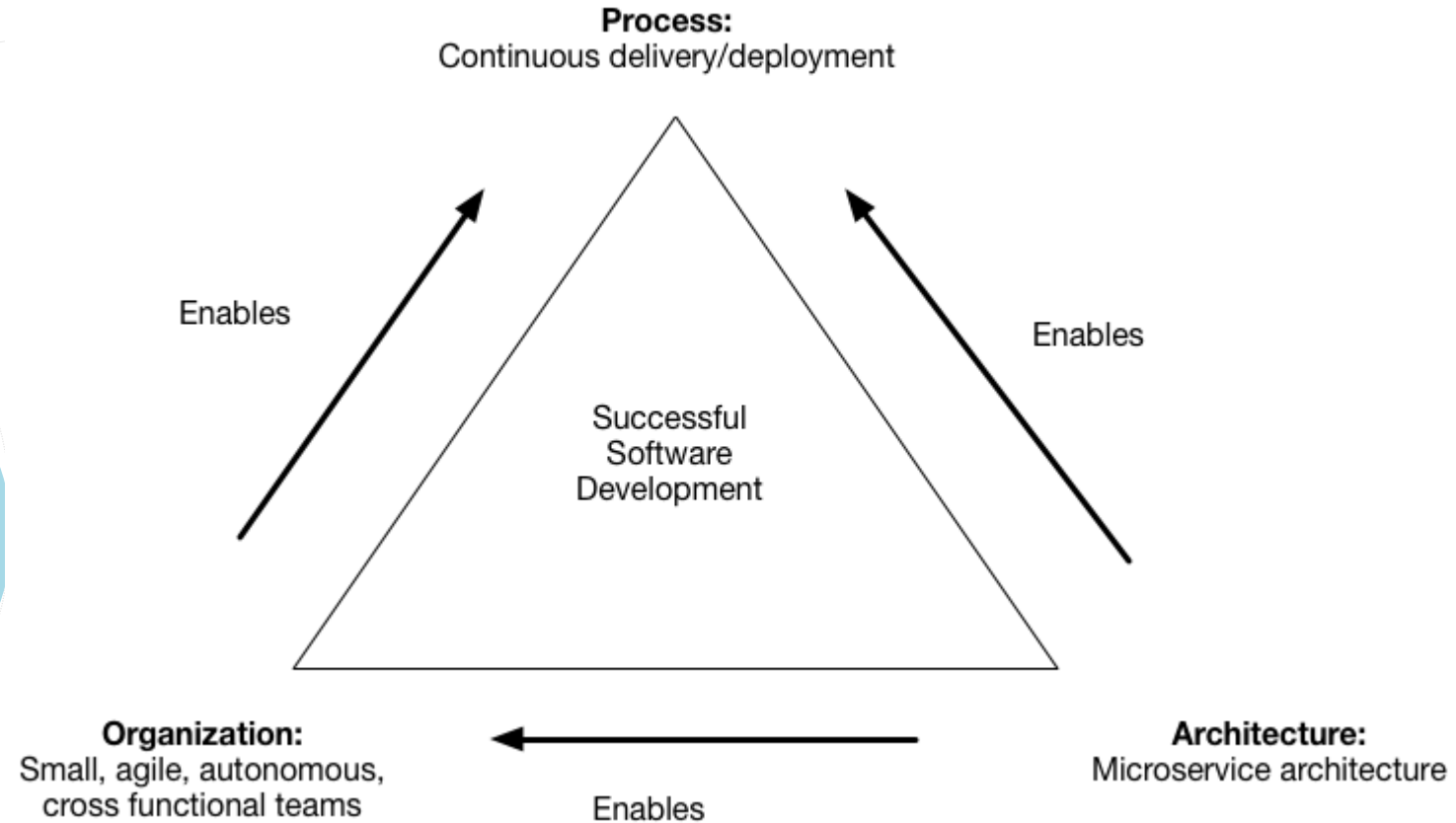


# Software Architectural Design

“การออกแบบสถาปัตยกรรมซอฟต์แวร์”



# Topics Covered

- ภาพรวม
- สิ่งที่ต้องรู้ก่อนออกแบบสถาปัตยกรรม
- Deployment architecture
  - Patterns
- Logical architecture
  - Patterns

# ภาพรวม

- เมื่อวิเคราะห์ระบบจนเข้าใจที่มาที่ไปได้แล้ว
  - ก่อนสร้างระบบ ต้องมองภาพใหญ่ให้ออก (Large-scale organization)
- วงการคอมพิวเตอร์ นิยม คำว่า “สถาปัตยกรรม” เพื่อให้เห็นถึงความสำคัญของโครงสร้าง ทำให้มีการแปลความ สถาปัตยกรรมซอฟต์แวร์ ไว้หลายแบบ
  - 1) องค์ประกอบและความสัมพันธ์ขององค์ประกอบซอฟต์แวร์
  - 2) สิ่งที่เกี่ยวข้องกับซอฟต์แวร์และยากต่อการเปลี่ยนแปลงในอนาคต
  - 3) การกำหนดโครงสร้างของซอฟต์แวร์ ด้วยการนำแต่ละส่วนการทำงานมากำหนดวิธีการทำงานร่วมกันผ่านส่วนต่อประสาน (Interfaces) ที่กำหนดไว้
  - 4) อะไรก็ตามที่มีความสำคัญกับซอฟต์แวร์

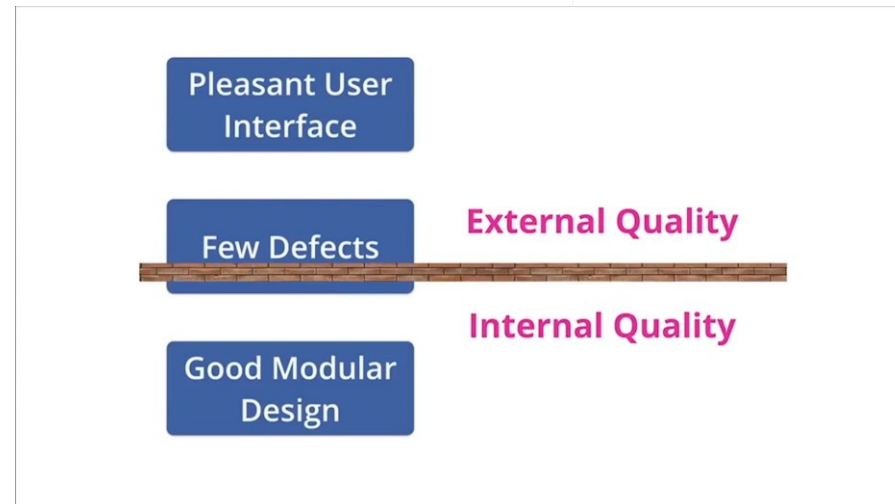
## ภาพรวม (ต่อ)

- “การออกแบบสถาปัตยกรรมซอฟต์แวร์” จึงมีหลายความหมาย
  - การออกแบบวิธีการจัดองค์ประกอบหลักของระบบ “How to organize your system”
  - การออกแบบการทำงานร่วมกันของสิ่งที่เปลี่ยนแปลงได้ยากในอนาคต
  - การออกแบบโครงสร้างของซอฟต์แวร์ และการกำหนดการทำงานร่วมกัน
  - การออกแบบอะไรก็ตามที่มีความสำคัญกับซอฟต์แวร์

สิ่งที่ได้จากกระบวนการนี้คือ Deployment Architectural Model และ  
Logical Architecture Model

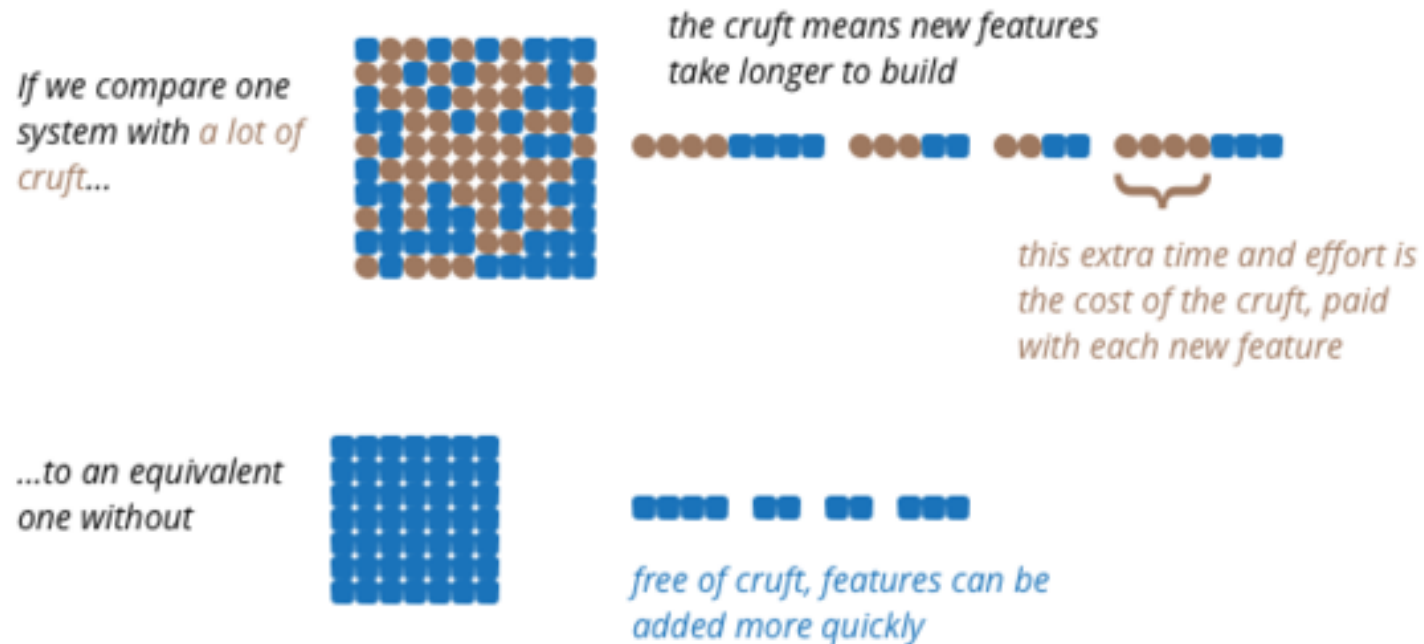
## ภาพรวม (ต่อ)

- การออกแบบสถาปัตยกรรม เกี่ยวข้องโดยตรงกับ “ความต้องการเชิงคุณภาพ (NFR)” เช่น
  - ลูกค้าบอกว่า “การแข่งขันทางการตลาดจะเป็นตัวเร่งให้มีการเปลี่ยนแปลง”
    - ออกแบบโดยเน้น ซอฟต์แวร์ที่สร้างต้องเพิ่มฟีเจอร์ ปรับแต่ง บำรุงรักษาได้ง่าย – Adaptivity + Maintainability
  - ลูกค้าบอกว่า “ซอฟต์แวร์ต้องทำงานเร็ว ระบบห้ามหยุดทำงาน”
    - ออกแบบโดยเน้น Performance + Reliability



