**บทนำ**

ฉันได้ยินเกี่ยวกับ Domain Driven Design และพบกับ Eric Evans ที่การประชุมของนักสถาปัตยกรรมระดับเล็กในยอดเขาที่จัดโดย Bruce Eckel ในฤดูร้อนของปี 2005 การประชุมได้รับการเข้าร่วมโดยบุคคลที่ฉันเคารพอย่างมาก รวมถึง Martin Fowler, Rod Johnson, Cameron Purdy, Randy Stafford, และ Gregor Hohpe

กลุ่มดูเหมือนจะประทับใจกับวิสัยทัศน์ของ Domain Driven Design และกระตือรือร้นที่จะเรียนรู้เพิ่มเติมเกี่ยวกับมัน ฉันยังรู้สึกว่าทุกคนปรารถนาว่าแนวคิดเหล่านี้จะเป็นเช่นนี้กับกลุ่มคนทั่วไปมากขึ้น เมื่อฉันสังเกตเห็นว่า Eric ใช้โมเดลโดเมนเพื่อพูดคุยเกี่ยวกับการแก้ไขบางอย่างจากที่กลุ่มพูดคุยกัน และเขาให้ความสำคัญกับโดเมนธุรกิจมากกว่าการตลาดเทคโนโลยี ฉันรู้ทันทีว่าวิสัยทัศน์นี้เป็นสิ่งที่ชุมชนต้องการอย่างแรงแล้ว

เราในชุมชนการพัฒนาสถาปัตยกรรมธุรกิจ โดยเฉพาะชุมชนการพัฒนาเว็บไซต์ได้รับผลกระทบจากปีหลายๆ ของการตลาดเทคโนโลยีที่เอาเราออกไปจากการพัฒนาซอฟต์แวร์เชิงวัตถุที่เหมาะสม ในชุมชน Java การสร้างโมเดลโดเมนที่ดีหายไปในการตลาดของ EJB และโมเดลคอนเทนเนอร์/คอมโพเนนต์ในปี 1999-2004 โชคดีที่เทคโนโลยีกำลังเปลี่ยนแปลงและประสบการณ์ร่วมกันของชุมชนการพัฒนาซอฟต์แวร์กำลังนำเรากลับสู่แนวคิดเชิงวัตถุเก่าแก่ อย่างไรก็ตาม ชุมชนนั้นยังขาดวิสัยทัศน์ที่ชัดเจนเพื่อนำไปใช้ในการพัฒนาซอฟต์แวร์เชิงวัตถุในองค์กรใหญ่ นี่เป็นเหตุผลที่ฉันคิดว่า DDD มีความสำคัญ

นับเป็นความน่าเสียดายที่นอกจากกลุ่มของนักสถาปัตยกรรมระดับสูงเล็กน้อยแล้ว ฉันรับรู้ว่ามีน้อยมากที่รู้จัก DDD ซึ่งเป็นเหตุผลที่ InfoQ ได้มอบหมายให้เขียนหนังสือเล่มนี้

ฉันหวังว่าโดยการเผยแพร่สรุปและนำเสนอขั้นพื้นฐานของ DDD อย่างสั้น ๆ และอ่านได้ง่าย และทำให้เป็นไปได้ว่าความคิดนี้จะกลายเป็นหลักสูตรหลักในอนาคต โดยที่เราจะเผยแพร่ฉบับพิมพ์ขนาดกระเป๋าราคาไม่แพงพร้อมให้ดาวน์โหลดฟรีบน InfoQ นั่นเอง

หนังสือเล่มนี้ไม่มีการนำเสนอแนวคิดใหม่ๆใดๆ แต่พยายามสรุปสาระสำคัญของ DDD ให้อย่างกระชับ โดยส่วนใหญ่นำเอาหนังสือเล่มเดิมของ Eric Evans เป็นหลัก รวมถึงแหล่งอื่นๆที่เกี่ยวข้องเช่น Applying DDD ของ Jimmy Nilsson และห้องสนทนา DDD ต่างๆ หนังสือเล่มนี้จะช่วยให้คุณเข้าใจเรื่องพื้นฐานของ DDD ได้อย่างรวดเร็ว แต่ไม่สามารถแทนที่ตัวอย่างและกรณีศึกษาจำนวนมากที่ได้รับการจัดเตรียมไว้ในหนังสือของ Eric หรือตัวอย่างที่เสนอในหนังสือของ Jimmy ฉันขอแนะนำให้คุณอ่านทั้งสองหนังสือเหล่านี้อย่างสม่ำเสมอ ในขณะเดียวกัน หากคุณเห็นด้วยว่า DDD เป็นสิ่งที่ชุมชนต้องการจะมีส่วนร่วมและเป็นการรับรู้เฉพาะของกลุ่มของเรา กรุณาแชร์หนังสือนี้และงานของ Eric ต่อคนอื่น ๆ บ้าง

ซอฟต์แวร์เป็นเครื่องมือที่สร้างขึ้นเพื่อช่วยให้เราจัดการกับความซับซ้อนของชีวิตสมัยใหม่ของเราได้ ซอฟต์แวร์เป็นเพียงเครื่องมือที่ช่วยให้เราไปสู่จุดมุ่งหมาย และมักเป็นเรื่องที่เป็นจริงและใกล้ชิดกับชีวิตจริงๆ ตัวอย่างเช่นเราใช้ซอฟต์แวร์สำหรับควบคุมการจราจรทางอากาศ ซึ่งเชื่อมโยงโดยตรงกับโลกที่เราอยู่ ซึ่งเราต้องการบินจากสถานที่หนึ่งไปยังสถานที่อื่น และเราทำเช่นนั้นโดยใช้เครื่องจักรที่ซับซ้อน ดังนั้นเราจึงสร้างซอฟต์แวร์เพื่อประสานงานการบินของพันธมิตรที่อยู่ในอากาศหลายพันเครื่องในเวลาใดก็ตาม

