Faculty of
Information Technology
King Mongkut's Institute of Technology Ladkrabang

Chapter 2

Software Architecture & Decomposition Strategies

บรรยายโดย ผศ.ดร.ธราวิเชษฐ์ ธิติจรูญโรจน์ และอาจารย์สัญชัย น้อยจันทร์

คณะเทคโนโลยีสารสนเทศ

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

Outline

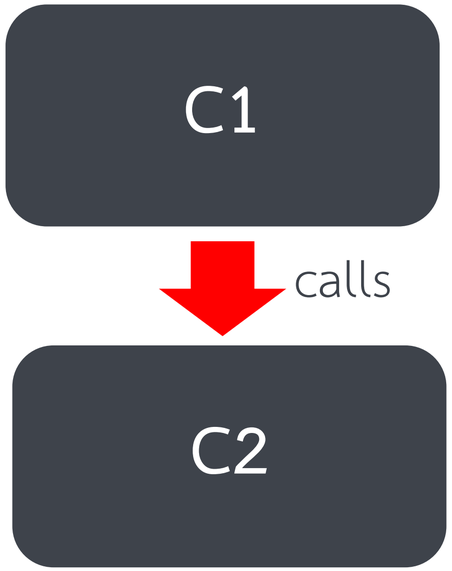
- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- Decomposition Strategies
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

Outline

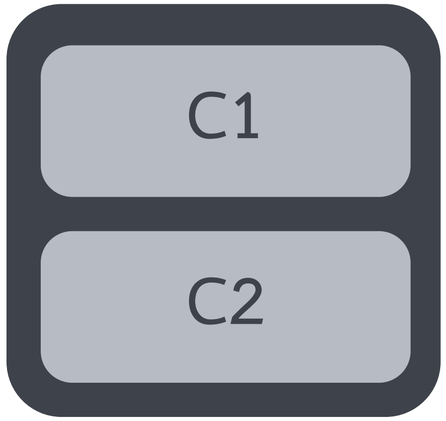
- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- Decomposition Strategies
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

ความสัมพันธ์ระหว่างคอมโพเนนต์

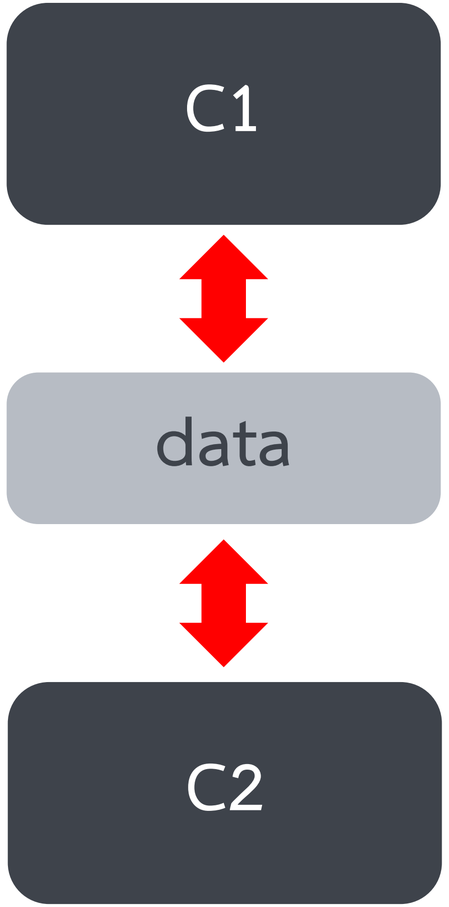
C2 *is part of* C1



C1 *uses* C2



C1 *is located with* C2



C1 *shares data with* C2

Outline

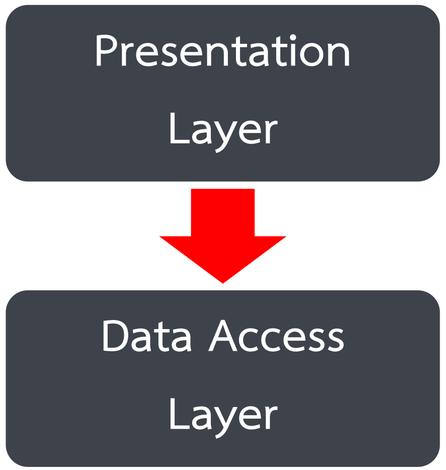
- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- Decomposition Strategies
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

ความแตกต่างระหว่าง Tier และ Layer

Logical Layer Architecture คือ การมองในมุมของการจัดการไฟล์โค้ดซึ่งแบ่งแยกไฟล์โค้ดตามหน้าที่การทำงาน เพื่อความสะดวกต่อการจัดการและการปรับปรุง อาทิเช่น Presentation Layer, Business Logic Layer, และ Data Access Layer เป็นต้น

Physical Tier Architecture คือ การมองในมุมของตำแหน่งที่ใช้จัดเก็บของไฟล์โค้ดขณะโปรแกรมกำลังทำงาน อาทิเช่น Client Tier, Web Tier และ Database Tier

สถาปัตยกรรมแบบ 2 เลเยอร์



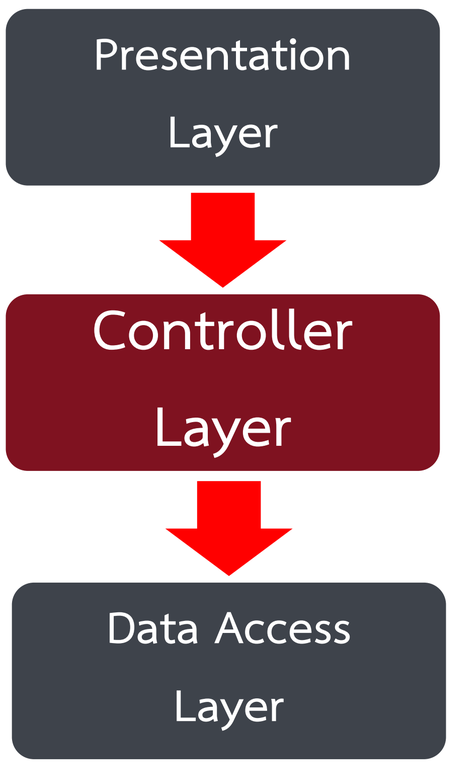
สถาปัตยกรรมแบบ 2 เลเยอร์จะประกอบไปด้วย 2 เลเยอร์หลัก ได้แก่ Presentation Layer และ Data Access Layer เท่านั้น โดยที่ Presentation Layer ทำหน้าที่สำหรับส่วนติดต่อกับผู้ใช้ และ Data Access Layer ทำในส่วนการติดต่อกับฐานข้อมูลเพื่อจัดการข้อมูล

คำถาม คือ “แล้ว Business Logic จะอยู่เลเยอร์ใด”

คำตอบ คือ “จะกระจายอยู่ทั้ง Presentation Layer และ Data Access Layer ขึ้นอยู่กับการออกแบบของนักพัฒนาโปรแกรม ”

การออกแบบด้วยสถาปัตยกรรมแบบ 2 เลเยอร์มีข้อดีคือ Simple เหมาะสมระบบที่ไม่ค่อยมีความซับซ้อน

สถาปัตยกรรมแบบ 3 เลเยอร์
(Component-based Architecture : CA)

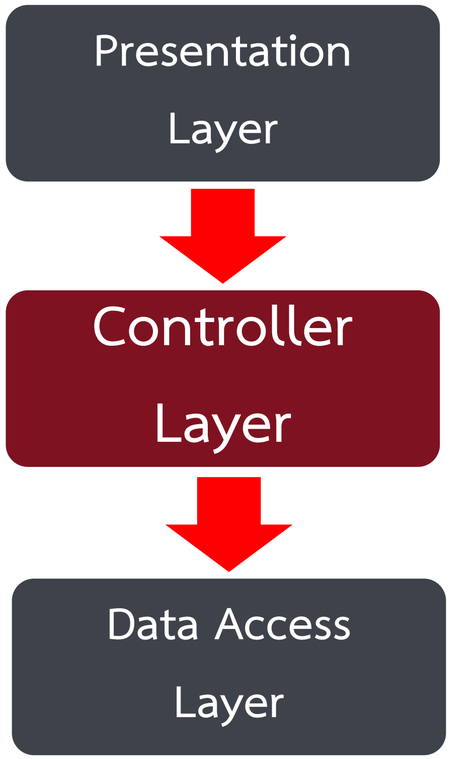


สถาปัตยกรรมแบบ 3 เลเยอร์จะประกอบไปด้วย 3 เลเยอร์หลัก ได้แก่ Presentation Layer, Controller Layer และ Data Access Layer โดยที่ Controller Layer คือ เลเยอร์ที่

- เกี่ยวข้องกับ Business Logic และงานที่สร้างมูลค่าเพิ่มได้
- เกี่ยวข้องกับ Core ธุรกิจ
- เกี่ยวกับการจัดการ Request ที่เข้ามา
- เกี่ยวข้องกับการ Preprocessing และ Transform ข้อมูลบางอย่างสำหรับ Presentation Layer หรือ Data Access Layer
- จัดการกับ Use Case ที่เกี่ยวข้อง

ซึ่งสามารถช่วยแก้ปัญหาการกระจาย Business Logic ของสถาปัตยกรรมแบบ 2 เลเยอร์ได้

สถาปัตยกรรมแบบ 3 เลเยอร์ (Component-based Architecture : CA)

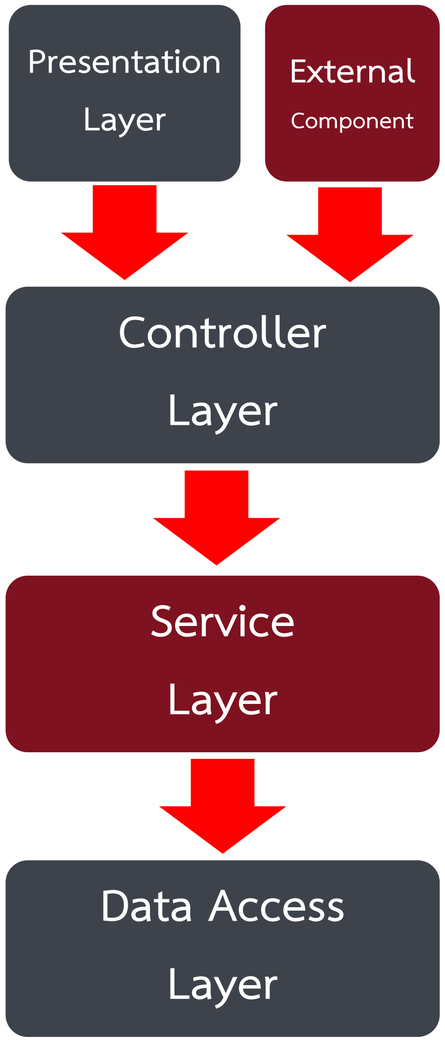


แต่อย่างไรก็ตาม สถาปัตยกรรมข้างต้นจะพบปัญหาหลัก ได้แก่

Sharing components ระหว่างแพลตฟอร์ม (ระหว่างแอปพลิเคชัน) ที่แตกต่างกันเป็น*เรื่องยาก* อาทิเช่น เมื่อ 2 คอมโพเนนต์ ต้องการแลกเปลี่ยนข้อมูลกัน ผู้ใช้ต้องรับข้อมูลจากแอปพลิเคชันหนึ่งด้วยตนเองและป้อนข้อมูลลงในแอปพลิเคชันอื่น

Code integration ทำให้เกิด Tightly coupled ทำให้*ยากต่อการเปลี่ยนแปลง*ใด ๆ

สถาปัตยกรรมแบบ 4 เลเยอร์



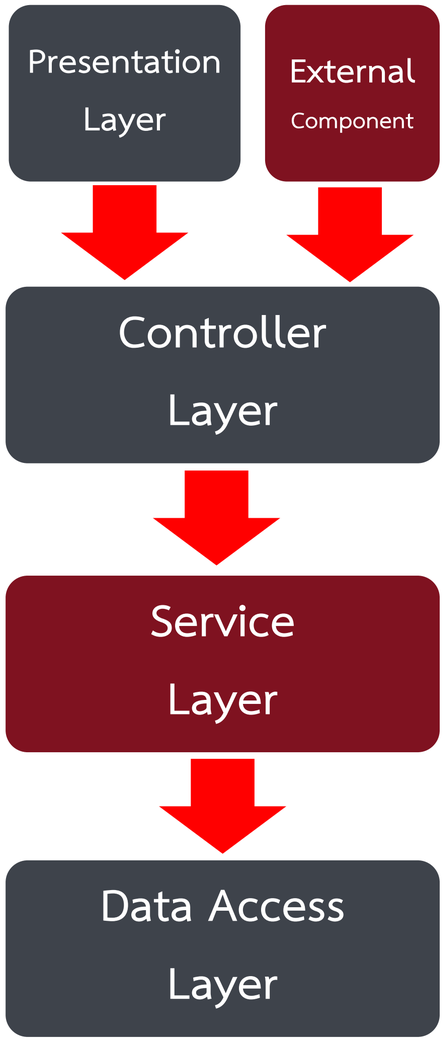
คือ สถาปัตยกรรมที่ออกแบบมาเพื่อรองรับให้ 2 คอมโพเนนต์ (ผู้ร้องขอบริการและผู้รับบริการ) สามารถสื่อสารกันข้ามโดเมนที่แตกต่างกันทางด้านเทคโนโลยีหรือคนละแพลตฟอร์ม (แอปพลิเคชัน)

ซึ่งประกอบไปด้วย 4 เลเยอร์ ได้แก่ Presentation Layer, Service Layer, Controller Layer และ Data Access Layer โดยมีหน้าที่ดังต่อไปนี้

- *Presentation Layer* มีหน้าที่นำเสนอข้อมูลและเป็นส่วนติดต่อกับผู้ใช้งาน

- *Data Access Layer* มีหน้าที่เชื่อมต่อกับฐานข้อมูลเท่านั้น

สถาปัตยกรรมแบบ 4 เลเยอร์



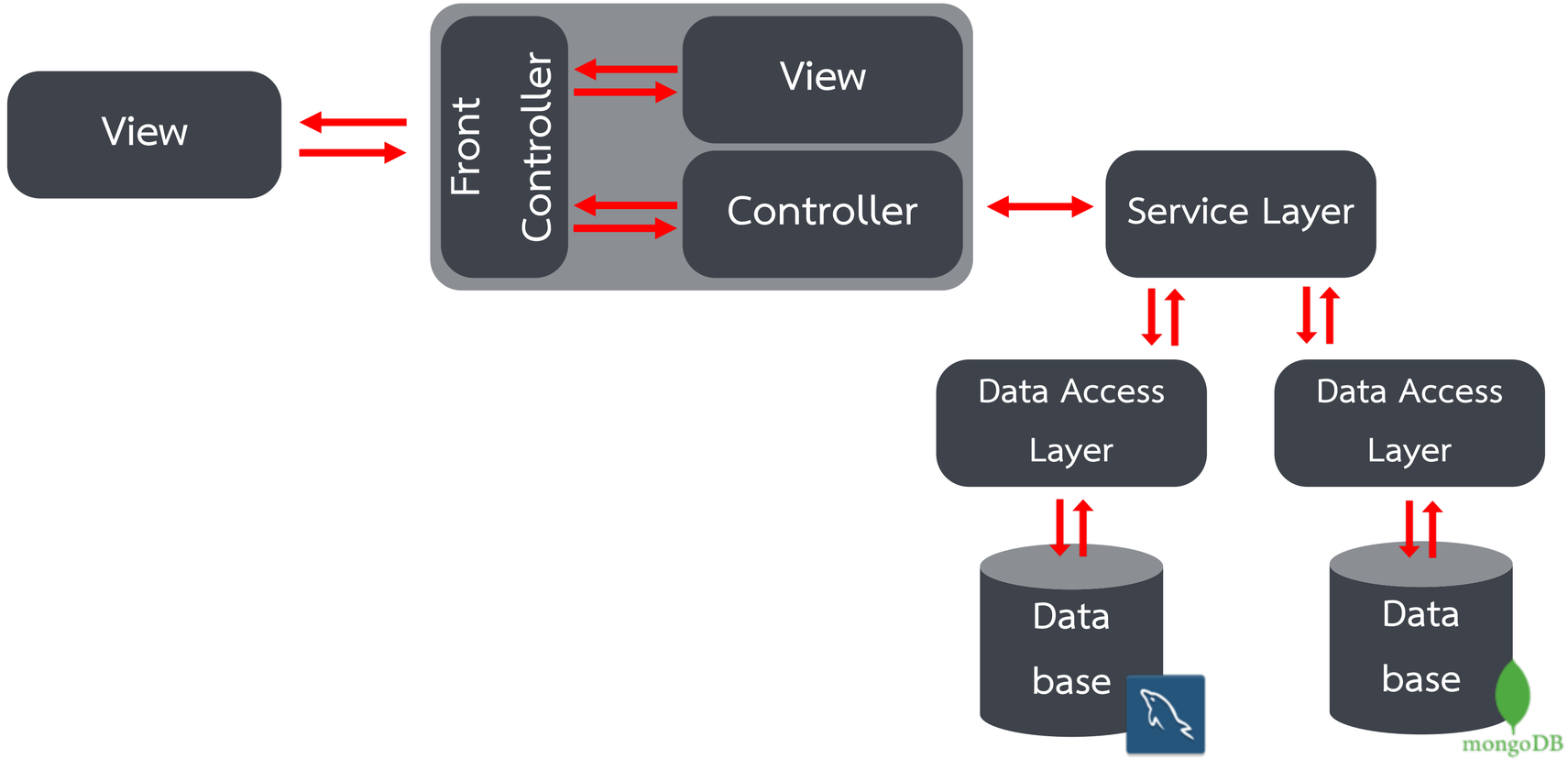
- ***Service Layer*** มีหน้าที่จัดเก็บ Business Logic ไว้ แล้วจะเรียกใช้งาน Data Access Layer แทน Controller Layer เพื่อหลีกเลี่ยงเหตุการณ์ที่ผู้ร้องขอบริการจะเข้าถึง Data Access Layer ได้โดยตรง สิ่งนี้ช่วยเพิ่มความปลอดภัยของระบบได้ นอกจากนี้ ยังช่วยลด Tightly coupled ให้เป็น Loose Coupling มากยิ่งขึ้น เช่น กรณีเปลี่ยนโครงสร้างของข้อมูลหรือชนิดฐานข้อมูล และยังสามารถทำ Transaction, Logging, Validate data

