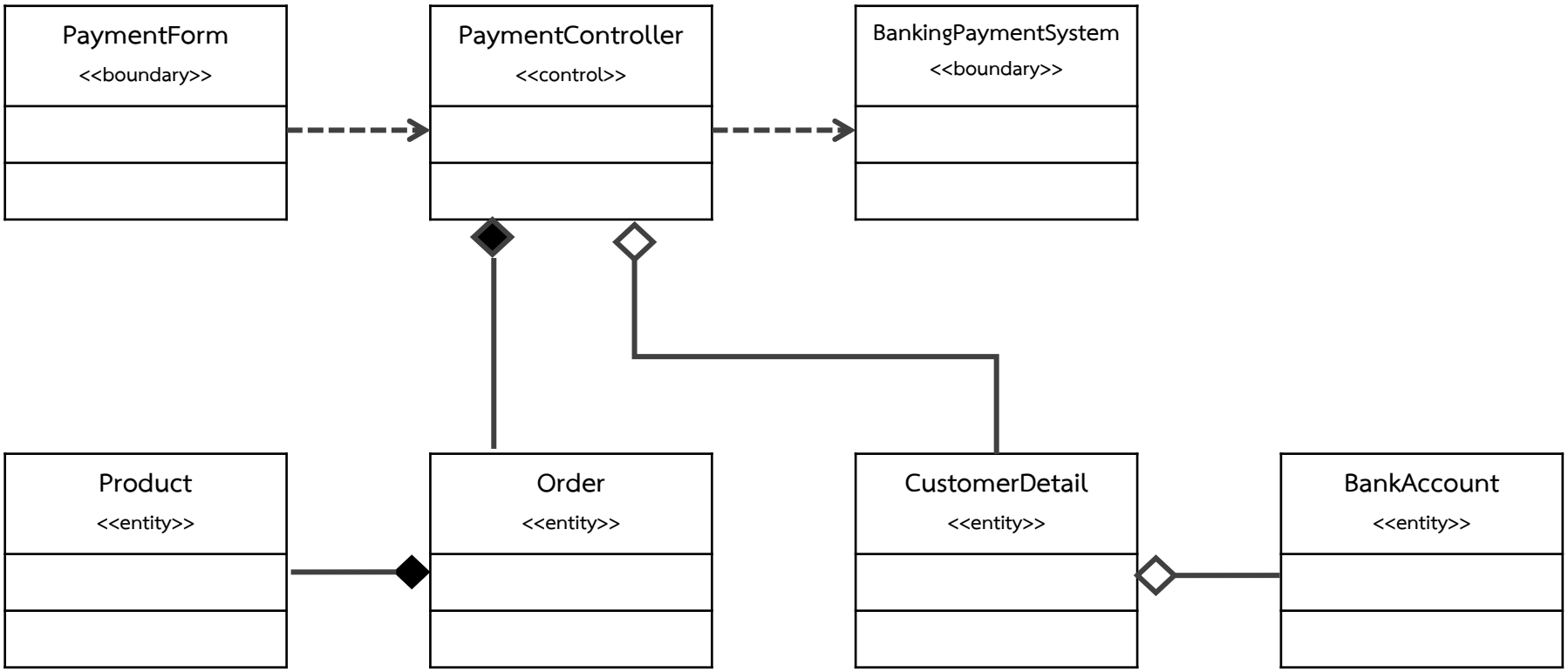
| | | |
|---|---|---|
| Dependency |  | `public class A {`<br>`  public void doSomething(B b) { ... }` |
| Aggregation |  | `public class A {`<br>`  private B _b;`<br>`  public void setB(B b) { _b = b; }` |
| Composition |  | `public class A {`<br>`  private B _b = new B();`<br>`}`<br><br>`public class A {`<br>`  private B _b;`<br>`  public A() {`<br>`  _b = new B(); }` |
| Inheritance |  | `public class A { ... }`<br>`public class B extends A { ....}` |
| Realization |  | `public interface A { ... }`<br>`public class B implements A { ....}` |