ซอฟต์แวร์ถูกสร้างขึ้นเพื่อช่วยให้เราจัดการกับความซับซ้อนในชีวิตสมัยปัจจุบันของเรา ซอฟต์แวร์เป็นเพียงเครื่องมือเพื่อทำให้เราได้ตามเป้าหมาย และจุดหมายที่สำคัญสำหรับซอฟต์แวร์นั้นเป็นสิ่งที่เป็นจริงและเชื่อมโยงกับความจำเป็นในชีวิตจริง ตัวอย่างเช่น เราใช้ซอฟต์แวร์สำหรับควบคุมการจราจรทางอากาศ ซึ่งเชื่อมโยงโดยตรงกับโลกที่เราดำเนินชีวิตอยู่ เราต้องการบินจากสถานที่หนึ่งไปยังอีกที่หนึ่ง และเราจะใช้เครื่องจักรที่ซับซ้อน เพื่อนำซอฟต์แวร์มาประสานงานในการบินของพันธมิตรที่อยู่ในอากาศพร้อมกันหลายพันเครื่อง

การออกแบบซอฟต์แวร์เป็นศิลปะ และเหมือนกับศิลปะใด ๆ มันไม่สามารถสอนและเรียนรู้เป็นวิทยาศาสตร์ที่แน่นอนได้ โดยใช้ทฤษฎีและสูตรต่าง ๆ ได้ เราสามารถค้นพบหลักการและเทคนิคที่มีประโยชน์ในการสร้างซอฟต์แวร์ตลอดกระบวนการสร้างซอฟต์แวร์ได้ แต่เราอาจไม่สามารถให้เส้นทางที่แน่นอนในการสร้างโมดูลโค้ดที่จะช่วยให้เหตุการณ์จริงสอดคล้องกับโดเมน โดยตรงได้ เหมือนกับรูปภาพหรืออาคาร ผลิตภัณฑ์ซอฟต์แวร์จะรวมความสมบูรณ์ของผู้ออกแบบและพัฒนามัน บางส่วนของเสน่ห์และความมั่นใจ (หรือขาดแคล้ว) ของผู้มีส่วนร่วมในการก่อตั้งและการเติบโตของมัน

มีวิธีการที่แตกต่างกันในการออกแบบซอฟต์แวร์ ตลอด 20 ปีที่ผ่านมา อุตสาหกรรมซอฟต์แวร์ได้ใช้วิธีการหลายวิธีในการสร้างผลิตภัณฑ์ แต่ละวิธีมีข้อดีและข้อเสียของตัวเอง เนื้อหาของหนังสือนี้จะเน้นไปที่วิธีการออกแบบที่เกิดขึ้นและพัฒนามาตลอด 20 ปีที่ผ่านมา แต่รวมตัวมากขึ้นในช่วงไม่กี่ปีที่ผ่านมา นั่นคือการออกแบบตามโดเมน (Domain-Driven Design) Eric Evans ได้มีส่วนร่วมอันมหาศาลในเรื่องนี้โดยเขียนเล่มหนังสือที่รวมความรู้และประสบการณ์ที่สะสมมานานเกี่ยวกับการออกแบบตามโดเมน สำหรับเนื้อหาที่ละเอียดอ่อนขึ้นเกี่ยวกับเรื่องนี้ เราขอแนะนำให้อ่านหนังสือของเขา "Domain-Driven Design: Tackling Complexity in the Heart of Software" ที่จัดพิมพ์โดย Addison-Wesley หมายเลขไอเอสเบิร์น: 0-321-12521-5

หลายความคิดและข้อสังเกตมีให้เรียนรู้ได้จากการติดตามกลุ่ม Diskussion ของ Domain Driven Design ที่ <http://groups.yahoo.com/group/domaindrivendesign> หนังสือเล่มนี้เป็นเพียงการแนะนำพื้นฐานเกี่ยวกับหัวข้อนี้เท่านั้น โดยมีจุดมุ่งหมายเพื่อให้คุณเข้าใจพื้นฐานของ Domain Driven Design แต่ไม่ได้มุ่งเน้นรายละเอียด โดยเราต้องการเรียนรู้แนวคิดและแนวปฏิบัติที่นำไปสู่การออกแบบซอฟต์แวร์ที่ดีในโลกของ Domain Driven Design โดยเฉพาะอย่างยิ่ง

**Floyd Marinescu**

**Co-founder & Chief Editor of InfoQ.com**

**2 - Domain-Driven Design คืออะไร**

การพัฒนาซอฟต์แวร์มักถูกนำไปใช้ในการออโตเมติกขบวนการต่างๆ ที่มีอยู่ในโลกจริงหรือให้การแก้ไขปัญหาทางธุรกิจต่างๆ โดยกระบวนการธุรกิจที่กำลังถูกออโตเมติกส่วนหนึ่งหรือปัญหาทางธุรกิจที่ต้องการใช้ซอฟต์แวร์จึงเป็นโดเมนของซอฟต์แวร์เหล่านั้น ดังนั้นเราต้องเข้าใจตั้งแต่ต้นว่าซอฟต์แวร์เกิดขึ้นมาจากโดเมนนี้และสัมพันธ์กับโดเมนนี้อย่างลึกซึ้ง

ซอฟต์แวร์ประกอบด้วยโค้ด บางครั้งเราอาจจะต้องการใช้เวลามากเกินไปกับโค้ด และมองซอฟต์แวร์เป็นเพียงวัตถุและเมธอดเท่านั้น

ในการผลิตรถยนต์ เป็นการใช้สมมติฐาน เพื่อให้เห็นภาพการผลิตซอฟต์แวร์ แรงงานที่มีส่วนร่วมในการผลิตรถยนต์ อาจเชี่ยวชาญในการผลิตชิ้นส่วนของรถยนต์ แต่ด้วยการเชื่อมโยงของชิ้นส่วนต่างๆ สามารถมองรถยนต์เป็นชุดของชิ้นส่วน แต่รถยนต์ก็ไม่ได้เป็นแค่นั้นเท่านั้น การผลิตรถยนต์ที่ดีเริ่มต้นจากวิสัยทัศน์ การเขียนสเปคอย่างละเอียดถูกต้องและตระหนักถึงการออกแบบ มีการออกแบบมากมาย เสริมเติม หลายเดือนหรือบางครั้งก็อาจใช้เวลาหลายปีในการออกแบบ โดยการเปลี่ยนแปลงและปรับปรุงจนถึงความสมบูรณ์ จนกระทั่งตระหนักถึงวิสัยทัศน์เดิมได้ การออกแบบไม่จำเป็นต้องอยู่บนกระดาษเท่านั้น มีการออกแบบโมเดลรถยนต์และทดสอบในสถานการณ์ต่างๆ เพื่อดูว่ามันจะทำงานได้หรือไม่ การออกแบบจึงได้รับการปรับปรุงตามผลการทดสอบ รถยนต์จึงถูกส่งเข้ากระบวนการผลิตและชิ้นส่วนถูกสร้างและประกอบเข้าด้วยกัน