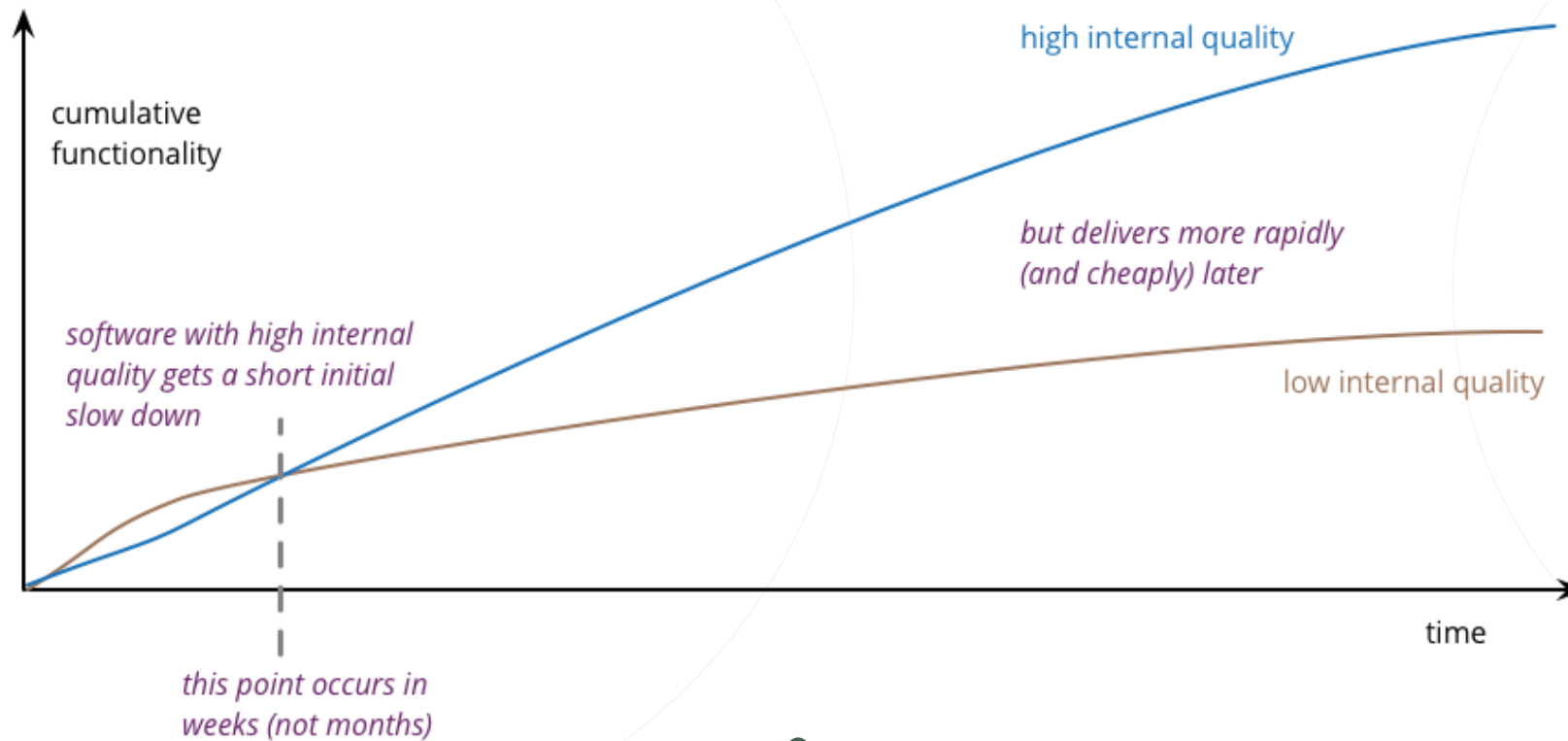
## ภาพรวม (ต่อ)

- ทำไมต้องออกแบบภาพใหญ่ (สถาปัตยกรรม) “Good architecture is something that supports its own evolution”



## ภาพรวม (ต่อ)

- การออกแบบสถาปัตยกรรม ส่งผลดีกับทีมพัฒนามากกว่าลูกค้า เพราะลูกค้า ไม่เห็นความแตกต่างของ internal quality

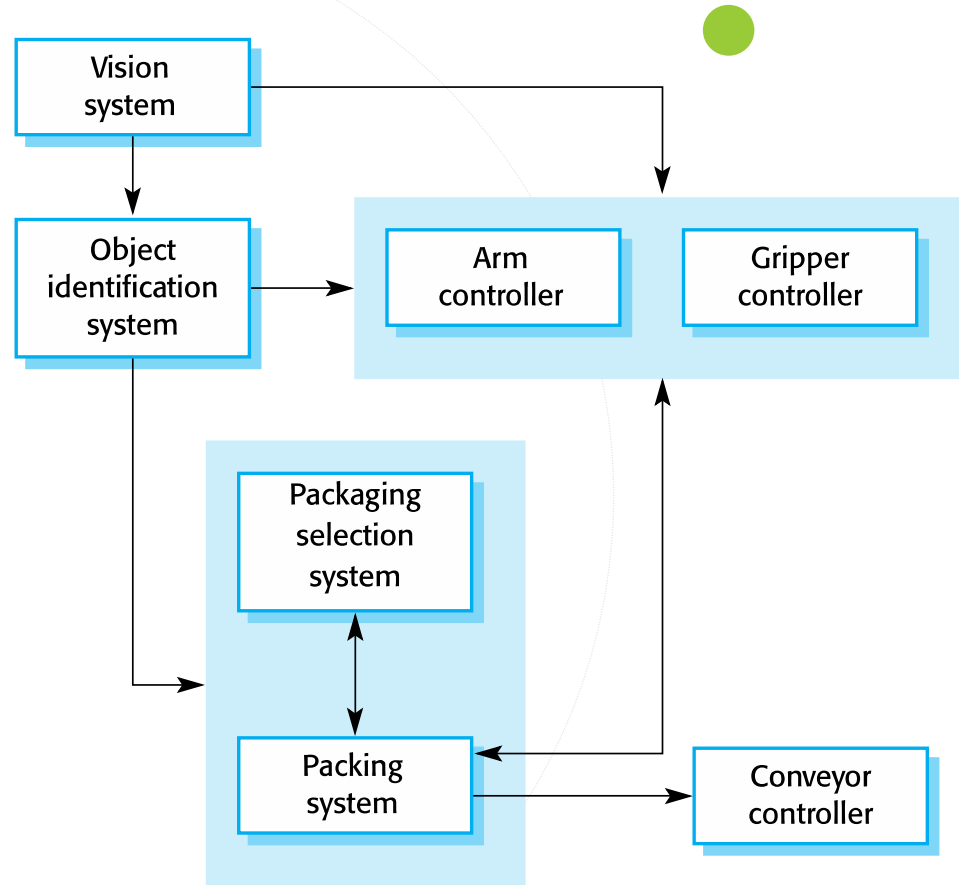


## ภาพรวม (ต่อ)

- การออกแบบสถาปัตยกรรม ทำให้มองเห็น Software Product-line ที่ควรมี
  - core function ของซอฟต์แวร์ (มักเกิดขึ้นซ้ำ ๆ กับระบบที่อยู่ในโดเมนเดียวกัน)
- ใช้อะไรในการออกแบบสถาปัตยกรรม ?
  - Block diagram: กล่องสี่เหลี่ยมแทนส่วนประกอบ, ลูกศรแทนการเชื่อมโยง
    - Bass et al. (2003) ให้ความเห็นว่า Block diagram เป็น poor architectural representations
  - UML (Deployment architecture)
    - Deployment diagram
  - UML (Logical architecture)
    - Component diagram
    - Package diagram

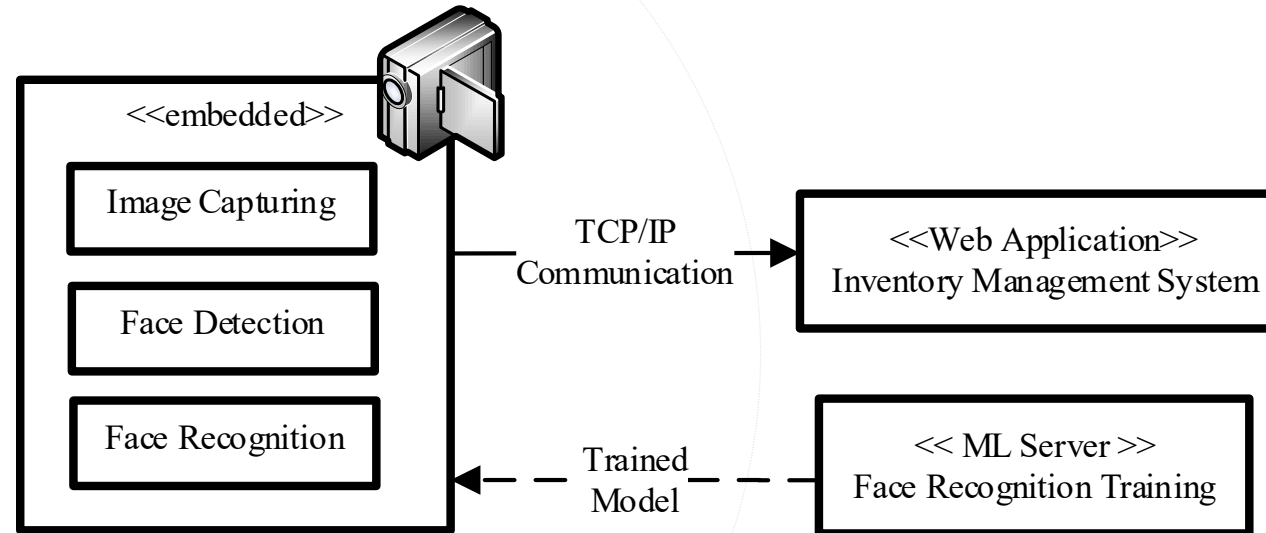


## ภาพรวม (ต่อ) : ตัวอย่าง block diagram ระบบหุ่นยนต์แพ็คของ



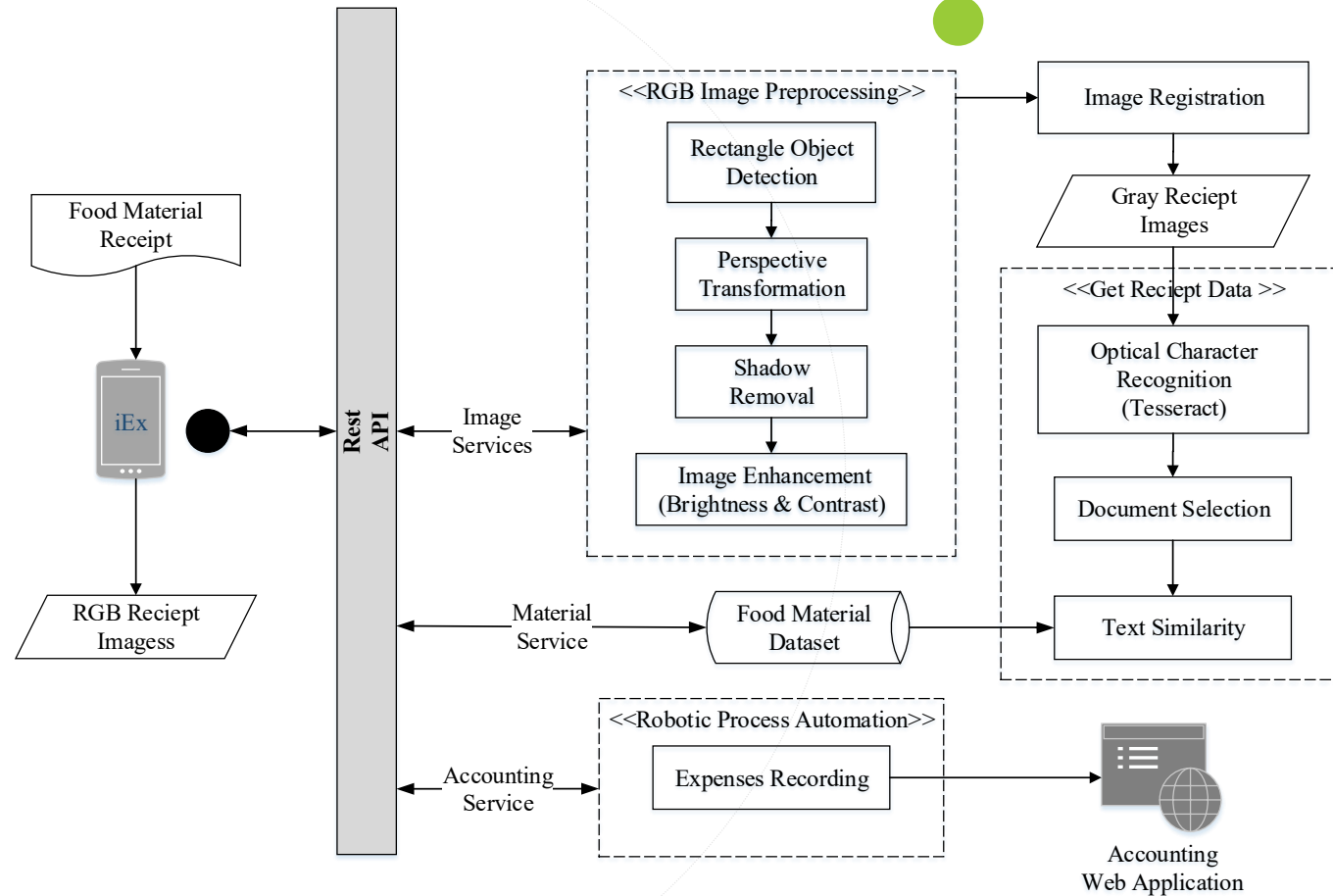
## ภาพรวม (ต่อ) : ตัวอย่าง block diagram