---

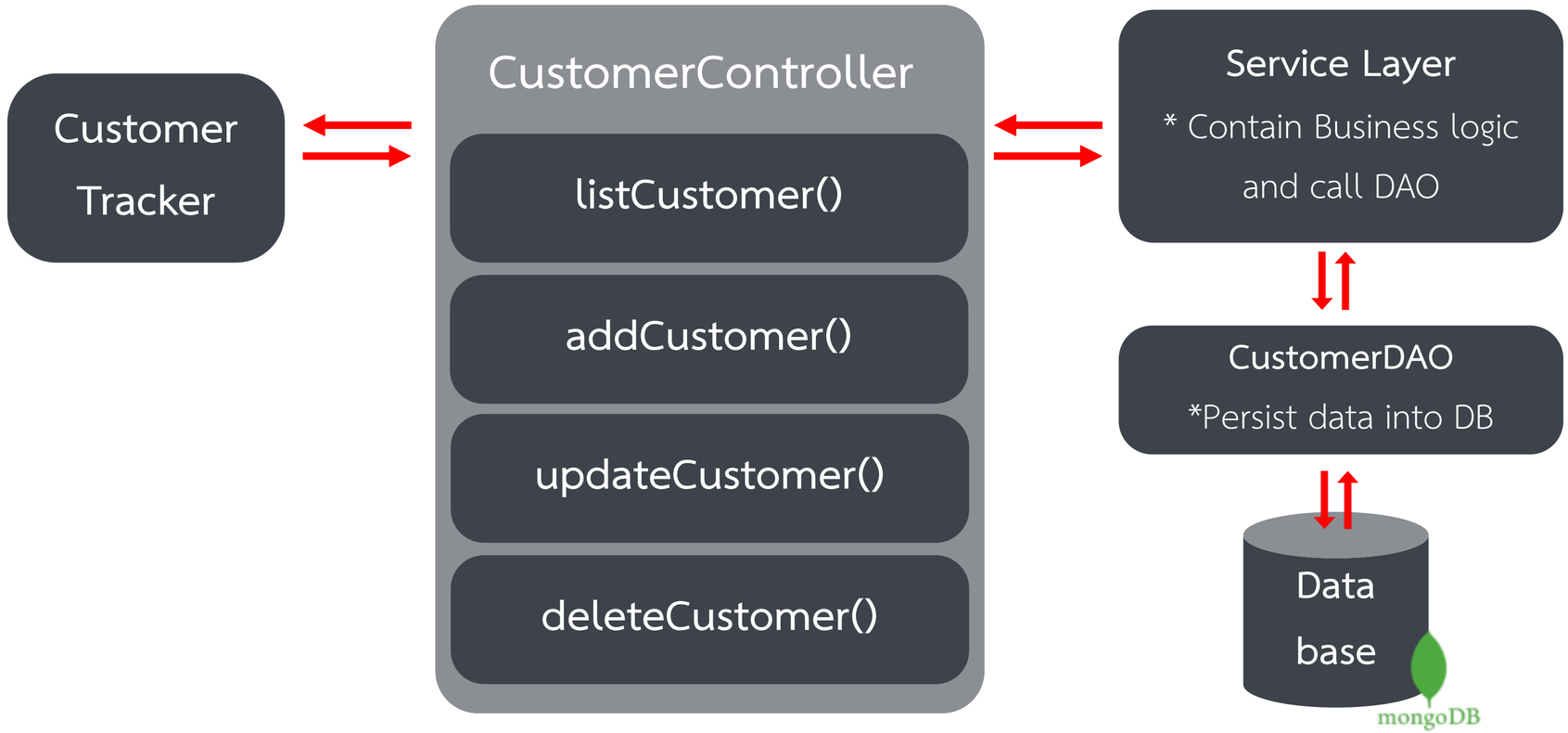
- ***Controller Layer*** มีหน้าที่จัดการกับ Request ที่เข้ามา, *การ Preprocessing และ Transform* ข้อมูลบางอย่างสำหรับ Presentation Layer หรือ Data Access Layer นอกจากนี้ ยังจัดการกับ Use Case ที่เกี่ยวข้อง

***แล้วแต่ ไม่แน่นอน**

สถาปัตยกรรมแบบ 4 เลเยอร์



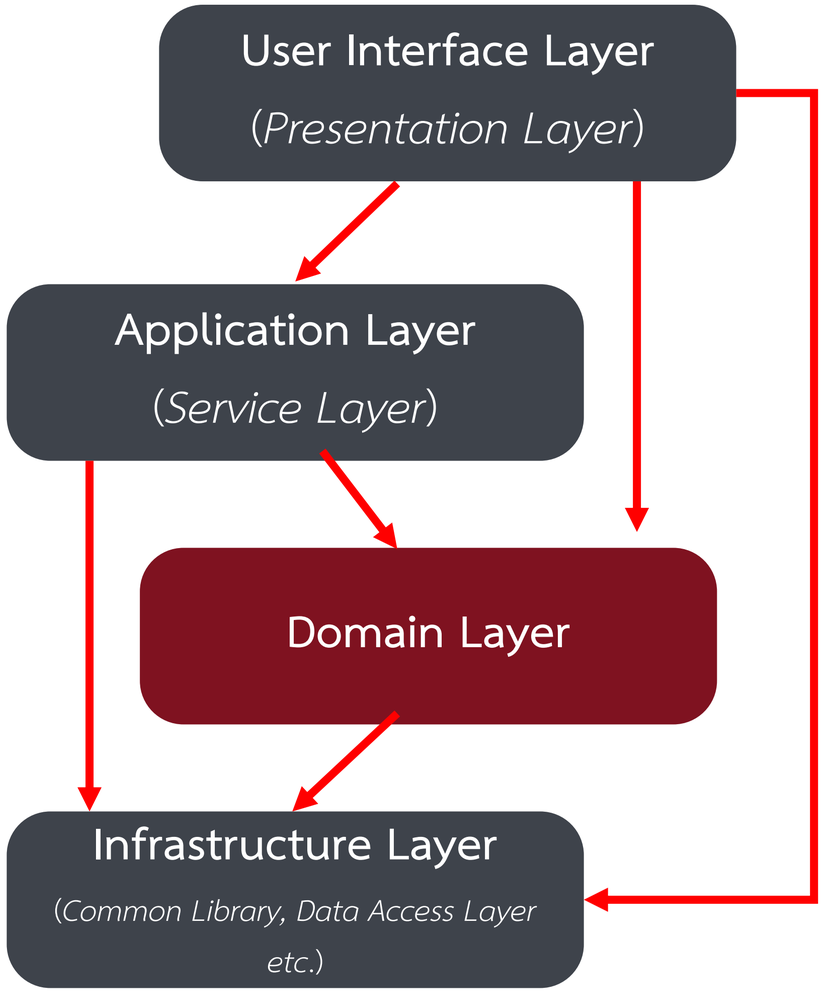
สถาปัตยกรรมแบบ 4 เลเยอร์



Outline

- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- Decomposition Strategies
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

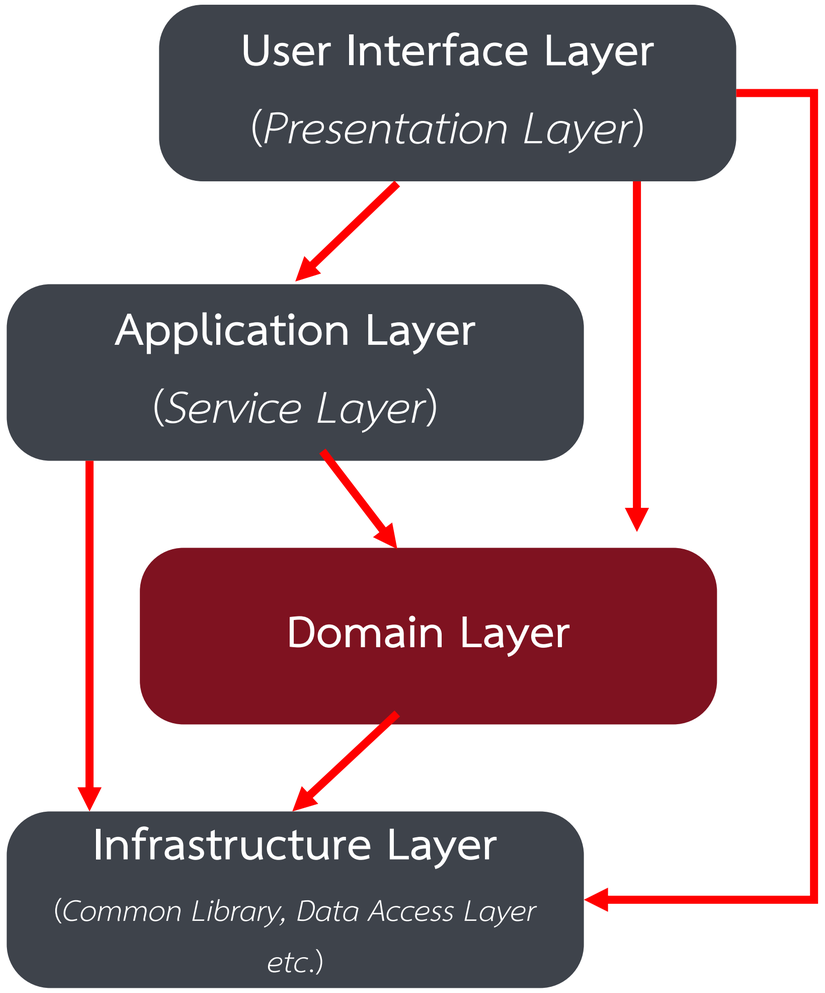
Domain-Driven Design (DDD)



Domain-Driven Design (DDD) เป็นสถาปัตยกรรมการพัฒนาซอฟต์แวร์ (Software Architecture) ที่ได้รับการแนะนำโดย Eric Evans ที่มีมาก่อน Agile และ Microservice การมองในรูปแบบที่อุดมคติที่เหมาะสมกับงานที่มองว่าเป็น Service มีขนาดเล็กและมีความคล่องตัวมาก

ซึ่งประกอบไปด้วย 4 เลเยอร์ ได้แก่ User Interface Layer, Application Layer, Domain Layer และ Infrastructure Layer โดยแต่ละเลเยอร์มีหน้าที่ดังนี้

Domain-Driven Design (DDD)



- *User Interface Layer* ทำหน้าที่เหมือนกับ Presentation Layer โดยนำเสนอข้อมูลและเป็นส่วนติดต่อกับผู้ใช้งาน
- *Application Layer* ทำหน้าที่เหมือนกับ Service Layer โดยจัดเก็บ Business Logic ไว้ แล้วจะเรียกใช้งาน DAL
- *Domain Layer* จะประกอบไปด้วย Entity, Value Object, Factory, Domain Service, Specification Pattern เป็นต้น
- *Infrastructure Layer* หน้าที่จะขึ้นอยู่กับมุมมองผู้ออกแบบ แต่โดยส่วนใหญ่สามารถมองได้ว่า Base Class, Common Library, Data Access Layer

Outline

- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- **Decomposition Strategies**
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

Outline

- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- Decomposition Strategies
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

Service Decomposition

Organization

System

Service Decomposition คือ กระบวนการแยกระบบที่มีขนาดใหญ่แบบ Monolithic ให้อยู่ในรูปแบบของ Service ย่อย ๆ หลาย ๆ อัน ซึ่งกระบวนการดังกล่าวไม่มีแนวทางปฏิบัติที่แน่นอน ในปี ค.ศ. 1967 ได้มีการนำเสนอ Conway's Law ขึ้น โดยมีใจความสำคัญว่า

> *“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.”*
>
> *Conway, M. E. (1968). How Do Committees Invent?. Datamation.*

กล่าวคือ ระบบจะออกแบบด้วยการแยกออกเป็นส่วน ๆ โดยลอกเลียนหรือคล้ายคลึงกับ

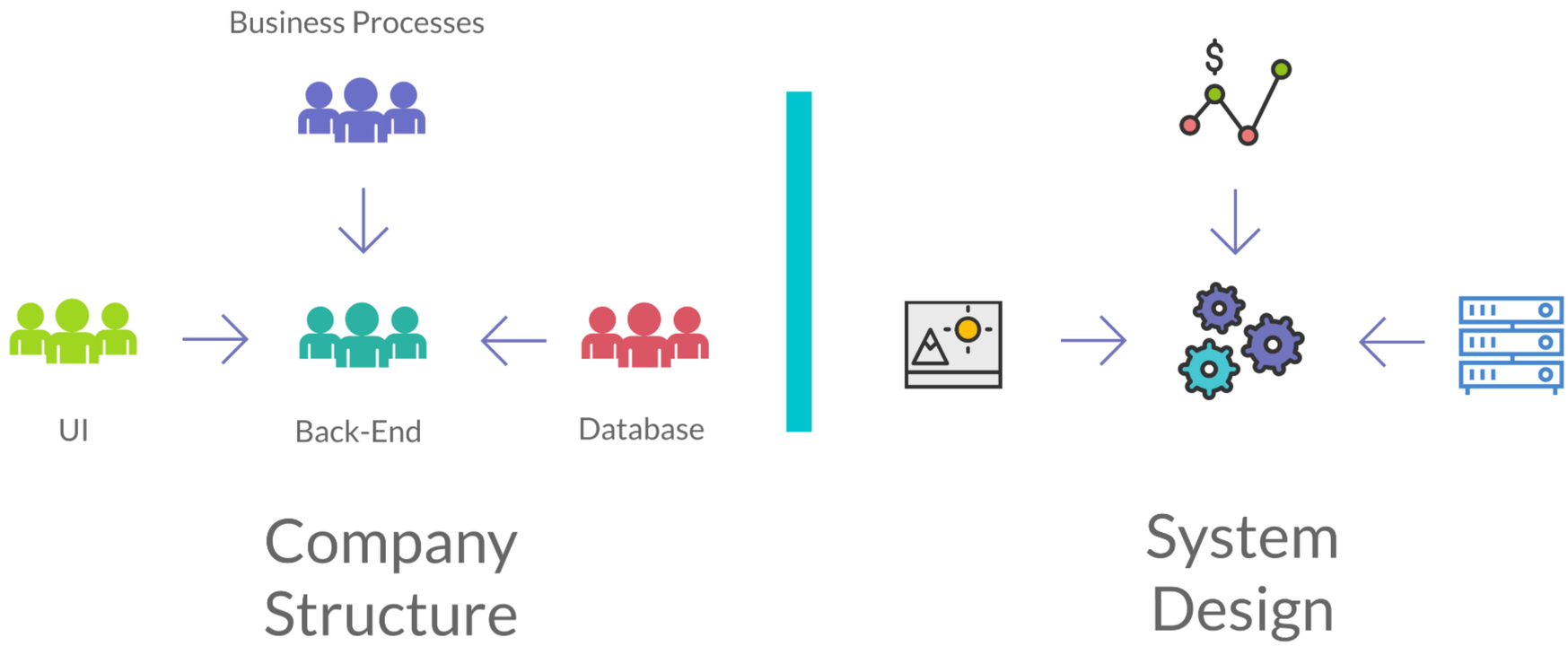
(1) วิธีการสื่อสารของคนหรือทีมในองค์กรนั้น ๆ (company communicate) หรือ

(2) โครงสร้างองค์กรในปัจจุบัน (current structure)

ซึ่งจะพบว่าการออกแบบข้างต้นซึ่งอาจจทำให้ได้รับระบบที่ล้าสมัยหรือไม่ค่อยมีคุณภาพ ซึ่งถือว่าล้าสมัยในยุคของ DevOps และ Agile นอกจากนี้ ยังทำให้ได้รับระบบที่เป็น Monolithic และ Tightly Coupled มาก

Service Decomposition

Conway's Law



อ้างอิง Conway's Law — LearnStuff.IO

Outline

- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- **Decomposition Strategies**
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

Service Decomposition

หลังจากนั้น เริ่มมีแนวความคิดที่จะพัฒนา Service ที่พึ่งพากันให้น้อยลง นักคอมพิวเตอร์ได้คิดค้นแนวคิดใหม่ คือ *Inverse Conway Manouvre Law* ซึ่งได้กล่าวไว้ว่า

> “Technology change is driving changing customer preferences and behavior, which in turn are driving organizational change across increasingly software-driven enterprises. The causality question behind Conway’s Law, therefore, is less about ***how changing software organizations can lead to better software***, but rather how companies can best leverage changing technology in order to transform their organizations,” ***says Bloomberg***.