การพัฒนาซอฟต์แวร์คล้ายกับนี้ ไม่สามารถเริ่มต้นโดยการนั่งพิมพ์โค้ดได้เลย แม้ว่าวิธีนี้จะเหมาะสำหรับกรณีที่ง่าย ๆ แต่สำหรับซอฟต์แวร์ที่ซับซ้อน เราไม่สามารถสร้างได้แบบนั้น

เพื่อสร้างซอฟต์แวร์ที่ดี คุณต้องรู้ว่าซอฟต์แวร์นั้นเกี่ยวข้องกับอะไร คุณไม่สามารถสร้างระบบธนาคารได้เลย หากคุณไม่มีความเข้าใจที่ดีเกี่ยวกับบริบทของการเงิน คุณจึงต้องเข้าใจดีว่าสิ่งที่เรียกว่า Domain คืออะไร ด้วยความเข้าใจเหล่านี้ จึงจะช่วยให้คุณสร้างซอฟต์แวร์ได้ดียิ่งขึ้น

เป็นไปได้หรือไม่ที่จะสร้างซอฟต์แวร์ธนาคารที่ซับซ้อนโดยไม่มีความรู้ด้านโดเมนที่ดี? ไม่มีทางเลย ไม่เคยเป็นเช่นนั้น เท่าที่รู้ ใครรู้จักธนาคารบ้าง? นักสถาปัตยกรรมซอฟต์แวร์หรือเปล่า? ไม่ใช่ พวกเขาเพียงแค่ใช้ธนาคารเพื่อเก็บเงินไว้เมื่อต้องการใช้ นักวิเคราะห์ซอฟต์แวร์? ไม่ได้จริงๆ พวกเขารู้จักการวิเคราะห์เรื่องที่กำหนดให้ เมื่อมีอุปสรรคทั้งหมดที่จำเป็น นักพัฒนา? ลืมมันไปเลย ใครแทนบ้าง? คนงานธนาคารแน่นอน ระบบธนาคารนั้นเข้าใจอย่างดีโดยคนภายในสถาบัน โดยผู้เชี่ยวชาญของพวกเขา พวกเขารู้จักรายละเอียดทั้งหมด ทุกๆ อย่าง กฎเกณฑ์ที่เป็นไปได้ทั้งหมด ปัญหาที่เป็นไปได้ทั้งหมด กฎข้อบังคับทั้งหมด ตรงนี้เป็นจุดเริ่มต้นที่เราควรจะเริ่มต้นเสมอ: ด้านโดเมน

เมื่อเราเริ่มโครงการพัฒนาซอฟต์แวร์ เราควรให้การสำคัญกับโดเมนที่ซอฟต์แวร์จะถูกใช้งานอยู่ จุดประสงค์ของซอฟต์แวร์ทั้งหมดคือการเพิ่มประสิทธิภาพให้กับโดเมนเฉพาะนั้น โดยเพื่อที่จะสามารถทำได้นั้น ซอฟต์แวร์ต้องเหมาะสมกับโดเมนที่มันถูกสร้างขึ้นมาเพื่อใช้งาน มิฉะนั้น ซอฟต์แวร์ก็จะนำความตึงเครียดเข้าสู่โดเมน ทำให้เกิดข้อผิดพลาด ความเสียหายและอาจทำให้เกิดความสับสนและความขัดแย้งในโดเมนด้วย

การออกแบบโดยใช้โดเมนมอเดลจะช่วยให้ผู้ที่ไม่มีความรู้เกี่ยวกับธนาคารสามารถเรียนรู้ได้มากมายโดยการอ่านโค้ดในโมเดลโดเมน นี่เป็นสิ่งที่สำคัญ โปรแกรมที่ไม่มีรากฐานที่ลึกซึ้งในโดเมนจะไม่สามารถรองรับการเปลี่ยนแปลงได้ดีในระยะยาว

การเริ่มต้นด้วยโดเมน จากนั้นจะทำอย่างไรต่อไป? โดเมนเป็นสิ่งหนึ่งในโลกนี้ ไม่สามารถเอามากรอกลงบนแป้นพิมพ์เพื่อกลายเป็นโค้ดได้อย่างง่ายดาย เราต้องสร้างการสรรหาของโดเมน และเราจะได้เรียนรู้จากผู้เชี่ยวชาญด้านโดเมนได้มากมาย แต่ความรู้เบื้องต้นนี้จะไม่สามารถแปลงเป็นโครงสร้างซอฟต์แวร์ได้อย่างง่ายดาย เราจึงต้องสร้างนวัตกรรมของโดเมน ซึ่งเป็นแบบจำลองของโดเมน เรียกได้ว่าโดเมนโมเดล ตาม Eric Evans โดเมนโมเดลไม่ใช่แผนภูมิเพียงแค่อย่างเดียว แต่เป็นความคิดที่แผนภูมินั้นจะต้องสื่อความหมาย โดเมนโมเดลไม่ได้แค่เป็นความรู้ที่อยู่ในหัวของผู้เชี่ยวชาญด้านโดเมน แต่เป็นแบบจำลองที่เราสร้างขึ้นจากความรู้ที่ได้จากผู้เชี่ยวชาญด้านโดเมน แผนภูมิสามารถแสดงและสื่อสารโดเมนโมเดลได้ เช่นโค้ดที่เขียนอย่างรอบคอบและรัดกุม หรือประโยคภาษาอังกฤษที่เขียนอย่างถูกต้อง โดยสามารถเป็นแบบจำลองของโดเมนได้อย่างมีประสิทธิภาพ