### Smart Inventory Access Monitoring System (SIAMS) using Embedded System with Face Recognition



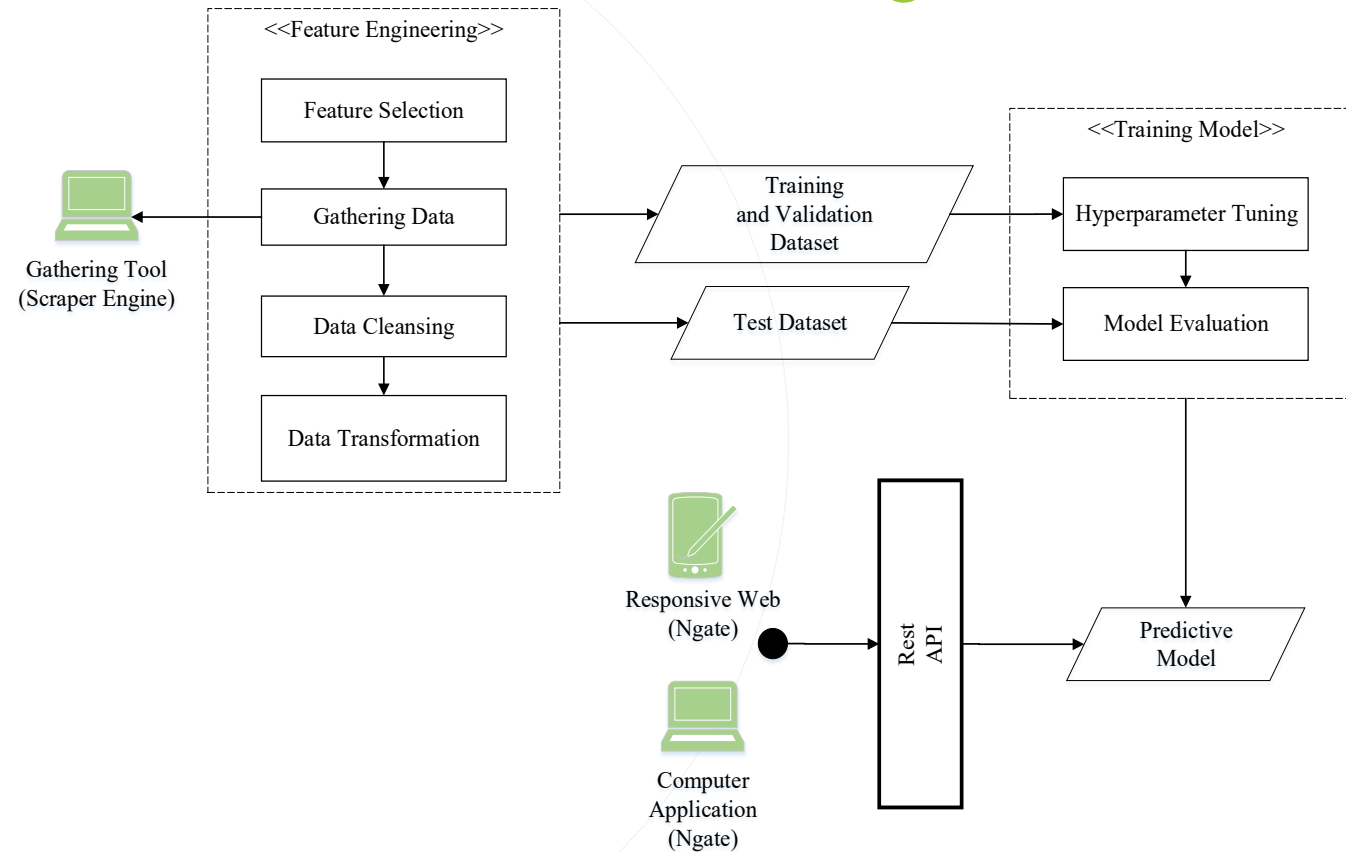
## ภาพรวม (ต่อ) : ตัวอย่าง block diagram

### Intelligent Expenditure Recording System for Small Restaurant



## ภาพรวม (ต่อ) : ตัวอย่าง block diagram

### An Enhancing Web Scraping technique using Machine Learning





## Architectural design decisions

สิ่งที่ควรรู้ก่อนออกแบบสถาปัตยกรรม

# สิ่งที่ควรรู้ก่อนการออกแบบสถาปัตยกรรม

## 1. สำรวจสถาปัตยกรรมต้นแบบที่เผยแพร่ไว้ (โดยผู้เชี่ยวชาญ)

- แม้รายละเอียดการทำงานภายในระบบแต่ละอันจะแตกต่างกัน แต่หากอยู่ในโดเมนเดียวกันแล้ว จะมีสถาปัตยกรรมที่ใกล้เคียงกันมาก
- เช่น Peer-to-Peer , Master-Slave, Client-Server, Decentralize เป็นต้น

## 2. ลักษณะการใช้หน่วยประมวลผล

- จำนวน core CPU หรือ จำนวนโพรเซส ที่ถูกใช้งานในขณะ Runtime
- เช่น embedded system ที่ทำงานในลักษณะ single process ก็ไม่จำเป็นต้องออกแบบเพื่อรองรับการประมวลผลแบบกระจาย (มีผลโดยตรงกับ performance)

# สิ่งที่ควรรู้ก่อนการออกแบบสถาปัตยกรรม (ต่อ)

## 3. ลูกค้านั้นคุณภาพด้านใดเป็นพิเศษ (NFR)

- Performance
  - ออกแบบ**รวม**เป็นองค์ประกอบขนาดใหญ่ มากกว่าแยกออกเป็นองค์ประกอบย่อย ซึ่งจะช่วยลดการสื่อสารระหว่างองค์ประกอบได้
  - ติดตั้งระบบไว้ในเครื่องเดียวกัน ไม่กระจายเครื่องเพื่อลดการสื่อสารระหว่างเครือข่าย
- Security
  - ใช้สถาปัตยกรรมแบบแยกชั้น (Layered architecture) โดยจัดให้องค์ประกอบที่มีความสำคัญสูงสุดอยู่ด้านในสุด (นับจากส่วนต่อประสานผู้ใช้เข้ามา)
  - การเข้าถึง ข้ามชั้นไม่ได้ ต้องเป็นลำดับต่อกันไป (Linear access)
- Safety
  - แยกส่วนการทำงานที่ต้องการความปลอดภัยสูงให้อยู่ภายในองค์ประกอบเดียวกัน (หากเป็นไปได้ ก็ให้มีจำนวนองค์ประกอบน้อยที่สุด) เพื่อให้ง่ายต่อการปิดระบบเมื่อมีความผิดพลาดเกิดขึ้น

# สิ่งที่ควรรู้ก่อนการออกแบบสถาปัตยกรรม (ต่อ)

## 3. ลูกค้านั้นคุณภาพด้านใดเป็นพิเศษ (NFR)

- Reliability
  - การออกแบบให้มีระบบทำงานทดแทนระบบหลักได้
  - ออกแบบให้กู้คืนระบบได้ อาจเป็นบางส่วนหรือทั้งหมด ขึ้นอยู่กับข้อกำหนด
  - ระบบทนทานต่อการกระทำที่ผิดพลาด
- Maintainability
  - แยกการทำงานของระบบออกเป็นองค์ประกอบย่อย (fine-grain component) ให้ง่ายต่อการแก้ไข แก้แล้วไม่กระทบกับเป็นวงกว้าง



# สิ่งที่ควรรู้ก่อนการออกแบบสถาปัตยกรรม (ต่อ)

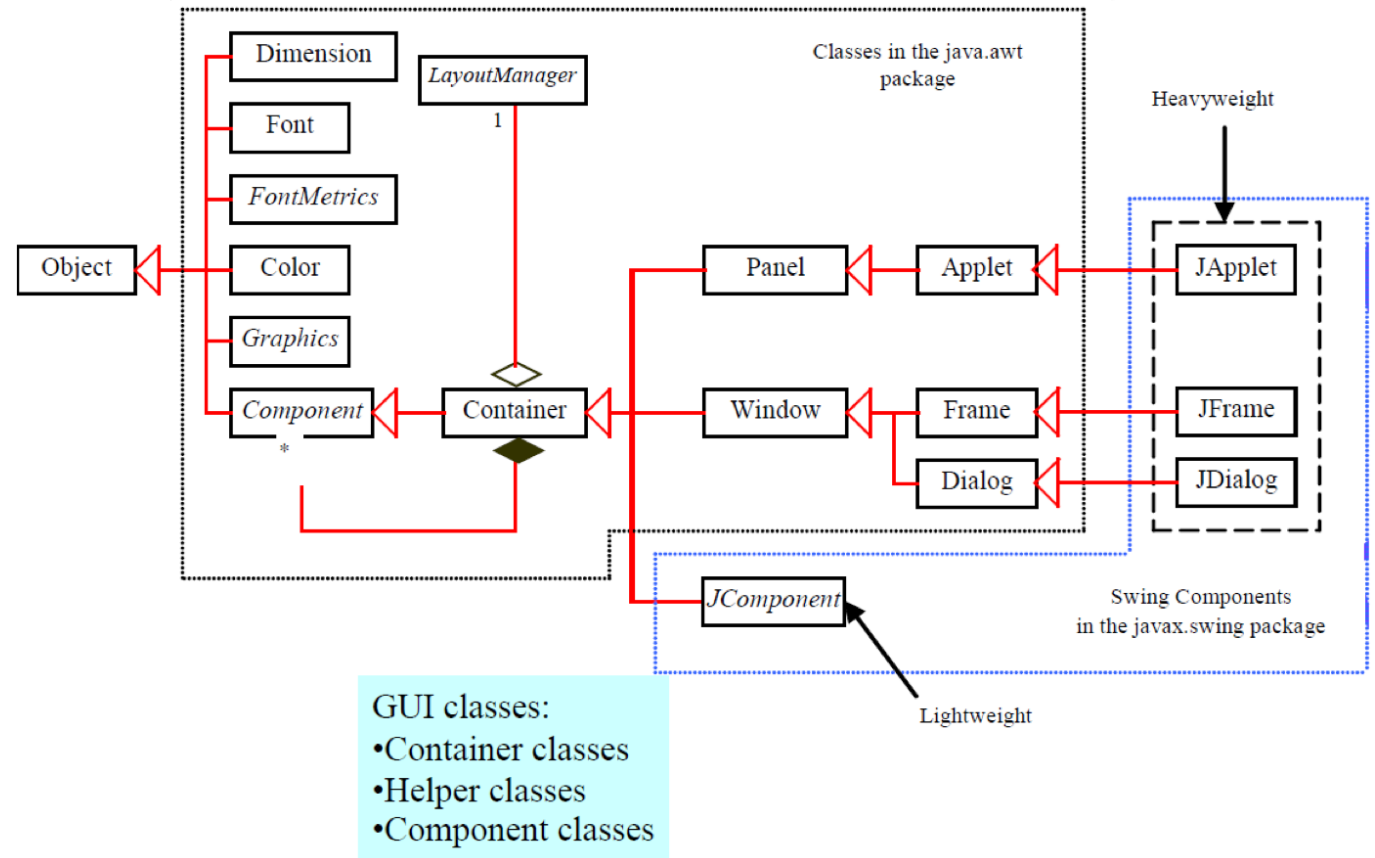
## 3. ลูกค้าเน้นคุณภาพด้านใดเป็นพิเศษ (NFR)