**Ref:** • (content) http://usna86-techbits.blogspot.com/2012/11/uml-class-diagram-relationships.html

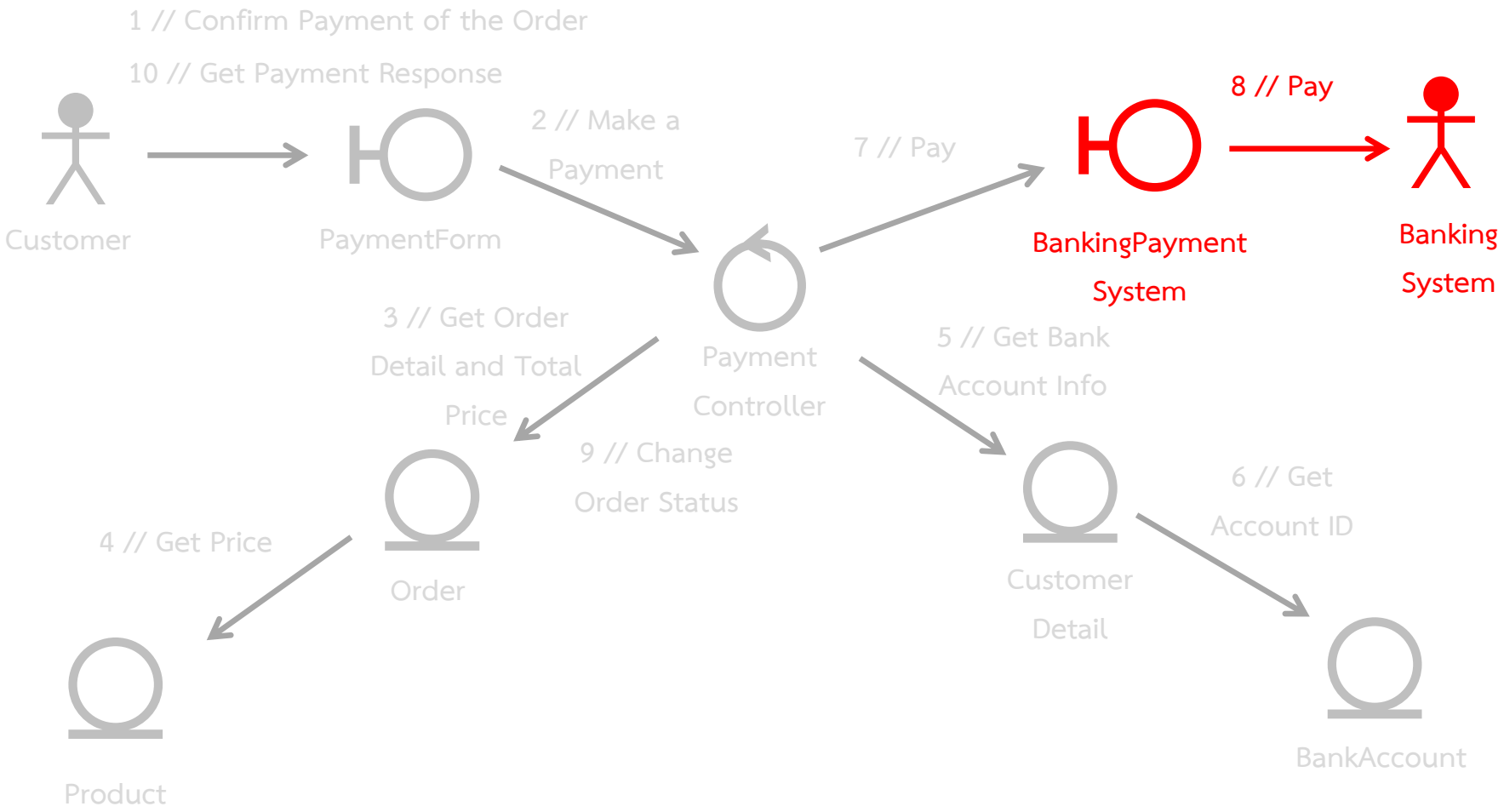
# Class Diagram: Payment



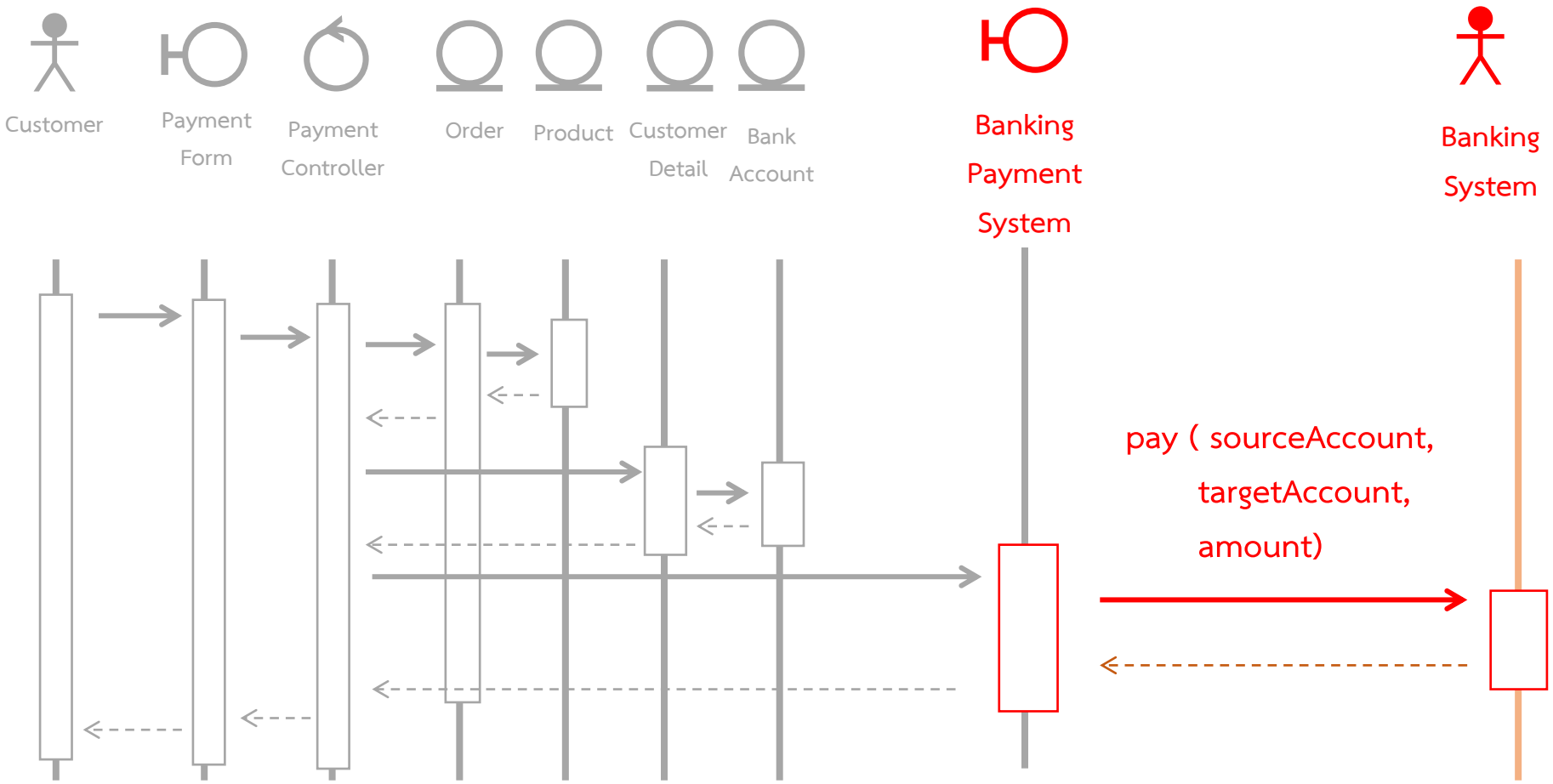
# Class Diagram: Payment



# Collaboration Diagram: Payment

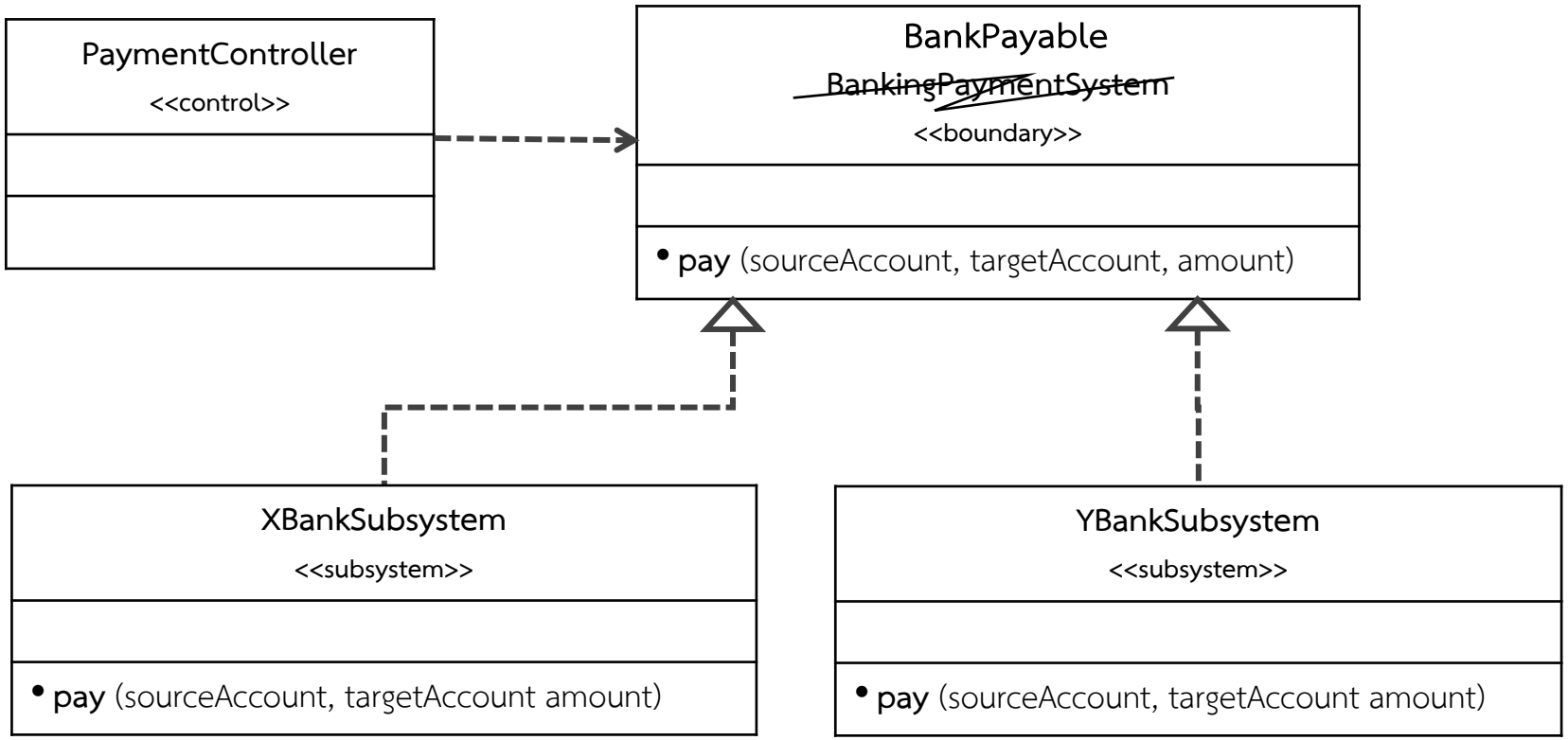


# Sequence Diagram: Payment



# Class Diagram: Payment

## Class Realization



# Example Code for Banking Payment

```
class PaymentController {
  public void run() {
    BankPayable payment = (BankPayable)beanfactory.getBean("BankService");
    payment.pay(customerBankAccoungID, shopBankAccountID, totolPrice);
  }
}
```

Configuration file "spring.xml"