โมเดลเป็นการแสดงภายในของโดเมนเป้าหมายของเรา และมันเป็นสิ่งที่จำเป็นมากในขั้นตอนการออกแบบและพัฒนา ระหว่างการออกแบบเราจะจำได้และอ้างอิงมากมายต่อโมเดล เพราะโลกรอบตัวเรามีมากเกินไปสำหรับการจัดการด้วยความจำของมนุษย์ แม้ว่าโดเมนที่เฉพาะเจาะจงจะเป็นไปได้ว่ามันจะมากกว่าที่จะจัดการได้ในเวลาเดียวกัน ดังนั้นเราจำเป็นต้องจัดระเบียบข้อมูล ทำระบบ แบ่งแยกให้เป็นชิ้นเล็ก ๆ จัดกลุ่มชิ้นเหล่านั้นเป็นโมดูลที่มีเหตุผล และจะต้องดูแลเรื่องละเอียดของชิ้นงานแต่ละชิ้น และแยกส่วนอื่น ๆ ของโดเมนออกไปบ้าง โดเมนมีข้อมูลมากเกินไปที่จะรวมทั้งหมดลงในโมเดล และส่วนใหญ่จะไม่จำเป็นต้องพิจารณาเพิ่มเติม นี่เป็นภาระที่ต้องพิจารณาด้วยตัวเองว่าจะเก็บอะไรไว้ และจะทิ้งอะไรไป มันเป็นส่วนหนึ่งของการออกแบบและการสร้างซอฟต์แวร์ โปรแกรมธนาคารเช่นเดียวกันจะเก็บที่อยู่ของลูกค้า แต่ไม่จำเป็นต้องสนใจเรื่องสีตาของลูกค้า เป็นต้น นั่นเป็นกรณีที่ชัดเจน แต่กรณีอื่น ๆ อาจไม่ชัดเจนเท่านั้น

แบบจำลอง (Model) เป็นส่วนสำคัญของการออกแบบซอฟต์แวร์ ที่จำเป็นต้องมีเพื่อจัดการกับความซับซ้อน ทั้งหมดที่เราคิดเกี่ยวกับโดเมนจะถูกสังเคราะห์เข้าด้วยกันเป็นแบบจำลองเดียวกัน โดยแบบจำลองนี้จะเกี่ยวข้องกับข้อมูล โครงสร้าง และกระบวนการต่าง ๆ ของโดเมนนั้น ๆ ซึ่งจะช่วยให้เราเข้าใจและพัฒนาซอฟต์แวร์ได้ง่ายขึ้น โดยขั้นตอนการสร้างแบบจำลองจะขึ้นอยู่กับประเภทของโดเมนและขอบเขตของโครงการซอฟต์แวร์นั้น ๆ ด้วย นั่นเป็นสิ่งที่ดี แต่มันต้องออกจากสมองของเราด้วย มันไม่ได้มีประโยชน์มากนักถ้ามันยังอยู่ในนั้น ใช่ไหม? เราต้องสื่อสารโมเดลนี้กับผู้เชี่ยวชาญด้านโดเมน นักออกแบบเหล่านี้และนักพัฒนาด้วย ตัวแบบ (model) เป็นสารอาหารสำคัญของซอฟต์แวร์ แต่เราต้องสร้างวิธีการแสดงออกและสื่อสารกับผู้เชี่ยวชาญด้านโดเมน (domain experts) ผู้ออกแบบและนักพัฒนาอื่นๆ เพื่อให้เข้าใจและเห็นภาพของตัวแบบนั้น ได้อย่างชัดเจน เราไม่ได้อยู่คนเดียวในกระบวนการนี้ดังนั้นเราต้องแบ่งปันความรู้และข้อมูล และเราต้องทำมันอย่างดี แม่นยำ สมบูรณ์และไม่มีความกำกวม มีวิธีที่แตกต่างกันเพื่อทำให้เกิดการสื่อสารได้ ได้แก่การวาดแผนภาพ ใช้กรณีการใช้งาน วาดภาพ รูปภาพ เป็นต้น อีกวิธีหนึ่งคือการเขียน ทำให้เราสามารถเขียนเกี่ยวกับวิสัยทัศน์ของเราเกี่ยวกับโดเมนได้ วิธีสุดท้ายคือภาษา สามารถสร้างภาษาเพื่อสื่อสารเกี่ยวกับปัญหาที่เกี่ยวกับโดเมนได้ แต่เราจะรายละเอียดเพิ่มเติมทั้งหมดในภายหลัง แต่จุดสำคัญคือเราต้องสื่อสารแบบชัดเจนเกี่ยวกับโมเดล

การออกแบบโค้ด (code design) นั้นต่างจากการออกแบบซอฟต์แวร์ (software design) โดยการออกแบบซอฟต์แวร์คือการสร้างโครงสร้างของบ้านที่เหมือนกับมองภาพทั้งหมด ส่วนการออกแบบโค้ดจะเน้นที่รายละเอียด เช่นตำแหน่งของรูปภาพบนผนังบางพื้นที่ การออกแบบโค้ดเป็นสิ่งสำคัญอย่างมาก แต่ไม่มีความสำคัญเท่ากับการออกแบบซอฟต์แวร์ การออกแบบโค้ดที่ผิดพลาดส่วนใหญ่สามารถแก้ไขได้อย่างง่ายดาย ในขณะที่การออกแบบซอฟต์แวร์ที่ผิดพลาดจะต้องใช้ค่าใช้จ่ายสูงมากในการซ่อมแซม การย้ายรูปภาพไปทางซ้ายอาจจะเป็นเรื่องง่าย แต่การแตกไปยังฝั่งหนึ่งของบ้านเพื่อทำใหม่อีกครั้งนั้นเป็นเรื่องต่างหาก แม้ว่าผลงานสุดท้ายจะดี ก็ยังต้องมีการออกแบบโค้ดอย่างดี เพื่อให้ได้รหัสที่สะอาด และสามารถบำรุงรักษาได้ดี ในการออกแบบโค้ดนั้น รูปแบบการเขียนโค้ดจะช่วยให้การออกแบบโค้ดดียิ่งขึ้น และควรนำเอารูปแบบการเขียนโค้ดที่ดีมาใช้เมื่อจำเป็น

