01076105, 01076106

Object Oriented Programming

Object Oriented Programming Project

Use Case Diagram

Software Development Life Cycle

- วงจรการพัฒนาซอฟต์แวร์

Software Development Life Cycle

- Requirements คือ การหาความต้องการของซอฟต์แวร์ เป้าหมายเพื่อจะตอบคำถามว่า ซอฟต์แวร์นี้ใช้สำหรับทำอะไร และทำงานอะไรได้บ้าง (What?)
  Output ของขั้นตอน Requirement คือ Software Specification ซึ่งโดยทั่วไปจะอยู่ในรูปแบบของข้อความ
- Analysis คือ การนำเอาความต้องการมาวิเคราะห์ ซึ่งการวิเคราะห์จะมีหลายระดับ ตั้งแต่วิเคราะห์ว่าซอฟต์แวร์ควรมีโครงสร้างการทำงานอย่างไร มีขั้นตอนการทำงานอย่างไร อาจรวมไปถึงมีส่วนติดต่อผู้ใช้แบ่งเป็นกี่ส่วน โดยส่วนขั้นตอนการทำงานหรือมักเขียนในรูปแบบ Use Case Diagram และ Use Case Description (How?)
  Output ของขั้นตอนนี้ คือ Use Case Diagram

Software Development Life Cycle

- Design คือ การนำเอาการวิเคราะห์มาออกแบบ โดยแบ่งออกเป็นการออกแบบส่วนติดต่อผู้ใช้ (User Interface Design) และการออกแบบโครงสร้างการทำงานของโปรแกรม (Software Design)
  โดยทั่วไปขั้นตอน Analysis กับ Design มักทำควบคู่กัน เรียกว่า Analysis and Design
  Output ของการออกแบบมักอยู่ในรูปของ Diagram โดยส่วนของโครงสร้างการทำงานในแบบ Object Oriented จะได้เป็น Class Diagram, Sequence Diagram และอื่นๆ ในส่วนของ UI Design ก็จะได้เป็น Wireframe หรือ UI Screen
- Coding คือ การนำเอาการออกแบบที่ได้ทำไว้ มาเขียนเป็นซอฟต์แวร์ โดยให้มีผลการทำงานตามที่ออกแบบไว้ โดยอาจมีรูปแบบการพัฒนาหลายแบบ เช่น Agile, Scrum, Extreme Programming, Lean หรืออื่นๆ

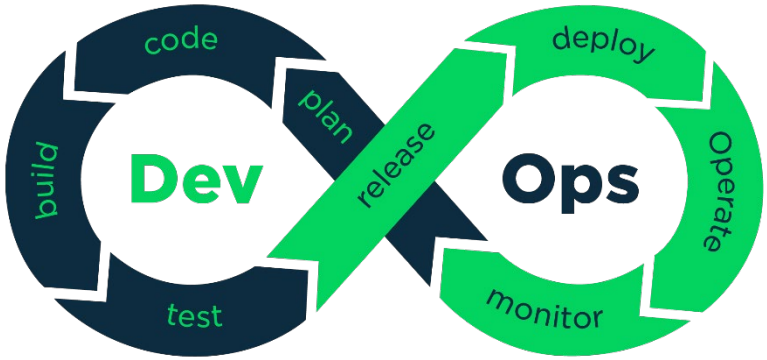
Software Development Life Cycle

- Testing คือ ขั้นตอนการทดสอบโปรแกรม ซึ่งในปัจจุบัน ถือได้ว่าเป็นขั้นตอนที่มีความสำคัญมาก ขนาดที่มีตำแหน่งงานที่ทำเรื่องการทดสอบโปรแกรมโดยเฉพาะ การทดสอบโปรแกรมแบ่งออกเป็น
  - Unit Testing คือการทดสอบระดับฟังก์ชันหรือคลาส
  - Integrate Testing คือการทดสอบเมื่อนำโปรแกรมมารวมกัน
  - User Acceptance Test (UAT) คือ การทดสอบในรูปแบบที่มีการจำลองการใช้งานจริง

  ปัจจุบัน Testing มีการพัฒนาเป็น Automate Testing คือ มีซอฟต์แวร์สำหรับช่วยการทดสอบโดยเฉพาะ โดยมีการจัดทำเป็น Test Script

Software Development Life Cycle

- Deployment คือ ขั้นตอนการนำซอฟต์แวร์ไปสู่การใช้งานจริง ตั้งแต่การติดตั้งเครื่องคอมพิวเตอร์ การติดตั้งซอฟต์แวร์พื้นฐาน จนถึงการติดตั้งซอฟต์แวร์ที่พัฒนา ในปัจจุบันงานด้านการ Deployment ได้พัฒนาไปมาก เนื่องจากการแข่งขันทางธุรกิจทำให้ความต้องการ**ความถี่**ของการ Deployment เพิ่มขึ้นอย่างมาก ซอฟต์แวร์บางตัวอาจมีการ Deploy ทุกวัน เนื่องจากมีการเพิ่ม feature ต่างๆ เข้าไปในซอฟต์แวร์ และ**เวลา**ที่ใช้ในการ Deploy ต้องสั้นด้วย เช่น Netflix สามารถ Deploy ได้ภายใน 1 นาที การจะบรรลุทั้งความถี่และเวลาที่ใช้ในการ Deploy ต้องอาศัยการทำงานที่เรียกว่า DevOps ซึ่งจะหาวิธีการร่วมกันระหว่าง Coder, Tester และ DevOps เพื่อให้การปรับปรุงซอฟต์แวร์ทำได้เร็วและบ่อยตามต้องการได้



การหา Requirement

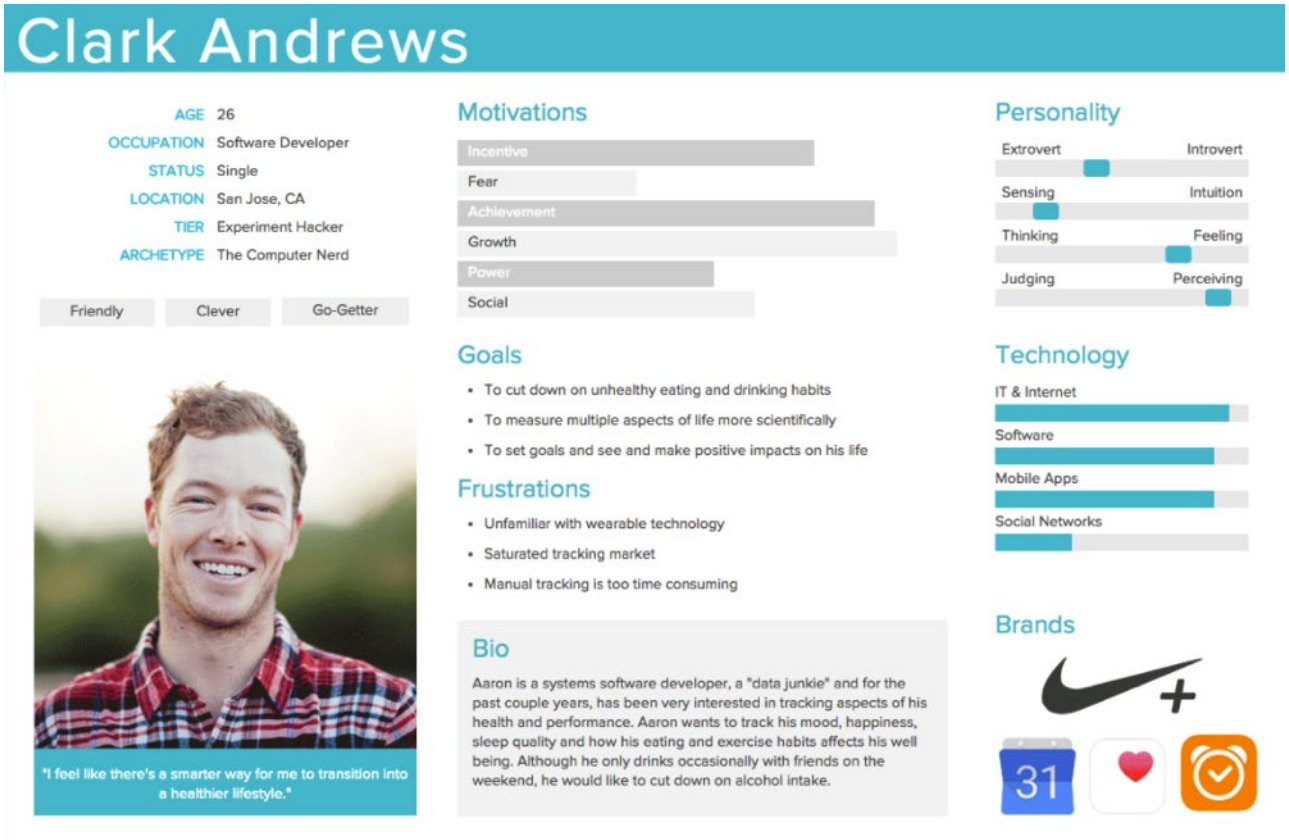
- การหา Requirement หรือ หาความต้องการของซอฟต์แวร์ เป็นงานที่สำคัญมากงานหนึ่ง เพราะ Requirement ถือเป็น “สารตั้งต้น” ของซอฟต์แวร์ หาก Requirement ไม่ครบ หรือ ผิดพลาด ก็จะพาให้ซอฟต์แวร์ที่พัฒนาขึ้นผิดพลาดตามไปด้วย

- Software Developer ที่มีทักษะในด้าน Requirement จะเป็นบุคลากรที่ทรงคุณค่ามากในองค์กร เพราะเป็นงานที่หาคนเก่งยาก และ “ฝึก” ได้ยาก

การหา Requirement

- การหา Requirement จะมีเครื่องมือหลายชนิดที่นิยมใช้งาน

- Persona คือ แบบจำลองบุคคลที่รวบรวมลักษณะเฉพาะ พฤติกรรม แรงจูงใจ และความต้องการของกลุ่มผู้ใช้กลุ่มใดกลุ่มหนึ่งในกลุ่มเป้าหมายทั้งหมด



การหา Requirement

- Persona มักใช้กับซอฟต์แวร์ประเภทใช้ในวงกว้าง เช่น Shopee, Banking, Social, E-commerce หรืออื่นๆ ซึ่งมีกลุ่มผู้ใช้หลากหลาย ความต้องการแตกต่างกัน
- จึงต้องจัดทำ Profile สมมติ ที่เป็นตัวแทนของผู้ใช้แต่ละกลุ่ม ประกอบด้วย
  - ข้อมูลส่วนตัว (Demographic), ความสนใจ, พฤติกรรม, การใช้เทคโนโลยี
  - เป้าหมาย (Goal) เป็นส่วนที่สำคัญ ควรเป็น Goal เฉพาะที่เกี่ยวข้องกับ Software ที่จะพัฒนา เช่น หากเป็นเรื่องการท่องเที่ยว เป้าหมายก็ควรเกี่ยวกับการท่องเที่ยว
  - ความขัดข้อง (Frustration) บางทีจะเรียก pain point โดยเป็นสิ่งที่คนนั้นมีความต้องการ แต่มีอุปสรรคในการได้มาซึ่งความต้องการ เช่น อยากได้แผนท่องเที่ยวที่ถูกใจโดยไม่ต้องใช้เวลาในการ Survey นาน
- หากทำ Persona ได้ครอบคลุม ผู้ออกแบบซอฟต์แวร์จะทราบว่าผู้ใช้ต้องการอะไร

การหา Requirement

- User Story เป็นอีกเครื่องมือที่นิยมใช้กัน ในการหาความต้องการของผู้ใช้ โดยมักมีรูปแบบดังนี้

> *As.....*
> *I want.....*
> *So that.....*

- As หมายถึง ผู้ใช้ในแต่ละ Role, I want คือ ต้องการจะทำอะไร, so that ดังนั้น ผู้พัฒนาซอฟต์แวร์จะมีอะไรมาตอบสนองความต้องการ เช่น

As ผู้ซื้อสินค้าจากร้าน A (สมมติว่าขายเสื้อผ้า)

I want ต้องการ เลือก เสื้อ 2 ตัวมาเปรียบเทียบข้างๆ กันได้

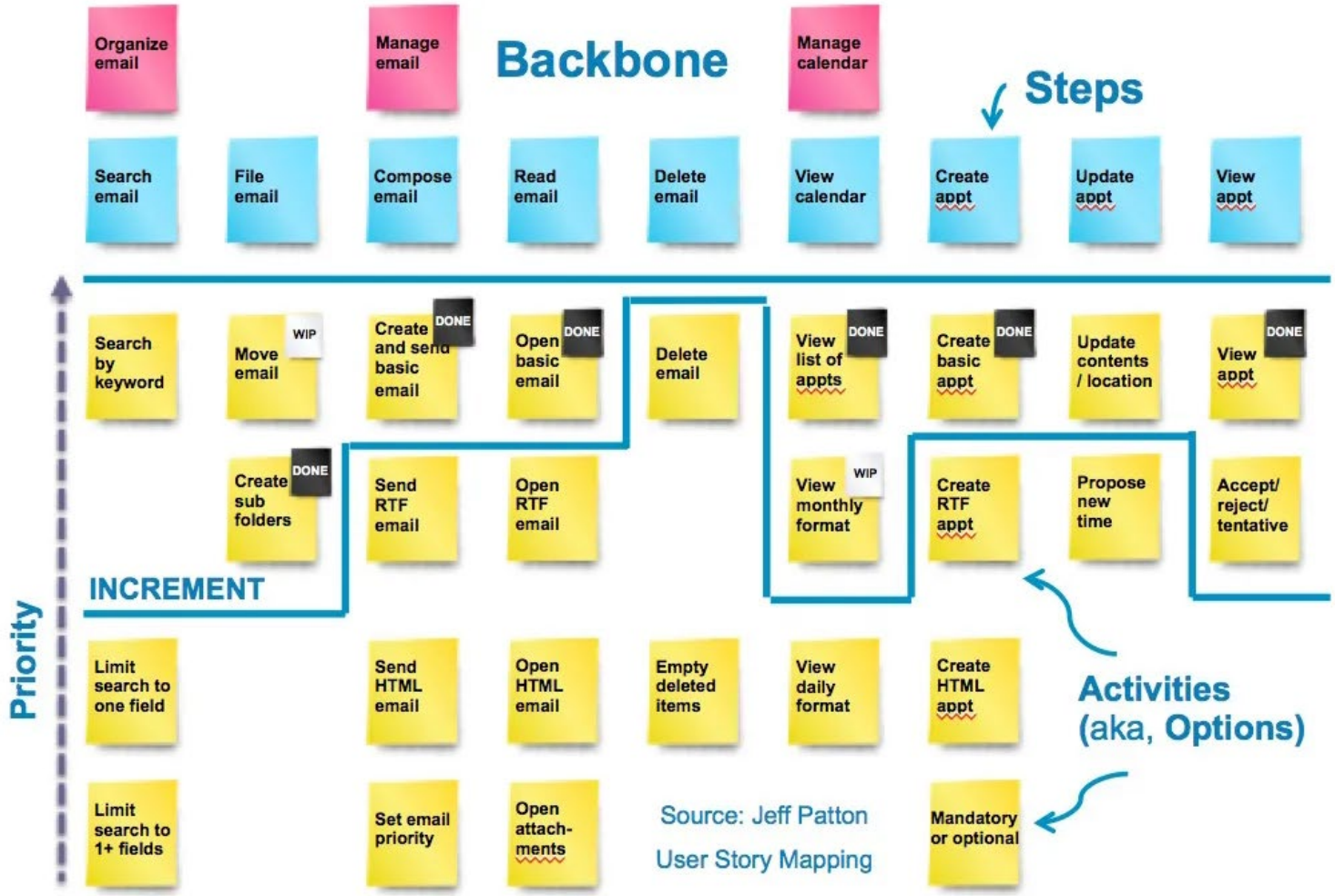
So that ซอฟต์แวร์ควรมี Feature นำ 2 สินค้ามา compare กัน