หมายความว่า การเปลี่ยนแปลงโครงสร้างองค์กรไม่ได้ทำให้ได้รับระบบหรือโปรแกรมที่ดีหรือตอบโจทย์ขึ้น แต่เป็นการเปลี่ยนแปลงของเทคโนโลยีที่ส่งผลต่อพฤติกรรมของลูกค้ามากกว่าที่ส่งผลทำให้ได้รับโปรแกรมที่ดีและตอบโจทย์ภาคธุรกิจ อาทิเช่น Mobile Banking นอกจากนี้ ยังสามารถนำการเปลี่ยนแปลงด้านเทคโนโลยีไปใช้ปรับเปลี่ยนองค์กรให้มีประสิทธิภาพการทำงานที่ดีขึ้นได้

Service Decomposition

ซึ่ง***การมองตรงกันข้าม***จากแนวคิดของ Conway's Law จะพบว่า

- **Independent** ทำให้โครงสร้างของระบบมีความเป็นอิสระต่อกันมากขึ้น ลดการพึ่งพากันลง
- **Self-contained service** ทำงานจบในตัว Service เอง
- **Team working independently** ทีมที่พัฒนาทำงานได้อย่างเป็นอิสระ

ดังนั้น การออกแบบระบบตามแนวคิดของ Microservice ที่เน้นความรวดเร็วและความยืดหยุ่นในแง่ต่าง ๆ เนื่องจากพอแต่ละ Service ลดการพึ่งพากันและกันมากขึ้น ส่งผลให้ Service มีความเป็นอิสระต่อและทำให้ระบบสามารถเติบโตได้เร็วขึ้น จึงสามารถสรุปได้ว่าแนวคิด “Inverse Conway Manouvre Law” มีความเหมาะสมกับการออกแบบระบบตามแนวคิดของ Microservice มากกว่าแนวคิด “Conway Manouvre Law”

Service Decomposition

ซึ่งสามารถสรุปแบบคร่าว ๆ ได้ใน 3 มุมมอง ได้แก่ Service, Communication และ Team ดังต่อไปนี้

Service

- ✓ Service ต้อง**ประกอบมาจากชุดคำสั่งเล็ก ๆ (Function) หลายอัน**ที่ทำงานเกี่ยวข้องกันมาก ๆ
- ✓ Service ต้องได้รับการ**ออกแบบตามแนวคิด “Common Closure Principle”** กล่าวคือ สิ่งที่เปลี่ยนแปลงร่วมกันควรถูกรวมเข้าด้วยกัน เพื่อให้มั่นใจว่าการเปลี่ยนแปลงแต่ละครั้งจะส่งผลต่อ Service เดียวเท่านั้น

Comm

- ✓ Service ต่าง ๆ จะต้อง**เชื่อมโยงกันแบบ Loosely coupled** ซึ่งแต่ละ Service เป็น API และ Service สามารถ**เปลี่ยนแปลงได้แต่ต้องไม่ส่งผลกระทบต่อ Client หรือ Service** อื่นที่มาเรียกใช้งาน

Team

- ✓ แต่ละ Service ควร**มีขนาดเล็กพอที่จะพัฒนาได้ด้วยคนประมาณ 6 - 10 คน** (“two pizza”)
- ✓ **หนึ่งทีมควรพัฒนา Service อย่างน้อยหนึ่ง** Service (กรณีที่รับผิดชอบมากกว่า 1 Service แต่ละ Service ต้องเป็นอิสระต่อกัน)
- ✓ ทีมต้อง **Develop และ Deploy Service ของตนเอง** และมีการทำงานร่วมกันกับทีมอื่นน้อยที่สุด

Outline

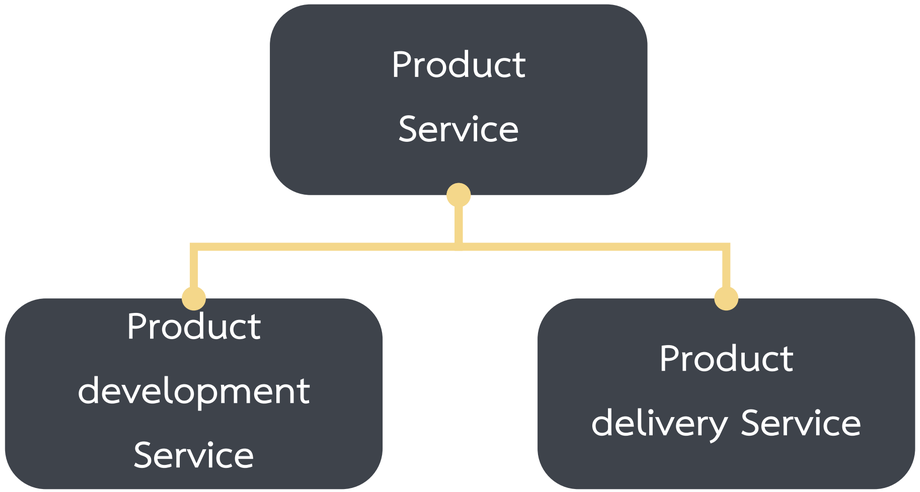
- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- **Decomposition Strategies**
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

Decompose by BUSINESS CAPABILITY Pattern (การแยกส่วนตามความสามารถทางธุรกิจ)

เป็นแนวคิดจากการออกแบบตามโครงสร้างและลักษณะของธุรกิจ โดยการแบ่งแยกหรือกำหนด Service ต่าง ๆ ได้จาก (1) ความสามารถทางธุรกิจที่ทำให้***สร้างมูลค่าเพิ่ม***ได้ หรือ (2) ***Entity*** ที่สอดคล้องหรือเกี่ยวข้องกับธุรกิจ อาทิเช่น

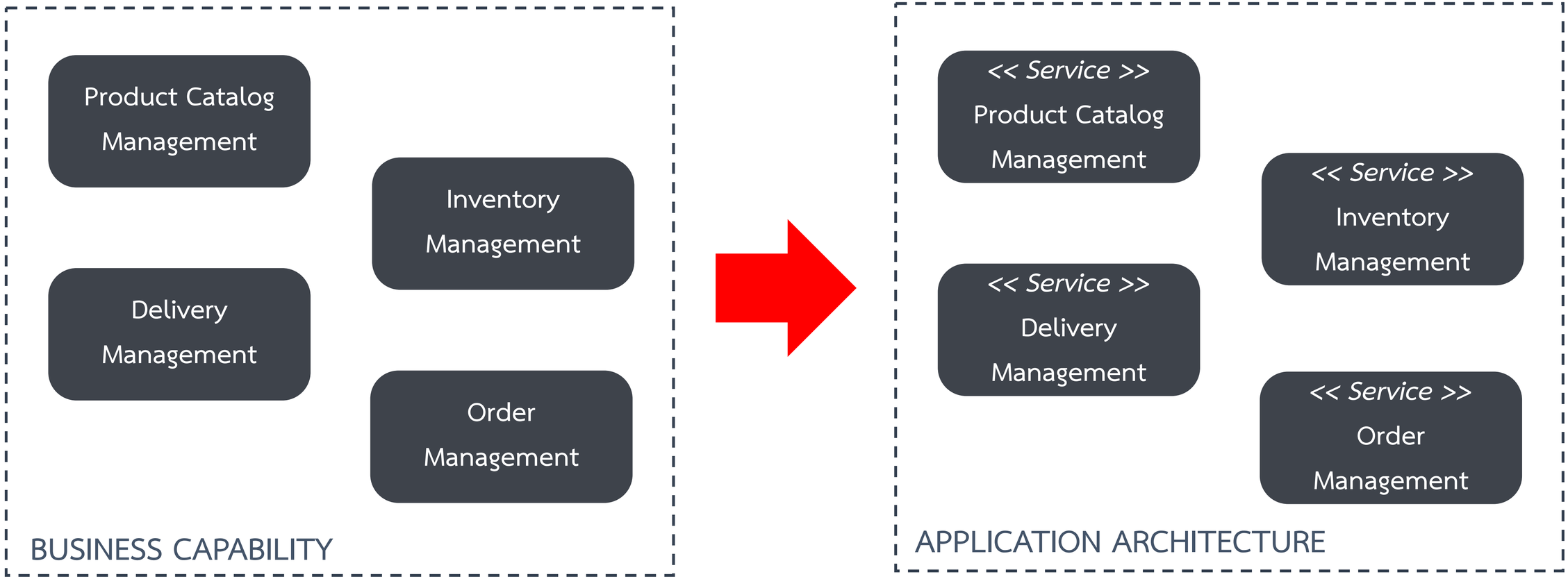
- ✓ Order Management มีหน้าที่รับผิดชอบคำสั่งซื้อ
- ✓ Customer Management มีหน้าที่รับผิดชอบต่อลูกค้า

ความสามารถทางธุรกิจมักจะมีโครงสร้างแบบ***ลำดับชั้นที่หลายระดับ*** ตัวอย่างเช่น



Decompose by BUSINESS CAPABILITY Pattern (การแยกส่วนตามความสามารถทางธุรกิจ)

ตัวอย่าง ร้านค้าออนไลน์ เมื่อได้รับการออกแบบตามแนวคิดของ Microservice และอาศัยการแยกส่วนตามความสามารถทางธุรกิจจะได้ผลลัพธ์ดังต่อไปนี้



Decompose by BUSINESS CAPABILITY Pattern (การแยกส่วนตามความสามารถทางธุรกิจ)

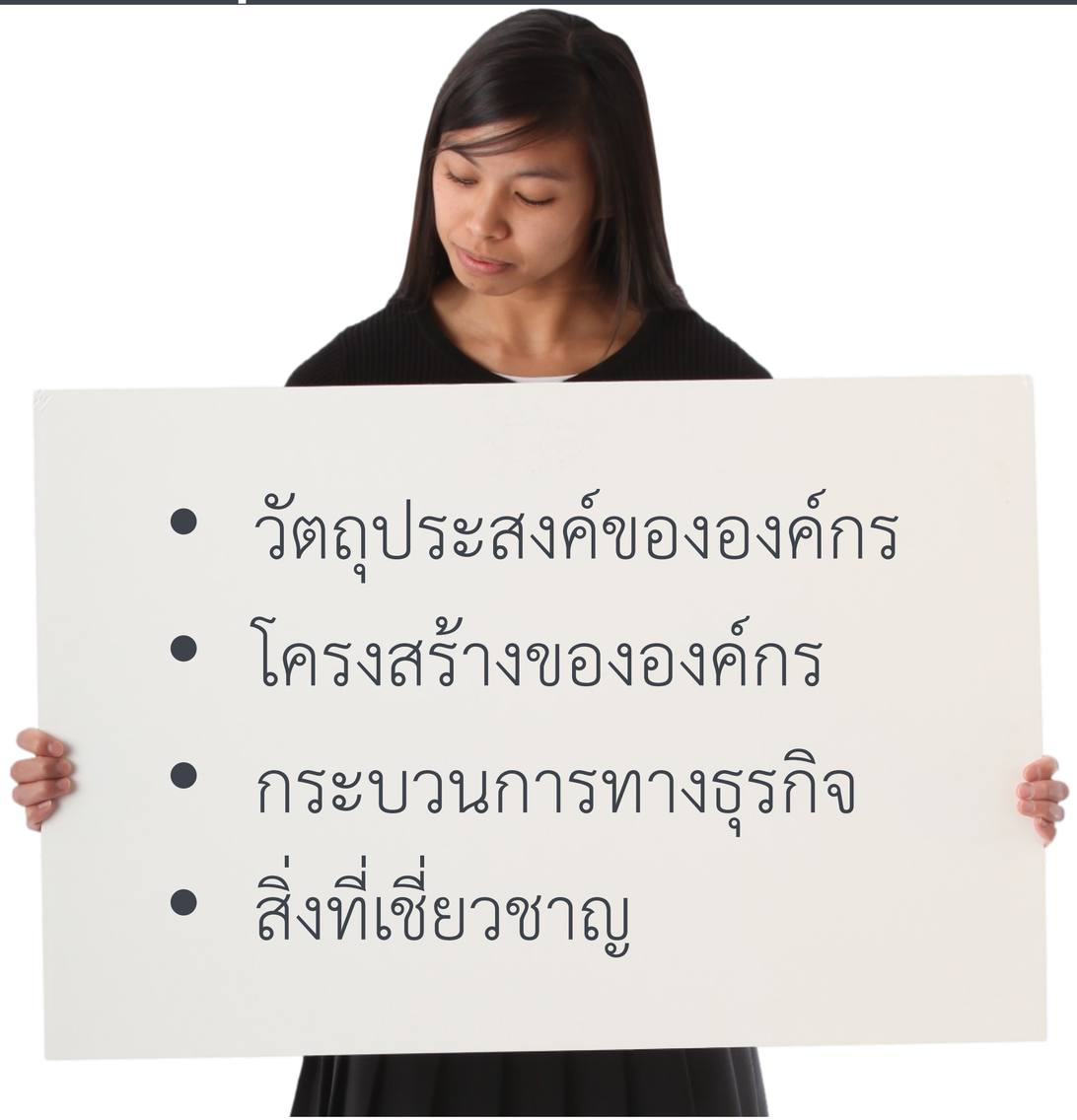
ข้อดีของการแยกส่วนตามความสามารถทางธุรกิจ ได้แก่

- เกิดความ Stable เนื่องจาก Business Logic มีความนิ่งหรือไม่ค่อยเปลี่ยนแปลงจะส่งผลให้ไม่ค่อยมีการปรับหรือแก้ไขระบบด้วย เป็นการเอาความต้องการของภาคธุรกิจเป็นหลัก
- ได้ส่งมอบระบบที่ตรงตามความต้องการทางธุรกิจมากกว่าทางเทคนิคหรือเทคโนโลยีสมัยใหม่ (บางครั้งคนพัฒนาเลือกใช้เทคโนโลยีสมัยใหม่ แต่ไม่ได้ประเมินความเสี่ยงหรือเลือกใช้แบบขาดความชำนาญ หรือเลือกใช้ไม่เหมาะกับภาคธุรกิจ)
- Service มีการติดต่อเพื่อแลกเปลี่ยนข้อมูลหรือร้องขอบริการแทนการทำงานภายในมากขึ้น
- Service มีความเกี่ยวข้องกันหรือสัมพันธ์กันแบบหลวม ๆ

Decompose by BUSINESS CAPABILITY Pattern

(การแยกส่วนตามความสามารถทางธุรกิจ)

หลักการในการระบุความสามารถทางธุรกิจควรเริ่มจากการ*ทำความเข้าใจ*ธรรมชาติและลักษณะการทำงานของธุรกิจก่อน โดย*การพิจารณา*จาก



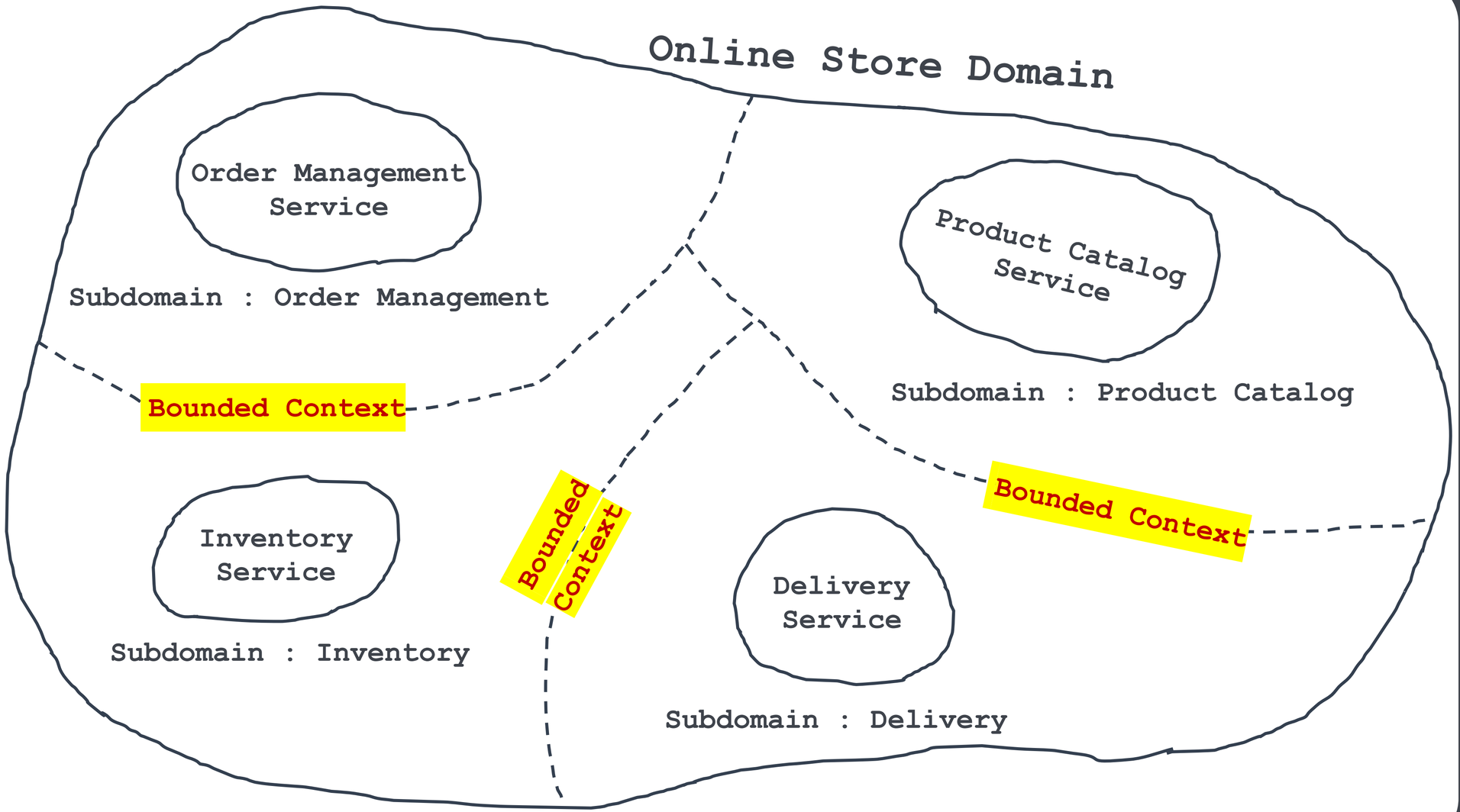
Outline

- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- **Decomposition Strategies**
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

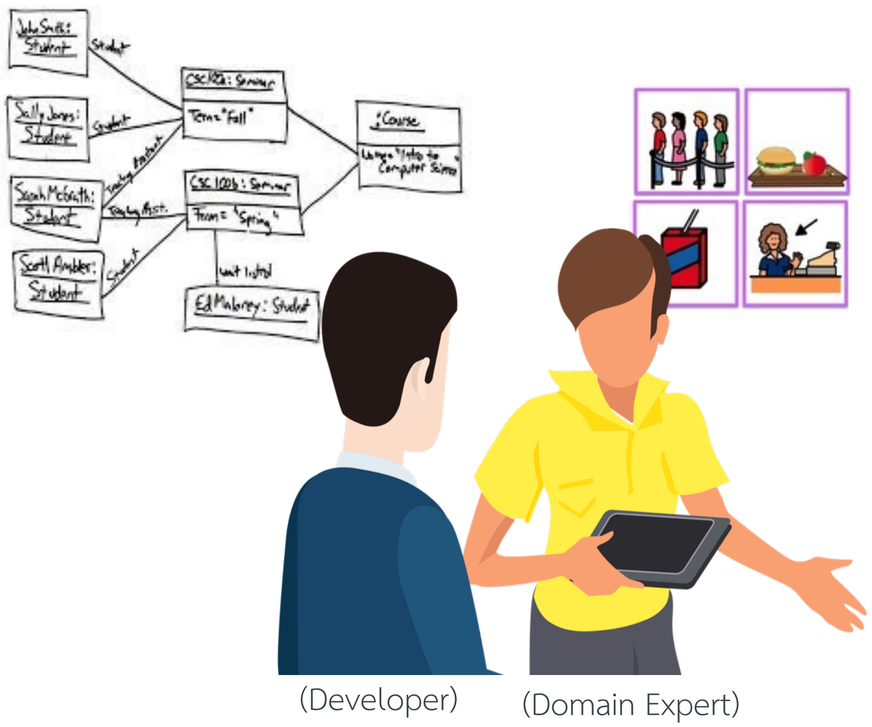
Domain-driven Design (DDD)

Decompose by DOMAIN OBJECTS Pattern

เป็นแนวคิดที่มองปัญหาของระบบและธุรกิจเป็นโดเมน (Domain) แต่ละโดเมนอาจจะประกอบด้วยหลายโดเมนย่อย ๆ ได้ (Subdomain) ซึ่งแต่ละโดเมนย่อยจะสอดคล้องกับส่วนต่าง ๆ ของธุรกิจ