แนวทางในการออกแบบซอฟต์แวร์มีหลายแนวทาง แบบหนึ่งคือวิธีการออกแบบแบบน้ำตก (waterfall design method) วิธีนี้ประกอบด้วยขั้นตอนหลายขั้นตอน โดยผู้เชี่ยวชาญด้านธุรกิจจะเสนอเกณฑ์ที่ต้องการ ซึ่งจะถูกสื่อสารไปยังนักวิเคราะห์ธุรกิจ นักวิเคราะห์ธุรกิจจะสร้างโมเดลโดยพึ่งข้อมูลเหล่านั้น แล้วส่งผลการวิเคราะห์ให้กับนักพัฒนา ซึ่งจะเริ่มต้นเขียนโค้ดตามที่ได้รับมา วิธีนี้เป็นการไหลเดียวของความรู้ แม้ว่าวิธีนี้จะเป็นแนวทางที่เก่าแก่ในการออกแบบซอฟต์แวร์ และใช้ได้บางครั้งโดยมีความสำเร็จในหลายปี แต่มีข้อจำกัดและข้อบกพร่อง ปัญหาหลักคือไม่มีการตอบกลับจากนักวิเคราะห์ธุรกิจถึงผู้เชี่ยวชาญด้านธุรกิจหรือจากนักพัฒนาถึงนักวิเคราะห์ธุรกิจ

วิธีการที่มีอีกแบบคือ Agile methodologies เช่น Extreme Programming (XP) วิธีการเหล่านี้เป็นการรวมตัวกันเพื่อต่อต้านการใช้วิธีการ Waterfall ที่มีข้อจำกัดและข้อบกพร่อง โดยเป็นผลมาจากความยากที่จะคาดเดาทุกข้อกำหนดได้แบบสมบูรณ์ไว้แต่แรก โดยเฉพาะเมื่อมีการเปลี่ยนแปลงกฎเกณฑ์ การสร้างโมเดลที่ครอบคลุมทุกภาคส่วนของโดเมนแบบสมบูรณ์ตั้งแต่แรกยิ่งไปกว่านั้นยิ่งเป็นเรื่องยาก การออกแบบซอฟต์แวร์ต้องใช้เวลาในการคิดเยอะแยะและบางครั้งคุณไม่สามารถมองเห็นปัญหาทั้งหมดตั้งแต่เริ่มต้นได้ นอกจากนี้ยังไม่สามารถคาดการณ์ผลข้างเคียงหรือข้อผิดพลาดของการออกแบบได้เสมอไปเช่นกัน อีกปัญหาที่ Agile พยายามแก้ไขคือการติดตั้ง “analysis paralysis” หรือความกลัวที่สมาชิกในทีมจะตัดสินใจออกแบบและพัฒนาไม่ได้เพราะกลัวที่จะเกิดข้อผิดพลาด จนทำให้ไม่มีความคืบหน้าเลยโดยสมบูรณ์ Agile นำเสนอการตัดสินใจในการออกแบบแต่ไม่ได้ต้องการการออกแบบล่วงหน้า แทนที่จะใช้ความยืดหยุ่นในการดำเนินการ และผ่านการพัฒนาต่อเนื่องด้วยการมีผู้เข้าร่วมธุรกิจอยู่เสมอ รวมถึงการทำการปรับปรุงโค้ดอย่างสม่ำเสมอ เที้ยงซ้ำเติม ทีมพัฒนาจะได้เรียนรู้เพิ่มเติมเกี่ยวกับโดเมนลูกค้าและสามารถผลิตซอฟต์แวร์ที่ตอบสนองความต้องการของลูกค้าได้ดีขึ้น

วิธีการ Agile มีปัญหาและข้อจำกัดของตัวเอง วิธีนี้สนับสนุนความเรียบง่าย แต่ทุกคนมีมุมมองของตัวเองว่ามันหมายถึงอะไร นอกจากนี้ การ Refactoring ต่อเนื่องที่ผู้พัฒนาทำโดยไม่มีหลักการออกแบบที่มั่นคงจะสร้างโค้ดที่ยากต่อการเข้าใจหรือเปลี่ยนแปลง และในขณะที่วิธีวางแผนแบบ waterfall อาจนำไปสู่การออกแบบโดยมีการเคลื่อนไหวมากเกินไป (over-engineering) ความกลัวที่จะเกิดการออกแบบโดยมีการเคลื่อนไหวมากเกินไปอาจเป็นเหตุผลหนึ่งที่ส่งผลต่อความกลัวของการออกแบบที่ลึกซึ้งและคิดอย่างเต็มความสามารถ

หนังสือเล่มนี้นำเสนอหลักการของการออกแบบด้วยการตระหนักถึงโดเมน (Domain Driven Design) ซึ่งเมื่อนำมาใช้ จะช่วยเพิ่มประสิทธิภาพของกระบวนการพัฒนาในการจำลองและดำเนินการแก้ไขปัญหาที่ซับซ้อนในโดเมน ในทางกลับกัน การออกแบบและการพัฒนาสามารถทำงานร่วมกันเพื่อสร้างตัวแก้ไขที่ดีกว่าได้ การออกแบบที่ดีจะเร่งความเร็วในการพัฒนา ในขณะที่ข้อมูลตอบรับจากกระบวนการพัฒนาจะเสริมสร้างการออกแบบให้ดียิ่งขึ้น

**การสร้างความรู้ในเชิงพื้นฐาน**

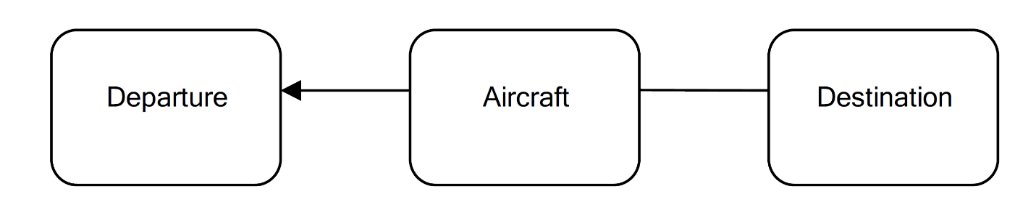
เรามาพิจารณาตัวอย่างของโครงการระบบควบคุมเครื่องบินและวิธีการสร้างความรู้ในโดเมนนี้กันดีกว่า