การหา Requirement

- จาก User Story ใน slide ก่อนหน้า อาจจะยังไม่ชัดเจน ที่จะอธิบายได้ว่า story นี้ developer จะต้องทำอะไรบ้าง
- ดังนั้นในขั้นตอนต่อไปจะมีการกำหนดสิ่งที่เรียกว่า DOD หรือชื่อเต็มชื่อ Definition of Done ให้กับแต่ละ Story ซึ่งก็ตรงกับชื่อ คือ จะรู้ได้อย่างไรว่า Software ที่พัฒนาขึ้นตอบสนองกับ Story นั้นแล้ว ดังนั้นต้องกำหนดรายละเอียดลงไป เช่น
  - ✓ **ระบบจะต้องมี ช่อง checkbox สำหรับใช้เลือกสินค้าที่จะเปรียบเทียบ โดยเลือกได้ไม่เกิน 2 ช่องเท่านั้น**
  - ✓ **ระบบจะต้องมีปุ่มที่เขียนว่า “compare” โดยเมื่อกดจะเปิดหน้าต่างใหม่แล้วนำสินค้า 2 ชิ้นมาแสดงในรูปแบบเปรียบเทียบ**
  - ✓ **รูปแบบเปรียบเทียบต้องแสดงผล...**
  - ✓ **...**

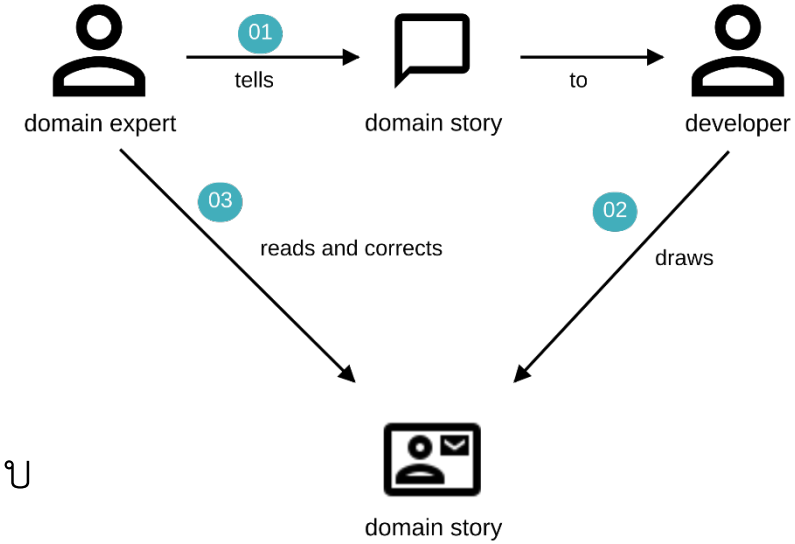
การหา Requirement

- ขั้นตอนในการทำ User Story ยังมีขั้นตอนตามหลังอีกมาก ซึ่งไม่กล่าวในวิชานี้



Domain Storytelling

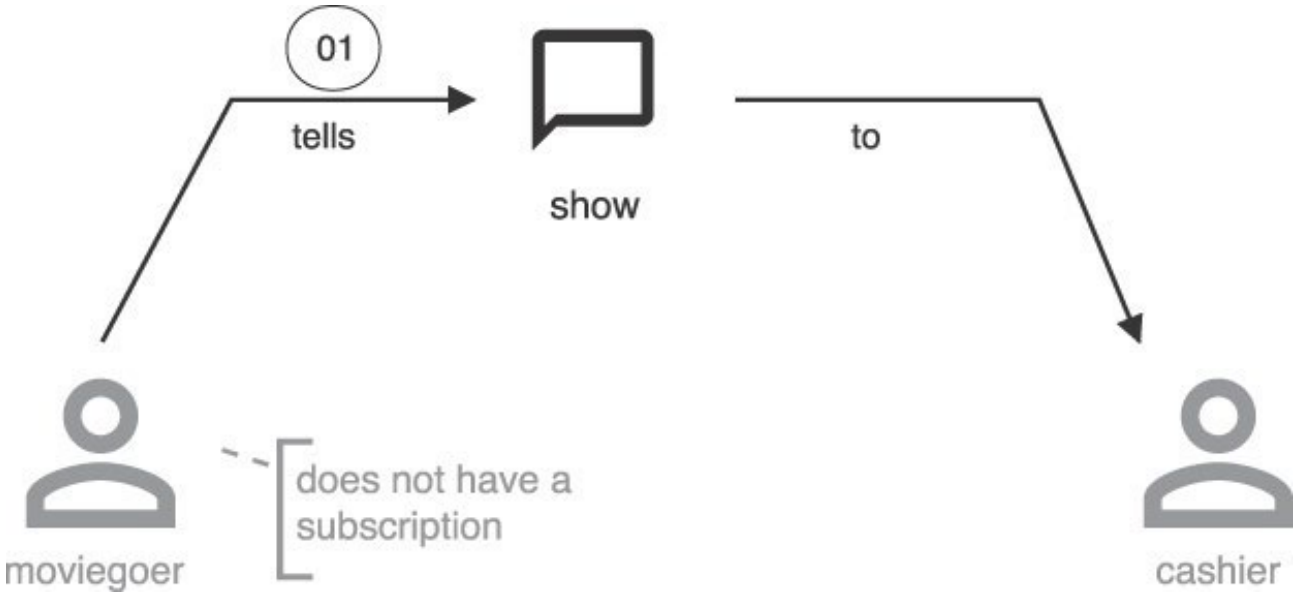
- เราจะลองใช้วิธีการที่เรียกว่า Domain Storytelling (DST) โดยคำว่า Domain มีความหมายถึง เรื่องราว หรือ กรอบ หรือ ขอบเขตที่จะพัฒนาซอฟต์แวร์ วิธีการนี้นำเสนอโดย Stefan Hofer and Henning Schwentner เพื่อใช้ทำความเข้าใจร่วมกันระหว่างเจ้าของงาน และ ผู้พัฒนาซอฟต์แวร์
- DST มีองค์ประกอบ 3 อย่าง ได้แก่
  - Actor หมายถึงผู้ใช้ของระบบ
  - Work Objects หมายถึง สิ่งต่างๆ ที่ไม่ใช่มนุษย์
  - Activity หมายถึง กิจกรรมที่เกิดขึ้นในระบบ



- สามารถศึกษาเพิ่มเติมได้จาก https://domainstorytelling.org/

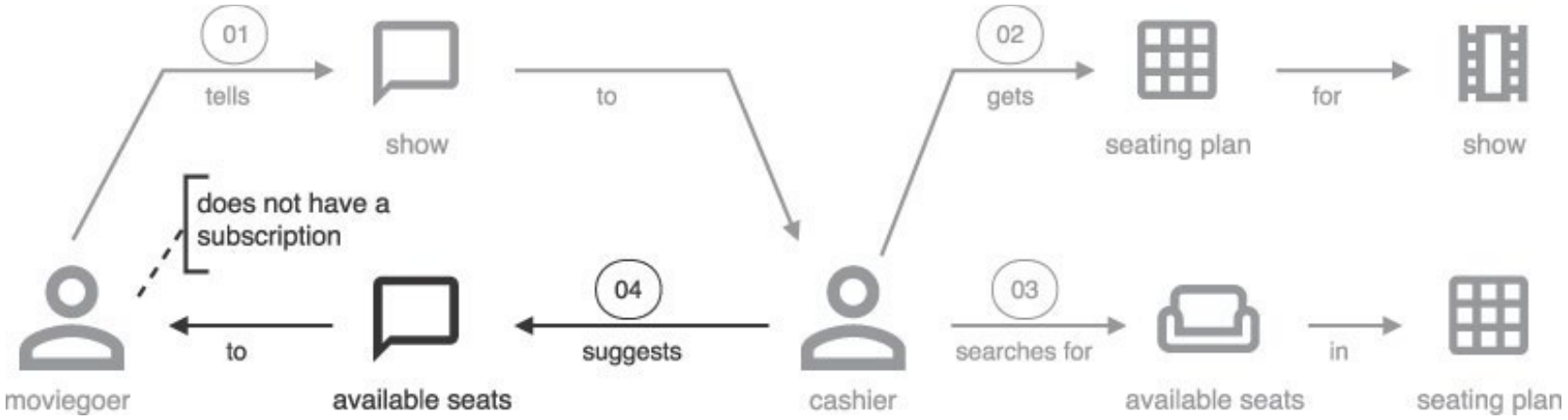
Domain Storytelling

- จะยกตัวอย่างโรงภาพยนต์แห่งหนึ่ง ซึ่งต้องการนำซอฟต์แวร์เข้ามาใช้
- เริ่มต้นต้องมี ผู้ดูหนัง ซึ่งขอเรียกว่า “คอหนัง” (moviegoer) ซึ่งวาดเป็นรูปคนและมีชื่อ Actor กำกับ บอกชื่อหนังที่ต้องการดูและรอบฉายให้กับ จนท. ซึ่งขอตั้งชื่อเป็น cashier
- ก็เขียนได้ตามรูป สำหรับ ป้ายกำกับ (Annotation) มีไว้เพื่อรายละเอียดของ Actor



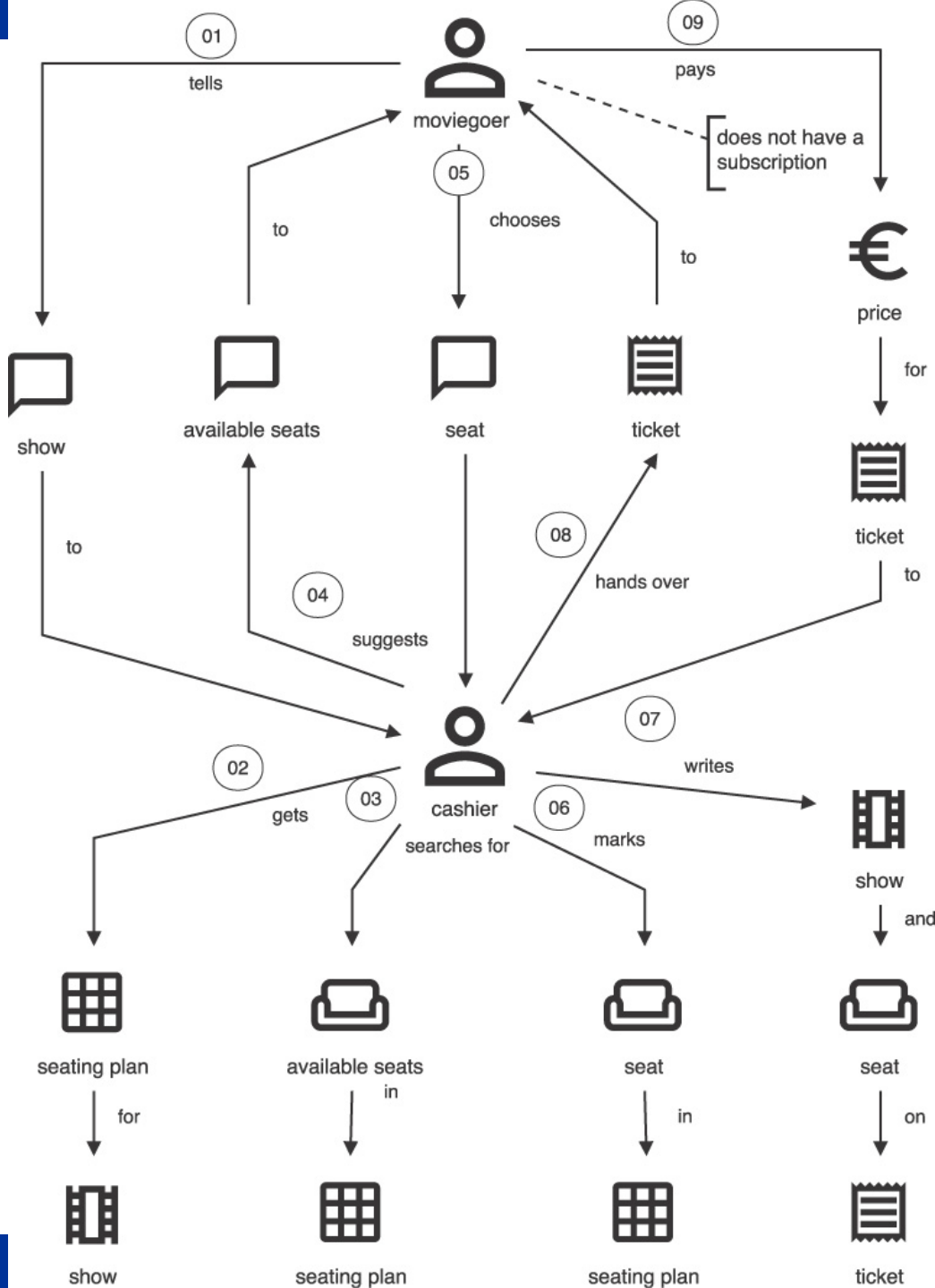
Domain Storytelling

- จากนั้น cashier จะหยิบผังที่นั่งมาดู (2) และ ดูว่ามีที่นั่งว่างตรงไหนบ้างในผังที่นั่ง (3) จากนั้นก็บอกที่ว่างให้กับคอหนัง (4) สามารถเขียนขั้นตอนได้ตามรูป
- จะเห็นว่าแผนผังสามารถแสดงขั้นตอนการทำงานได้ละเอียด และ เข้าใจได้ทั้งสองฝ่าย