- กรณีต้องการ NFRs ที่ออกแบบสวนทางกัน เช่น ลูกค้าต้องการทั้ง Performance และ Maintainability
  - ให้พิจารณาแต่ละส่วนว่าต้องการอะไรมากที่สุด และแยกการออกแบบของส่วนนั้น ๆ  
จึงค่อยเลือกสถาปัตยกรรมที่เหมาะสมให้แต่ละส่วน

# สิ่งที่ควรรู้ก่อนการออกแบบสถาปัตยกรรม (ต่อ)

## 4. ระดับความสัมพันธ์ภายในการทำงานแต่ละส่วนที่ต้องการ (Cohesion & Coupling)

- Cohesion หมายถึง โปรแกรมที่ตอบสนองงานเดียวกันถูกจัดให้อยู่ด้วยกัน
- Coupling หมายถึง โปรแกรมที่จะทำงานได้สมบูรณ์ ต้องอาศัยโปรแกรมจากส่วนอื่น (พึ่งพากลุ่มอื่น)
- ตามทฤษฎีแล้ว ควรออกแบบให้การทำงานแต่ละส่วนมี cohesion สูง และให้ coupling ต่ำ



# สิ่งที่ควรรู้ก่อนออกแบบสถาปัตยกรรม (ต่อ)

## 5. จะสร้างการทำงานแต่ละส่วนด้วยแนวคิดภาษาแบบใด (Programming Paradigms)

- Procedural
- Object Oriented
- Function
- Logic

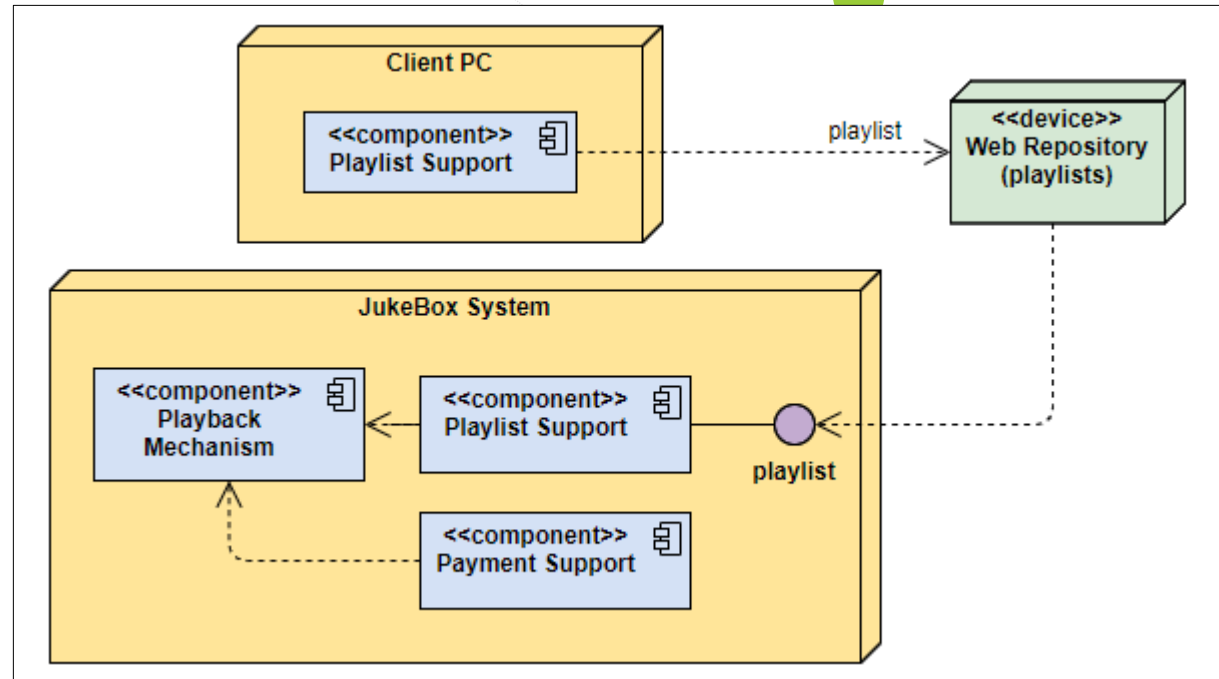
# Deployment architecture

- Overview
- Deployment architecture patterns
  - Master-slave architecture
  - Two-tier client-server architecture
  - Multi-tier client-server architecture
  - Peer-to-peer architecture
    - decentralized p2p
    - semi centralized

# Deployment architecture

- เป็นสถาปัตยกรรมที่พิจารณาจากการนำไปติดตั้ง / นำไปใช้
- เกี่ยวข้องกับ
  - ระบบปฏิบัติการ (Operating System)
  - ฮาร์ดแวร์ (Hardware)
  - เครือข่าย (Network)
  - อุปกรณ์ทางกายภาพ (Physical devices)

# Deployment architecture (ต่อ)



ที่มา: JukeBox System, Online Visual Paradigm

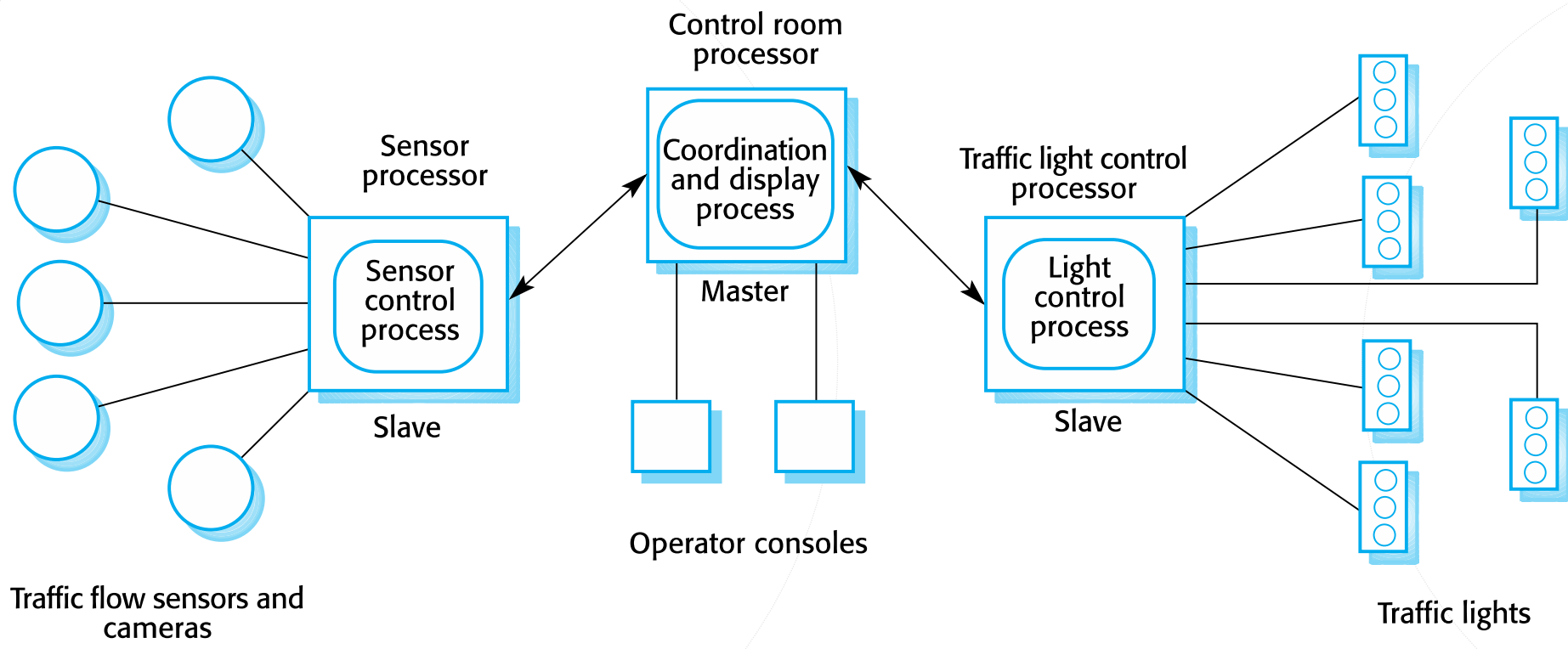
## Deployment architecture patterns: Master-slave architecture

- มักนำไปใช้กับ Real time system
- ระหว่าง Master และ Slave แยกกันประมวลผลข้อมูล
- เจ้านาย (Master) รับผิดชอบในการคำนวณ ติดต่อสื่อสาร และควบคุมการทำงานของ Slave
- บ่าว (Slave) ทำงานในสิ่งที่ถูกมอบหมายให้ทำ

The diagram illustrates a distributed control system for traffic management, organized into three main functional areas connected by a central communication bus.

- Sensor Processor (Slave):** This unit is connected to **Traffic flow sensors and cameras**. It contains a **Sensor control process**.
- Control room processor (Master):** This central unit contains a **Coordination and display process**. It is connected to **Operator consoles**.
- Traffic light control processor (Slave):** This unit is connected to **Traffic lights**. It contains a **Light control process**.

Arrows indicate bidirectional communication between the Sensor processor and the Control room processor, and between the Control room processor and the Traffic light control processor.