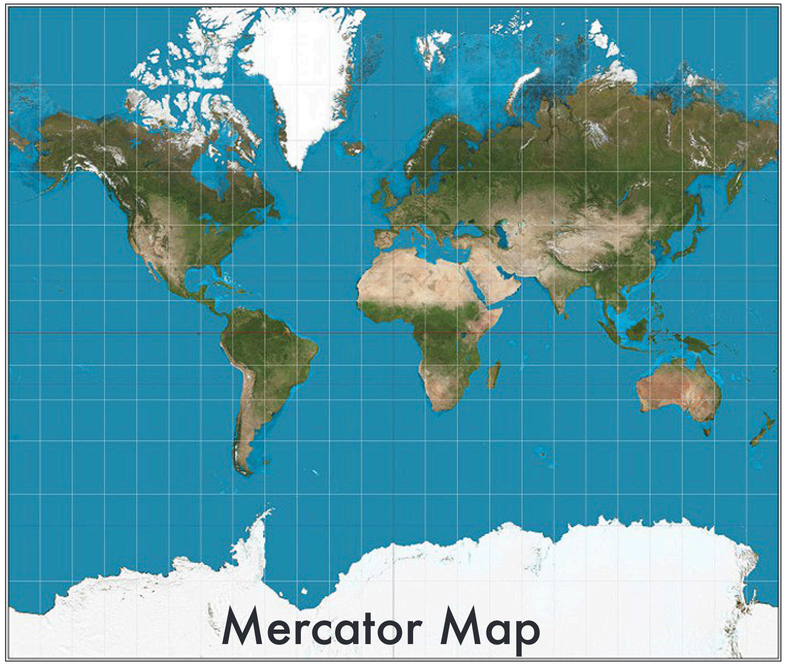
Domain-driven Design: DDD

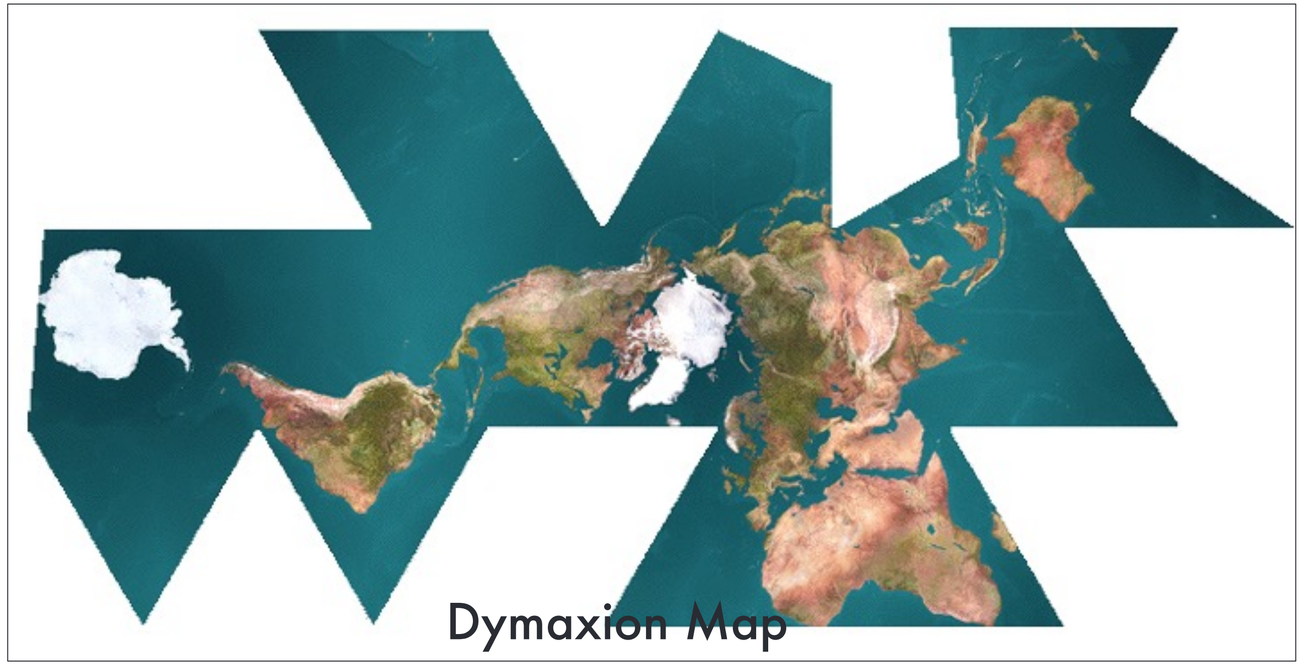


การออกแบบ Domain-Driven Design (DDD) เป็นวิธีการออกแบบระบบที่นิยมนำมาใช้กับสถาปัตยกรรมแบบ Microservice ที่ได้รับการแนะนำโดย Eric Evans ซึ่ง DDD เป็นการออกแบบที่คาดหวังให้ผู้พัฒนา (Developer) สื่อสารกับผู้เชียวชาญเฉพาะด้าน (Domain Expert) ให้เข้าใจตรงกัน ทำให้ผู้พัฒนาสามารถมองระบบในมุมมองเดียวกันกับที่ผู้เชียวชาญเฉพาะด้านมอง นอกจากนี้ ยังช่วยให้ผู้พัฒนาออกแบบระบบ กระบวนการทำงาน และภาษา (ศัพท์เทคนิค) ให้สอดคล้องกับที่ผู้เชียวชาญเฉพาะด้านใช้งาน ซึ่งผู้เชียวชาญเฉพาะด้านแต่ละคนเข้าจะมี (1) มุมมองต่องาน (2)แนวคิดต่าง ๆ (3) ขั้นตอนการดำเนินงาน (4) หลักการที่ใช้ตัดสินใจ (5) การเรียกสิ่งต่าง ๆ (ภาษา) ต่องานแตกต่างกัน ซึ่งสิ่งเหล่านี้มักถูกเรียกว่าแบบจำลอง (Model)

ผู้เชียวชาญเฉพาะด้าน (Domain Expert) คือ ใครบางคนในภาคธุรกิจที่มีความรู้ ความเข้าใจ หรือ ความชำนาญในส่วนที่นักพัฒนากำลังพัฒนาอยู่ (Business logic)

Domain-driven Design: DDD

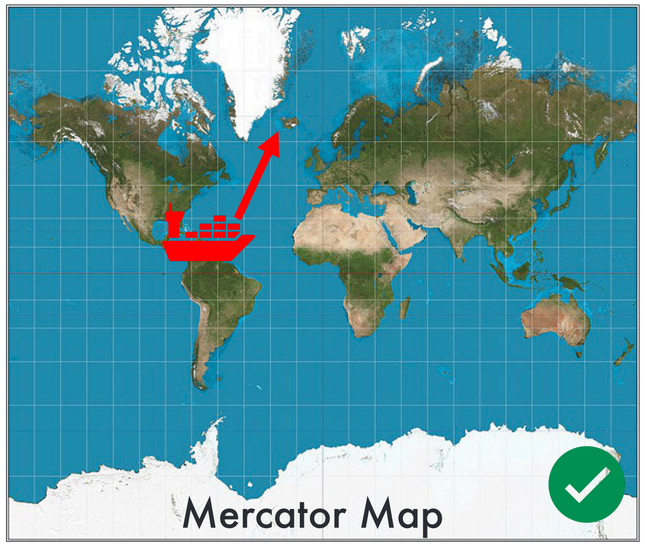


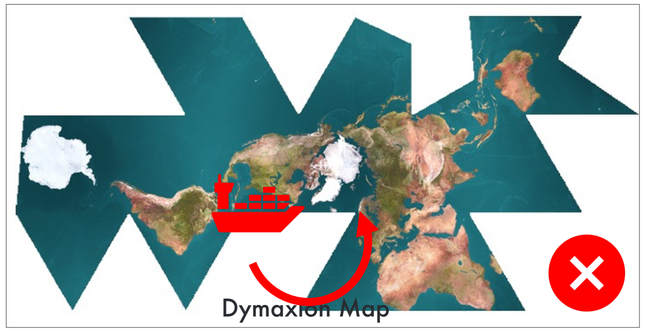


อ้างอิง https://futuremaps.com/blogs/news/top-10-world-map-projections

Eric Evans ได้ยกตัวอย่างว่า แผนที่โลก 2 แบบ ข้างต้นเปรียบเสมือนแบบจำลองที่แตกต่างกันและเหมาะสมกับงานคนละชนิดกัน โดยที่ แผนที่แบบ Mercator เหมาะสำหรับการใช้เพื่อนำทางสำหรับการเดินทาง (เดินเรือ) ขณะที่ แผนที่แบบ Dymaxion เป็นรูปแบบที่แสดงอัตรส่วนและขนาดของแต่ละทวีปตามความเป็นจริง

Domain-driven Design: DDD



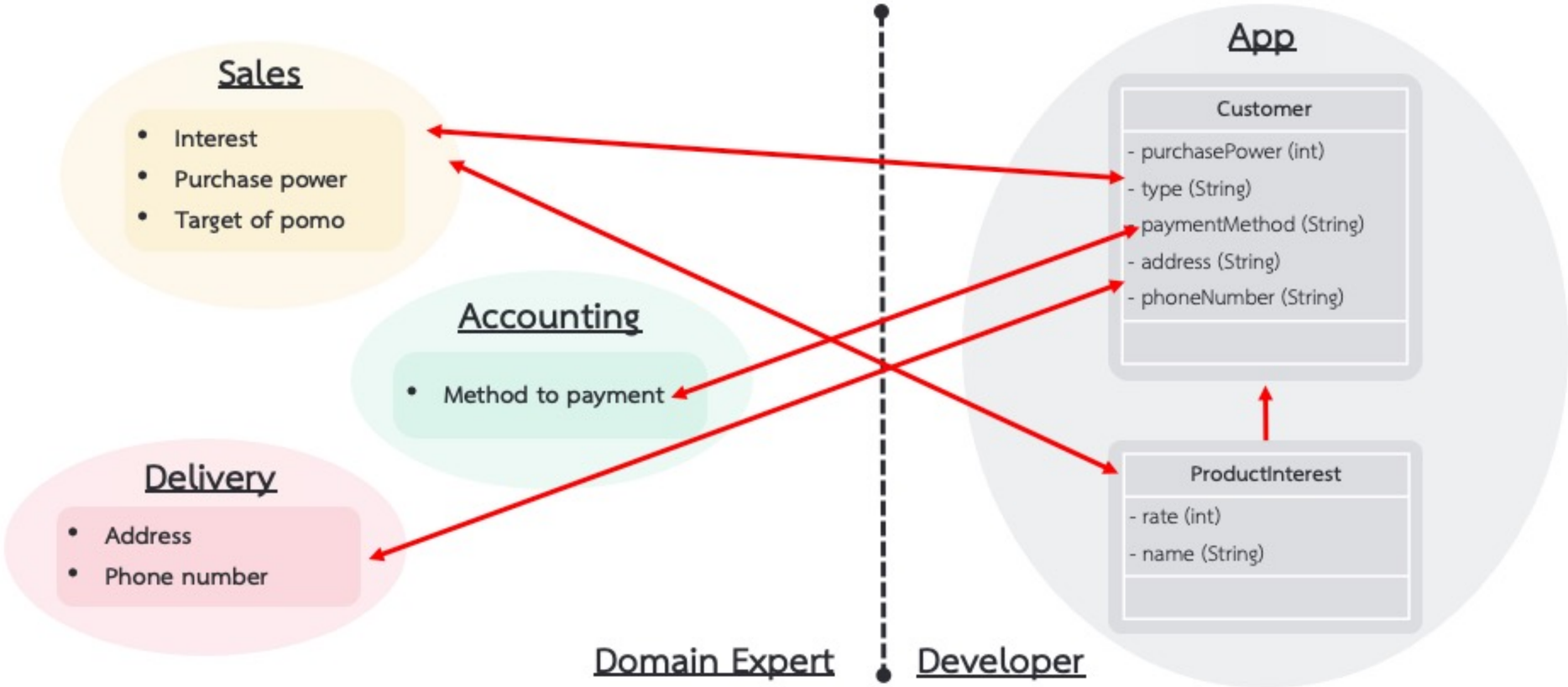


ถ้านักศึกษาลองมองว่าคลาสไดอะแกรมที่ออกแบบสำหรับระบบ Microservice ก็เปรียบเสมือนแผนที่ Mercator และ Dymaxion ที่สามารถสะท้อนคุณสมบัติของข้อมูลในระบบงานออกมาได้แตกต่างกัน ทำให้งานบางอย่างง่ายขึ้นและบางอย่างยากขึ้น ด้วยเหตุนี้ นักศึกษาควรออกแบบระบบตาม “*ปัญหาที่ต้องการแก้ไข ไม่ใช้ออกแบบตามข้อจำกัดของเทคโนโลยี หรือออกแบบตามฐานข้อมูล*” เพราะความเป็นจริงแล้วผู้เชียวชาญมีประสบการณ์มานานทำให้เขาทราบว่าปัญหาของงานไหนควรจะต้องมองในมุมมองหรือพิจารณาปัญหานี้ในรูปแบบใด ดังนั้น การออกแบบระบบด้วย DDD จึงสนับสนุนให้ผู้พัฒนาและผู้เชียวชาญออกแบบระบบร่วมกันเสมอ

Domain-driven Design: DDD

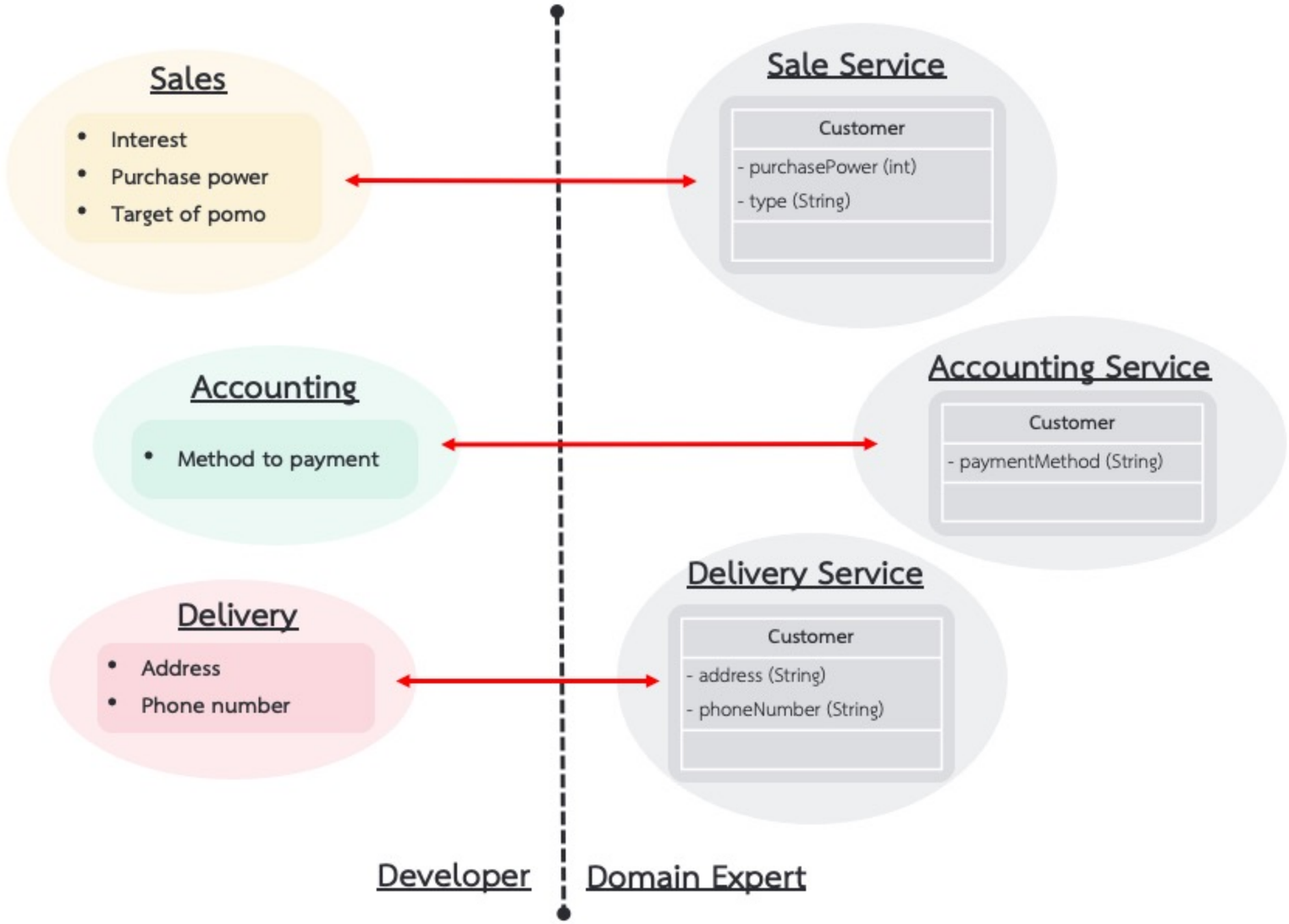
นอกจากนี้ การสื่อสารด้วยภาษาหรือคำศัพท์แบบเดียวกันจะช่วยทำให้การสื่อสารชัดเจนมากขึ้น อาทิเช่น ในระบบงานใหญ่มีความเป็นไปได้ที่จะต้องทำงานร่วมกับ Domain expert หลายคน ซึ่งแต่ละคนอาจจะใช้คำศัพท์ที่แตกต่างกัน หรือมีมุมมองต่อสิ่งใดสิ่งหนึ่งแตกต่างกัน เพราะพวกเขารับผิดชอบงานคนละส่วนกัน จากตัวอย่างข้างต้น จะพบว่าคำว่า Customer ของแต่ละ Domain expert จะมีความหมายแตกต่างกันไป ทั้งนี้ ขึ้นอยู่กับบริบท (Context) ของแต่ละคน

Domain-driven Design: DDD



ส่วนใหญ่นักพัฒนาพยายามเอาความหมายของคำว่า Customer ในแต่ละ **Context มารวมกันเพื่อสร้างโมเดลของ Customer** ที่ใกล้เคียงกับโลกความเป็นจริงขึ้นมา แต่นักศึกษาควรคิดก่อนว่าถ้าทำแบบนี้แล้วโมเดลที่ได้รับจะเหมาะสมกับปัญหานั้น ๆ หรือไม่ เพราะแต่ละโมเดลเหมาะสมหรือสามารถตอบโจทย์ปัญหาได้แตกต่างกันไป

Domain-driven Design: DDD



ดังนั้น ถ้านักศึกษาออกแบบโมเดลตามภาษาและมุมมองของ Domain Expert จะพบว่าโมเดลที่ได้จะสะท้อนการใช้งานในมุมมองที่ถูกต้องและเหมาะสมของแต่ละ Context ได้ดีกว่า นอกจากนี้ ยังช่วยให้เราสื่อสารกับ Domain Expert ได้ดีขึ้น