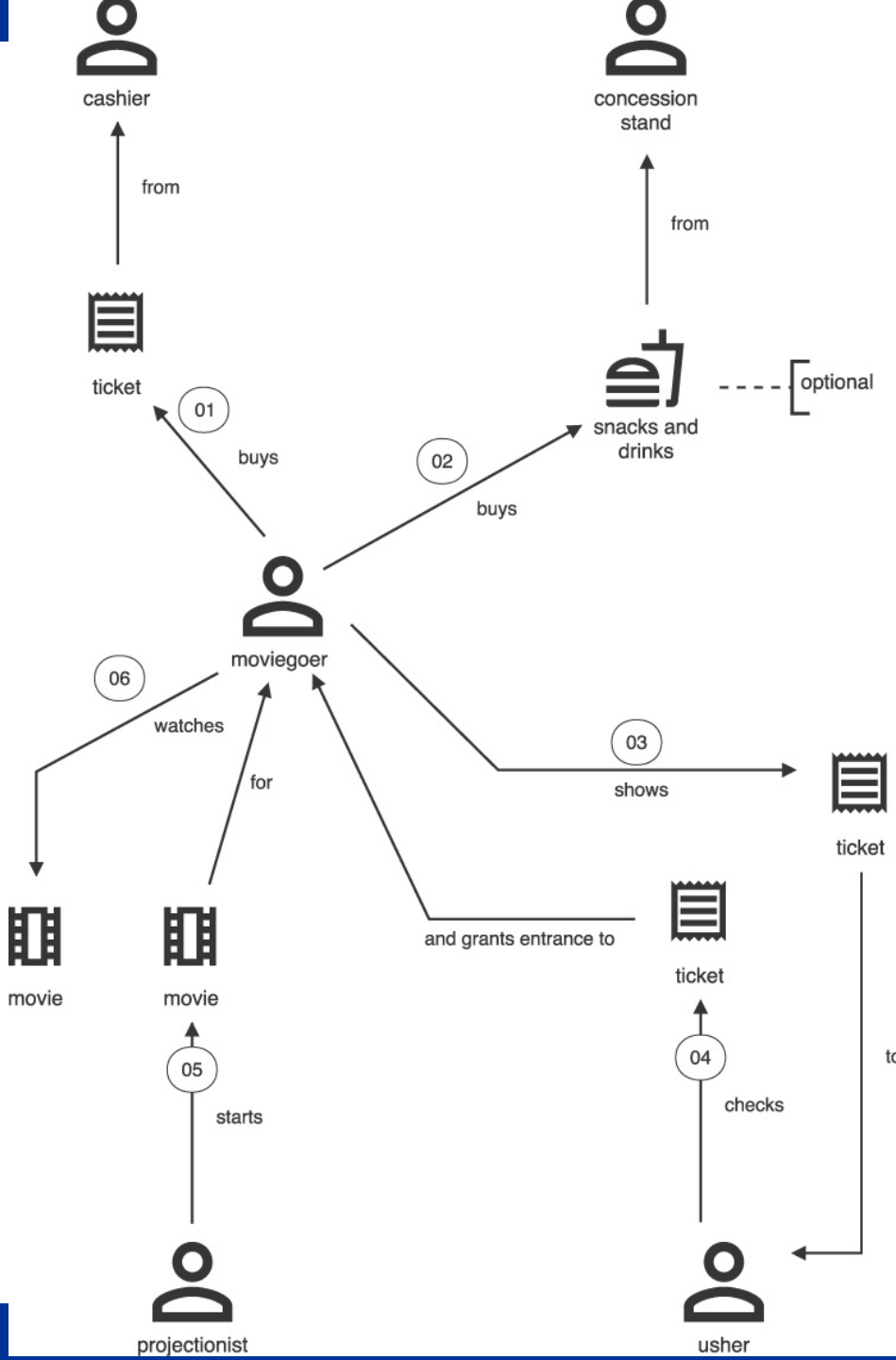
Domain Storytelling

- เมื่อทำแบบนี้ไปเรื่อยๆ dev ก็จะเข้าใจระบบของ user มากขึ้นเรื่อยๆ โดย user เองก็ทบทวนขั้นตอนได้ง่ายเช่นกัน
- วิธีการแบบนี้แม้จะดูเหมือนว่าเปลืองเวลา แต่มีข้อดี คือ จะได้ requirement มาครบ
- เทียบกับการต้องกลับไปถาม user หลายรอบ ถึงจุดตกหล่นวิธีการนี้อาจจะประหยัดเวลามากกว่าก็ได้



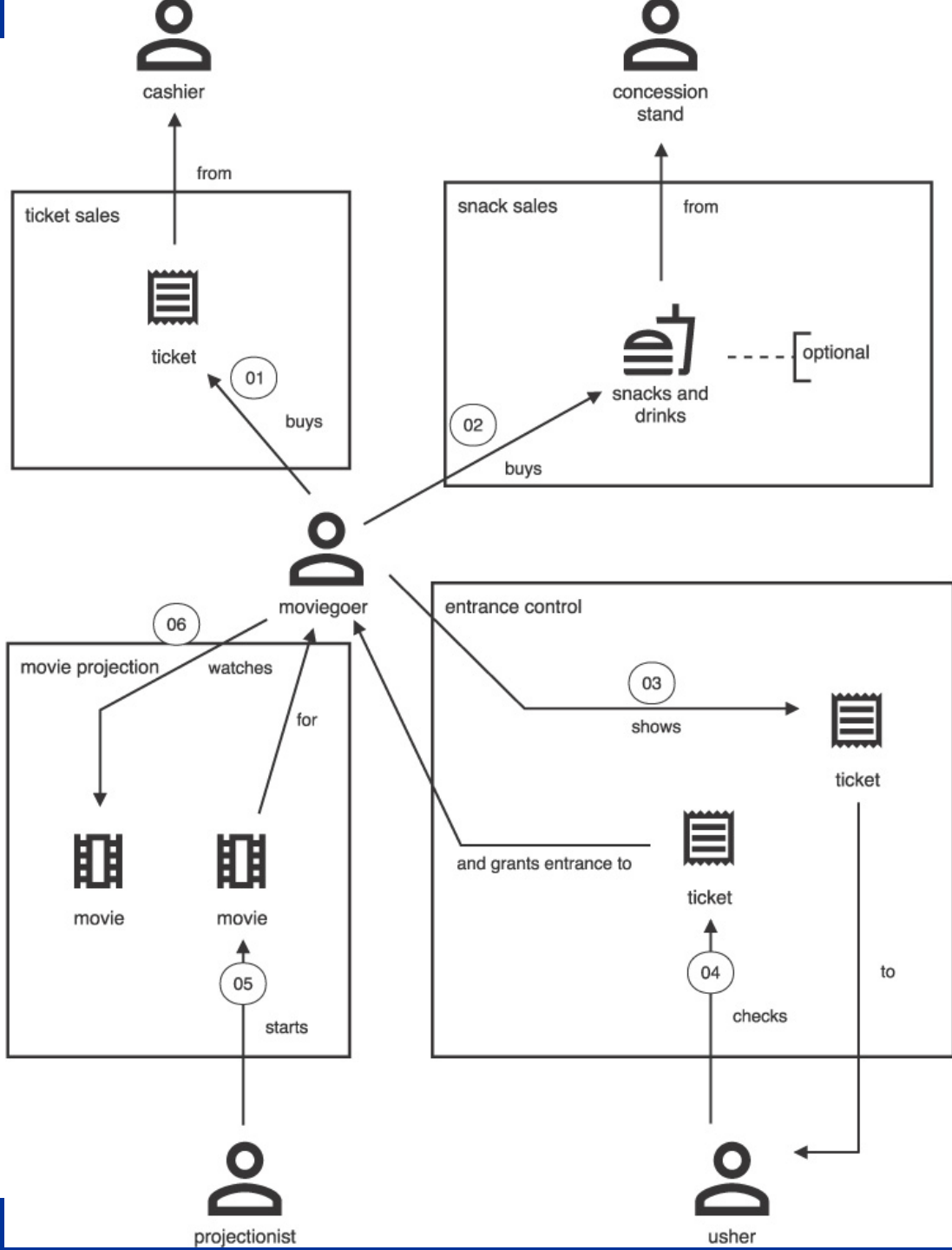
Domain Storytelling

- หลังจากที่วาดภาพตาม slide ก่อนหน้า ซึ่งข้อเสีย คือ ถ้าระบบใหญ่ๆ อาจจะดูยุ่งเหยิงมาก
- เราอาจเขียนอีก diagram เพื่อแสดงภาพรวมของระบบ จากรูปจะเห็นงานย่อยๆ คือ
  - ซื้อขายตั๋ว
  - ซื้อขายอาหาร
  - การเข้าชมภาพยนต์
  - การฉายหนัง



Domain Storytelling

- จากนั้นจะนำการทำงานมาดำเนินการจัดกลุ่ม ซึ่งจะเป็นไปตามงานย่อยที่กล่าวถึงก่อนหน้านี้ ได้แก่
  - ซื้อขายตั๋ว
  - ซื้อขายอาหาร
  - การเข้าชมภาพยนต์
  - การฉายหนัง

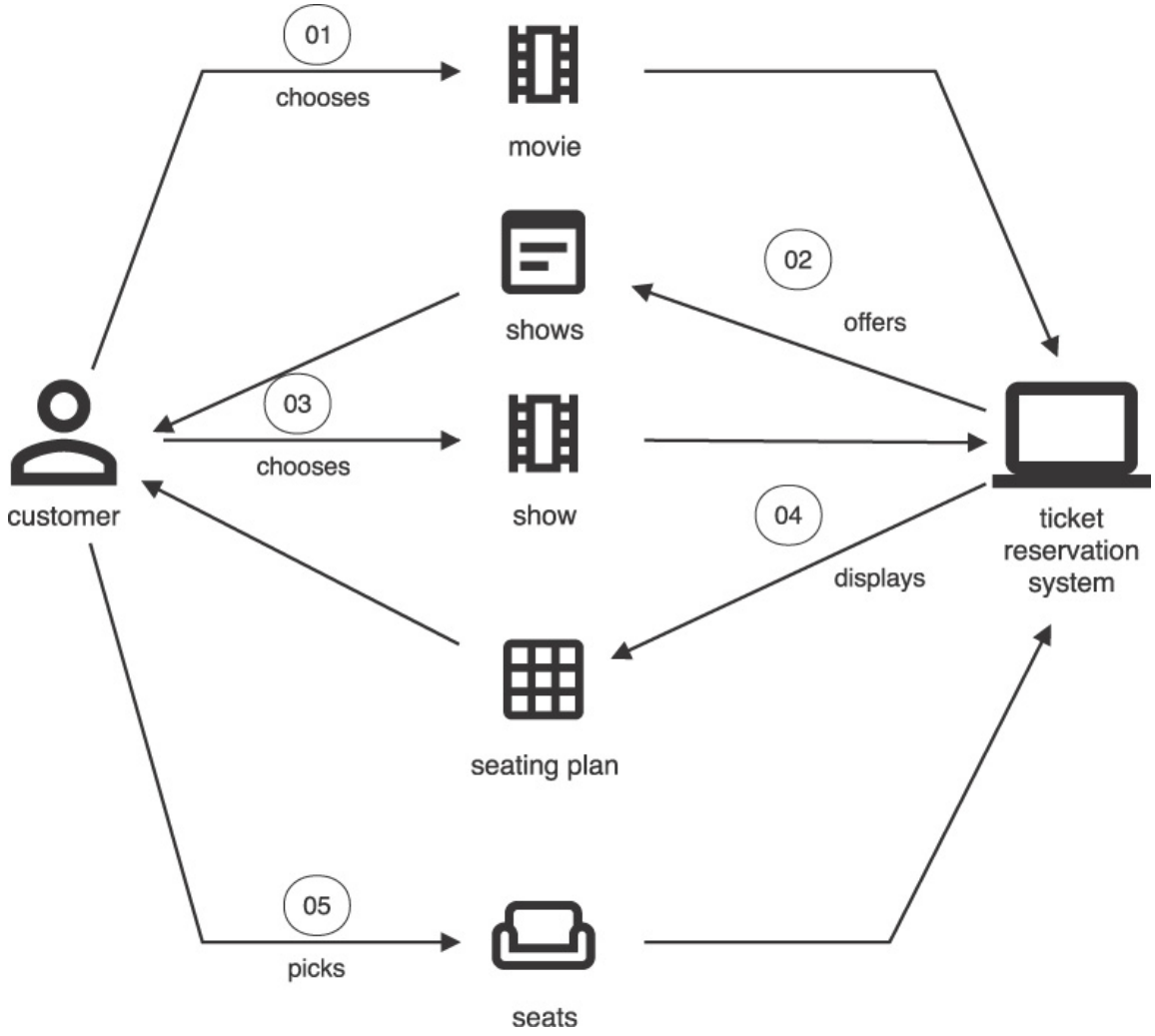


Domain Storytelling

- Domain stories ทำให้เห็นว่า ใคร who (actor) ทำอะไร (activity) กับสิ่งใด (work objects) และกับ ใครอีก (other actors)
- Domain stories จะเกี่ยวข้องอยู่รอบๆ Actor
- Actors จะเขียนไว้ที่เดียวใน domain story แต่ work objects สามารถปรากฏได้หลายครั้ง
- Activities จะทำหน้าที่เชื่อมโยง Actor และ Work Object เข้าด้วยกันเพื่อสร้างเป็นประโยค
- ประโยคแต่ละประโยคควรมีความหมายสมบูรณ์เมื่อคุณอ่านออกเสียง