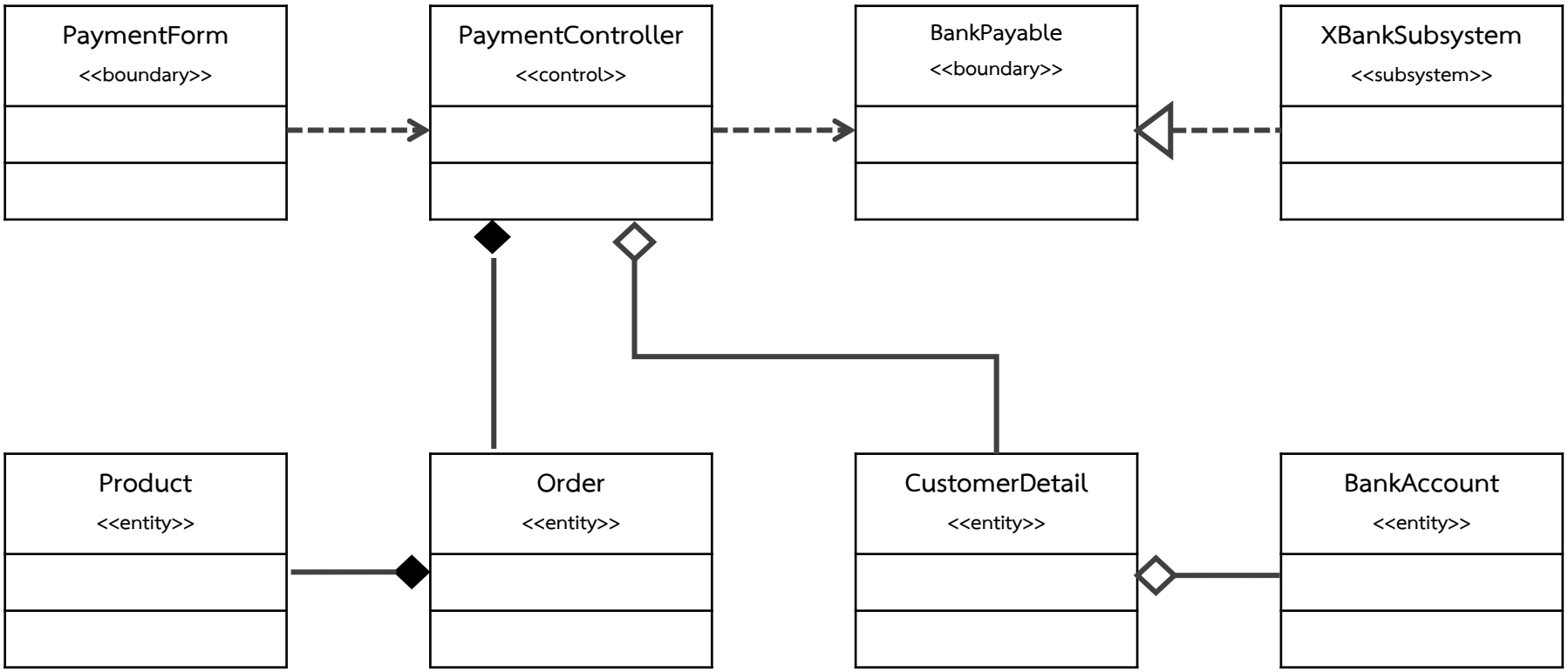
```
<bean id="BankingService" class="XBankSubsystem"></bean>
```

```
interface BankPayable {
  bool pay ( sourceAccount, targetAccount, amount);
}
```

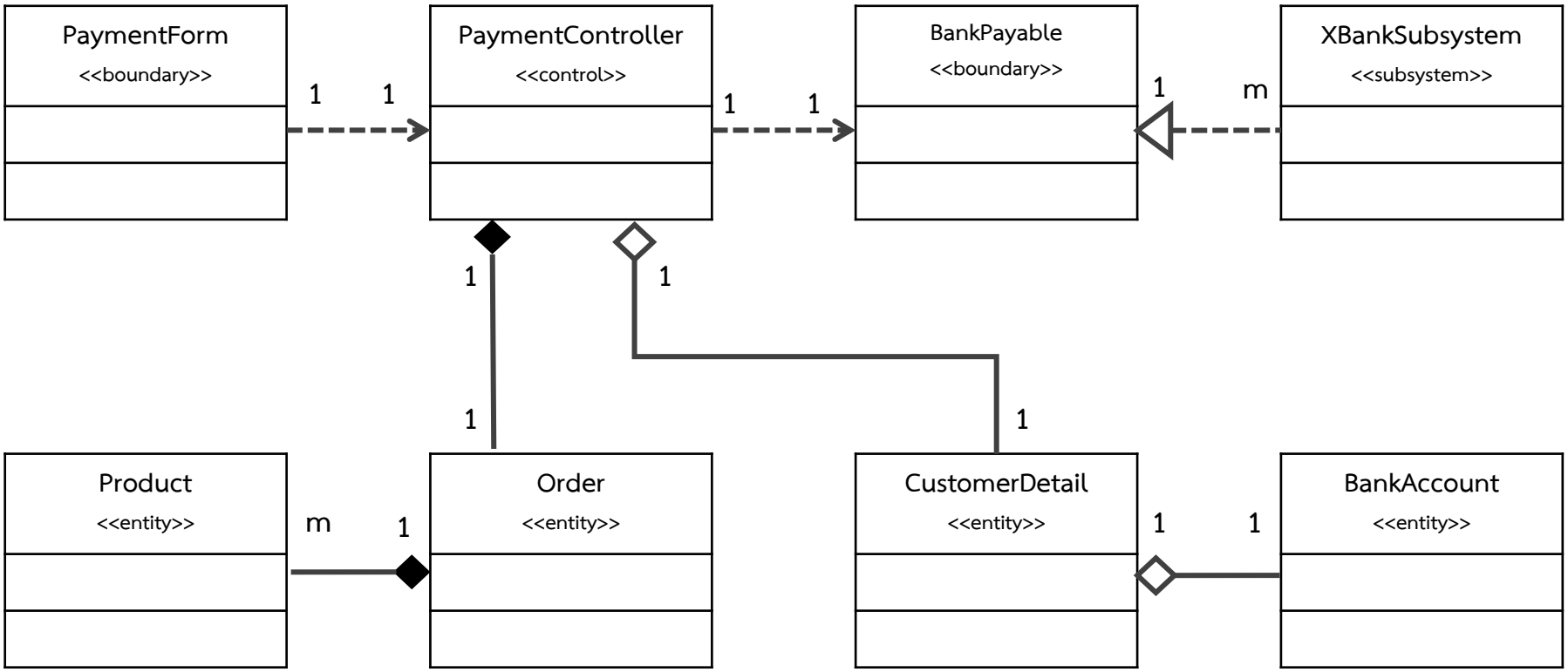
```
Class XBankSubsystem implements BankPayable {
  bool pay ( sourceAccount, targetAccount, amount){
    httpRequest("https://XBANK.com/pay?s=" + sourceAccount + "&t=" + targetAccount + "&p=" + amount );
  }
}
```

```
Class YBankSubsystem implements BankPayable {
  bool pay ( sourceAccount, targetAccount, amount){
    httpRequest("https://YBANK.com/transfer/" + sourceAccount + "/" + targetAccount + "/" + amount );
  }
}
```