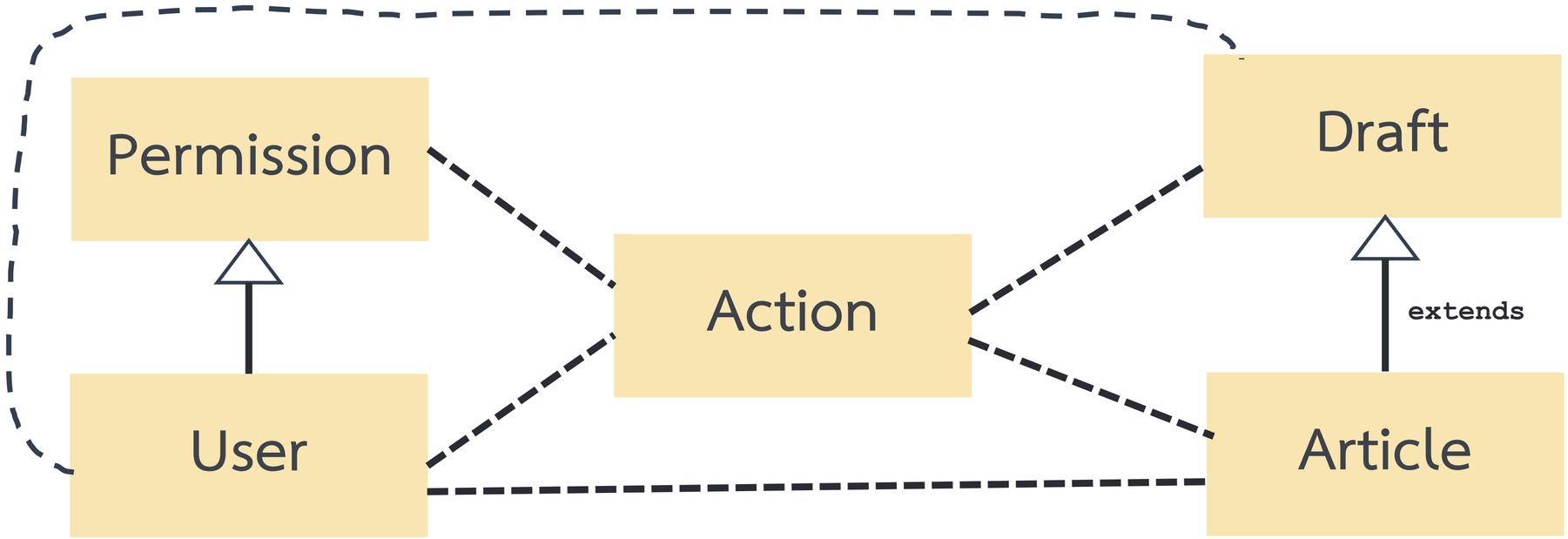
Domain-driven Design: DDD

กรณีศึกษา เว็บไซต์เขียนบทความ ถ้าผู้ใช้งานแจ้งความต้องการดังต่อไปนี้

- ผู้เขียนบทความ (Writer)
  - สามารถเพิ่ม แก้ไข และลบร่างบทความ (Draft) ได้
  - สามารถเพิ่มผู้ตรวจทานบทความ (Reviewer)
  - สามารถตีพิมพ์บทความฉบับร่าง (Draft) ไปเป็นฉบับจริง (Article) ได้
- ผู้ตรวจทาน (Reviewer)
  - สามารถแก้ไขบทความฉบับร่าง (Draft) ได้
  - สามารถให้ให้ข้อเสนอแนะ (Comment) ต่อบทความฉบับร่าง (Draft) ได้
- ผู้อ่าน (Reader)
  - สามารถอ่านบทความฉบับจริง (Article)
  - สามารถให้ให้ข้อเสนอแนะ (Comment) ต่อบทความฉบับจริง (Article) ได้

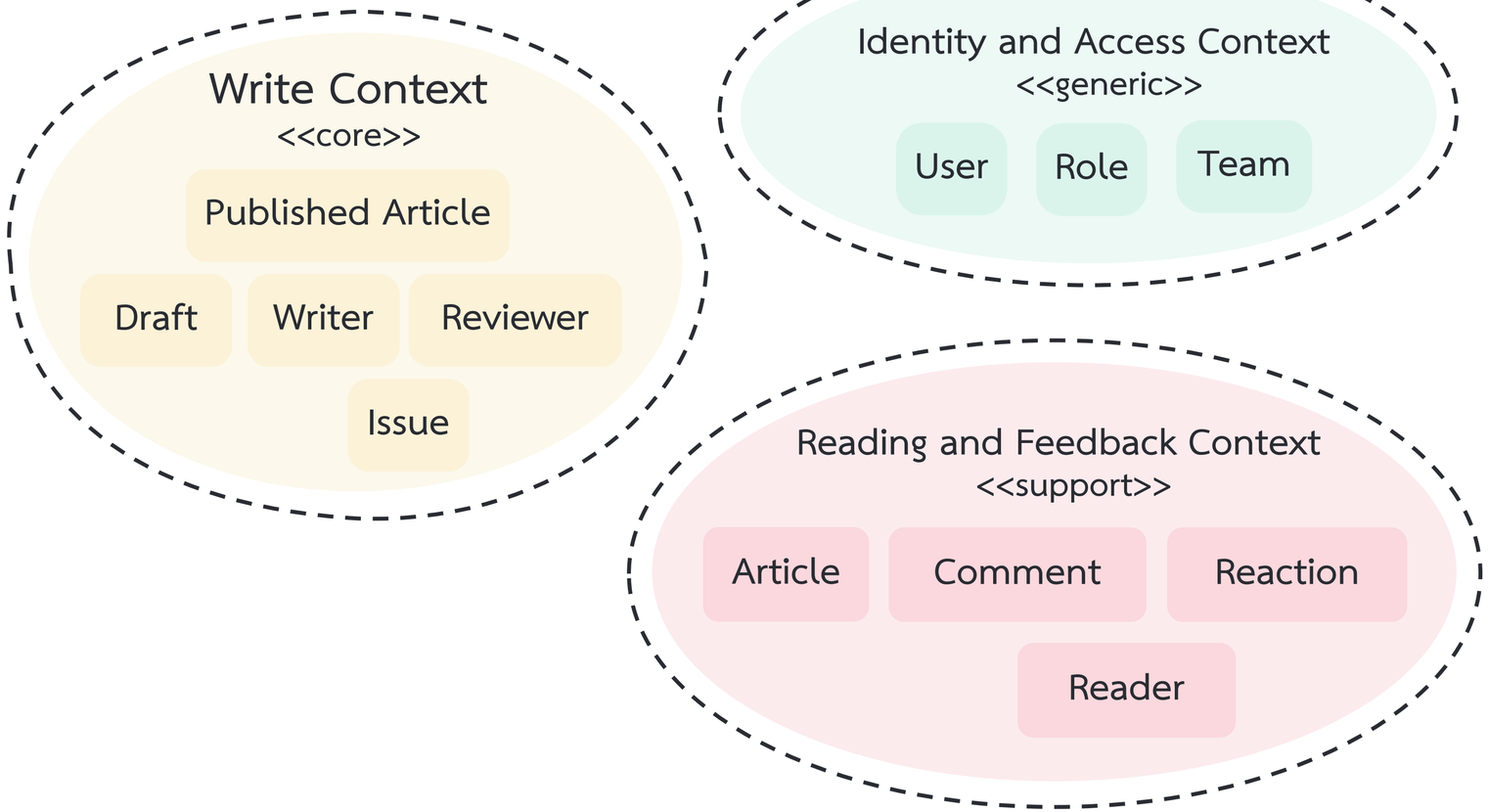
Domain-driven Design: DDD

กรณีศึกษา เว็บไซต์เขียนบทความ ถ้าผู้ใช้งานแจ้งความต้องการดังต่อไปนี้



ถ้านักศึกษานำความต้องการมาออกแบบเป็นคลาสไดอะแกรมทันทีอาจส่งผลทำให้ได้รับระบบที่ไม่ตอบโจทย์กับปัญหา เพราะการออกแบบด้วยคลาสไดอะแกรมมันจะอยู่บทสมมติฐานที่ว่า (1) ทุกคลาสสามารถ Interact ต่อกันได้อย่างอิสระ และ (2) ไม่มี context หรือ Boundary มากำกับ

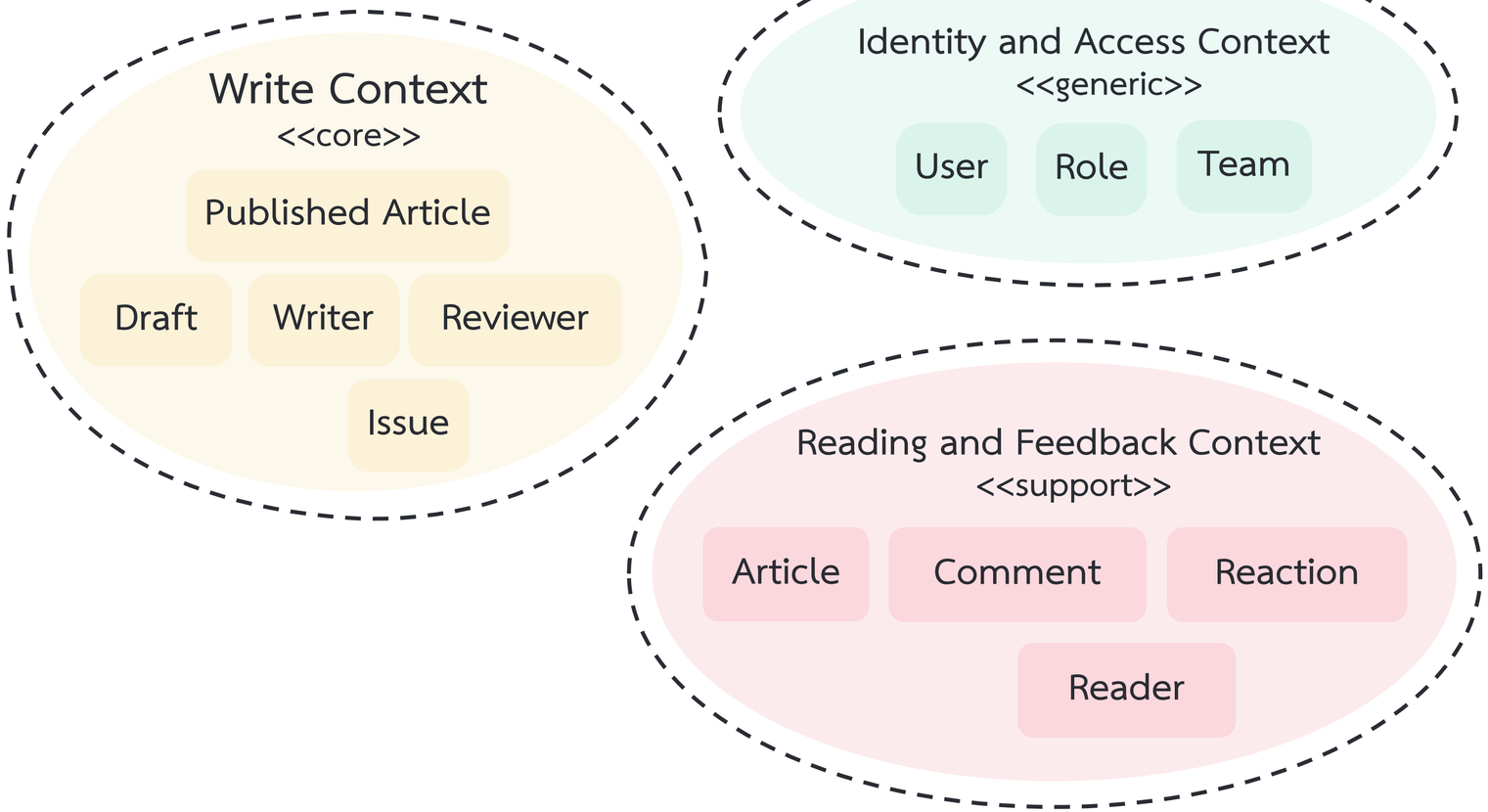
Domain-driven Design: DDD



ดังนั้น การออกแบบด้วย DDD นักศึกษาควรระบุให้ได้ก่อนว่าอะไรคือ

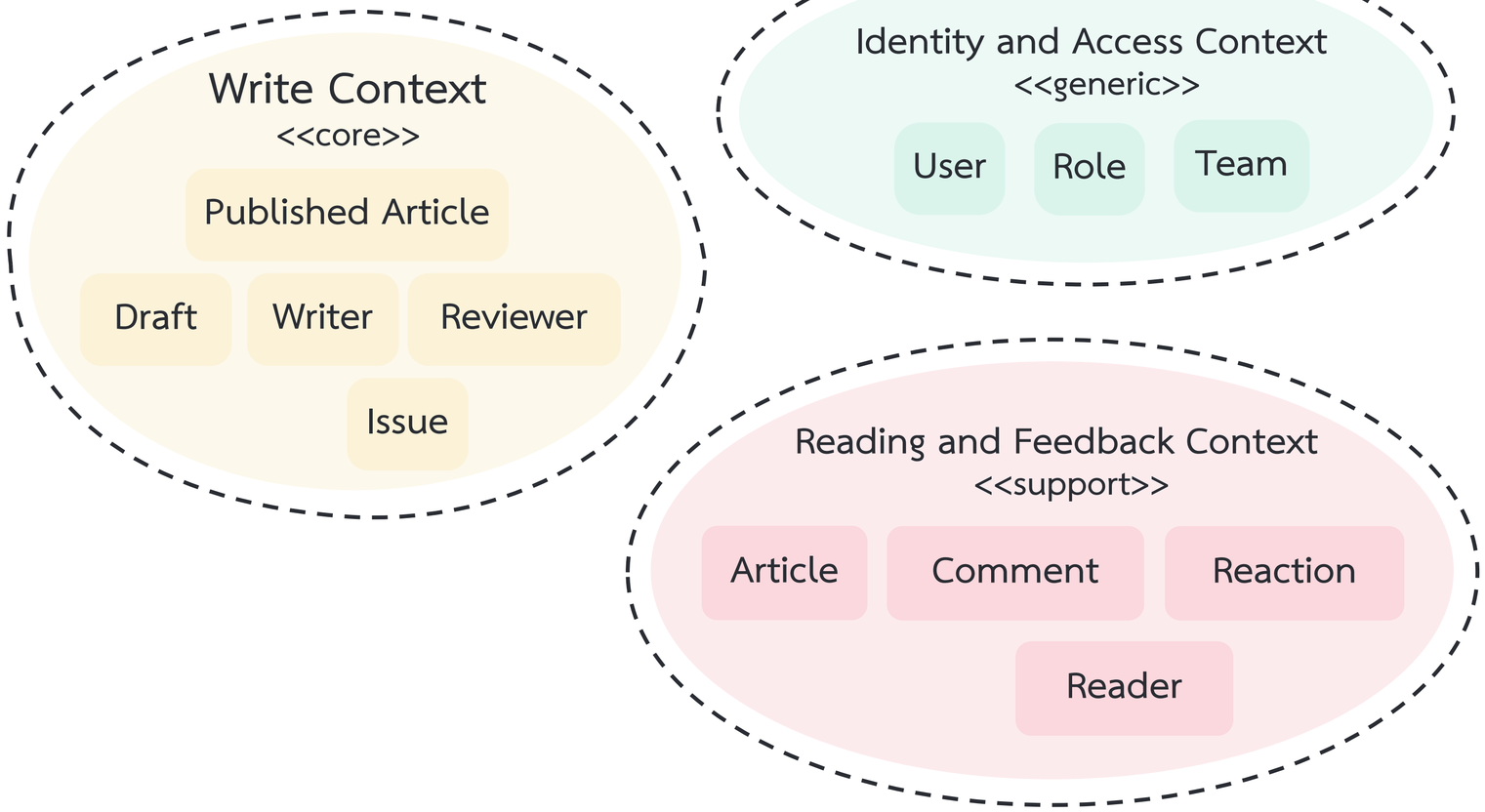
- Subdomain
- Bounded Context
- คำที่เรียกใช้

Domain-driven Design: DDD



Subdomain คือ ส่วนย่อยของโดเมนมีแนวความคิดที่ว่า “*ในระบบต่าง ๆ จะมีส่วนที่รับผิดชอบหรือทำหน้าที่เกี่ยวกับกฎกติกาและข้อมูลทางธุรกิจ ซึ่งเรียกว่า Domain Model โดยมีหลายส่วนประกอบเข้าด้วยกันและทำงานร่วมกันเพื่อให้ได้ผลลัพธ์ตามที่ต้องการ*”