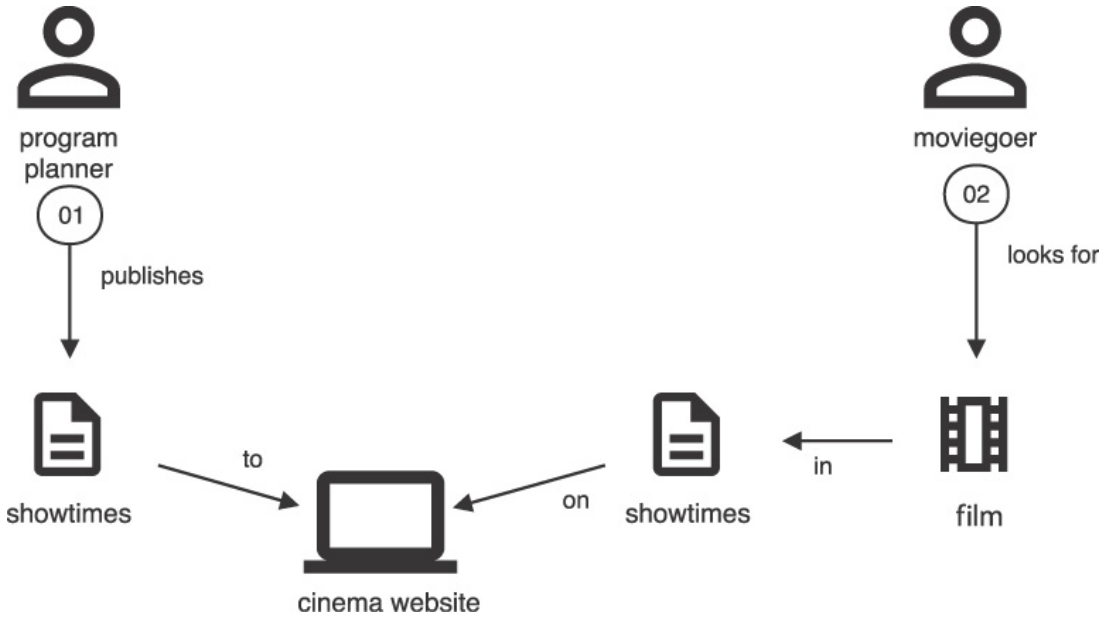
Domain Storytelling

- หากเป็นการ “เล่าเรื่อง” เพื่อจะพัฒนาซอฟต์แวร์ สามารถเขียนได้ดังนี้
  - ผู้ใช้เลือกภาพยนตร์
  - TRS แสดงรอบฉายให้ผู้ใช้
  - ผู้ใช้เลือกรอบฉาย
  - TRS แสดงผังที่นั่งให้ผู้ใช้
  - ผู้ใช้เลือกที่นั่ง
  - (ยังขาด confirm และชำระเงิน)



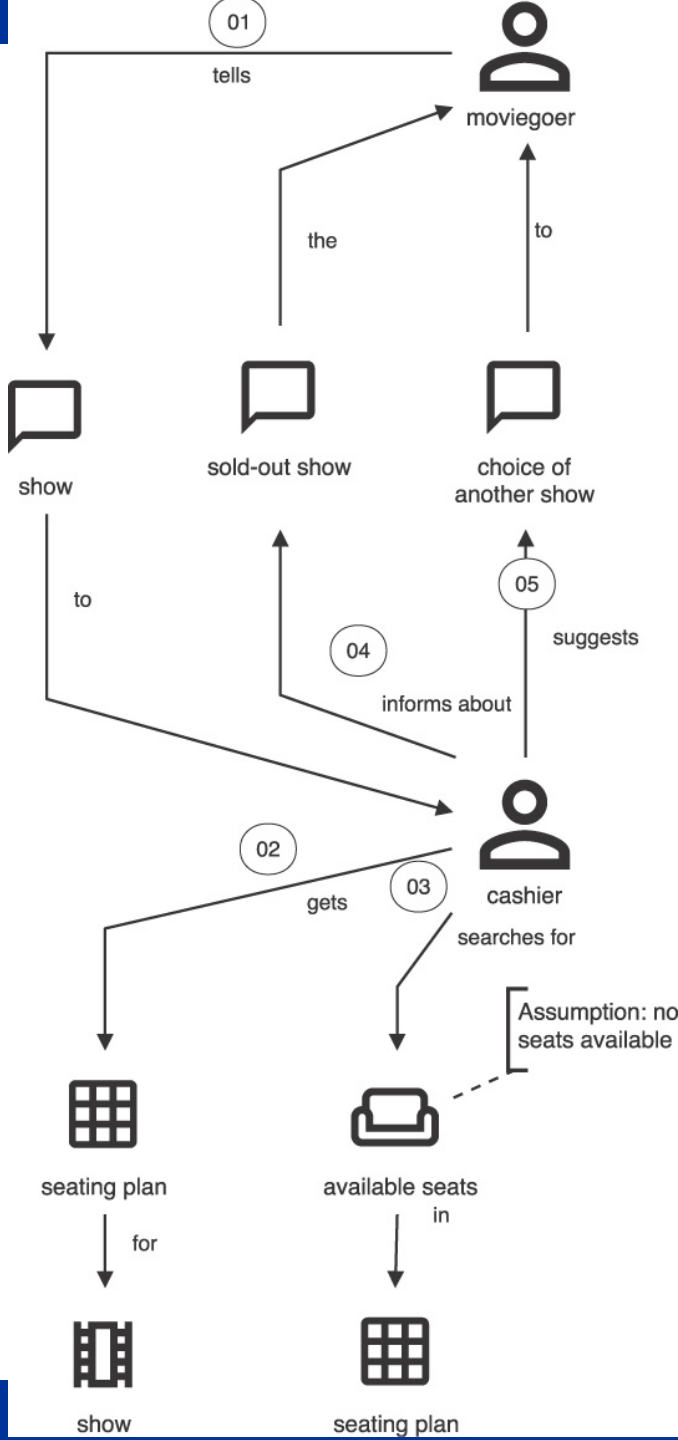
Domain Storytelling

- ในการออกแบบซอฟต์แวร์ นอกเหนือจากลูกค้าแล้ว ควรคำนึงถึงฝ่ายที่เป็น จนท. ของโรงหนังด้วย
- จากรูปจะเห็นว่ามีฟังก์ชัน ที่ program planner ทำหน้าที่ใส่ข้อมูลภาพยนต์และกำหนดรอบฉายลงในเว็บไซต์
- กรณีโรงภาพยนต์ ควรจะมีบทบาทของ จนท. ขายตั๋ว และควรรองรับทั้งแบบซื้อตั๋วหน้าโรง หรือ จองล่วงหน้า



Domain Storytelling

- ในการออกแบบซอฟต์แวร์ ควรคำนึงถึง กรณีปกติ และ กรณีพิเศษ
- จากภาพแสดงถึง กรณีที่ 1) คอหนัง บอกรอบฉายให้กับ cashier 2) cashier ตรวจสอบ ในผังที่นั่ง 3) พบว่าขายหมดแล้ว 4) แจ้งตั๋วขายหมดให้กับคอหนัง 5) cashier แนะนำเรื่องอื่น หรือ รอบฉายอื่น
- กรณีพิเศษต่างๆ ควรมองให้ครบถ้วน เพราะจะต้องเป็น feature ของซอฟต์แวร์ด้วย



Domain Storytelling

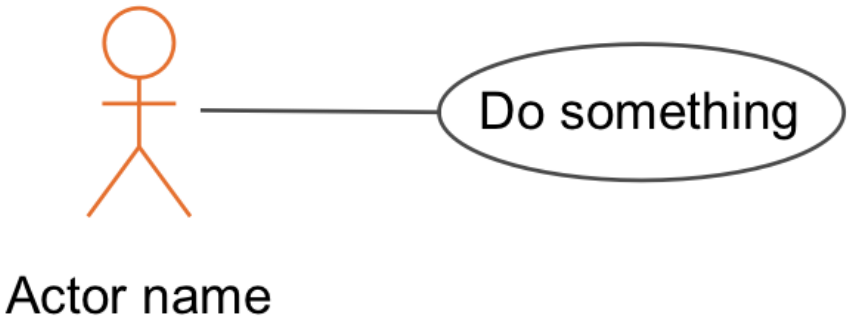
- เครื่องมือที่ใช้เขียน Domain Storytelling อาจเขียนบนกระดาษธรรมดา หรือซอฟต์แวร์วาด diagram ต่างๆ เช่น https://app.diagrams.net/ , miro.com
- เครื่องมือที่ออกแบบมาสำหรับวาด Domain Storytelling โดยตรงจะอยู่ที่เว็บไซต์ https://egon.io

Use Case Diagram

- Use Case Diagram เป็น Diagram มาตรฐานสำหรับการพัฒนาซอฟต์แวร์ ได้รับความนิยมและใช้งานทั่วไปในการพัฒนาซอฟต์แวร์
- Use Case Diagram ใช้สำหรับบอกว่าระบบทำอะไรได้บ้าง หรือ ความต้องการของผู้ใช้มีอะไรบ้าง โดยใน Use Case Diagram จะเห็นเป็นภาพรวมเท่านั้น
- Use Case Diagram จะบอกว่าผู้ใช้ของระบบแบ่งออกเป็นกี่กลุ่ม ผู้ใช้แต่ละกลุ่ม **สามารถ** ทำอะไรได้บ้างกับระบบที่จะพัฒนาขึ้น
- นอกจากนั้น Use Case Diagram ยังบอกความสัมพันธ์ของการทำงาน ว่าในแต่ละ Use Case มีการขึ้นต่อกันอย่างไร เช่น การกำหนดที่นั่งจะต้องอยู่ในกระบวนการจองตั๋ว ซึ่งหมายความว่า การจะกำหนดที่นั่งให้กับคนดู อยู่ๆ จะกำหนดเลยไม่ได้ จะต้องทำผ่านกระบวนการจองตั๋วเท่านั้น ซึ่งเมื่อแปลงเป็นซอฟต์แวร์แล้วจะเห็นลำดับการทำงานได้

Use Case Diagram

- สำหรับ Use Case Diagram จะมีองค์ประกอบเบื้องต้น 2 ส่วน คือ
  - Actor ซึ่งหมายถึง ผู้ใช้แต่ละกลุ่ม ใน Use Case Diagram มักจะเขียน Actor ที่เป็นตัวแทนของผู้ใช้ประเภทหนึ่งไว้ที่เดียว ยกเว้นกรณีจำเป็นอาจเขียนแยกได้ โดย Actor ไม่จำเป็นจะต้องเป็นคน อาจเป็นเครื่องจักร หรืออะไรก็ได้ที่ใช้ Software
  - Use Case หมายถึง การทำงานที่ Actor สามารถทำได้ มักใช้เป็นคำกริยา เพื่อบอกการกระทำ หรือ กิจกรรม
  - เส้นที่ลากระหว่าง Actor หมายถึงความสัมพันธ์ระหว่าง Actor กับ Use Case