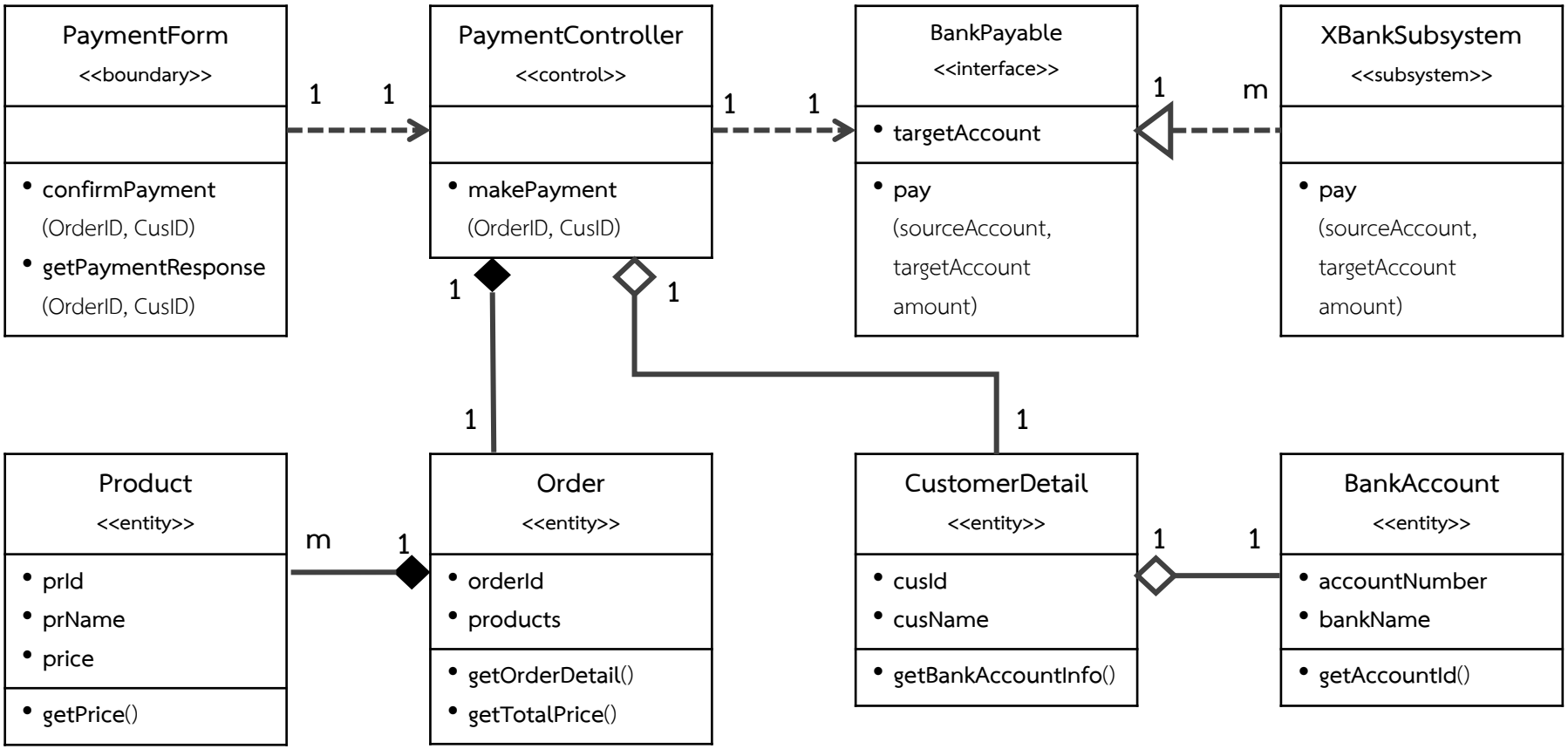
# Class Diagram: Payment



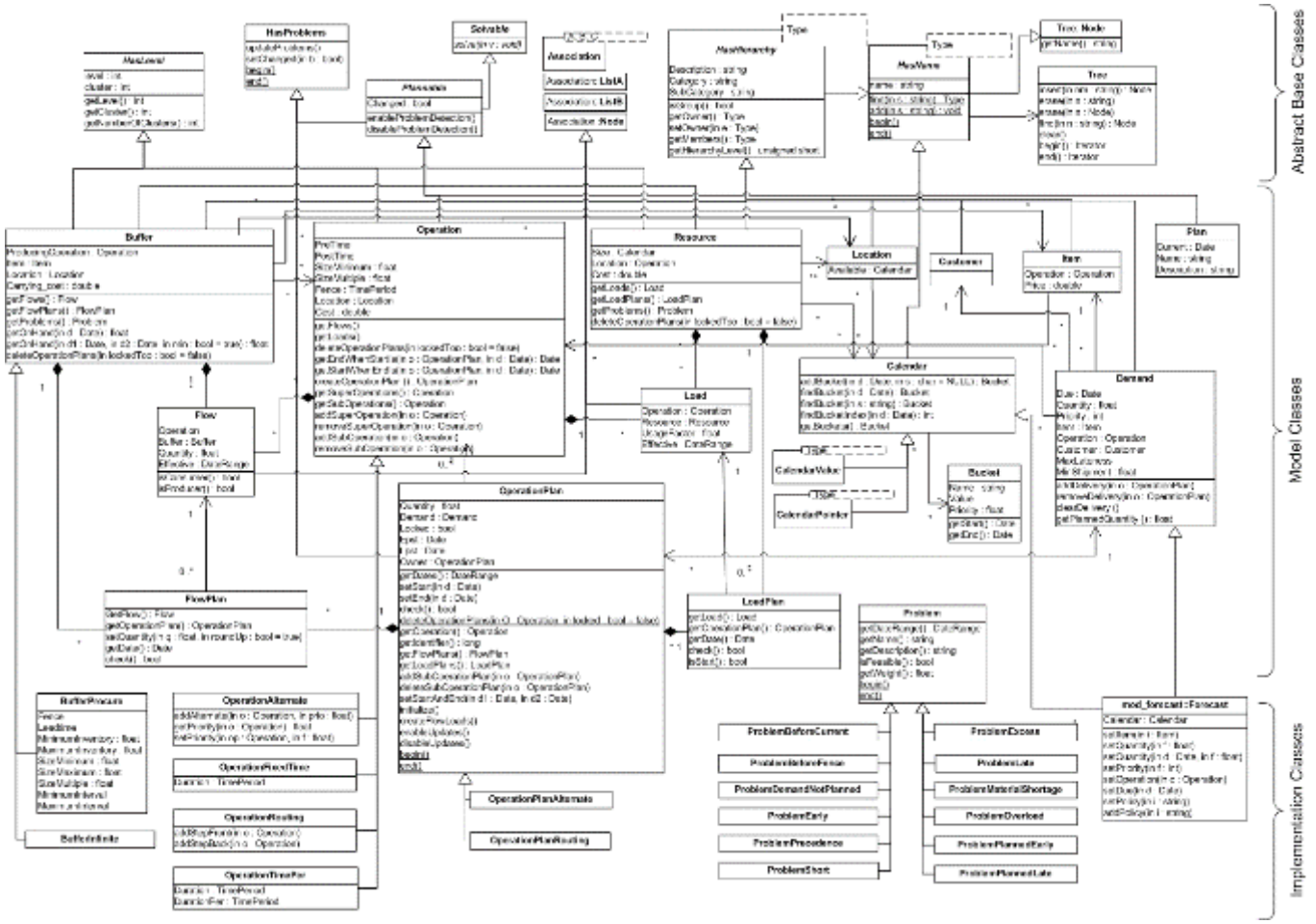
# Class Diagram: Payment



# Class Diagram: Payment