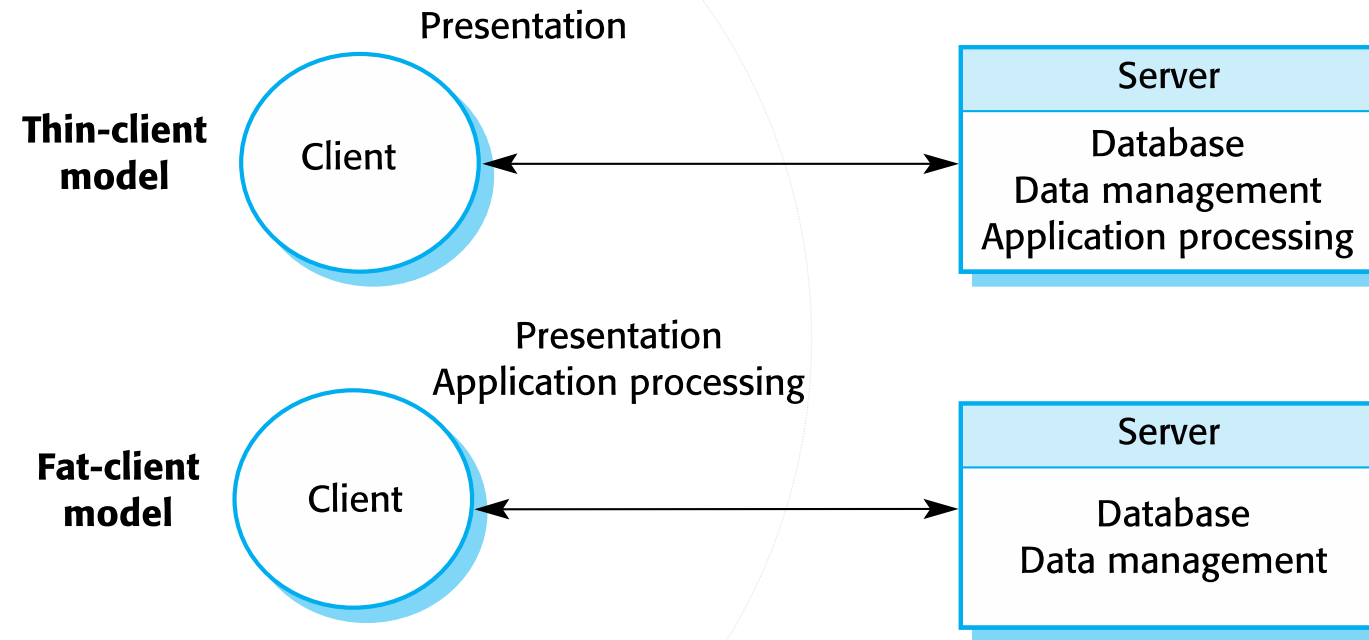




## Deployment architecture patterns: Two-tier client-server

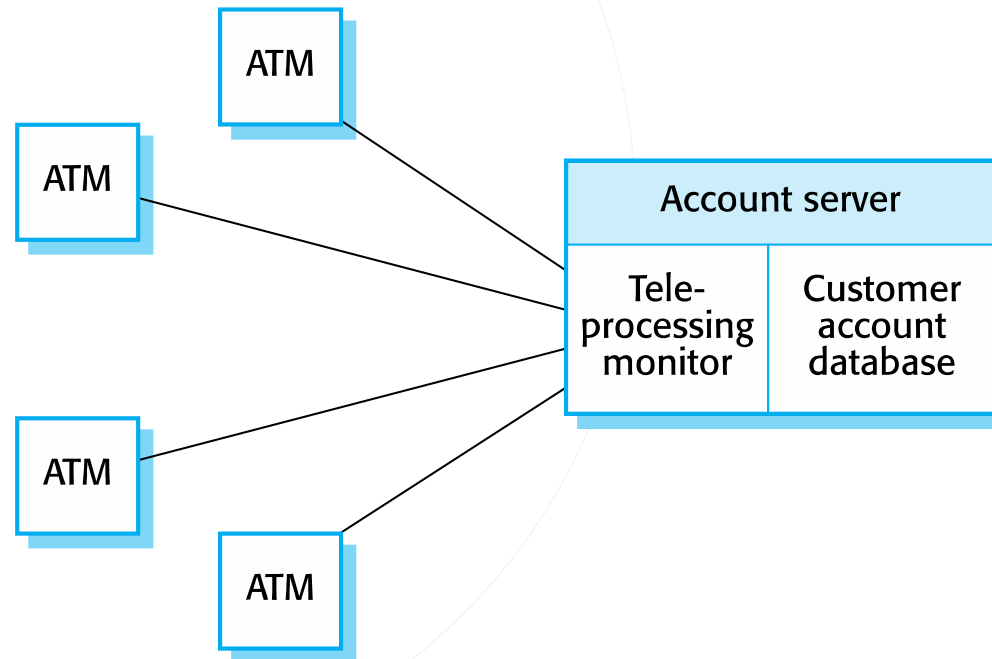
- **Logical server 1 + Clients** ซึ่งเป็นสถาปัตยกรรมแบบกระจายที่เรียบง่ายที่สุด
- แบ่งออกเป็น 2 ประเภทคือ
  - Thin-client model คือ ให้ client ทำหน้าที่เดียวคือ “แสดงผล” นอกนั้นให้โยนภาระการทำงานให้ server (ไม่ต้อง install client และ ต้องใช้พลังของ server และ network มาก)
  - Fat-client model คือ นอกจากจะให้ client ทำหน้าที่แสดงผลแล้ว ยังให้ client ทำ business logic บางส่วน (หรือทั้งหมด) ด้วย ซึ่งทำให้ server ทำหน้าที่จัดการข้อมูลเพียงอย่างเดียว (แต่ต้อง install client)

# Deployment architecture patterns: Two-tier client-server (ต่อ)



## Deployment architecture patterns: Two-tier client-server (ต่อ)

“A fat-client architecture for an ATM system”



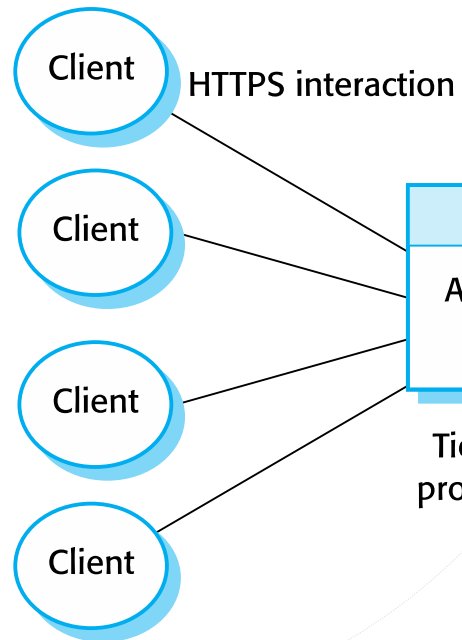
## Deployment architecture patterns: Two-tier client-server (ต่อ)

- ปัจจุบันเทคโนโลยี JavaScript ทำให้ web-base application เข้าใกล้ fat-client เพราะไม่ต้อง install client ก็ทำให้ client สามารถทำงานได้ด้วยตัวเองมากขึ้น
- เทคโนโลยี auto-update ช่วยลดปัญหาการติดตั้ง s/w ที่ client ได้ เนื่องจากมีตัวช่วยจัดการ dependencies

# Deployment architecture patterns: Multi-tier client-server

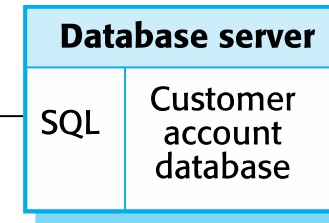
- เกิดขึ้นเพื่อเพิ่มขีดความสามารถในการให้บริการของสถาปัตยกรรมแบบ two-tier
- แยกการทำงานแต่ละส่วนให้อยู่คนละเครื่องกัน

Tier 1. Presentation



Tier 2. Application processing and data management

SQL query

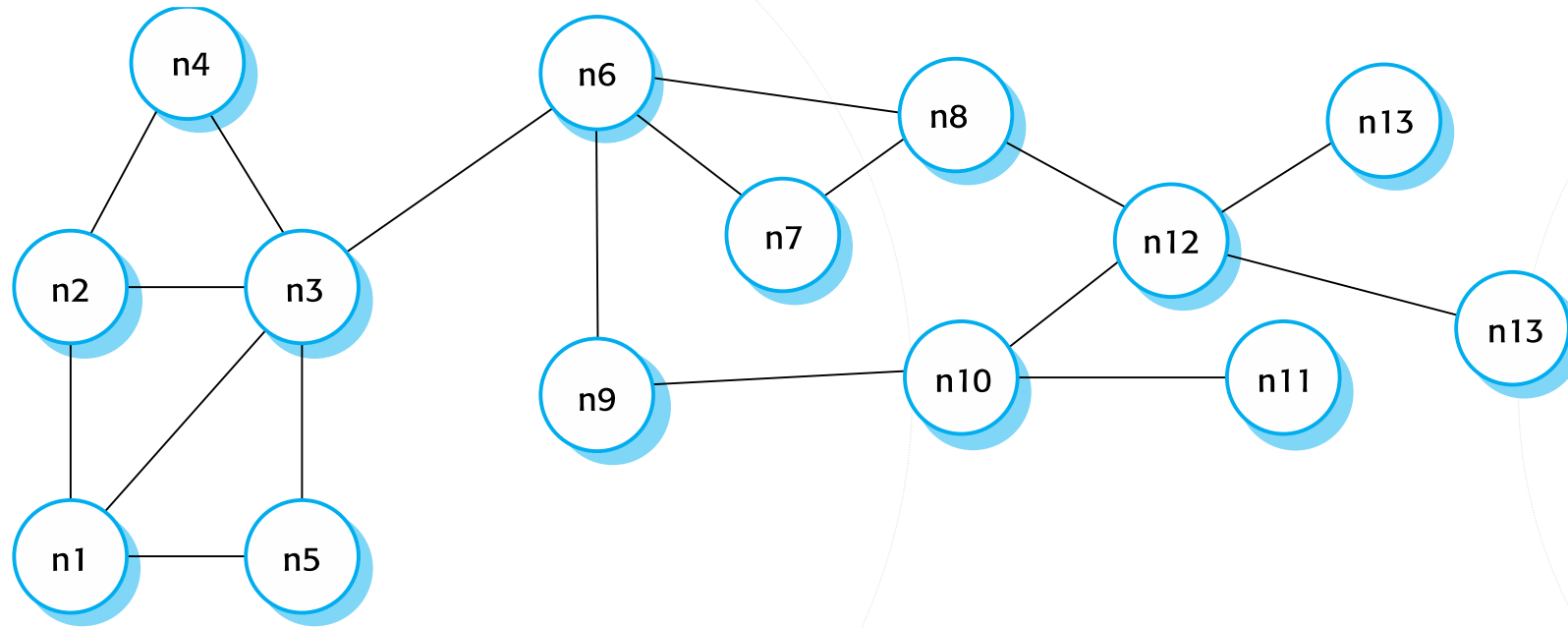


Tier 3. Database processing

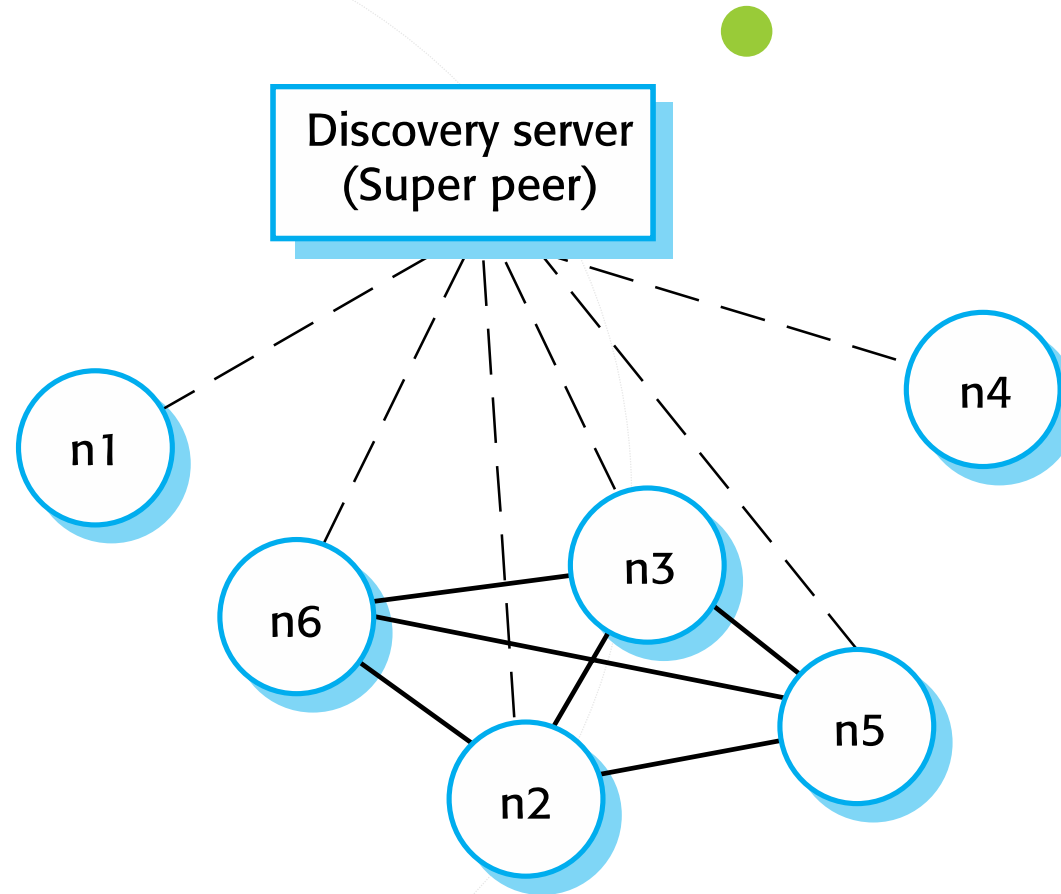
## Deployment architecture patterns: Peer-to-peer