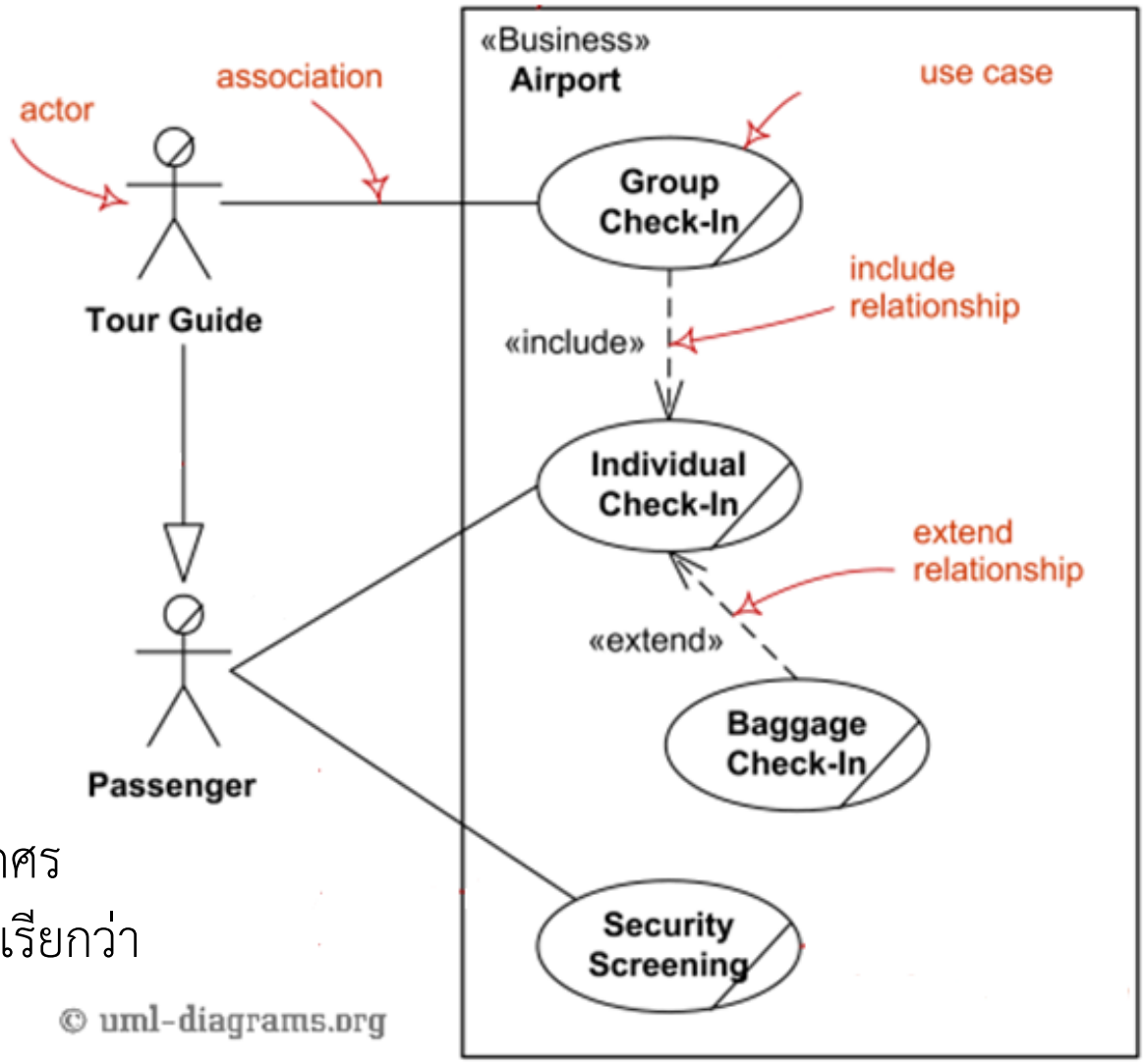
Use Case Diagram

สัญลักษณ์

- Use case
- Actor
- Connection

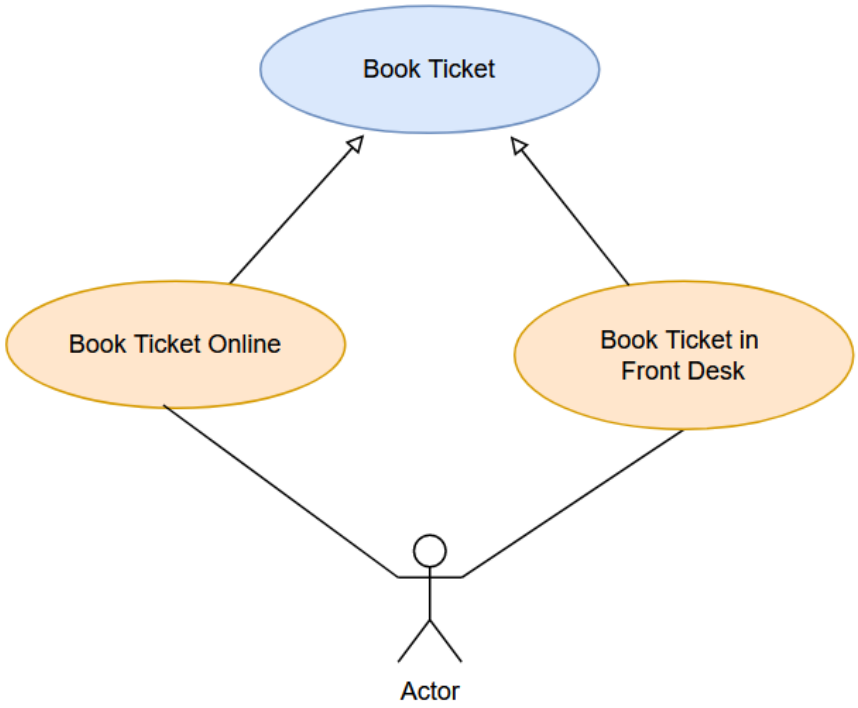
ความสัมพันธ์

- จะลากเส้นตรงระหว่าง Actor และ Use Case
- กรณีที่ Actor มีลักษณะเป็น subset จะใช้เครื่องหมายลูกศร โดยลูกศรวิ่งเข้า Superset (เรียกว่า Generalize)



Use Case Diagram

- ความสัมพันธ์แบบ Generalization ระหว่าง Use Case ใช้ในการอธิบายว่าในการทำงานใน Use Case หนึ่งสามารถทำได้มากกว่า 1 วิธี
- จากรูปแสดงให้เห็นว่า การซื้อตั๋ว สามารถทำได้ 2 วิธี

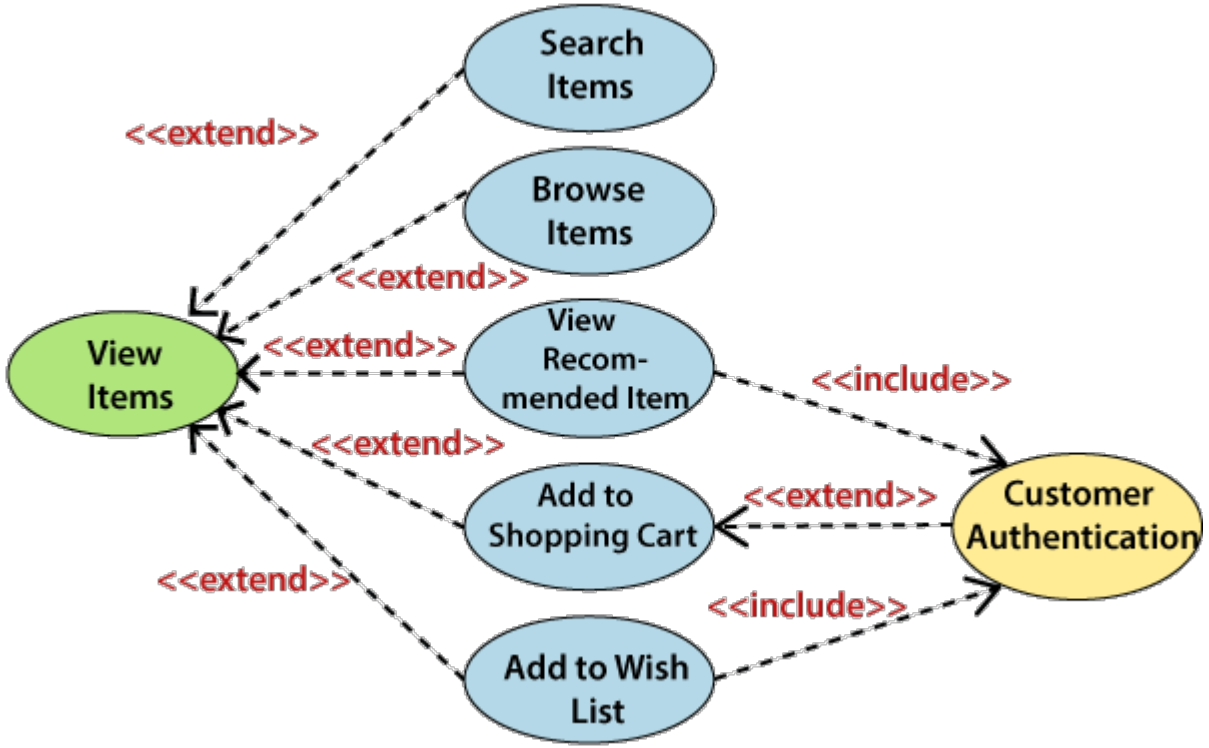


Use Case Diagram

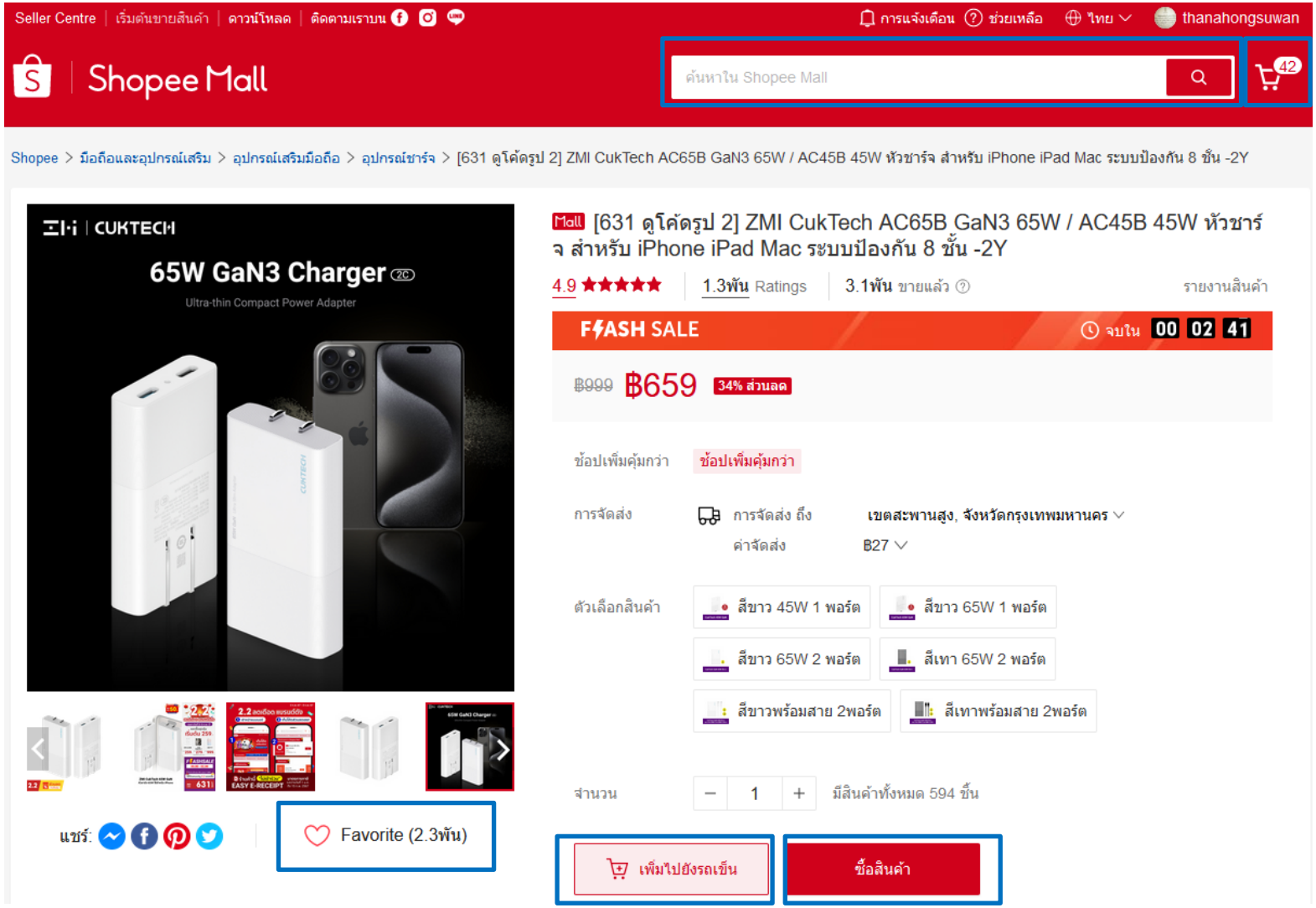
- ความสัมพันธ์แบบ *<<extends>>* ระหว่าง Use Case
  - <<extends>> จะใช้กับกรณีที่การทำงานบางอย่าง เป็นส่วนขยายของอีกงานหนึ่ง หมายถึง ต้องเกิดการทำงานแรกก่อน จึงจะเกิดการทำงานที่สองได้
  - เช่น ในเว็บ Shopee จะจ่ายเงินได้ ก็ต้องสั่งสินค้าก่อน
  - ในการเขียนความสัมพันธ์นี้ หัวลูกศรจะชี้ไปยัง use case ตัวที่มีการ extend ออกไป และหางลูกศรจะชี้ไปยัง use case ที่เป็นส่วนขยาย

Use Case Diagram

- จาก use case นี้ เมื่ออยู่ที่ view สินค้า จะสามารถค้นหาสินค้าอื่น สามารถดูสินค้าอื่น ดูสินค้าแนะนำ สามารถใส่ลงตะกร้า และ ใส่ลงใน Wish List ซ
- สังเกตุว่าหากจะ view recommendation และ add to wish list จะต้อง authen

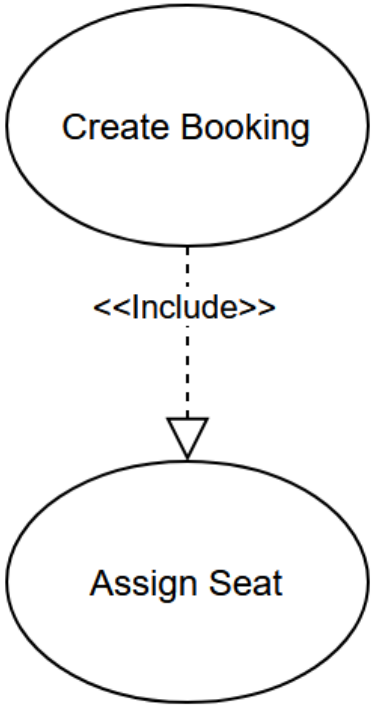


Use Case Diagram



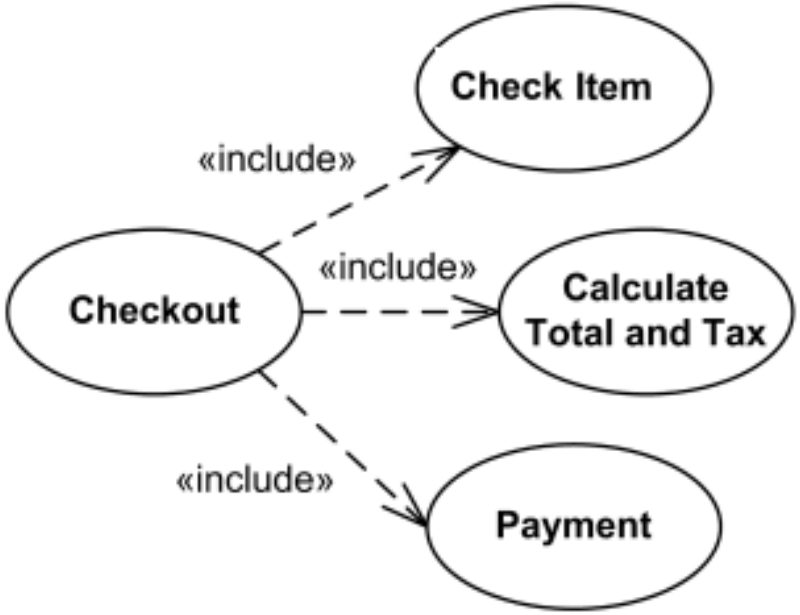
Use Case Diagram

- ความสัมพันธ์ระหว่าง Use Case แบบ *<<include>>*
  - <<include>> จะใช้แทนกรณีที่จะทำงานแรก จะต้องทำการทำงานที่ 2 ด้วย ถ้างานที่ 2 ยังไม่สำเร็จ งานแรกก็จะไม่สำเร็จตามไปด้วย
  - เช่น ในระบบโรงภาพยนต์ ก่อนที่จะจองตั๋วได้สำเร็จ จะต้องมีการเลือกที่นั่งก่อน
  - ลูกศรของความสัมพันธ์จะชี้ไปยัง use case ตัวที่ถูกใช้งาน