Domain-driven Design: DDD

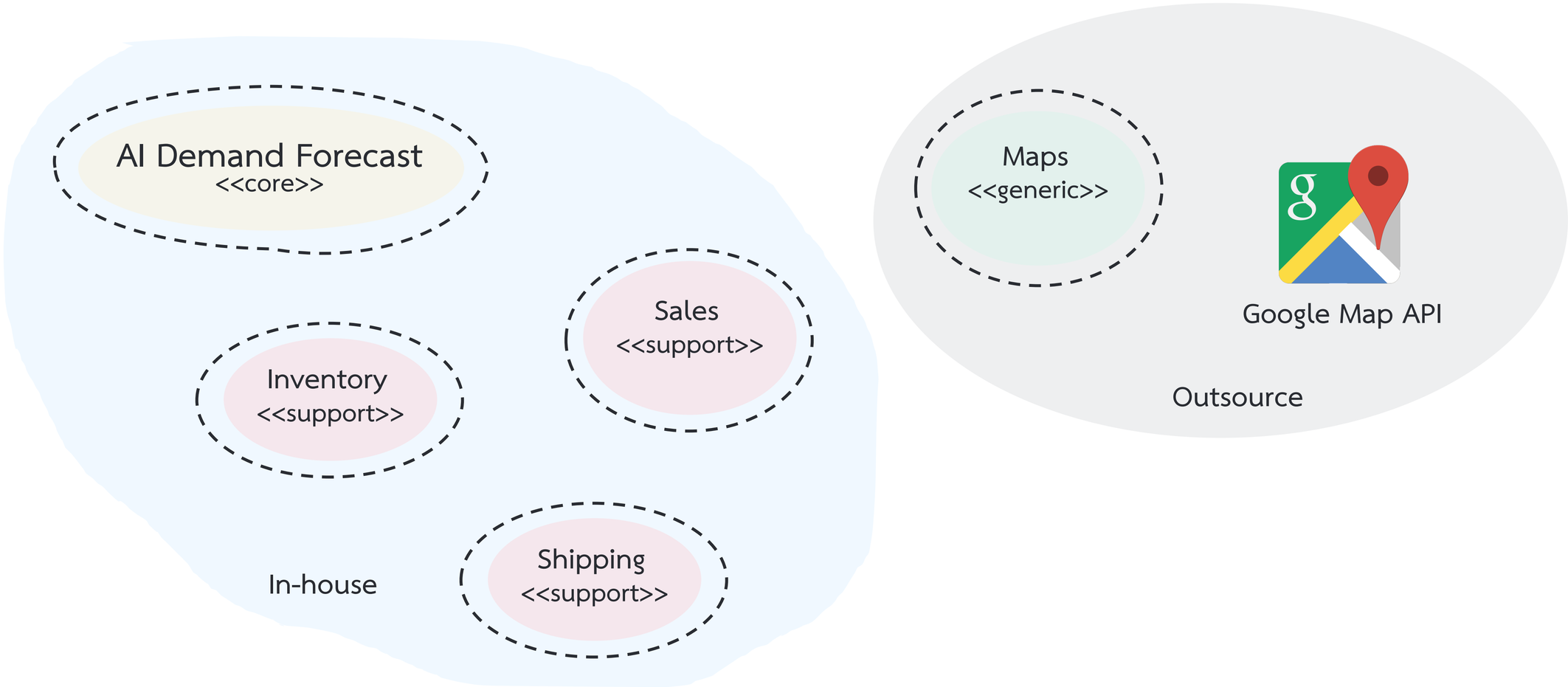


Subdomain สามารถแบ่งออกเป็น 3 ประเภท ได้แก่

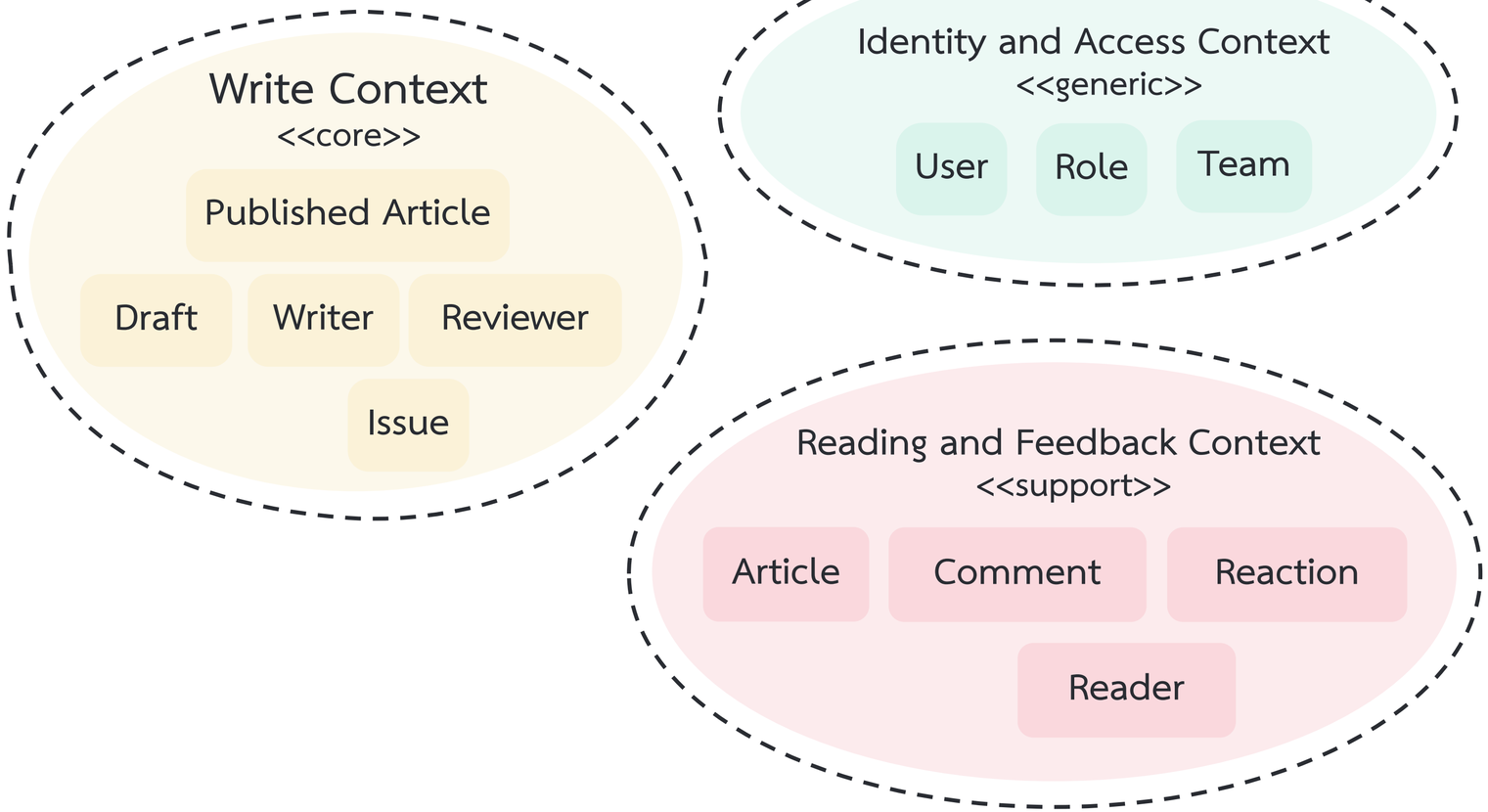
- **Core** คือ ส่วนสำคัญของระบบที่ทำให้ธุรกิจได้เปรียบคู่แข่ง หรือเป็นงานหลักของธุรกิจ
- **Support** คือ ส่วนประกอบที่ทำให้ core สมบูรณ์
- **Generic** คือ ส่วนทั่วไปของระบบสามารถหาซื้อได้หรือจ้าง outsource ได้

Domain-driven Design: DDD

ตัวอย่าง ระบบ Online Shopping ที่ระบบต้องสามารถทำนายความต้องการซื้อของลูกค้าได้



Domain-driven Design: DDD



Bounded Context

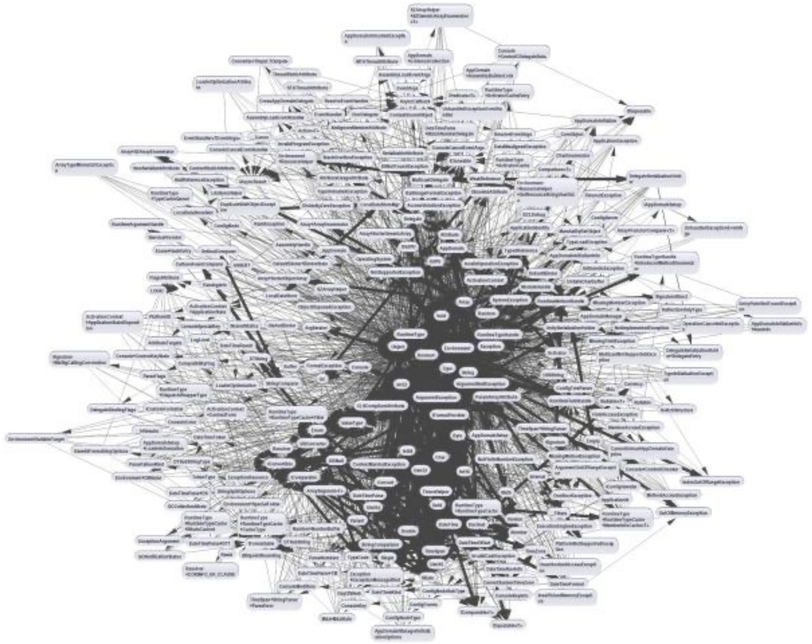
การระบุส่วนประกอบต่าง ๆ ได้นั้นจะอาศัยการพิจารณาที่ “Bounded Context” กล่าวคือ ***ขอบเขตของโดเมนที่รวม Business Logic และมี Data เพียงพอ***ต่อการทำงานเพื่อแก้ปัญหาของ Domain นั้น ซึ่งแต่ละ Domain มีหน้าที่และ***ความรับผิดชอบแตกต่างกัน***และจะ***ไม่ข้ามขอบเขตกัน***

โดยส่วนใหญ่ ถ้า subdomain เป็นส่วนของปัญหา แสดงว่า Bounded Context ควรเป็นส่วนของคำตอบ

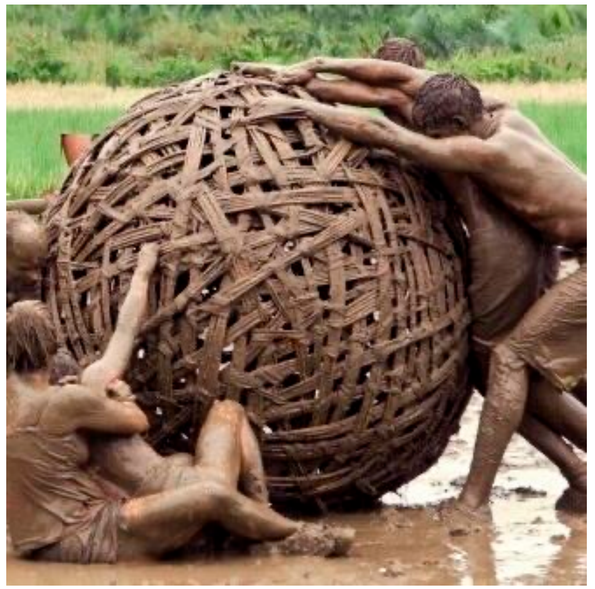
Domain-driven Design: DDD

Bounded Context

อย่างไรก็ตาม ถ้านักศึกษาออกแบบระบบไม่ได้ ถ้าจะเกิดเหตุการณ์ที่เรียกว่า “Big Ball of Mud” กล่าวคือ Component ต่าง ๆ เกิด dependency ไปทั่ว ซึ่งสามารถเกิดได้ทั้งระบบแบบ Monolith และ Microservice ที่ไม่ได้รับการกำหนด Bounded Context ที่ชัดเจน



https://twitter.com/mariofusco/status/1112332826861547520?lang=zh-Hant

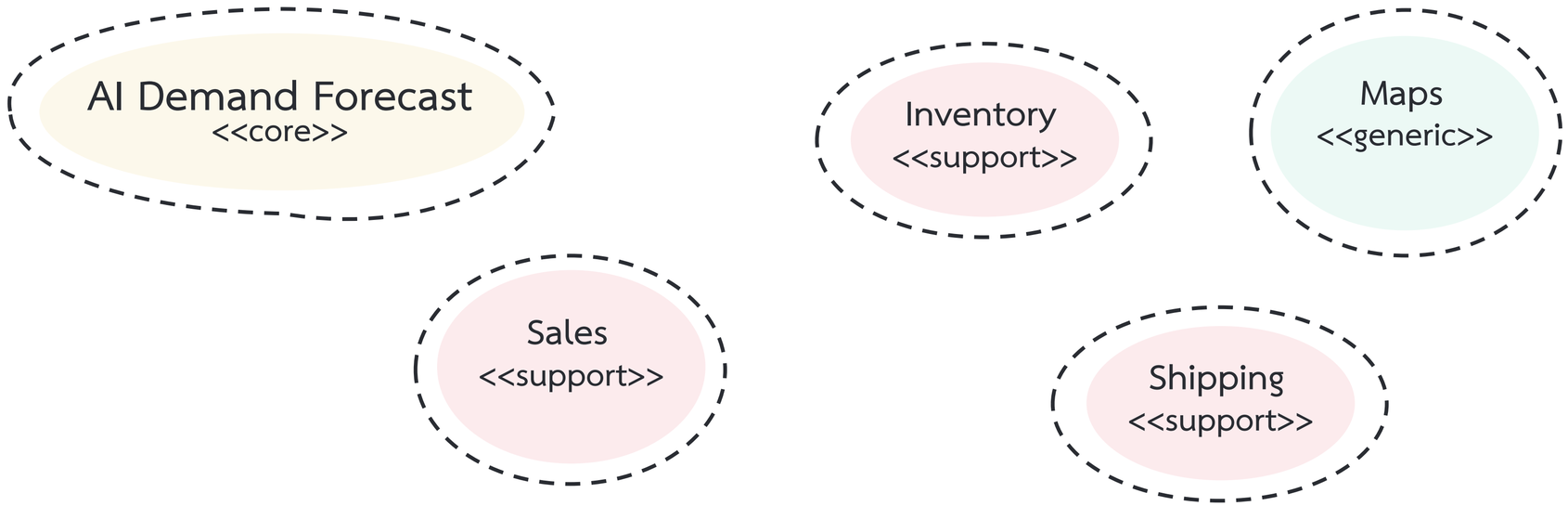


https://twitter.com/emendasoftware/status/462137921991376896

Domain-driven Design: DDD

Bounded Context

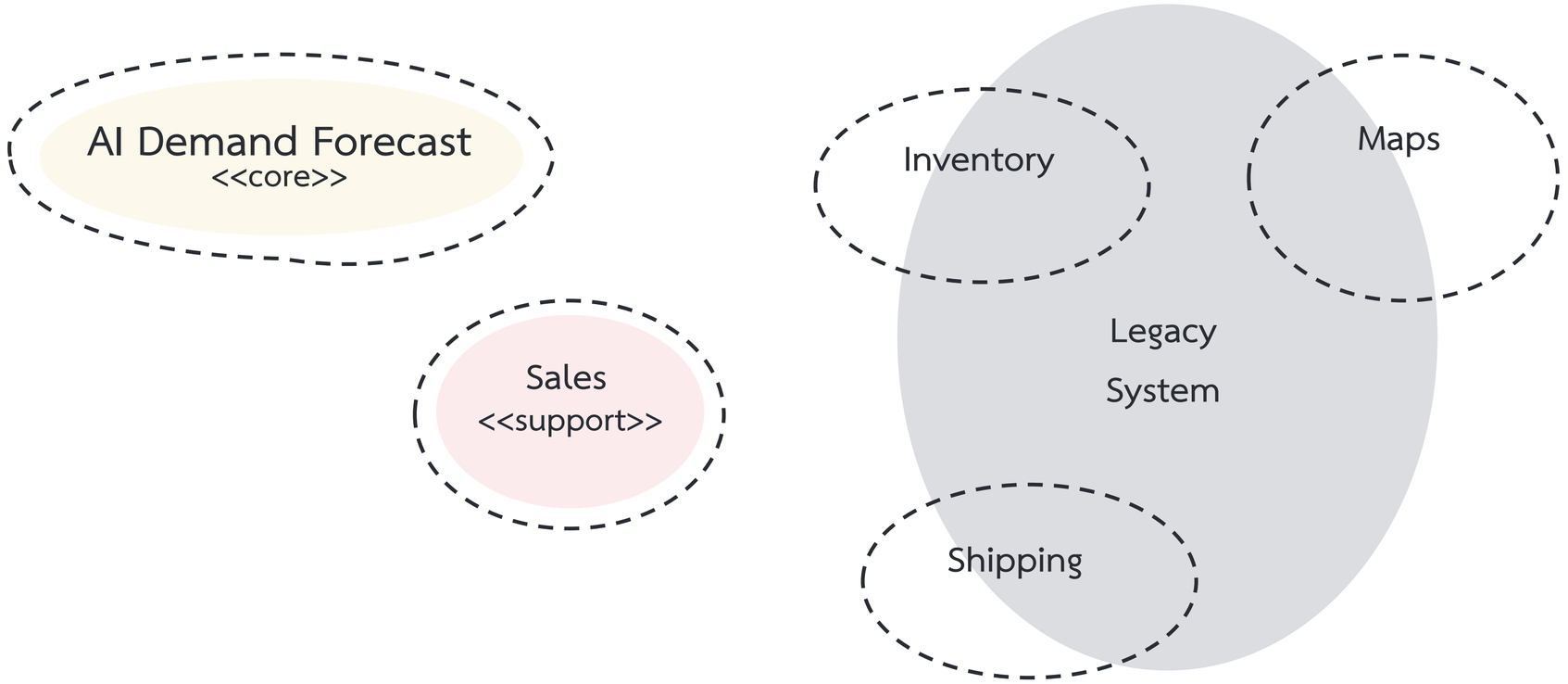
ซึ่งการออกแบบที่ดี นักศึกษาควรออกแบบให้ 1 subdomain ควรมีเพียง 1 Bounded Context และควรมี 1 Microservice เท่านั้น



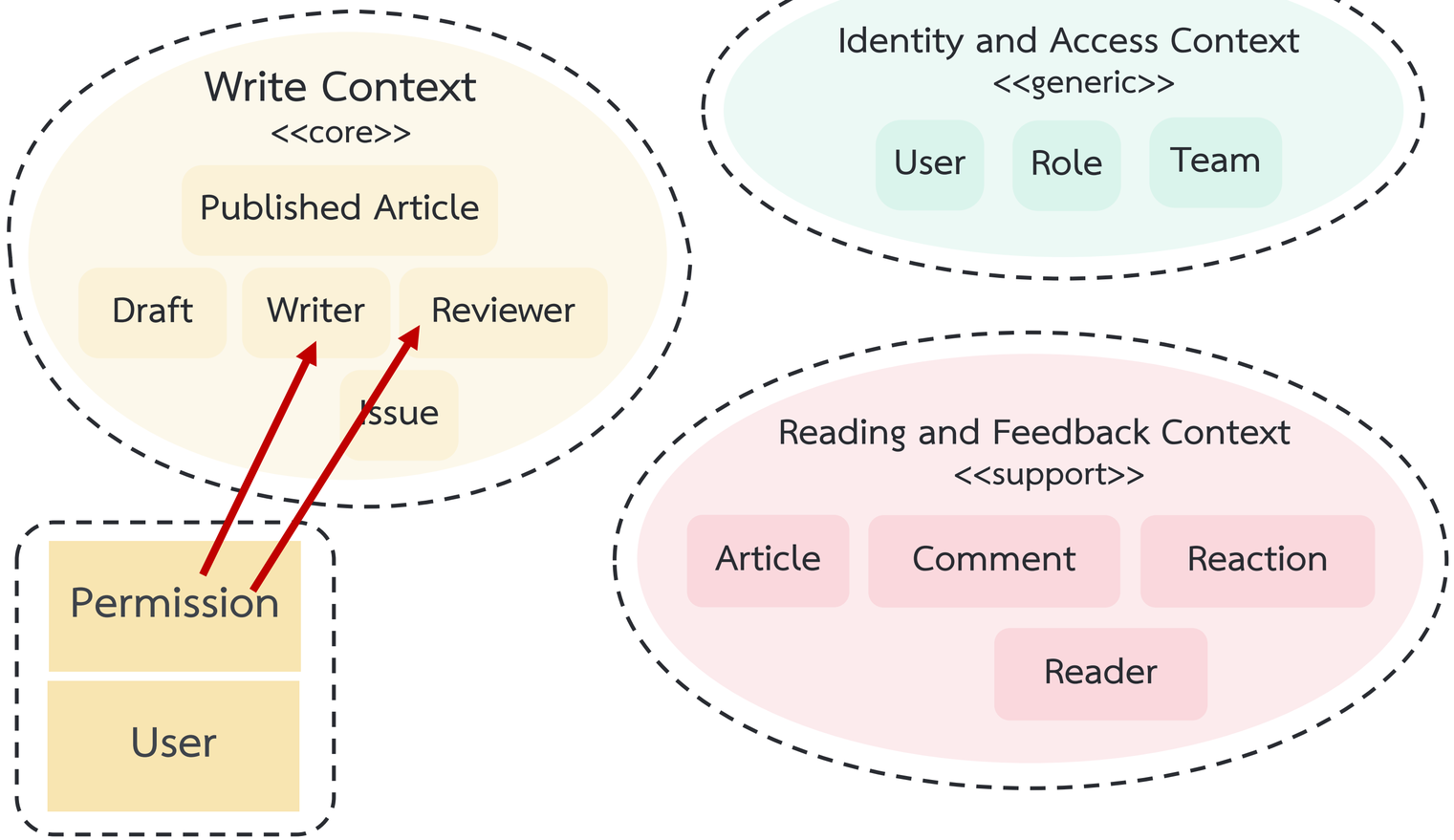
Domain-driven Design: DDD

Bounded Context

ซึ่งการออกแบบที่ดี นักศึกษาควรออกแบบให้ 1 subdomain ควรมีเพียง 1 Bounded Context และควรมี 1 Microservice เท่านั้น