- ไม่มีเครื่องใดเป็นตัวหลัก (Center) ในการให้บริการ แต่เครื่องที่ให้บริการอาจเป็นเครื่องใดเครื่องหนึ่งในเครือข่ายนั้น ๆ
- สถาปัตยกรรมนี้ออกแบบมาเพื่อดึงพลังความสามารถของเครื่องคอมพิวเตอร์ในเครือข่ายมาช่วยกันทำงาน
- ตัวอย่างเช่น การแชร์ไฟล์ผ่าน BitTorrent protocol

# Deployment architecture patterns: Decentralized Peer-to-peer




# Deployment architecture patterns: Semi centralized Peer-to-peer







## Logical architecture

- Overview
  - Logical architecture patterns
    - MVC
    - Layered
    - Micro service
- 

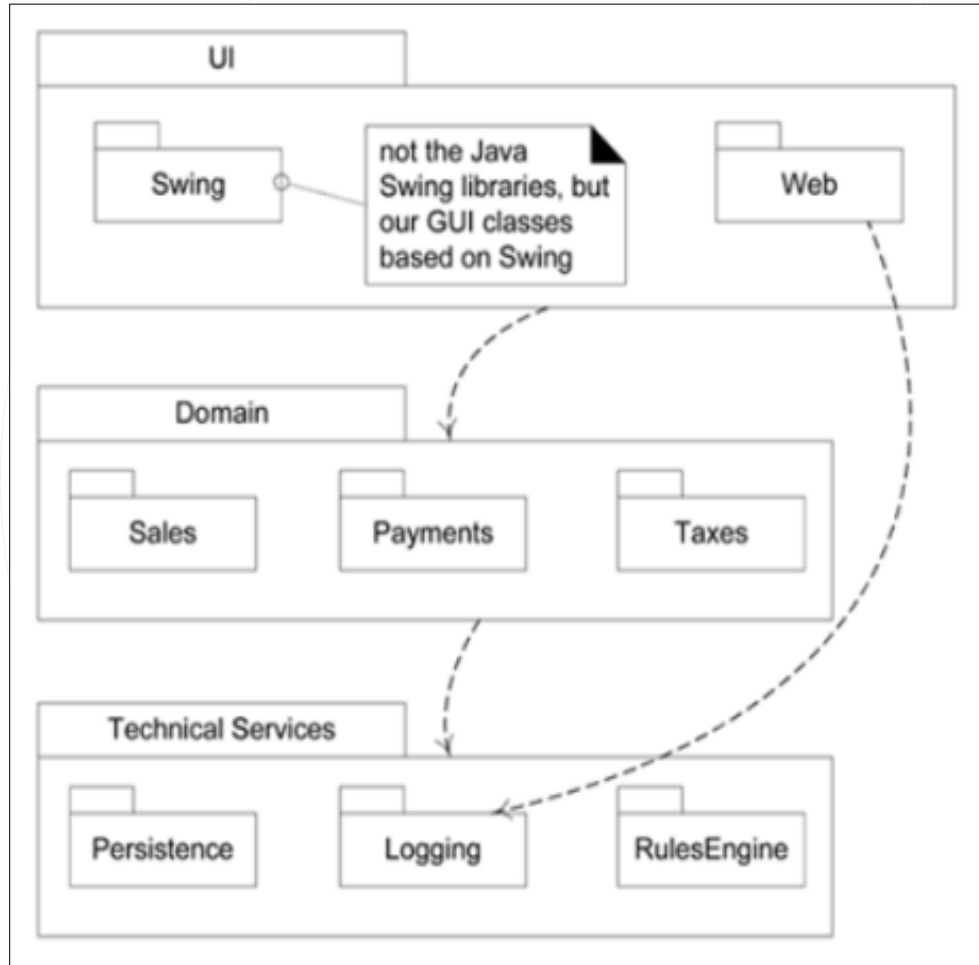
# Logical architecture

- แสดงถึงโครงสร้างสำคัญที่เกิดขึ้นในชั้นความคิด ไม่ยึดติดกับโครงสร้างทางกายภาพ เช่น
  - การกำหนด Package ของคลาส
  - การกำหนด Namespace ของคลาส
- นิยมนำแผนภาพยูเอ็มแอลชื่อ แผนภาพแพ็คเกจ (Package diagram) มาใช้สื่อความหมาย

# ไวยากรณ์พื้นฐานของแผนภาพแพ็คเกจ

สัญลักษณ์	ชื่อเรียก	การนำไปใช้
	Package	ใช้เพื่อแสดงกลุ่มการทำงานของสิ่งใดสิ่งหนึ่ง เช่น กลุ่มคลาส หรือ กลุ่มของแพ็คเกจเป็นต้น
	Dependency	ความไม่เป็นอิสระจากกันระหว่างแพ็คเกจ หรือ การที่แพ็คเกจใดแพ็คเกจหนึ่งอาศัยการทำงานจากแพ็คเกจอื่น ในที่นี้คือ Package A อาศัยการทำงานจาก Package B
	Nested package	ใช้เพื่อแสดงว่าแพ็คเกจหนึ่งสามารถเป็นที่รวบรวมของแพ็คเกจอื่นได้

# ตัวอย่างแผนภาพแพ็คเกจ



```
// --- UI Layer
```

```
com.mycompany.nextgen.ui.swing
com.mycompany.nextgen.ui.web
```

```
// --- DOMAIN Layer
```

```
// packages specific to the NextGen project
com.mycompany.nextgen.domain.sales
com.mycompany.nextgen.domain.payments
```

```
// --- TECHNICAL SERVICES Layer
```

```
// our home-grown persistence (database) access layer
com.mycompany.service.persistence
```

```
// third party
org.apache.log4j
org.apache.soap.rpc
```

```
// --- FOUNDATION Layer
```

```
// foundation packages that our team creates
com.mycompany.util
```

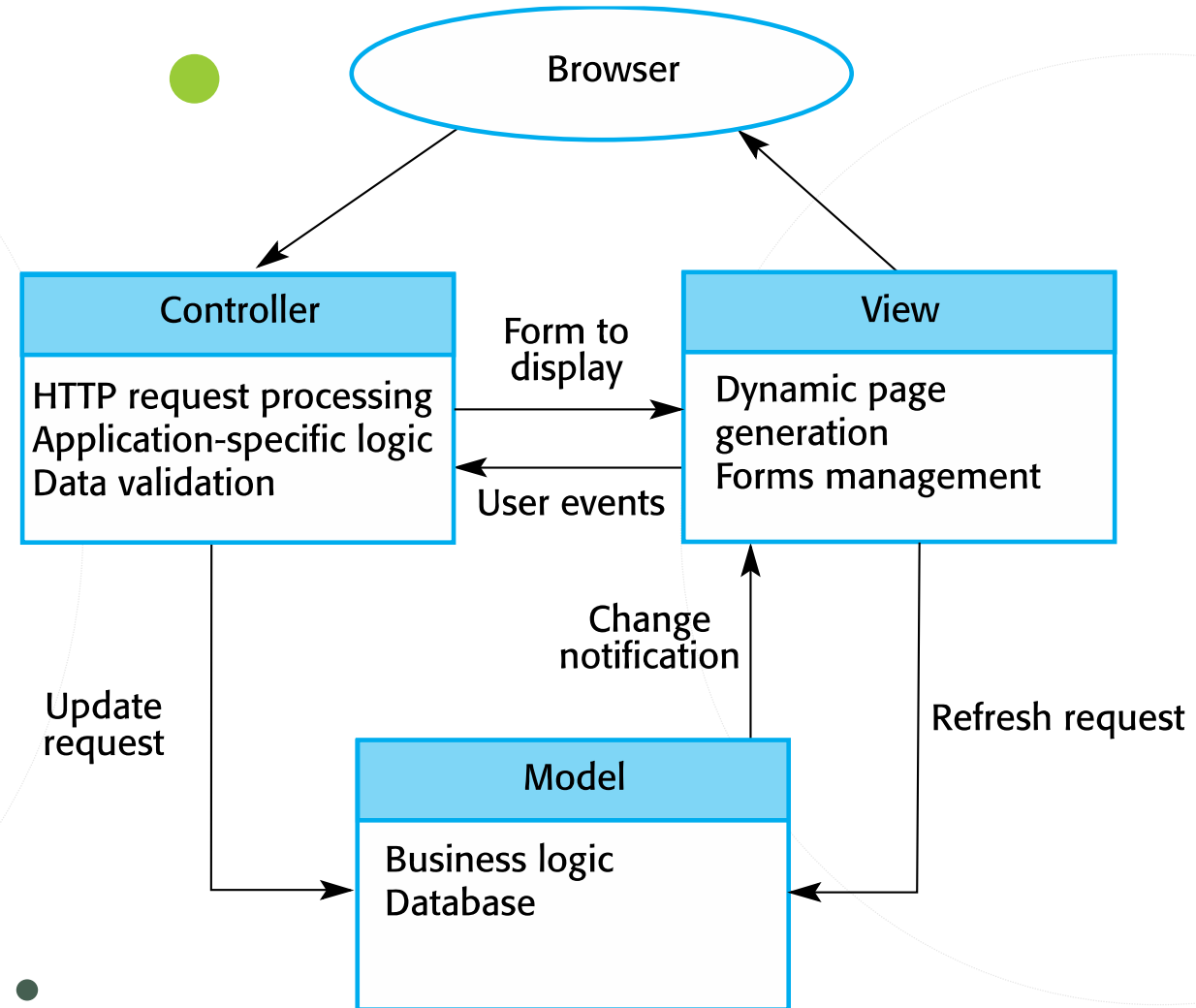
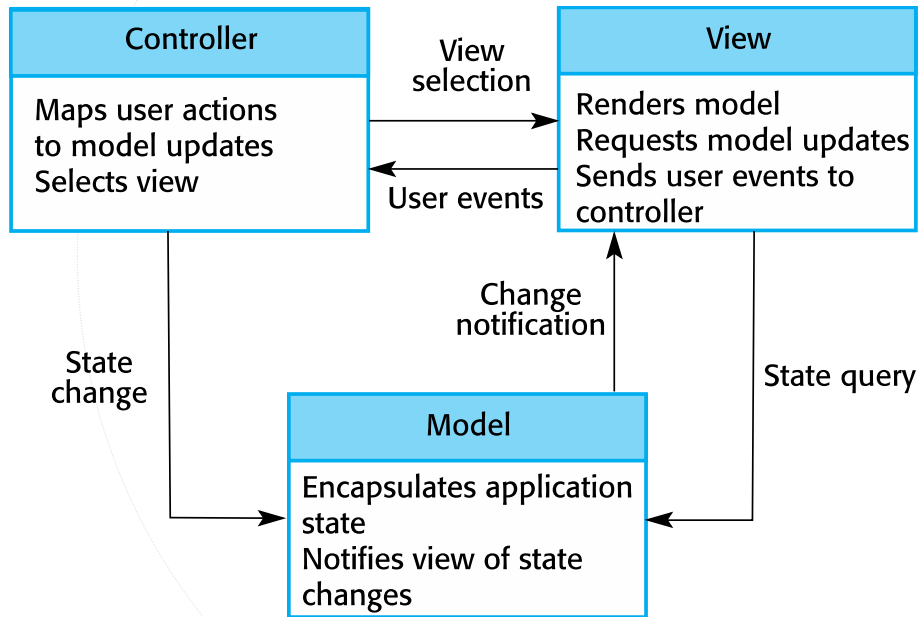
# Logical architecture patterns: Model – View - Controller