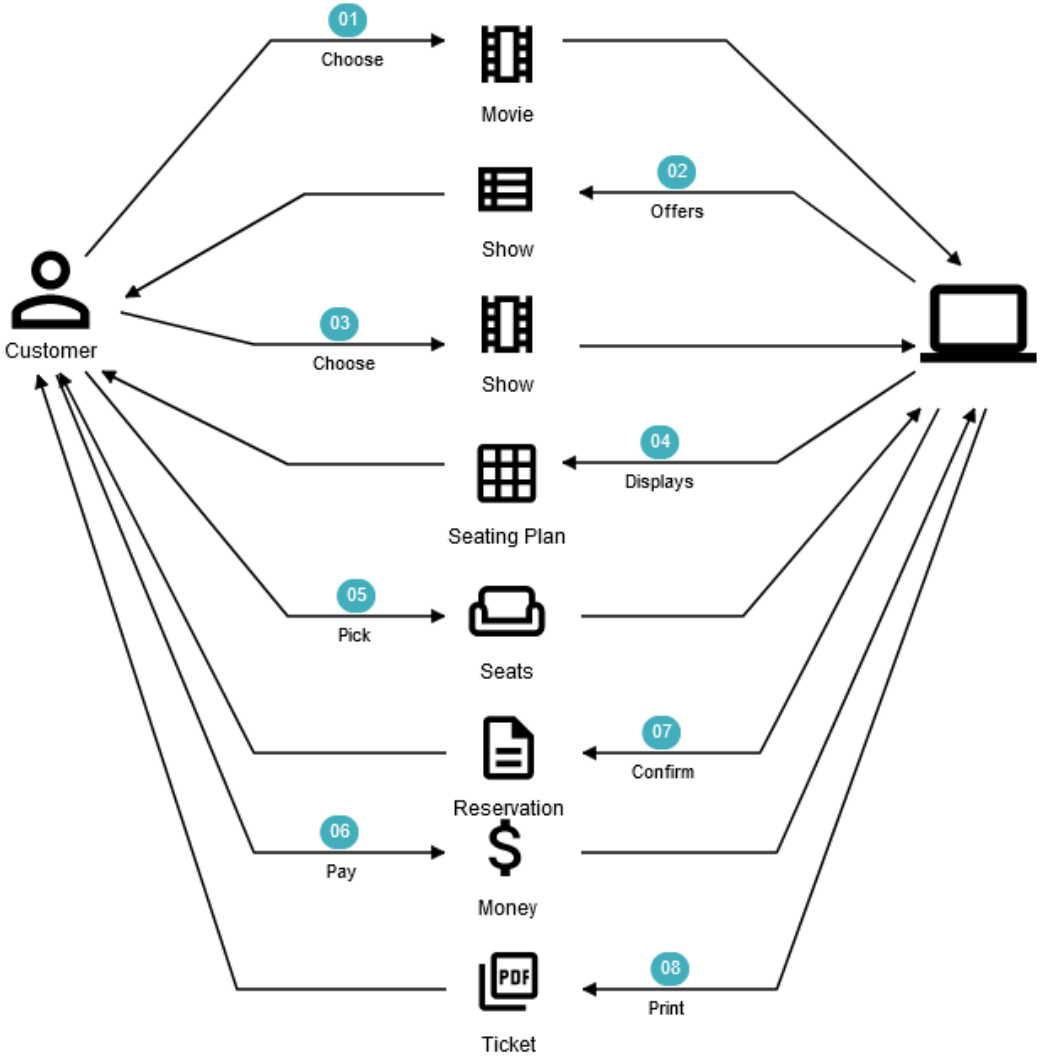
Use Case Diagram

- จากรูปจะเห็นว่าการจะ check out หรือ การซื้อสินค้า ได้ จะต้องรวมถึงการตรวจสอบสินค้า การคำนวณยอดรวม และ การชำระเงิน



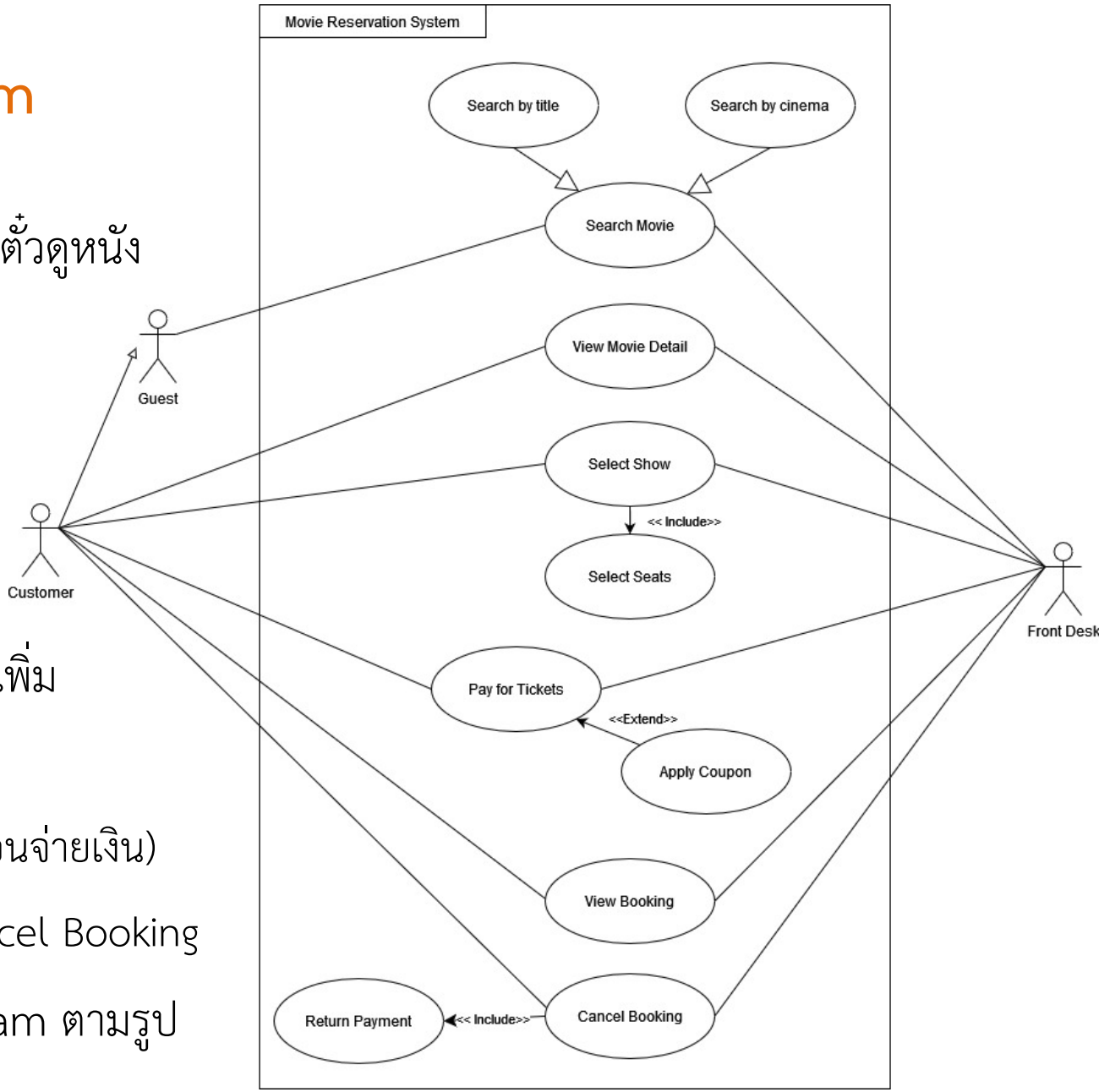
Use Case Diagram

- เราสามารถ Map จาก Domain Storytelling ให้เป็น Use Case Diagram ได้ เช่น สมมติว่ามีกิจกรรมตามรูป
- จะแยกกิจกรรมได้ดังนี้
  - เลือก หนัง ที่จะดู -> ระบบแสดง รอบฉาย
  - เลือก รอบฉาย -> ระบบแสดงที่นั่ง
  - เลือกที่นั่ง -> ระบบยืนยันการจอง
  - ชำระเงิน -> ได้ตั๋ว



Use Case Diagram

- สรุปกิจกรรมของการซื้อตั๋วดูหนัง
  - View Movie Detail
  - Select Show
  - Select Seats
  - Pay for tickets
- และเพื่อให้ครบถ้วน จะเพิ่ม
  - Search Movie
  - Apply Coupon (ตอนจ่ายเงิน)
  - View Booking/Cancel Booking
- จะได้ Use Case Diagram ตามรูป

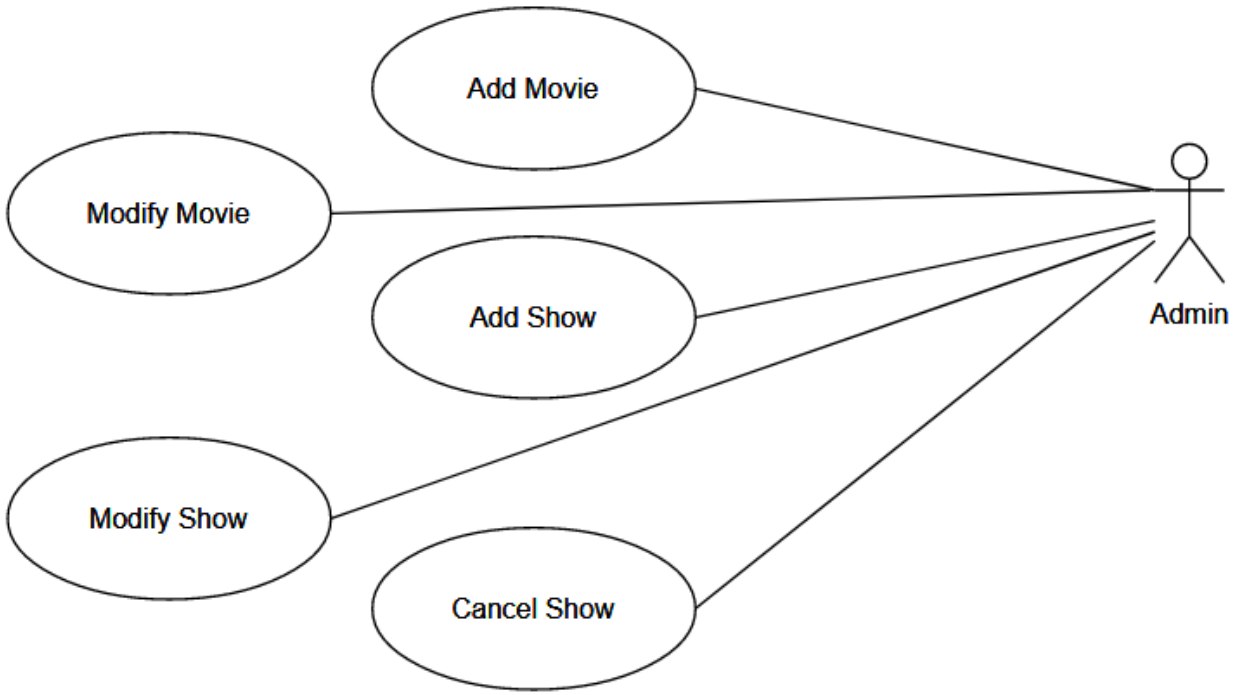


Use Case Diagram

- การเขียน Use Case ให้พิจารณาการทำงานของระบบ ให้ระบุกิจกรรมที่เกิดขึ้น โดยกิจกรรมควรมีลักษณะเบ็ดเสร็จในตัวเอง ถ้าเทียบกับ Application อาจจะเป็นการทำงานของ 1 หน้าเว็บ
- ให้สังเกตุ Action ในหน้าเว็บ เช่น ปุ่มกด ทุกปุ่มให้สงสัยว่าเป็นกิจกรรม แม้ว่าปุ่มนั้นกดแล้วจะไม่เปิดหน้าใหม่ เช่น Add to cart กดแล้วคล้ายกับหน้าจอเหมือนเดิม แต่เป็นการสั่งให้เกิดการเปลี่ยนแปลงที่เบื้องหลังแล้ว
- ให้แน่ใจว่าทุกส่วนของระบบ จะต้องมีกิจกรรมแสดงใน Use Case Diagram
- จากนั้นให้พิจารณาว่าแต่ละ Use Case ต้องรวมการทำงานใน Use Case ด้วยหรือไม่ ถ้ามีให้ใส่ความสัมพันธ์แบบ Include
- จากนั้นให้พิจารณาว่าในแต่ละ Use Case สามารถทำต่อจาก Use Case อื่นหรือไม่ ถ้ามีให้ใส่ความสัมพันธ์แบบ Extend