Domain-driven Design: DDD



<u>คำที่เรียกใช้</u>

คือ ภาษาเฉพาะที่เรียกใช้งานกันภายในแต่ Bounded Context ที่ทุกคนในทีม Developer หรือ Domain Expert เข้าใจตรงกัน และไม่กำกวม

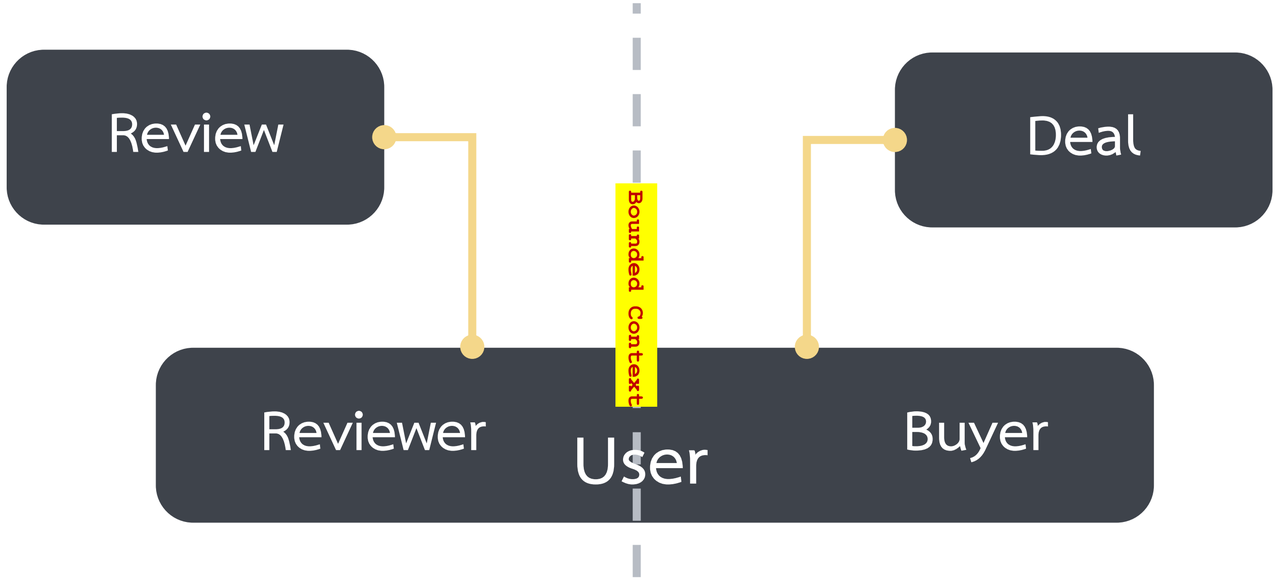
โดยทั่วไป 2 โดเมนหรือส่วนย่อยของโดเมนใด ๆ อาจจะมีคำใช้เรียก

- Entity อย่างเดียวกันด้วย***ชื่อที่ต่างกัน***
- Entity คนละอย่างกันด้วย***ชื่อเดียวกัน***

สิ่งเหล่านี้ทำให้กำหนดขอบเขตที่ชัดเจนได้ค่อนข้างยาก ยกตัวอย่างเช่น User เป็น Entity

Domain-driven Design: DDD

ตัวอย่างเช่น กรณีศึกษาคำว่า User ในโดเมนของ Review กับ Deal



สำหรับ Review Domain แล้ว ข้อมูล User ที่เกี่ยวข้องน่าจะเป็น จำนวนรีวิว และคำติชม มากกว่าเพศ หรือ อายุของ User ถ้ามองในแง่นี้จะพบว่า User ใน Domain นี้น่าจะเป็น *“Reviewer”*

สำหรับ Deal Domain แล้ว ข้อมูล User ที่เกี่ยวข้องน่าจะเป็น การซื้อ Deal เท่านั้น ถ้ามองในแง่นี้จะพบว่า User ใน Domain นี้น่าจะเป็น *“Buyer”*

จากตัวอย่างข้างต้นจะพบว่า ทั้ง Reviewer และ Buyer ต่างก็เป็น User ทั้งคู่เป็น Entity เดียวกัน เพียงแค่มีการใช้ข้อมูลที่ต่างกันขึ้นกับแต่ละ Domain ดังนั้น ควรจัดเก็บข้อมูลของ User ที่เกี่ยวกับ Review ไว้ที่ Review Service และเก็บข้อมูลของ User ที่เกี่ยวกับ Deal ไว้ที่ Deal Service ซึ่งไม่ควรนำข้อมูลทั้งหมดมาเก็บที่ User Service ที่เดียว

Domain-driven Design: DDD

ตัวอย่างเช่น กรณีศึกษาการฉีด vaccine ใน Context ต่าง ๆ ในโดเมน Warehouse, Scheduling และ Medication Record

สำหรับ Warehouse Domain จะสนใจว่าวัคซีนได้ถูกนำไปใช้งานหรือยัง	`vaccine.take_out()`
สำหรับ Scheduling Domain จะสนใจว่าพยาบาลฉีดวัคซีนให้กับคนไข้	`nurse.vaccinate(patient, vaccine)`
สำหรับ Medication Record Domain จะสนใจว่าคนไข้ได้รับวัคซีนจากพยาบาลคนไหน	`patient.take(vaccine, nurse)`

ถ้านักศึกษาลองปรับให้ใช้เหมือนกันจะพบว่า

สำหรับ Warehouse Domain จะสนใจว่าวัคซีนได้ถูกนำไปใช้งานหรือยัง	`nurse.vaccinate(patient, vaccine)`
สำหรับ Scheduling Domain จะสนใจว่าพยาบาลฉีดวัคซีนให้กับคนไข้	`nurse.vaccinate(patient, vaccine)`
สำหรับ Medication Record Domain จะสนใจว่าคนไข้ได้รับวัคซีนจากพยาบาลคนไหน	`patient.take(vaccine, nurse)`

นักศึกษาจะพบว่า (1) vaccine ใน Warehouse ต้องเก็บข้อมูลของ Entity ของ nurse และ patient เพิ่มเติมโดยไม่จำเป็น และ (2) มุมมองทางด้านการสื่อสารของฝาย Warehouse จะเริ่มสับสน เพราะแต่เดิมจะสนใจเพียงของในคลังเท่านั้น โดยไม่ได้สนใจว่าใครฉีดหรือฉีดให้ใคร (3) ทำให้เกิด dependency มากยิ่งขึ้น

Decompose by DOMAIN OBJECTS Pattern (การแยกส่วนตามโดเมน)

ถ้านักศึกษาสามารถออกแบบแต่ละ Service ได้ตรงตาม Bounded Context จะทำให้พัฒนา Service นั้นได้ด้วยทีมขนาดเล็กเนื่องจากขอบเขตที่จำกัดและงานที่ชัดเจน ทำให้ง่ายต่อการทำความเข้าใจ โดยทั่วไปแล้ว Bounded Context มักจะมีรูปแบบคล้ายคลึงกับโครงสร้างองค์กร

Decompose by DOMAIN OBJECTS Pattern (การแยกส่วนตามโดเมน)

การกำหนดรูปแบบการสื่อสารระหว่าง Service

จะเกิดหลังหลังจากนักศึกษาได้กำหนด Bounded Context เรียบร้อย ซึ่งมีโอกาสที่บาง Service จำเป็นต้องติดต่อกับ Service ใน Domain อื่น ทำให้การออกแบบ API ควรเลือกวิธีการสื่อสารกันให้เหมาะสมโดยพิจารณาจาก (1) ใช้งานได้ง่าย และ (2) ไม่ขึ้นกับภาษาที่ใช้พัฒนา เพื่อหลีกเลี่ยงการยึดติดกับเทคโนโลยี โดยส่วนใหญ่นิยมการสื่อสารแบบ REST ด้วยโปรโตคอล HTTP หรือ gRPC ด้วยโปรโตคอล HTTP 2 ซึ่งโปรโตคอลทั้งหมดข้างต้นจะไม่ขึ้นกับภาษาที่ใช้พัฒนา และเป็นการสื่อสารแบบ Asynchronous

Ref https://www.dineshonjava.com/application-decomposition-to-microservice/

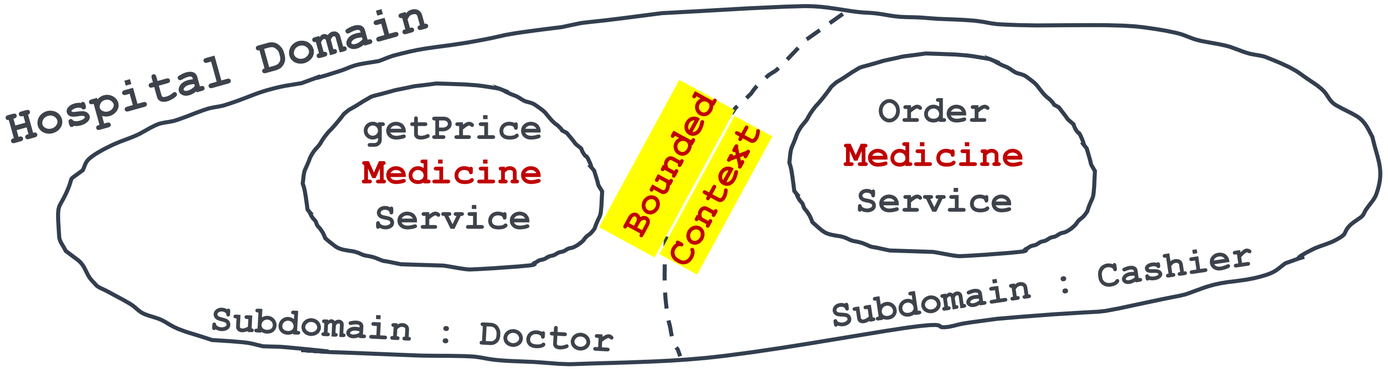
Decompose by DOMAIN OBJECTS Pattern (การแยกส่วนตามโดเมน)

ขั้นตอนที่ 1 แจกแจง Function ทั้งหมดในระบบออกมาพร้อมทั้งระบุ Entity ที่ใช้

ขั้นตอนที่ 2 จัดกลุ่ม Function เป็น Bounded Context เพื่อการแบ่งส่วนต่าง ๆ ให้ออกมาชัดเจน ซึ่งสามารถทำได้ 2 แบบ

- แบบจัดกลุ่มตาม***บทบาท (Role)*** ตัวอย่างเช่น ระบบ E-learning จะแบ่งออกเป็น 3 บทบาท ได้แก่ อาจารย์ นักศึกษา และผู้ดูแล เป็นต้น
- แบบจัดกลุ่มตาม***หน้าที่การทำงาน (Function)*** ตัวอย่างระบบ E-learning เช่น ระบบจัดการรายวิชา ระบบจัดการผู้เรียน ระบบจัดการแบบทดสอบ เป็นต้น

ถึงแม้ว่า Bounded Context จะต่างกัน อาจจะมี Entity เดียวกันแต่ใช้ข้อมูลใน Entity แตกต่างกันได้ อาทิเช่น ใน Context ของ**หมอ**กับ**เจ้าหน้าที่การเงิน**อาจเกี่ยวข้องกับ Entity ของ**ยา** โดยหมอจะใช้ข้อมูลในส่วนการสั่งยารักษา แต่เจ้าหน้าที่การเงินจะใช้ข้อมูลในส่วนราคาของยา



Decompose by DOMAIN OBJECTS Pattern (การแยกส่วนตามโดเมน)

ขั้นตอนที่ 3 ระบุชนิดหรือประเภทของโดเมนย่อย ๆ (Subdomain) ว่าเป็นประเภทใด โดยกำหนดได้ 3 ประเภทดังนี้

- Core คือ ส่วนที่สร้างความแตกต่างที่สำคัญสำหรับธุรกิจและเป็นส่วนที่สร้างมูลค่าเพิ่มที่สุดของระบบ
- Supporting คือ ส่วนที่เกี่ยวข้องและสนับสนุน Core แต่ไม่ได้สร้างมูลค่าเพิ่มให้ระบบ (สามารถจ้าง Outsource)
- Generic คือ ส่วนทั่ว ๆ ไป ไม่ได้เฉพาะเจาะจงสำหรับธุรกิจ (หาซื้อได้ตามท้องตลาด)

Decompose by DOMAIN OBJECTS Pattern (การแยกส่วนตามโดเมน)

ขั้นตอนที่ 4 การระบุความสัมพันธ์ระหว่าง Domain

สำหรับ Domain ใด ๆ นั้นมีโอกาสจะเรียกใช้งาน Entity ที่อยู่ใน Domain อื่นเสมอ ในขั้นตอนนี้จะเป็นการระบุว่า Domain ใดมีความเกี่ยวข้องหรือเชื่อมโยงกันบ้าง และมีความสัมพันธ์ไปในทางทิศใด

ขั้นตอนที่ 5 การระบุส่วนประกอบต่าง ๆ ใน Core Domain

โดยทั่วไปแล้ว Core Domain จะประกอบไปด้วย 3 หลัก ๆ ได้แก่ command, domain event และ aggregated ตามลำดับ

Service Decomposition

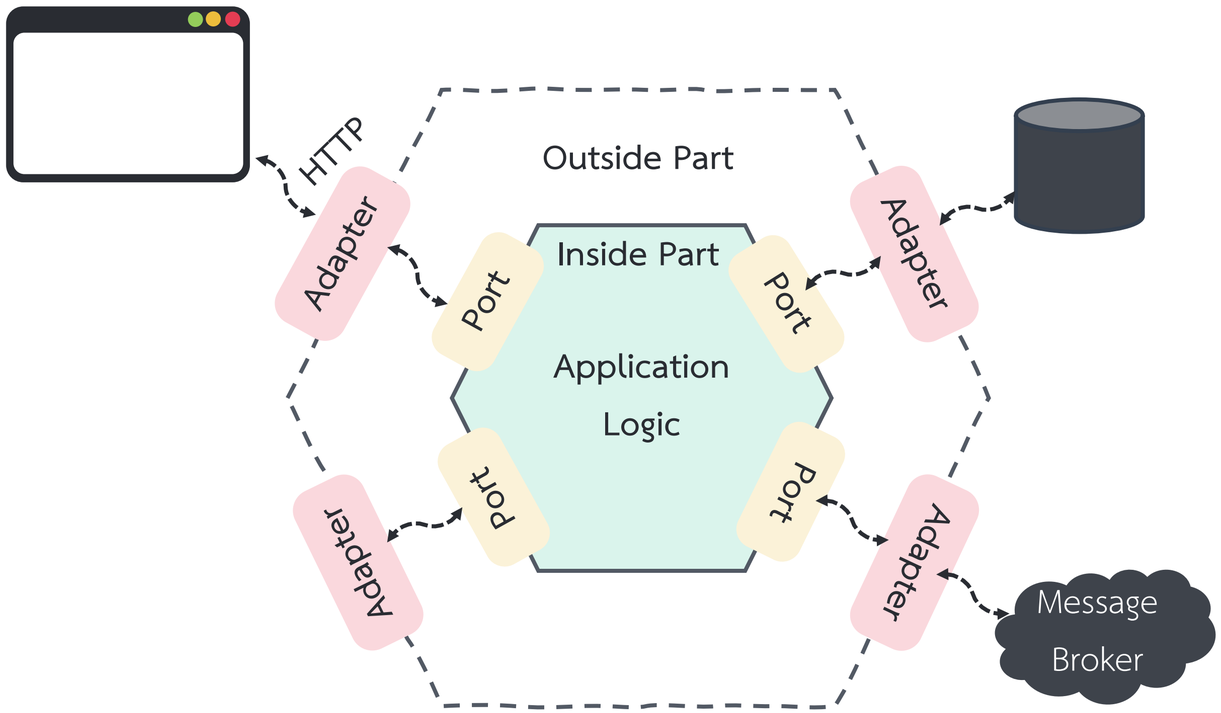
นอกจากนี้ การทำ Service Decomposition ด้วยการแยกส่วนตามความสามารถทางธุรกิจและการแบ่งแยกตามขอบเขต (Domain Objects) แล้วยังมีแนวความคิดอื่น ๆ อีก ได้แก่

- การแบ่งแยกตาม Action Verbs (งาน)
  - ✓ อาทิเช่น ระบบ Payment Service
- การแบ่งแยกตาม Nouns (คำนาม)
  - ✓ อาทิเช่น ระบบ Customer Service

Outline

- Software Architecture Styles
  - ความสัมพันธ์ระหว่างคอมโพเนนต์
  - สถาปัตยกรรมแบบ 2, 3 และ 4 เลเยอร์
  - Domain-driven Design (DDD)
- Decomposition Strategies
  - Conway Manouvre Law
  - Inverse Conway Manouvre Law
  - Business Capability Pattern
  - Domain Objects Pattern (or Domain-driven Design: DDD)
- Hexagonal Architecture in Java