- คือ สถาปัตยกรรมที่แบ่งส่วนการทำงานออกตามหน้าที่ ประกอบด้วย 3 ส่วน
  - Controller คือ ส่วนการควบคุมรีเคสที่เกิดขึ้นจากผู้ใช้ และส่งการทำงานต่อไปยัง View หรือ Model (ส่วนมากจะเรียกการทำงานไปยัง model ตามที่รีเคสกำหนด และเรียกใช้ View ตามความเหมาะสม)
  - View คือ ส่วนที่ทำหน้าที่แสดงผลข้อมูลต่อผู้ใช้
  - Model คือ ส่วนจัดการข้อมูล (Data) และการทำงานที่เกี่ยวข้องกับข้อมูล (Business rules)
  - กำหนดการเข้าถึงแต่ละส่วนเป็นแบบ 3 ทาง View -> Controller -> Model -> View
- เกิดจากแนวคิด การแยกส่วนและทำให้องค์ประกอบของซอฟต์แวร์เป็นอิสระจากกัน เพื่อจำกัดขอบเขตของผลกระทบเมื่อเกิดการเปลี่ยนแปลง
- นำไปใช้เมื่อ
  - ข้อมูลที่ระบบต้องจัดการ สามารถแสดงผลได้หลายรูปแบบ (View) และมีปฏิสัมพันธ์กับผู้ใช้ในหลายลักษณะ (Controller)
  - เมื่อไม่สามารถควบคุมการเปลี่ยนแปลงที่เกิดขึ้นกับระบบได้

## Logical architecture patterns: Model – View – Controller (ต่อ)

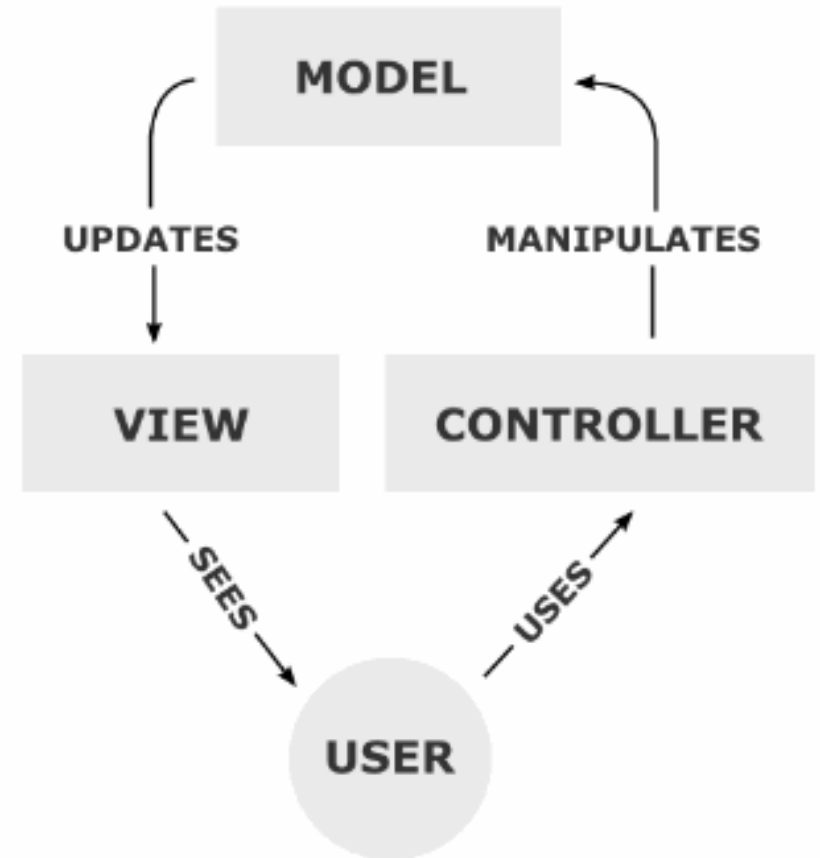
- Web-base ส่วนใหญ่ ใช้ Pattern นี้ในการบริหารจัดการปฏิสัมพันธ์ที่เกิดขึ้น
- คิดค้นขึ้นเพื่อให้ Font-End developer และ Back-end developer พัฒนาระบบไปพร้อมกันได้ และโค้ดไม่ทับกัน
- ข้อดี: การเปลี่ยนแปลงข้อมูลเป็นอิสระจากการแสดงผล ลดความซ้ำซ้อนของโค้ด ด้วยการสร้าง multiple views จาก model เพียงอันเดียว
- ข้อเสีย: โค้ดมีความซับซ้อน

# Logical architecture patterns: Model – View – Controller (ต่อ)



## Logical architecture patterns: Model – View – Controller (ต่อ)

- ชื่อ MVC เหมือนกัน  
แต่อาจมี “**ลักษณะการทำงาน**” ที่แตกต่างกัน  
ออกไป
- model เปรียบเหมือน “**สะพานเชื่อม**”  
ระหว่าง View และ Controller





## Example (Mini – MVC)

```
1  <?php
2  include(__DIR__.'\model.php');
3  include(__DIR__.'\controller.php');
4  include(__DIR__.'\view.php');
5  $modelObj = new Model();
6  $controllerObj = new Controller($modelObj);
7  $viewObj = new View($controllerObj, $modelObj);
8
9  if(isset($_REQUEST['action']) && !empty($_REQUEST['action'])){
10 |     $controllerObj->clicked();
11 | }
12
13 echo $viewObj->output();
```

# Example (Mini – MVC)

```
1  <?php
2  class Model{
3      private $text;
4      public function __construct(){
5          $this->text = "Please click here and look what gonna happen!";
6      }
7      public function setText($text){
8          $this->text = $text;
9      }
10     public function getText(){
11         return $this->text;
12     }
13 }
```

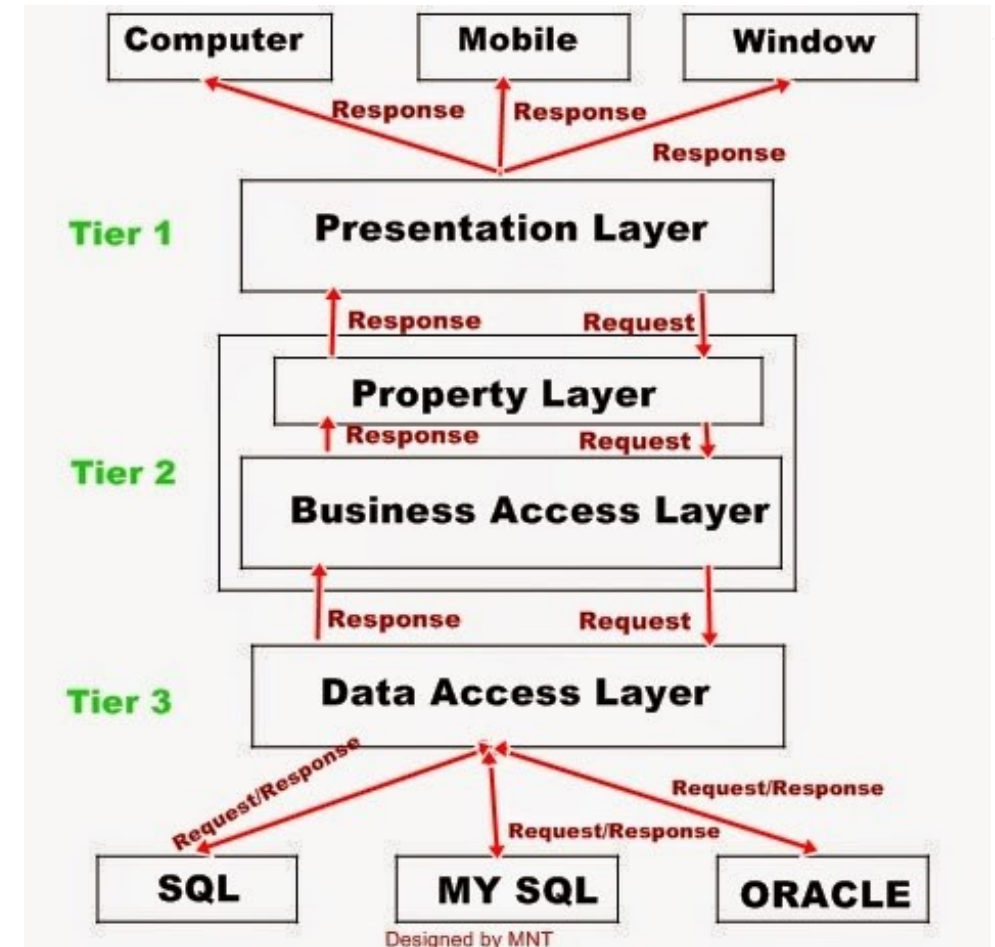
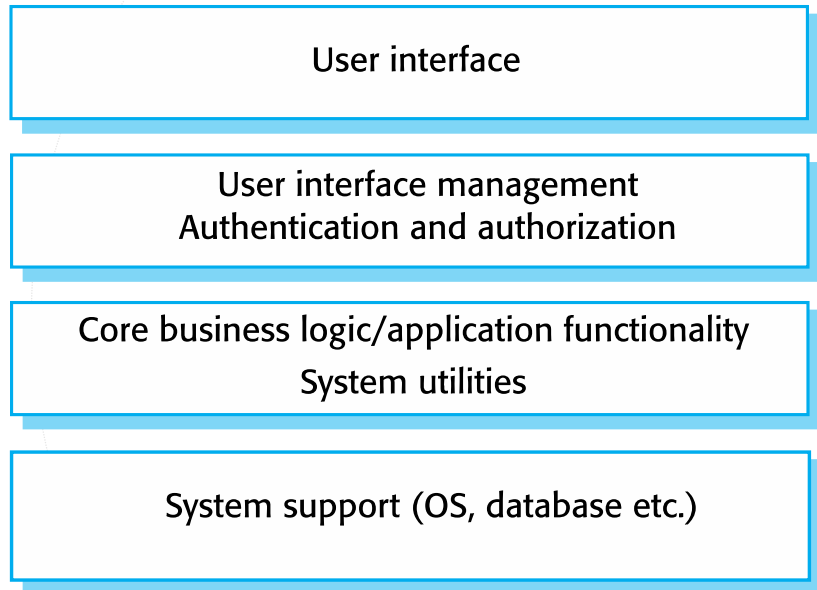
```
1  <?php
2  class Controller{
3      private $model;
4      public function __construct($model){
5          $this->model = $model;
6      }
7
8      public function clicked(){
9          $this->model->setText("This ain't MAGIC <br>");
10     }
11 }
```

```
1  <?php
2  class View{
3      private $model;
4      private $controller;
5      public function __construct($controllerObj,$modelObj){
6          $this->model = $modelObj;
7          $this->controller = $controllerObj;
8      }
9      public function output(){
10         return '<p><a href="index.php?action=clicked">'
11             . $this->model->getText(). '</a></p>';
12     }
13 }
```