มีเครื่องบินหลายพันเครื่องบินอยู่ในอากาศในขณะใดขณะหนึ่งทั่วโลก เราจะไม่พยายามอธิบายระบบควบคุมการจราจรทั้งหมด แต่จะให้ความสำคัญกับส่วนย่อยที่เล็กลงซึ่งคือระบบติดตามการบิน โครงการที่เสนอคือระบบติดตามการบินที่ติดตามทุกการบินในพื้นที่ที่กำหนด กำหนดว่าการบินนั้นเป็นตามเส้นทางที่กำหนดหรือไม่ และหากมีโอกาสชนกับเครื่องบินอื่นในท้องฟ้าไหม

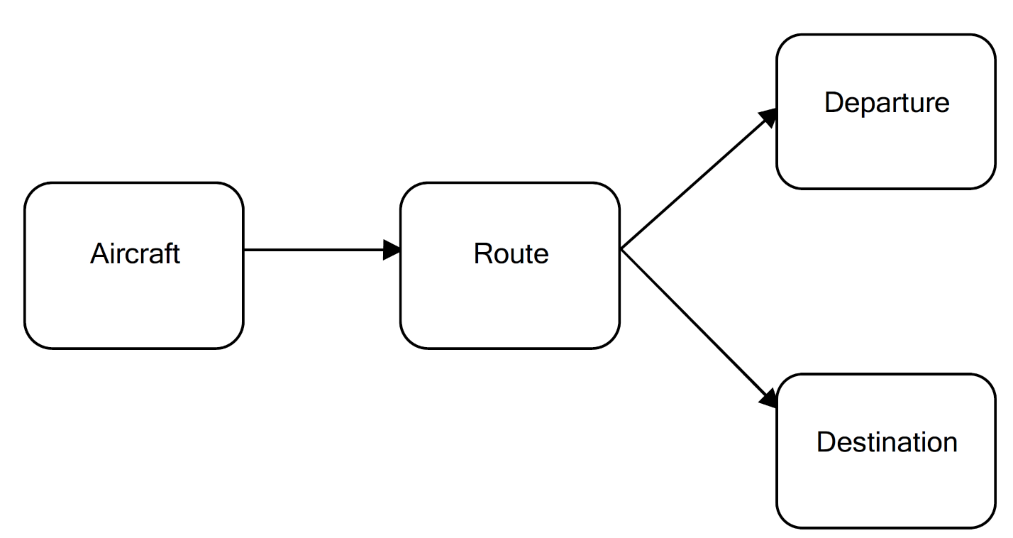
จากมุมมองการพัฒนาซอฟต์แวร์ เราจะเริ่มต้นจากไหนดี? ในส่วนก่อนหน้านี้เรากล่าวไว้ว่าเราควรเริ่มต้นด้วยการเข้าใจเรื่องโดเมน (domain) ซึ่งในกรณีนี้คือการติดตามการบินในอากาศ ผู้ควบคุมการจราจรทางอากาศเป็นผู้เชี่ยวชาญในสาขานี้ แต่ผู้ควบคุมไม่ใช่นักออกแบบระบบหรือผู้เชี่ยวชาญด้านซอฟต์แวร์ คุณไม่สามารถคาดหวังให้พวกเขามอบคำอธิบายที่สมบูรณ์ของโดเมนปัญหาให้คุณได้โดยตรง

ผู้ควบคุมการจราจรทางอากาศมีความรู้ที่มหาศาลเกี่ยวกับสาขางานของพวกเขา แต่ในการสร้างโมเดลจำเป็นต้องสกัดข้อมูลสำคัญและทำให้เป็นทั่วไป ผู้ควบคุมการจราจรทางอากาศมีความรู้ที่มหาศาลเกี่ยวกับสาขางานของพวกเขา แต่ในการสร้างโมเดลจำเป็นต้องสกัดข้อมูลสำคัญและทำให้เป็นทั่วไป เพื่อหาความเป็นระเบียบในปริมาณข้อมูลที่ดูจะไม่มีลำดับเราจำเป็นต้องเริ่มต้นที่ไหนบางที

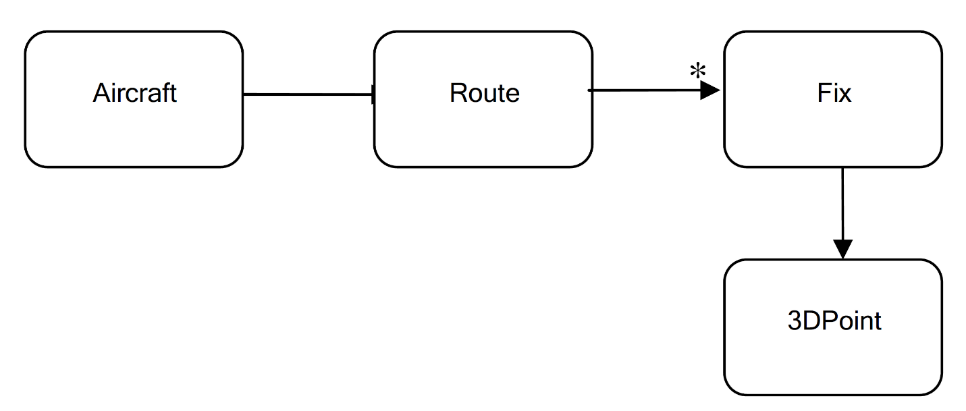
ผู้ควบคุมและคุณตกลงว่าแต่ละเครื่องบินมีสนามบินขึ้นลงต้นทางและปลายทาง ดังนั้นเรามีเครื่องบิน สนามบินขึ้นลงต้นทางและสนามบินปลายทางตามที่แสดงในรูปด้านล่าง