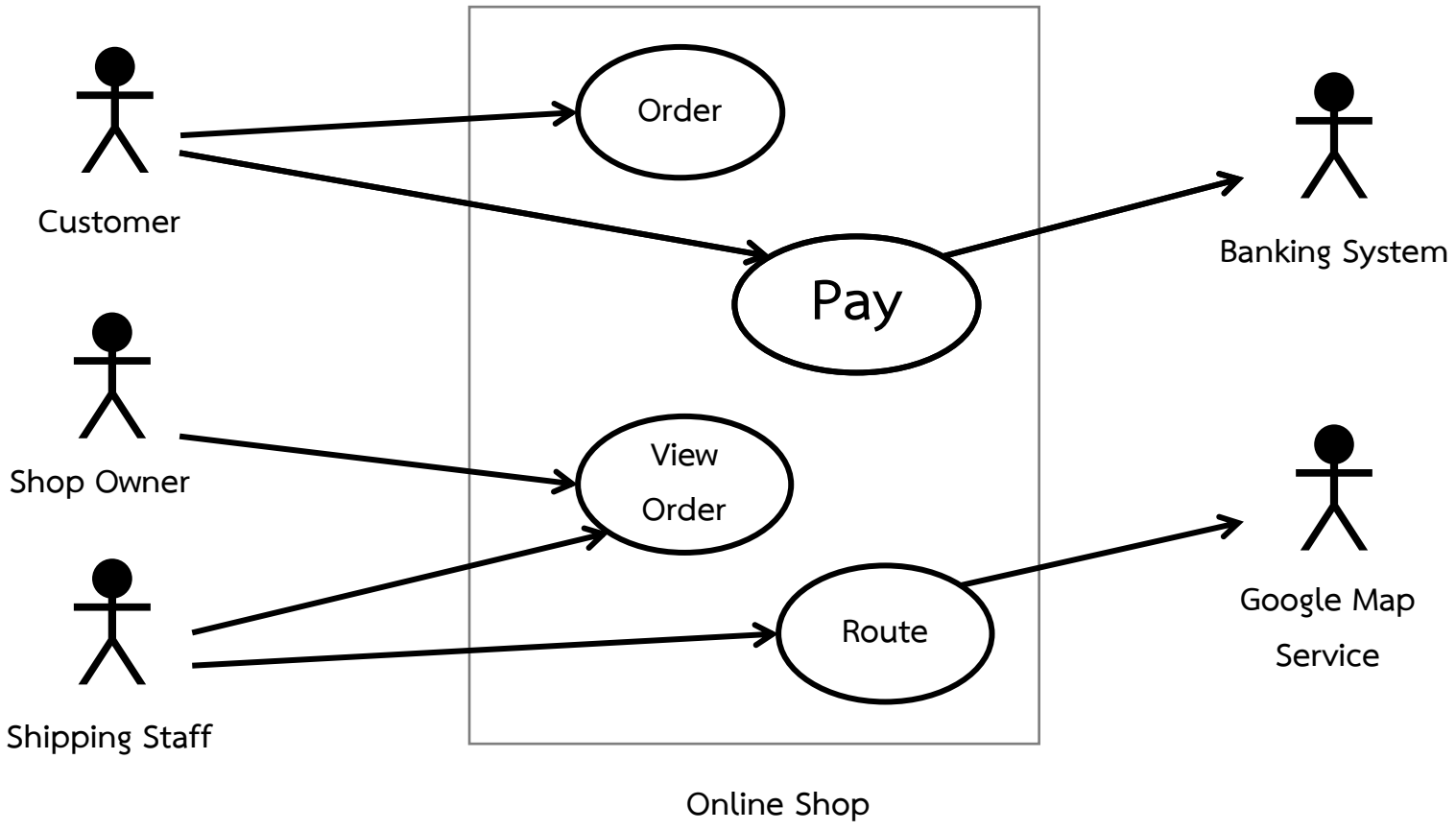
# Class Diagram of a Complex System



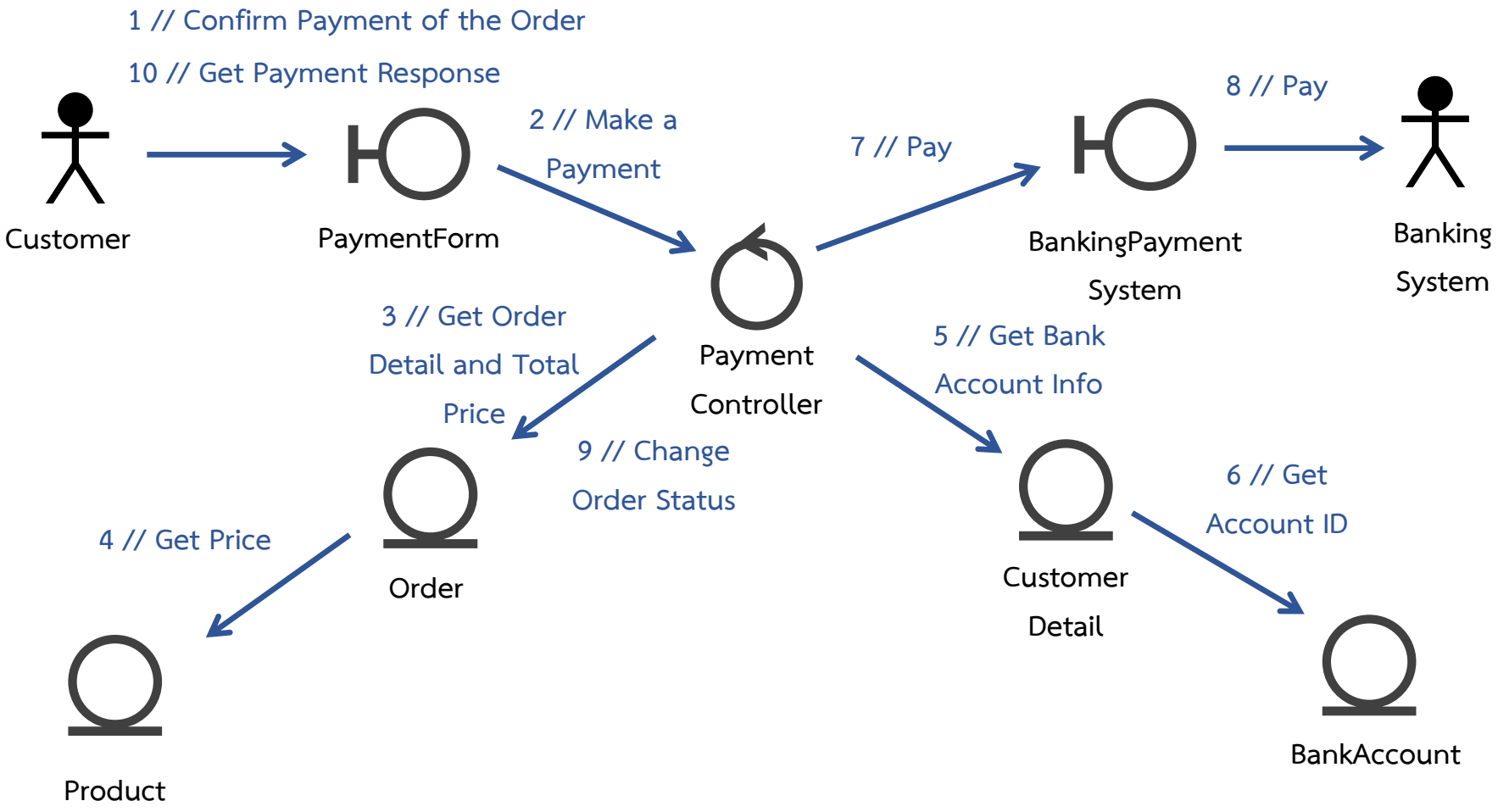

**Ref:** • (image) https://frepple.com/docs/4.0/developer-guide/planning-engine/class-diagram.html

# Summary

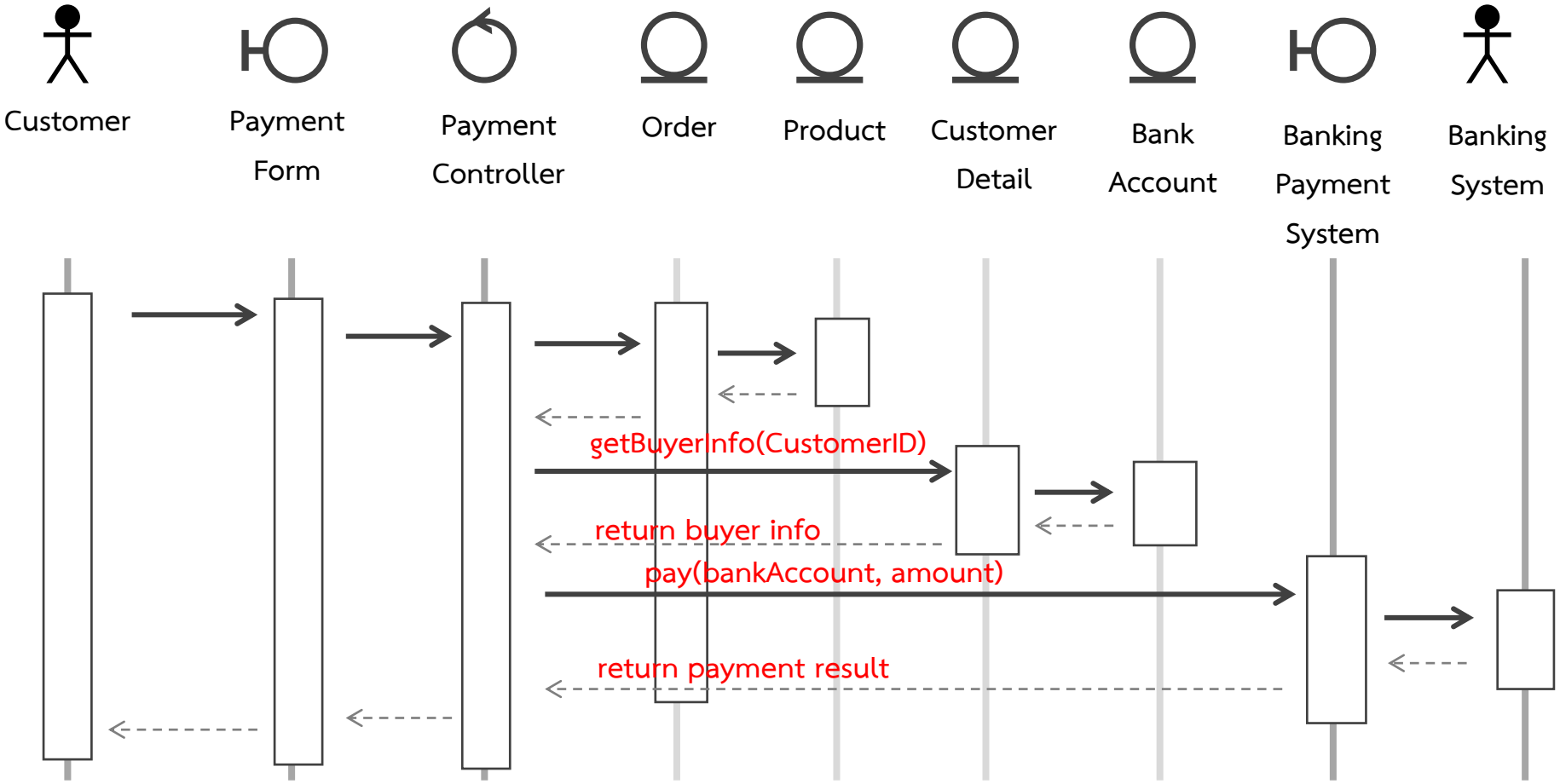
# Use Case Diagram



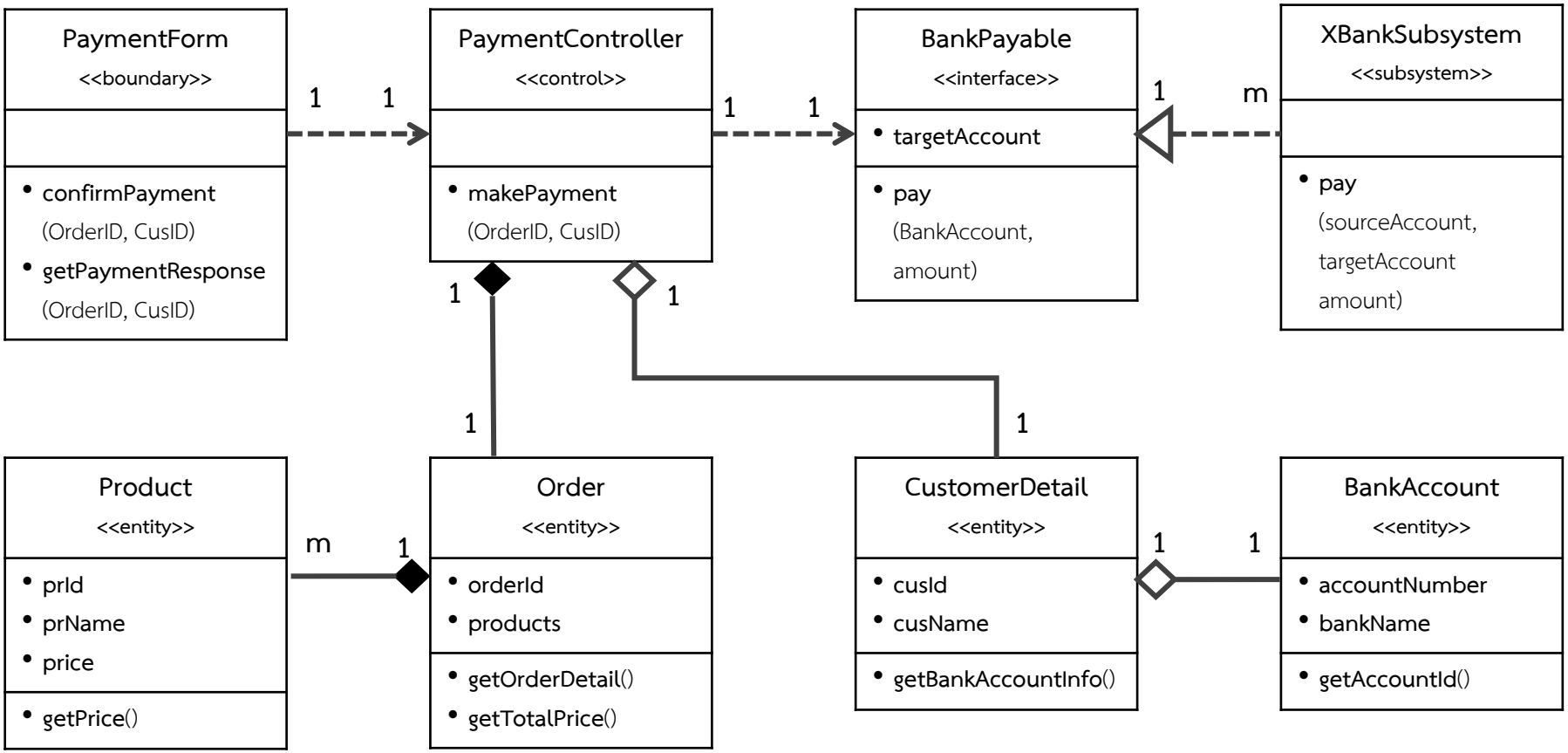
# Collaboration Diagram: Payment



# Sequence Diagram: Payment



# Class Diagram: Payment



“

When you have a blueprint
doesn't mean
you can't add another room.

”

Tom Graves