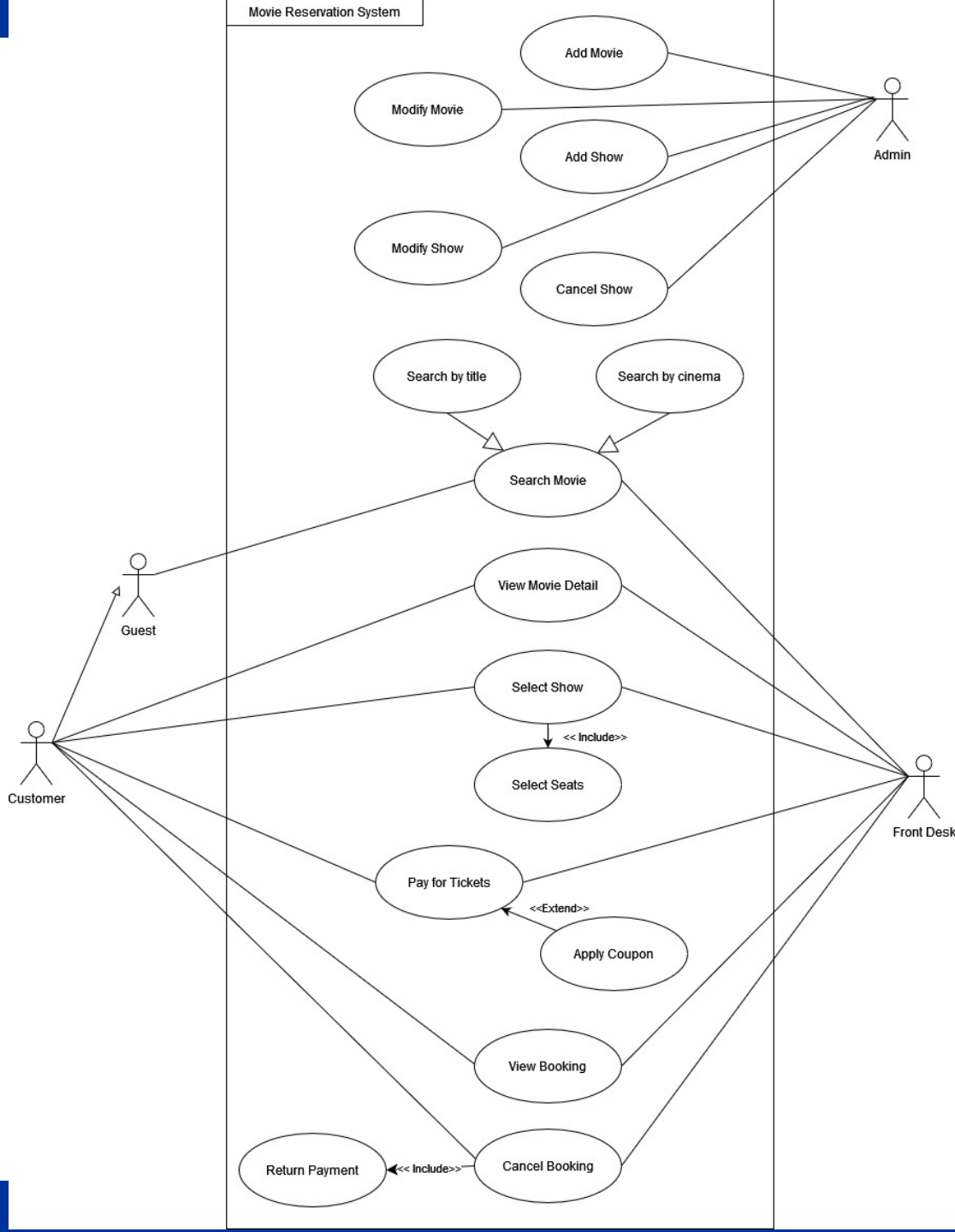
Use Case Diagram

- และอย่าลืมการทำงานเบื้องหลัง เช่น การเพิ่มภาพยนต์ การแก้ไขภาพยนต์ การเพิ่มรอบฉาย การแก้ไขรอบฉาย หรือ การยกเลิกรอบฉาย ซึ่งกระทำโดยเจ้าหน้าที่



Use Case Diagram

- สามารถเขียนเป็น Use Case Diagram รวมได้ดังนี้ โดยเสเ โดยกรอบสี่เหลี่ยมจะหมายถึงขอบเขตของระบบ (System) ที่จะพัฒนาขึ้น
- การตั้งชื่อ Use Case จะตั้งเป็นคำกริยา เพื่อแสดงว่า Use Case นั้นมีหน้าที่ใด
- สัญลักษณ์ Use Case ในที่อื่นอาจต่างไปจาก Slide นี้ แต่จะคล้ายกัน



Use Case Diagram

- จะเห็นได้ว่าการเขียน Use Case Diagram ที่ดี จำเป็นจะต้องมีขั้นตอนการทำงานที่ชัดเจนเสียก่อน จึงจะเขียน Use Case Diagram ออกมาได้ครบถ้วน
- เมื่อมี Use Case Diagram ที่แสดงความสัมพันธ์ที่ครบถ้วนก็จะช่วยให้การพัฒนาโปรแกรม สามารถทำได้ตรงตามความต้องการได้

Use Case Description

- Use Case Description คือ คำอธิบายรายละเอียดการทำงานของ Use Case แต่ละ Use Case อย่างไรก็ตามอาจเขียน Use Case Description เฉพาะ Use Case หลัก

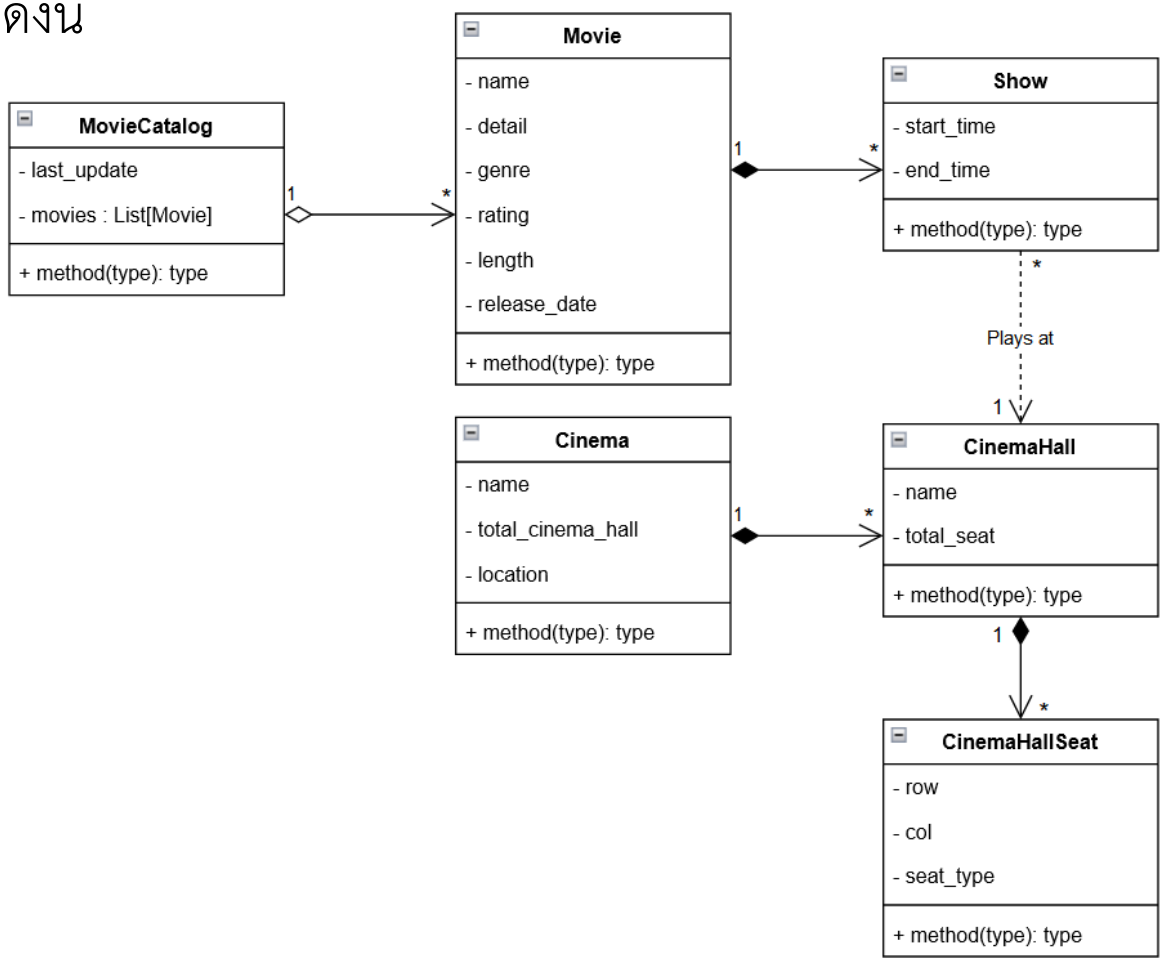
Use Case Name	จองตั๋วภาพยนตร์ (Create Booking)
Actor	Customer, Front Desk Officer
Description	กระบวนการจองตั๋วภาพยนตร์ เมื่อต้องการเข้ามาชมภาพยนตร์
Normal Course	1. เมื่อผู้ใช้เลือกภาพยนตร์ที่ต้องการชม และ เลือกรอบที่ต้องการชม จะเข้าสู่การจองตั๋ว 2. ระบบจะแสดงรายละเอียดของภาพยนตร์ที่เลือก วันที่ รอบฉาย ภาษาที่ฉาย ชื่อโรงภาพยนตร์ และ โรงย่อย 3. ระบบจะแสดง Layout ของเก้าอี้นั่ง รูปแบบของเก้าอี้นั่ง พร้อมทั้งราคาของที่นั่งแต่ละประเภท 4. หากผู้ใช้เลือกที่นั่ง ระบบจะแสดงที่นั่งที่ผู้ใช้เลือก พร้อมทั้งแสดงราคารวมของการจองทั้งหมด 5. หากผู้ใช้เลือก ส่วนลด หรือ โปรโมชัน ระบบจะเรียกส่วนงานของส่วนลดมาทำงาน
Alternate Course	1. กรณีที่นั่งเต็ม 2. กรณีผู้ใช้ 2 คนเลือกที่นั่งเดียวกัน

Class Diagram

- เราสามารถเชื่อมโยง Domain Storytelling กับ Class Diagram ได้
  - Actor : ปกติจะเป็น Object หนึ่งของระบบ กรณีมี Actor หลายกลุ่ม ให้มองหาความคล้ายคลึงและใช้ Inheritance
  - Work Objects : จะเป็น Object เช่นในระบบโรงภาพยนต์ จะมี Movie, Show, Seating Plan, Seat, Reservation
  - อาจเติม Object อื่นๆ เช่น Seat จะต้องอยู่ในโรงภาพยนต์ (Cinema Hall) และ กรณีที่เป็นแบบ Cineplex ก็จะมีหลายโรงภาพยนต์

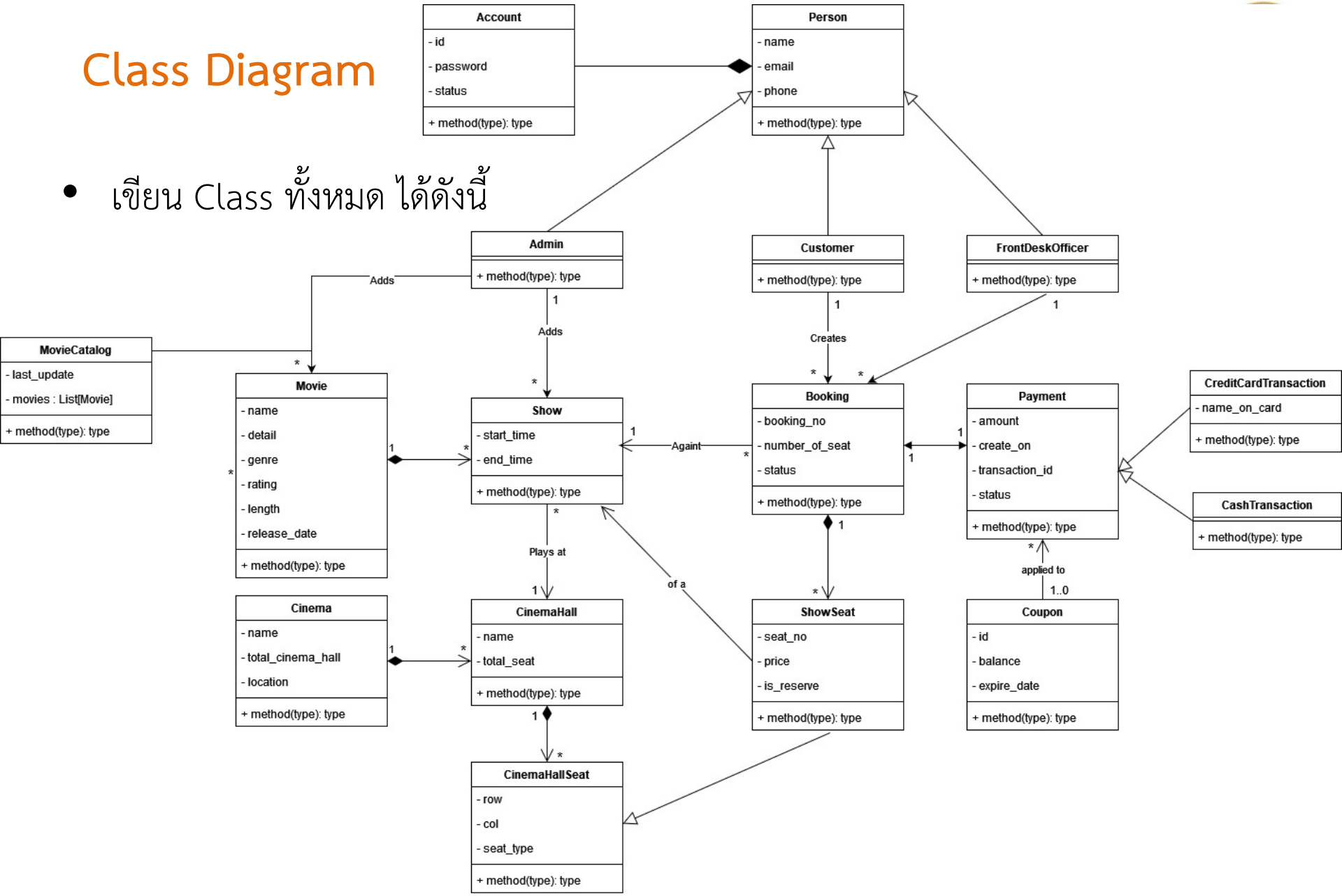
Class Diagram

- ในส่วนของข้อมูลพื้นฐานได้แก่ โรงภาพยนต์ ภาพยนต์ สามารถเขียนเป็น Class Diagram ดังนี้