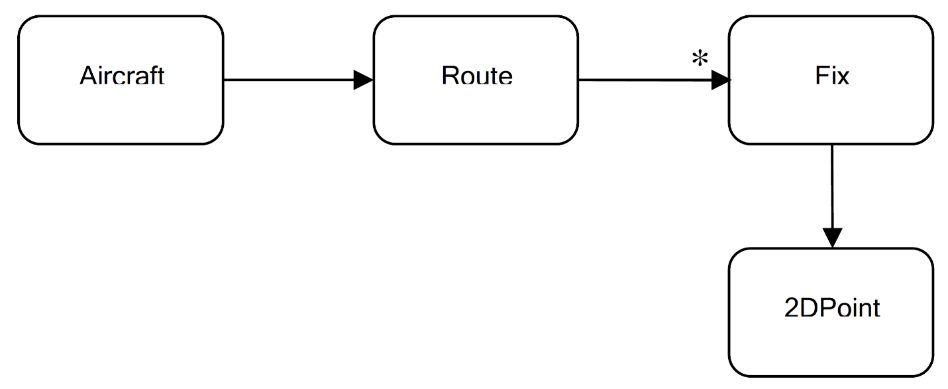
โอเค โดยอากาศยานจะบินขึ้นจากสถานที่หนึ่งและลงจอดในสถานที่อื่น แต่เกิดอะไรขึ้นในอากาศ? เส้นทางการบินที่มันผ่านไปเป็นอย่างไร? จริงๆ แล้วเราสนใจมากกว่าเกี่ยวกับสิ่งที่เกิดขึ้นขณะที่มันอยู่ในอากาศ ตัวควบคุมบอกว่าแต่ละเครื่องบินได้รับแผนการบินที่ควรอธิบายการเดินทางทางอากาศทั้งหมด ฟังเกี่ยวกับแผนการบิน คุณอาจคิดในใจว่าเป็นเกี่ยวกับเส้นทางที่เครื่องบินตามขณะที่อยู่ในอากาศ หลังจากพูดคุยกันต่อไป คุณได้ยินคำว่า "route" ที่น่าสนใจมาก และคุณตอบสนองอย่างทันที เพราะมีเหตุผลที่ดีอยู่ด้วย เส้นทางนั้นมีแนวความคิดสำคัญในการเดินทางโดยเครื่องบิน นั่นคือเครื่องบินทำอะไรขณะบินอยู่ พวกมันตามเส้นทาง มันเป็นเรื่องชัดเจนว่าจุดออกและปลายทางของเครื่องบินก็เป็นจุดเริ่มต้นและจุดสิ้นสุดของเส้นทางด้วย ดังนั้น ไม่ใช่การเชื่อมโยงเครื่องบินกับจุดต้นทางและปลายทาง แต่ดูเหมือนจะเป็นธรรมชาติมากกว่าที่จะเชื่อมโยงกับเส้นทาง ซึ่งในเทิร์นนั้นจะเชื่อมโยงกับจุดต้นทางและปลายทางที่เกี่ยวข้อง



พูดคุยกับผู้ควบคุมเครื่องบินเกี่ยวกับเส้นทางที่เครื่องบินตามไป คุณพบว่าจริงๆแล้วเส้นทางประกอบไปด้วยส่วนย่อยๆ ซึ่งรวมกันเป็นเส้นทางที่เบี่ยงเบนจากจุดออกเดินทางไปยังจุดหมายของเรา การวาดเส้นตรงควรผ่านไปที่จุดที่กำหนดไว้ล่วงหน้า ดังนั้นเส้นทางสามารถพิจารณาได้เป็นชุดของจุดที่ต่อเนื่องกัน โดยที่จุดเหล่านั้นเป็นจุดที่กำหนดไว้ล่วงหน้าแล้วว่าต้องผ่านไปในเส้นทางนั้นๆ ในจุดนี้ คุณไม่ได้มองการออกเดินทางและปลายทางเป็นจุดปลายทางของเส้นทางแล้ว แต่เพียงแค่เป็นสองจุดจากหนึ่งในจุดประสงค์นั่นเอง น่าจะแตกต่างออกไปจากวิธีการควบคุมที่ผู้ควบคุมมองเห็น แต่เป็นการแยกสิ่งที่จำเป็นที่ช่วยให้ง่ายต่อการใช้งานต่อไป การเปลี่ยนแปลงที่เกิดขึ้นจากการค้นพบเหล่านี้ มีดังนี้:



แผนภาพแสดงส่วนประกอบอีกองค์ประกอบหนึ่ง คือ ข้อความที่ว่าแต่ละจุดซ่อมเป็นจุดในพื้นที่ที่ตามมาด้วยเส้นทาง และมันถูกแสดงเป็นจุดสามมิติ แต่เมื่อคุณพูดคุยกับผู้ควบคุม คุณจะค้นพบว่าเขาไม่เห็นเหมือนกับนั้น ในความเป็นจริงเขาเห็นเส้นทางเป็นการโปรเจกชันของการบินของเครื่องบินบนโลก การแก้ไข (The fixes) คือจุดบนพื้นผิวโลกที่กำหนดโดยพิกัดละติจูดและลองจิจูดอย่างไม่เหมือนกัน เพื่อนำมาใช้ในการปรับปรุงหรือแก้ไขปัญหาที่เกิดขึ้น ดังนั้นแผนภาพที่ถูกต้องคือ:



คุณกำลังสนทนากับผู้เชี่ยวชาญด้านหนึ่ง เพื่อแลกเปลี่ยนความรู้ สิ่งที่เกิดขึ้นที่นี่คืออะไรจริงๆ? คุณเริ่มถามคำถาม และพวกเขาตอบ ในขณะที่พวกเขาทำเช่นนั้น พวกเขาขุดค้นแนวคิดสำคัญจากโดเมนการจราจรทางอากาศออกมา แนวความคิดเหล่านั้นอาจออกมาไม่สมบูรณ์และไม่เป็นระบบ แต่ยังคงเป็นสิ่งที่สำคัญต่อการเข้าใจด้านที่เกี่ยวข้อง คุณต้องเรียนรู้เกี่ยวกับโดเมนอย่างมากที่สุดจากผู้เชี่ยวชาญ และโดยการถามคำถามที่ถูกต้องและประมวลผลข้อมูลในทางที่ถูกต้อง คุณและผู้เชี่ยวชาญจะเริ่มสร้างภาพรวมของโดเมน นั่นคือ โมเดลโดเมน มุมมองนี้ไม่สมบูรณ์และไม่ถูกต้อง แต่มันเป็นจุดเริ่มต้นที่คุณต้องการ พยายามค้นหาแนวคิดสำคัญของโดเมนได้เลย