Hexagonal Architecture in Java

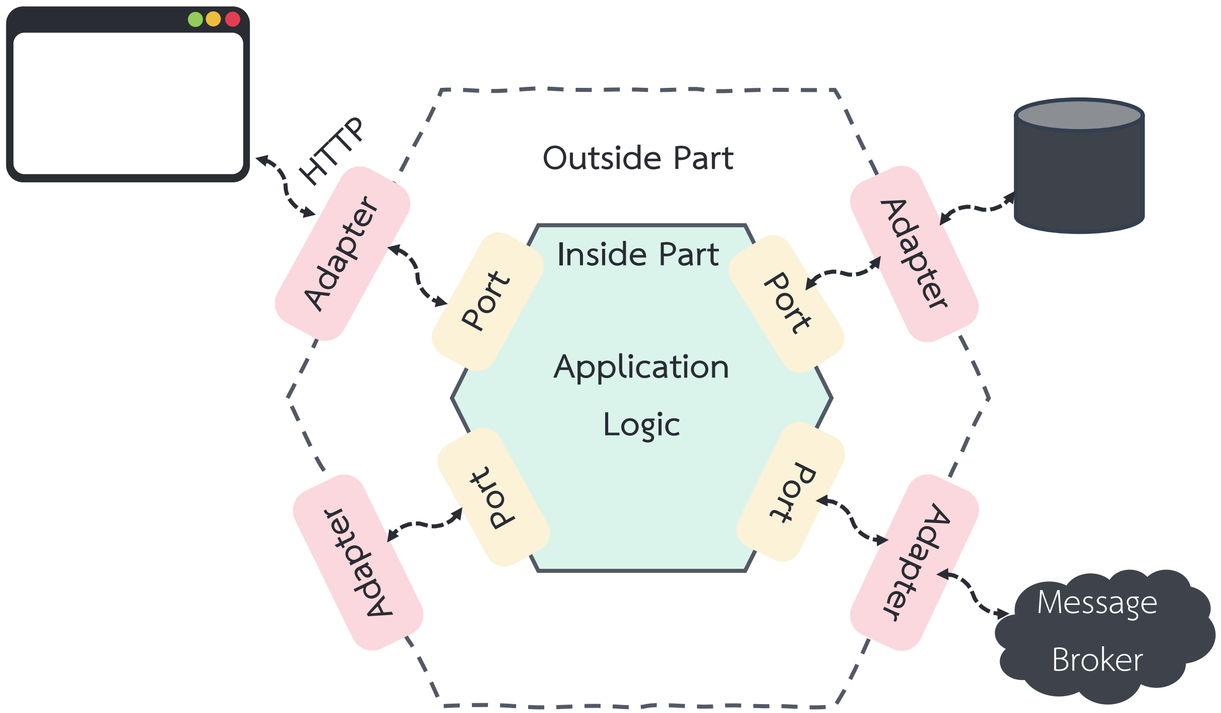


Hexagonal Architecture (หรือเรียกว่า Ports & Adapters Architecture) ได้รับการคิดค้นในปี ค.ศ. 2006 โดย Alistair Cockburn ซึ่ง Hexagonal Architecture เป็นสถาปัตยกรรมสำหรับการออกแบบระบบรูปแบบหนึ่งที่อยู่บนหลักการที่ว่า “Hexagonal Architecture which makes the software easy to maintain, manage, test, and scale ” โดยที่ Hexagonal Architecture ได้แบ่ง application ออกเป็น 2 ส่วนหลัก ได้แก่

Inside Part คือ ส่วนของ core business logic ของ application (หรือ เรียกว่า Application core)

Outside Part คือ เป็นส่วนของ ฐานข้อมูล, UI, และ messaging queues เป็นต้น

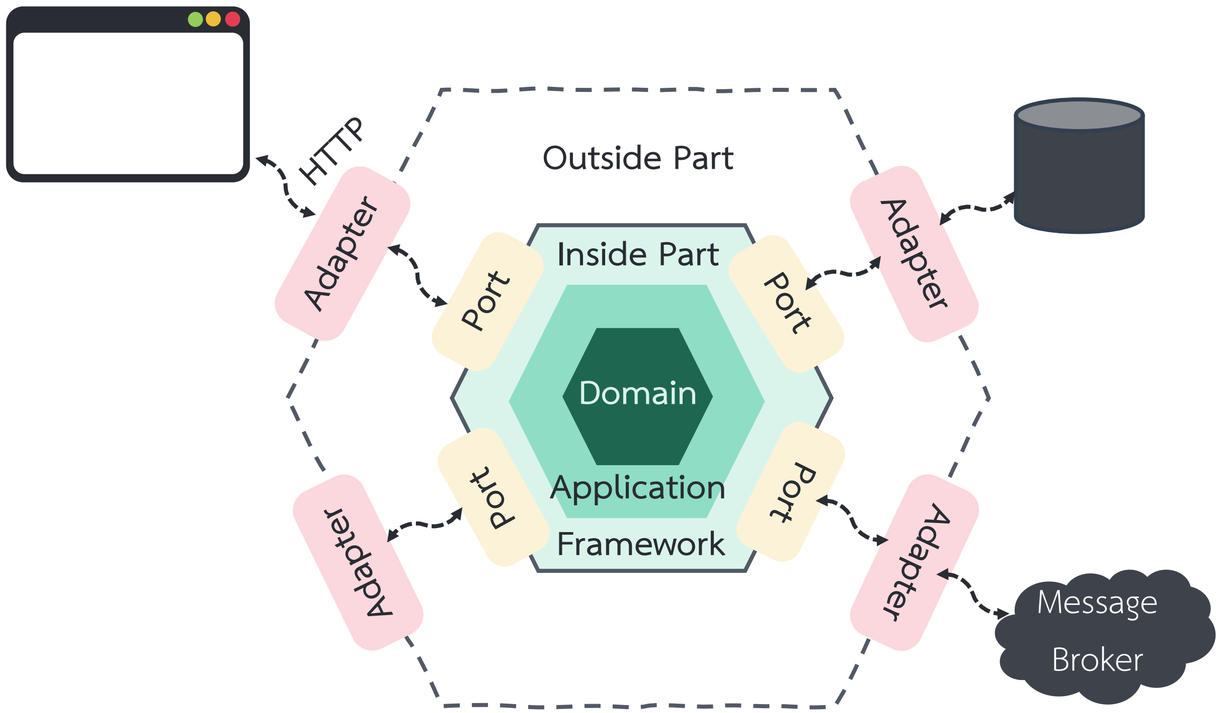
Hexagonal Architecture in Java



ซึ่งทั้ง 2 ส่วนจะสื่อสารกันผ่านสิ่งที่เรียกว่า Port และ Adapter โดยที่

- Port คือ ไฟล์ Interface ในภาษาจาวา
- Adapter คือ คลาสที่ได้รับการ implement ไฟล์ interface มา

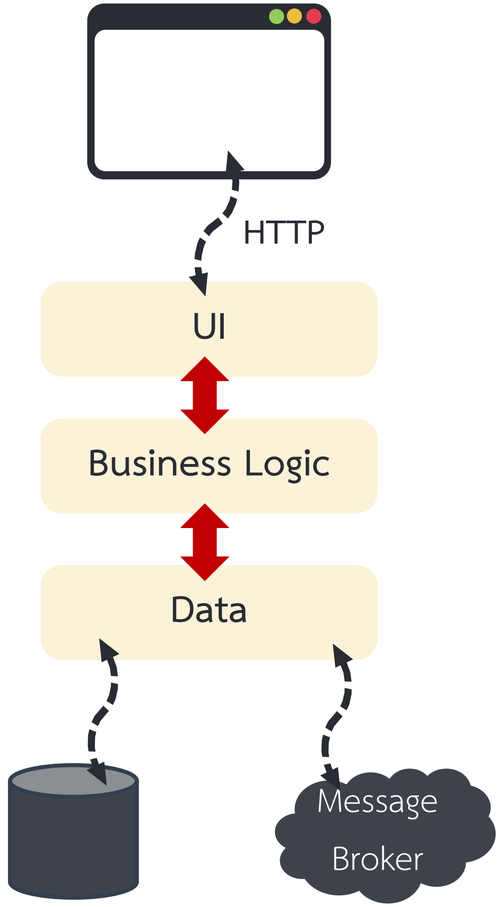
Hexagonal Architecture in Java

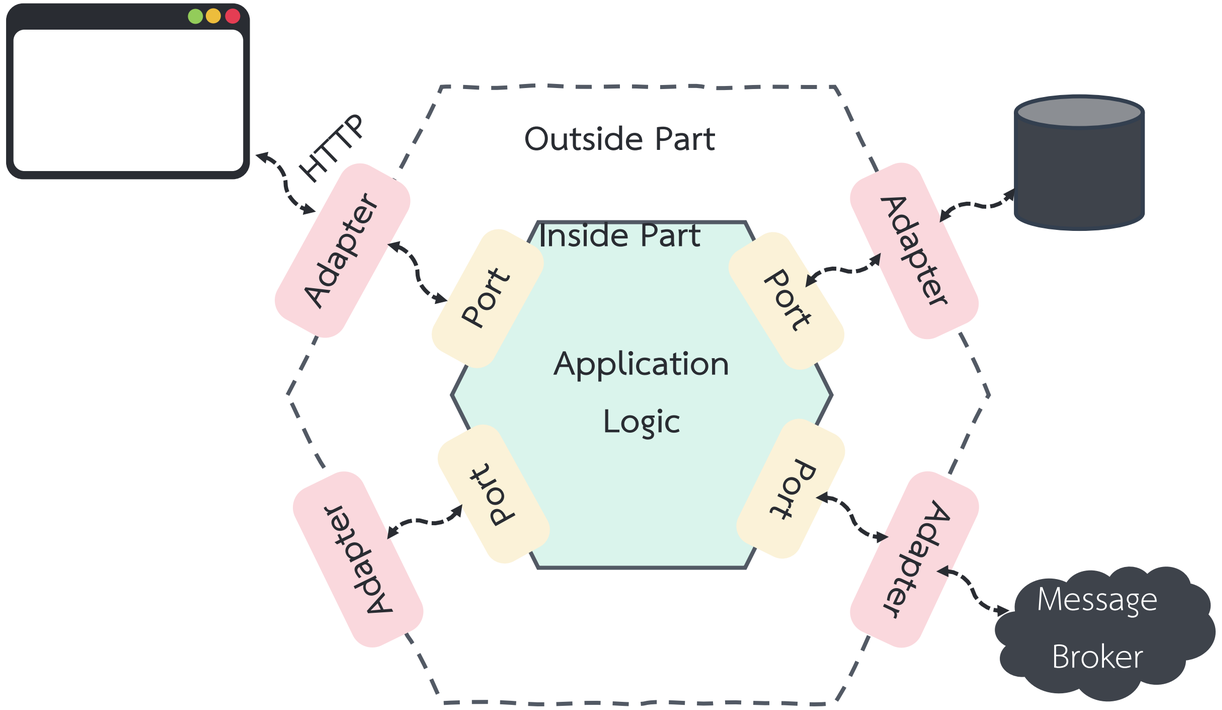


นอกจากนี้ Hexagonal Architecture ยังสามารถแบ่งออกเป็น 3 เลเยอร์ ได้แก่

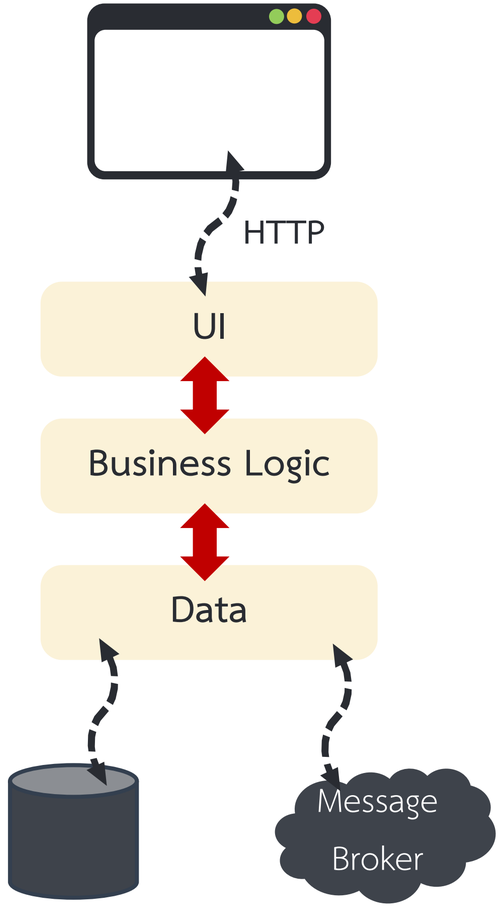
- **Domain** คือ เลเยอร์ของ core business logic
- **Application** คือ ตัวกลางระหว่างเลเยอร์ Domain กับเลเยอร์ Framework
- **Framework** คือ เลเยอร์ที่ใช้กำหนดว่าเลเยอร์ Domain จะ response ต่อ request ที่มาจาก Outside part อย่างไร

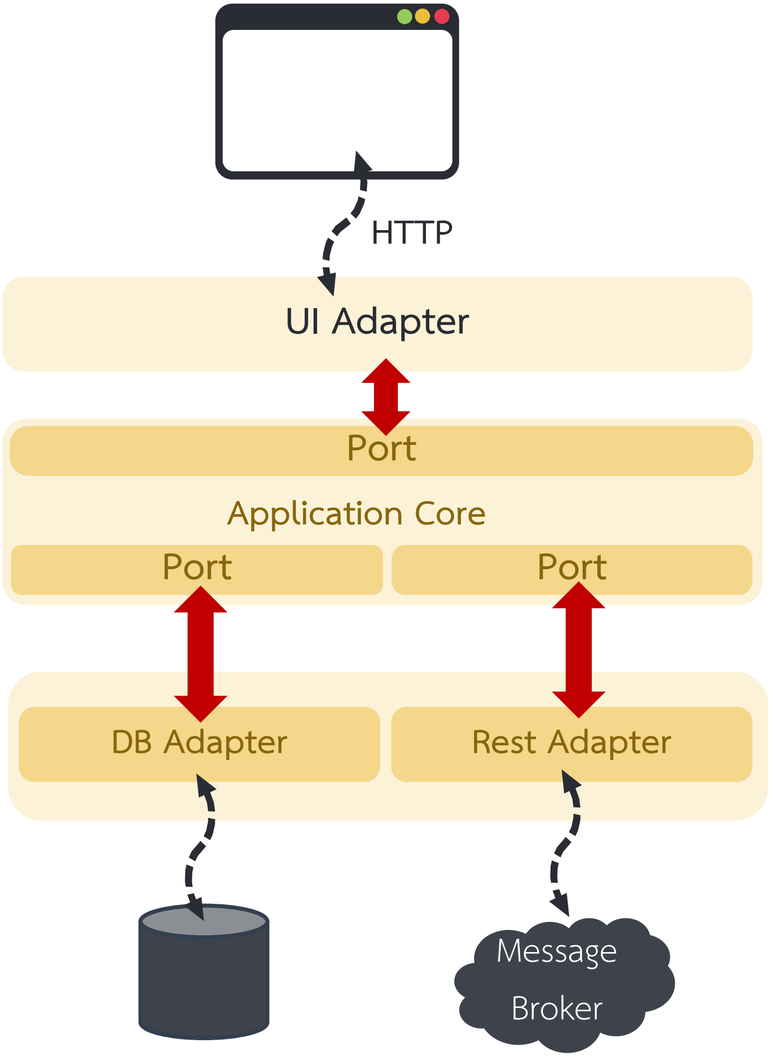
Hexagonal Architecture in Java



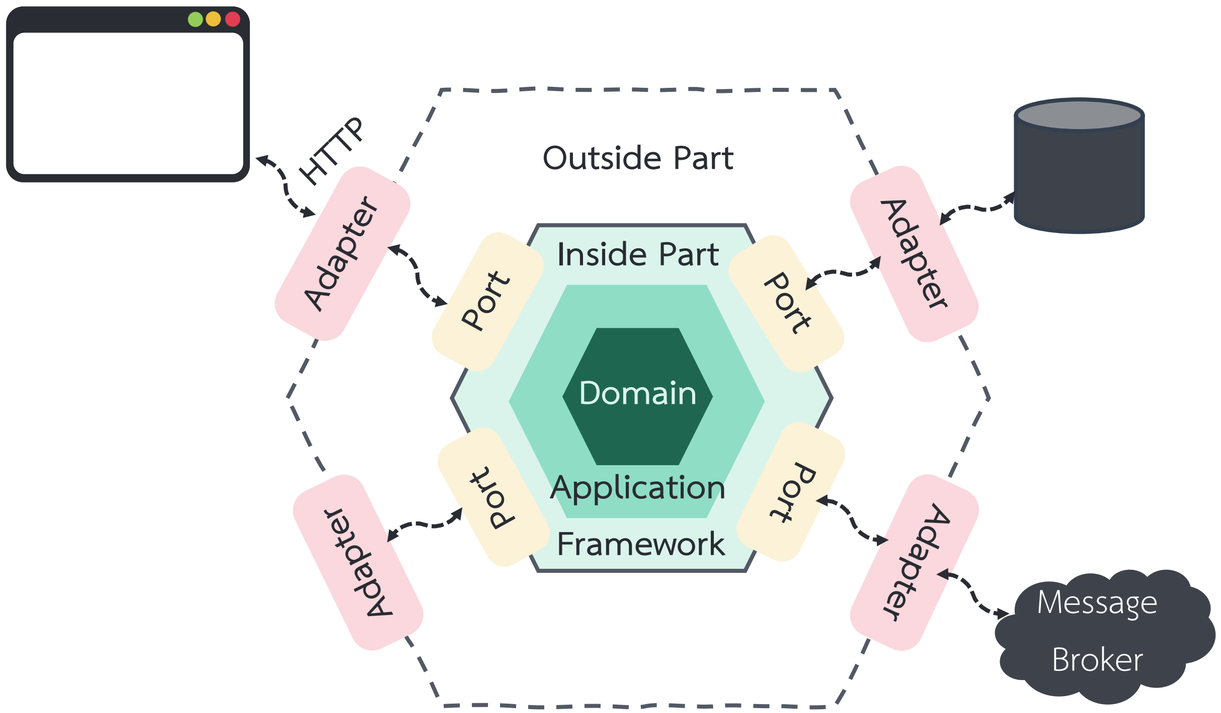


Hexagonal Architecture in Java





Hexagonal Architecture in Java



ข้อดีของการออกแบบด้วย Hexagonal Architecture

- **Easy to maintain** เนื่องจาก core logic ไม่เกี่ยวข้องกับ Component ภายนอกโดยตรง ทำให้เกิด loosely coupled ส่งผลให้ง่ายต่อการดูแลและบำรุงรักษาโดยปราศจากการกระทบกระเทือนถึง Component อื่น ๆ

- **Easy to adapt new changes** เนื่องจากแต่ละเลเยอร์เป็นอิสระต่อกัน ทำให้สะดวกต่อการเพิ่มลดหรือเปลี่ยน Component ภายนอกใหม่

- **Easy to test** จากข้อดีที่ได้กล่าวมาแล้วทำให้ง่ายต่อการเขียน Test case ทดสอบ โดยการจำลองตัว port และ adapter ขึ้นมาทำสอบเฉพาะ core business logic