## Logical architecture patterns: Layered

- ตั้งต้นมาจากแนวคิดการแยกส่วน และทำให้แต่ละส่วนมีความเป็นอิสระจากกัน
- แบ่งระบบออกเป็นชั้น โดยแต่ละชั้นแบ่งแยกตามฟังก์ชันที่ให้บริการอย่างชัดเจน
- การทำงานของชั้นบน ต้องพึ่งพาการทำงานชั้นล่าง (ที่ติดกัน)
- การเข้าถึงแต่ละชั้นเป็นแบบ linear (มีลำดับ ห้ามข้ามชั้น) ติดต่อกันผ่าน interface ที่กำหนดไว้เท่านั้น
  - Presentation Layer (UI) <--> Business Access Layer <-> Data Access Layer
- เหมาะกับ NFRs ที่เน้น Information Security แต่ไม่เน้น Performance
- ความแตกต่างระหว่าง Layered และ MVC
  - การเปลี่ยนแปลงที่ไม่เป็นอิสระจากกัน เพราะ layer บน อาศัยการทำงานของ layer ด้านล่าง
  - Layered เป็นการเข้าถึงแต่ละส่วนในระบบ Linear แต่ MVC เข้าถึงแบบ Triangular

# Logical architecture patterns: Layered (ต่อ)



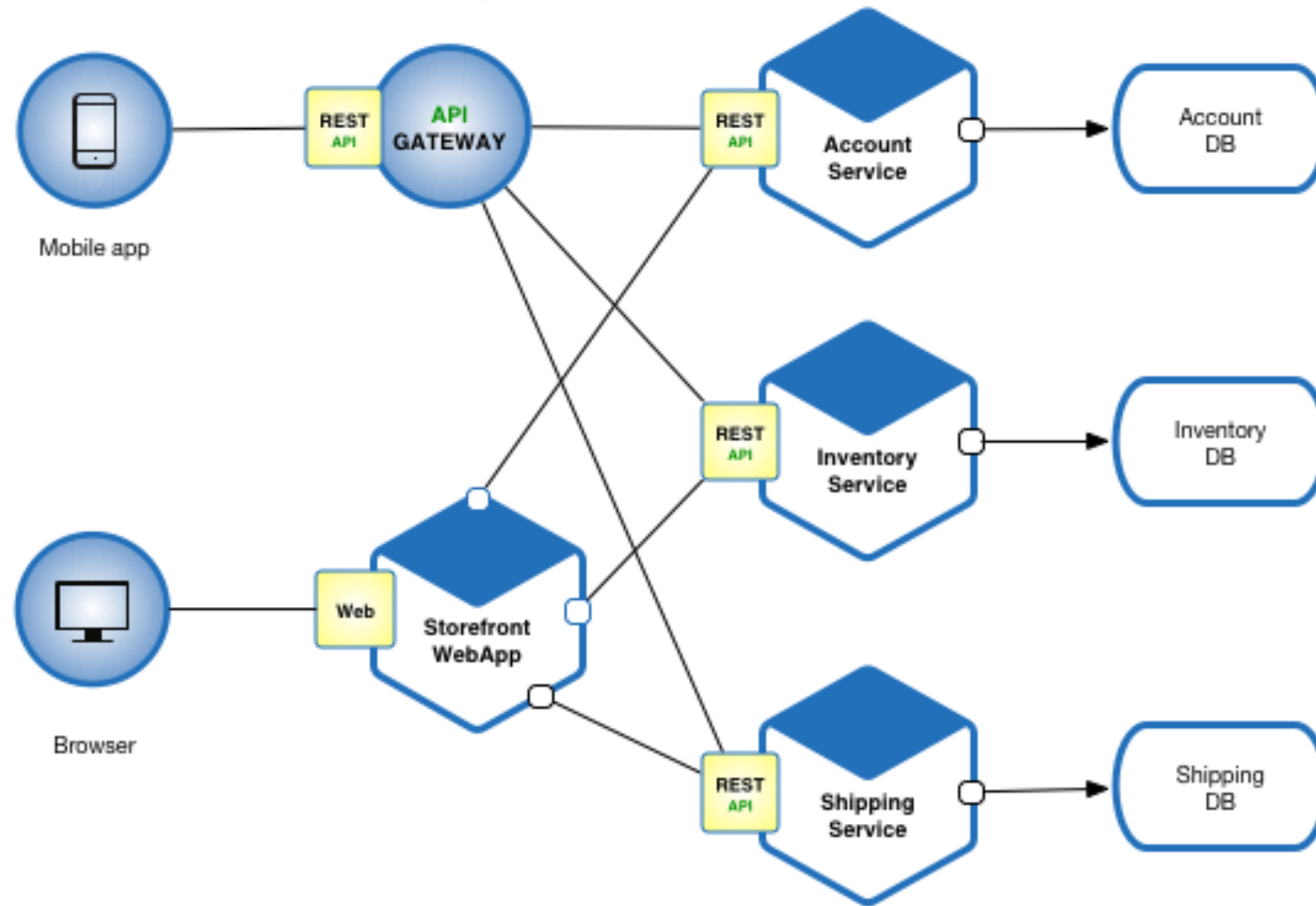
## Logical architecture patterns: Micro service

- การพัฒนาซอฟต์แวร์ ให้อยู่ในรูปแบบ “กลุ่มของซอฟต์แวร์บริการย่อยหลาย ๆ กลุ่ม”
- ซอฟต์แวร์ที่มีหน้าที่ให้บริการ มีความสามารถเฉพาะที่แตกต่างกันไปในแต่ละกลุ่ม
- โดยแต่ละกลุ่มจะรันแยกโปรเซส และมักให้บริการผ่าน HTTP API
- ซอฟต์แวร์แต่ละกลุ่มไม่จำเป็นต้องเขียนด้วยภาษาเดียวกัน และเชื่อมต่อฐานข้อมูลคนละฐานกันก็ได้
- การแก้ไขโค้ดของบริการหนึ่ง จะไม่ส่งผลกระทบกับบริการอื่น (Loose Coupling)

## Logical architecture patterns: Micro service (ต่อ)

Advantage	Disadvantage
ทำให้เกิดโครงสร้างของโมดูลที่ชัดเจน เหมาะกับระบบขนาดใหญ่	เสี่ยงต่อความผิดพลาด (เนื่องจากระบบปลายทาง ล้มเหลว)
การ Deploy ทำแยกกันได้	ต้องอาศัยการดูแลระบบอย่างสม่ำเสมอจากหลายส่วน งาน
รองรับเทคโนโลยีที่หลากหลาย	การนำไปใช้มีความซับซ้อน เพราะหนึ่งโปรแกรม = หนึ่งเซิร์ฟเวอร์ $\geq$ หนึ่งโปรเซส จึงทำให้การบริหารคน ดูแลยาก

## Logical architecture patterns: Micro service (ต่อ)



## Logical architecture patterns: Micro service (ต่อ)

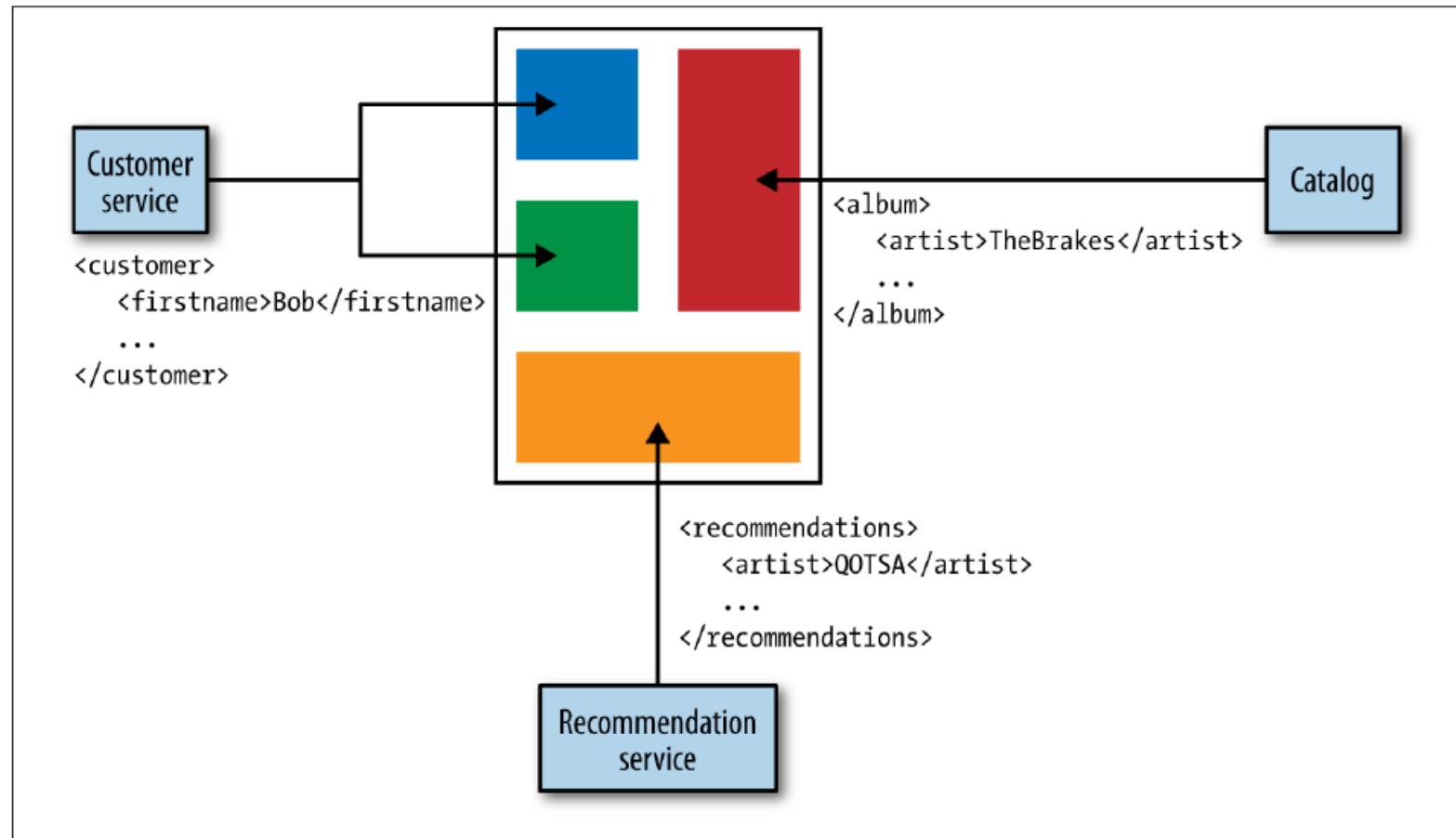


Figure 4-7. Using multiple APIs to present a user interface



# อ้างอิง

- Software engineering 10<sup>th</sup> edition, Ian Sommerville
- <https://martinfowler.com/architecture/>
- <https://blogs.msdn.microsoft.com/malaysia/2013/08/02/layered-architecture-for-net/>
- <https://coderwall.com/p/l-a79g/the-principles-of-the-mvc-design-pattern>
- <https://blog.eventuate.io/2017/01/04/the-microservice-architecture-is-a-means-to-an-end-enabling-continuous-deliverydeployment/>
- Building Microservice 1 st edition, Sam Newman