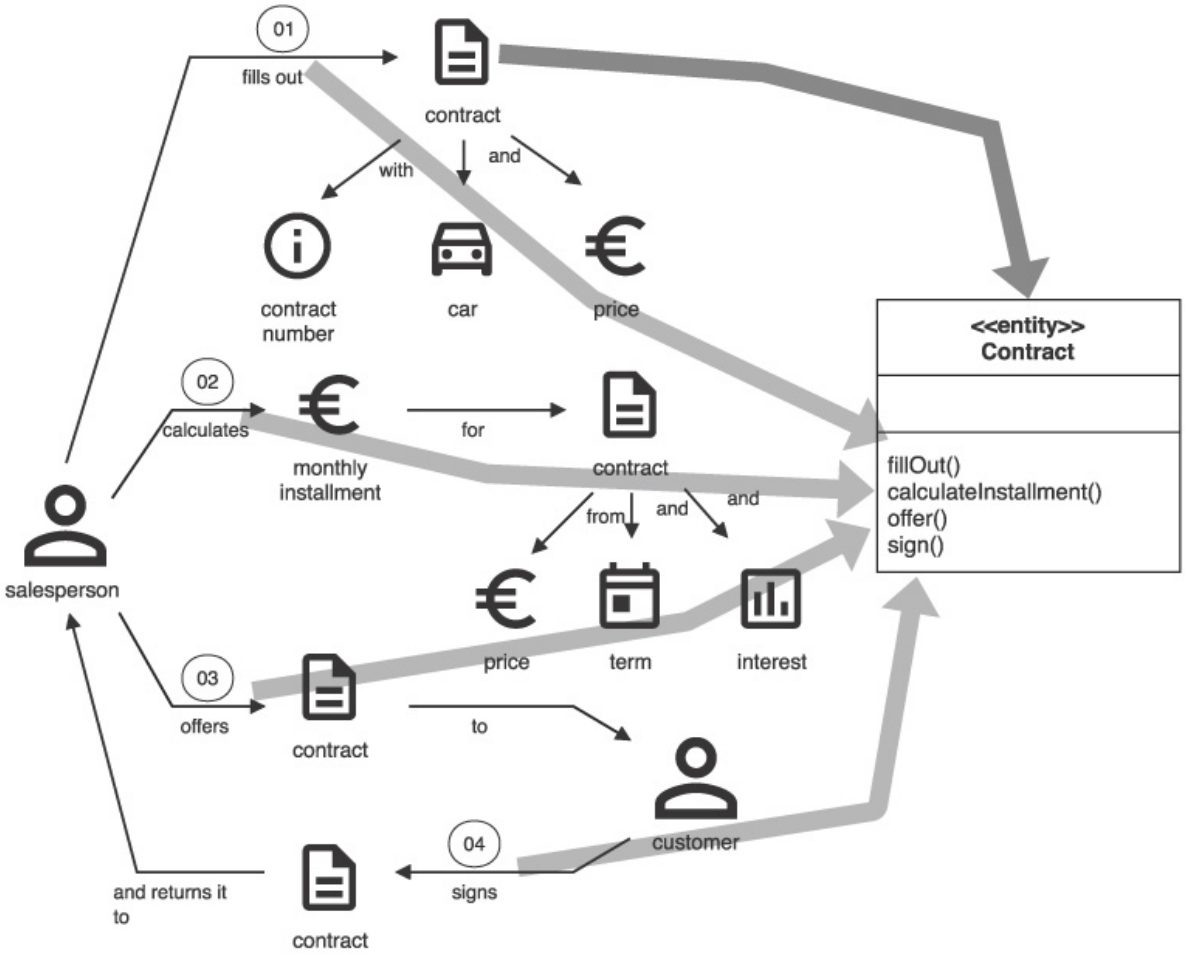
Class Diagram

- เขียน Class ทั้งหมด ได้ดังนี้



Class Diagram

- ส่วน Activity จะถูกแปลงเป็น method ของ class



Class Diagram

- หลักการออกแบบ Class Diagram ที่ควรคำนึงถึง

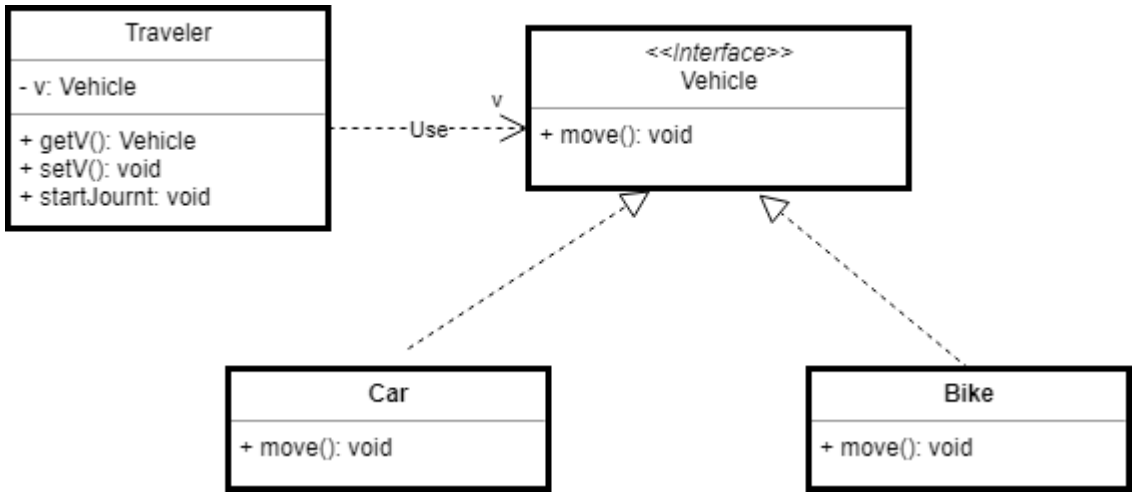
1. **Creator :** ใครเป็นผู้สร้าง object หรือ instance และ ใครเป็นผู้เก็บ object หรือ instance ปกติแล้วใครเป็นผู้ใช้ object นั้นควรจะเป็นคนสร้าง object และควรมีปฏิสัมพันธ์กับ object นั้น เช่น Account เป็นผู้สร้าง transaction ก็ควรจะเป็นผู้เก็บ
2. **Information Expert :** ข้อมูล Attribute และ Method ควรเก็บในคลาสที่เกี่ยวข้องหรือ “รู้เกี่ยวกับ” เรื่องนั้นๆ มากที่สุด เช่น การฝากเงิน ถอนเงิน คลาสที่เป็น Information Expert คือ Account ดังนั้น การฝากเงิน ถอนเงิน ควรจะอยู่ที่ Account

Class Diagram

- หลักการออกแบบ Class Diagram ที่ควรคำนึงถึง

3. **Low Coupling :** Coupling คือ การขึ้นต่อกันของ object คือ แต่ละ object ต้องขึ้นกับ object อื่นมากน้อยแค่ไหน หรือมี impact of change มากแค่ไหน เช่น หากออกแบบให้ การฝากเงิน ถอนเงิน อยู่ที่ตู้ ATM ถ้าเพิ่มโจทย์เป็นการฝากถอนทำที่ counter ได้ ก็มี impact of change มาก

   ตัวอย่าง เรื่องการเดินทาง แต่เดิมอาจเดินทางโดยรถยนต์ แต่ถ้าเผื่อการเดินทางประเภทอื่นไว้ล่วงหน้า ถ้ามีการเปลี่ยนแปลง ก็จะปรับ code ไม่มาก

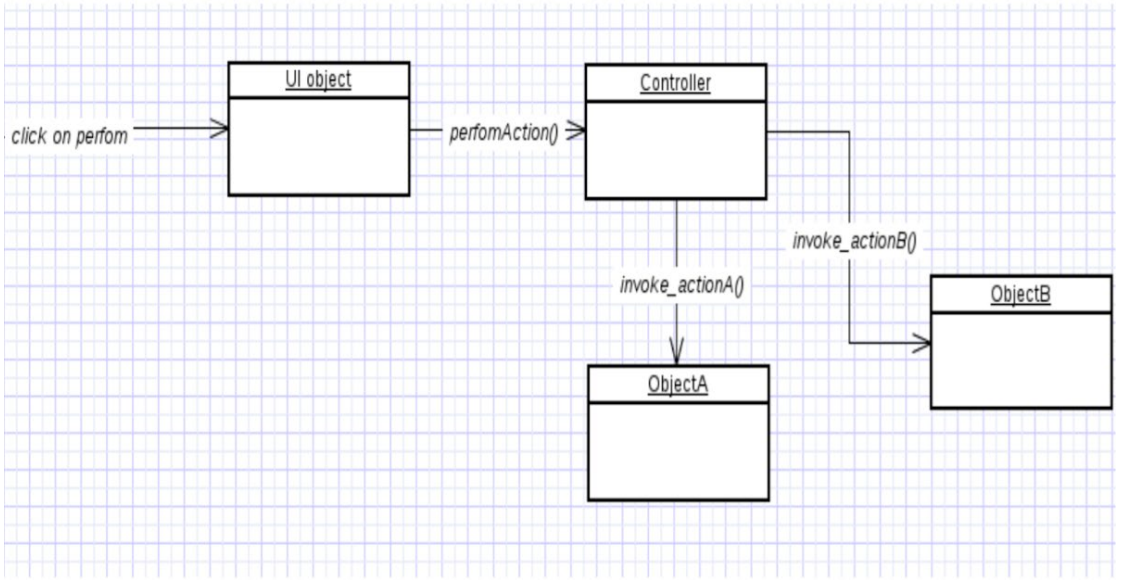


Class Diagram

- หลักการออกแบบ Class Diagram ที่ควรคำนึงถึง

4. **Controller :** คลาสจะแบ่งออกเป็นคลาสพื้นฐาน ที่ทำหน้าที่เก็บข้อมูลของระบบ เช่น Account, ATM และอื่นๆ แต่จะมีคลาสอีกแบบที่ทำหน้าที่ประสานงาน หรือ กระจายการทำงาน เช่น คลาส Bank คลาสชนิดนี้ จะเรียกว่า Controller โดยทำหน้าที่เก็บ List ของ Object และ Method กลาง ทำหน้าที่รับ request จาก UI

ในระบบที่ใหญ่ขึ้น อาจมี Controller หลายตัวก็ได้ เช่น payment controller, delivery controller

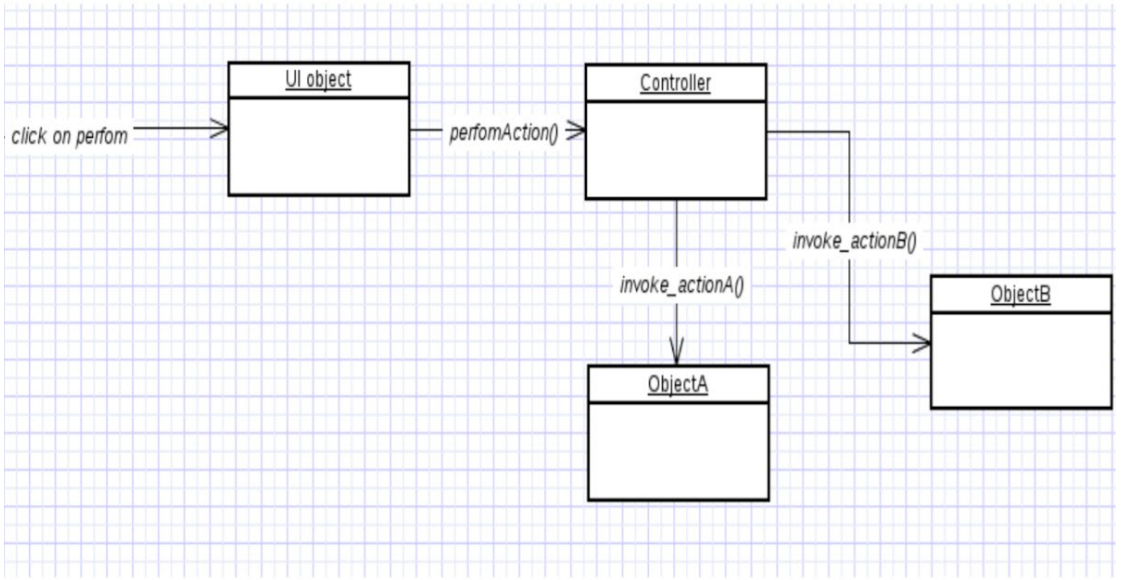


Class Diagram

- หลักการออกแบบ Class Diagram ที่ควรคำนึงถึง

4. **Controller :** คลาสจะแบ่งออกเป็นคลาสพื้นฐาน ที่ทำหน้าที่เก็บข้อมูลของระบบ เช่น Account, ATM และอื่นๆ แต่จะมีคลาสอีกแบบที่ทำหน้าที่ประสานงาน หรือ กระจายการทำงาน เช่น คลาส Bank คลาสชนิดนี้ จะเรียกว่า Controller โดยทำหน้าที่เก็บ List ของ Object และ Method กลาง

ในระบบที่ใหญ่ขึ้น อาจมี Controller หลายตัวก็ได้ เช่น payment controller, delivery controller



For your attention