นี่เป็นส่วนสำคัญของการออกแบบ โดยทั่วไปจะมีการอภิปรายในระยะยาวระหว่างนักออกแบบซอฟต์แวร์หรือนักพัฒนากับผู้เชี่ยวชาญในด้านนี้ ผู้เชี่ยวชาญซอฟต์แวร์ต้องการแยกความรู้จากผู้เชี่ยวชาญด้านสาขาและพวกเขายังต้องแปลงมันให้เป็นรูปแบบที่มีประโยชน์ ในบางช่วง, พวกเขาอาจต้องการสร้างโปรโตไทป์เร็วๆนี้เพื่อดูว่ามันทำงานได้ดีแค่ไหนแล้ว ในขณะที่ทำเช่นนั้นพวกเขาอาจพบปัญหาบางอย่างกับโมเดลหรือวิธีการของพวกเขา และอาจต้องการเปลี่ยนโมเดล การสื่อสารไม่ได้เป็นแค่ทางเดียว จากผู้เชี่ยวชาญในโดเมนไปสู่สถาปนิกซอฟต์แวร์และต่อมาถึงนักพัฒนาโปรแกรมเท่านั้น มีการตอบกลับ (feedback) อีกด้วย ซึ่งช่วยให้มีโมเดลที่ดีขึ้นและเข้าใจโดเมนได้ชัดเจนและถูกต้องมากขึ้นด้วย ผู้เชี่ยวชาญโดเมนรู้ความเชี่ยวชาญของพวกเขาอย่างดี แต่พวกเขาจัดระบบและใช้ความรู้ของพวกเขาในวิธีที่เฉพาะเจาะจง ซึ่งไม่ได้เป็นวิธีการที่ดีที่สุดสำหรับการนำไปใช้ในระบบซอฟต์แวร์เสมอไปทุกครั้ง ความคิดวิเคราะห์ของนักออกแบบซอฟต์แวร์ช่วยหาคำศัพท์สำคัญในโดเมนขณะที่พูดคุยกับผู้เชี่ยวชาญในโดเมนและยังช่วยสร้างโครงสร้างสำหรับการพูดคุยในอนาคตตามที่จะเห็นในบทถัดไป เราผู้เชี่ยวชาญด้านซอฟต์แวร์ (สถาปนิกและนักพัฒนาซอฟต์แวร์) และผู้เชี่ยวชาญด้านโดเมนกำลังสร้างแบบจำลองของโดเมนร่วมกัน และแบบจำลองนี้คือสถานที่ที่สองพื้นที่เชี่ยวชาญเหล่านั้นพบกัน การดูเหมือนจะเป็นกระบวนการที่ใช้เวลามากเป็นอย่างมาก และจริงๆ แล้วเป็นเช่นนั้น เพราะในที่สุดจุดมุ่งหมายของซอฟต์แวร์คือการแก้ไขปัญหาธุรกิจในโดเมนชีวิตจริง ดังนั้นจึงต้องผสมผสานอย่างลงตัวกับโดเมนอย่างสมบูรณ์แบบ

**2 - ภาษาที่มีอยู่ทั่วไป (The Ubiquitous Language)**

**ความจำเป็นของภาษาที่เหมือนกัน**

บทก่อนหน้านี้ได้ยกตัวอย่างว่าจำเป็นต้องพัฒนาโมเดลของโดเมนโดยการทำงานร่วมกับผู้เชี่ยวชาญด้านโดเมน อย่างไรก็ตามวิธีการนี้มักมีความยากลำบากในตอนแรกเนื่องจากมีการขัดข้องในการสื่อสารพื้นฐาน นักพัฒนามีความคิดเต็มไปด้วยคลาส เมธอด อัลกอริทึม และแพทเทิร์น และมักจะเชื่อมโยงระหว่างแนวความคิดในชีวิตจริงกับศิลปะการเขียนโปรแกรม พวกเขาต้องการดูว่าคลาสอ็อบเจ็กต์จะถูกสร้างขึ้นอย่างไรและมีความสัมพันธ์กันอย่างไร พวกเขาคิดเช่นเดียวกับ inheritance, polymorphism, OOP เป็นต้น และพูดคุยกันเช่นนั้นตลอดเวลา และมันเป็นสิ่งปกติสำหรับพวกเขาที่จะทำเช่นนั้น เพราะนักพัฒนาจะยังคงเป็นนักพัฒนาอยู่เสมอ แต่ผู้เชี่ยวชาญด้านของโดเมนที่มักจะไม่รู้อะไรเกี่ยวกับสิ่งเหล่านี้ เขาไม่มีความรู้เกี่ยวกับ software libraries, frameworks, persistence และในบางกรณีอาจไม่รู้อะไรเกี่ยวกับฐานข้อมูลเลย พวกเขารู้เกี่ยวกับพื้นที่ความเชี่ยวชาญของตนเองเท่านั้น

ในตัวอย่างการตรวจสอบการจราจรทางอากาศ ผู้เชี่ยวชาญด้านโดเมนรู้เกี่ยวกับเครื่องบิน เส้นทาง ความสูงของเครื่องบิน ลองจิจูดและละติจูด พวกเขารู้เกี่ยวกับการเบนทางจากเส้นทางปกติและเกี่ยวกับแนวโน้มของเครื่องบิน และพวกเขาพูดถึงเรื่องเหล่านี้โดยใช้ภาษาคำศัพท์ของตนเองซึ่งไม่ใช่สิ่งที่เข้าใจได้ง่ายโดยผู้ที่ไม่ได้อยู่ในวงการนั้น

เพื่อเอาชนะความแตกต่างในสไตล์การสื่อสาร ในขณะที่เรากำลังสร้างโมเดล เราต้องสื่อสารเพื่อแลกเปลี่ยนความคิดเห็นเกี่ยวกับโมเดล การสื่อสารเกี่ยวกับองค์ประกอบที่เกี่ยวข้องในโมเดล ว่าเราเชื่อมต่ออย่างไร สิ่งที่เป็นประโยชน์และสิ่งที่ไม่เกี่ยวข้องกับโมเดล การสื่อสารในระดับนี้เป็นสิ่งที่สำคัญอย่างยิ่งสำหรับความสำเร็จของโครงการ หากมีคนพูดอะไรไปแล้วอีกคนไม่เข้าใจ หรืออย่างแย่ที่สุดเข้าใจผิด โอกาสในการประสบความสำเร็จของโครงการจะเป็นอย่างไรบ้าง?

โปรเจกต์จะเผชิญกับปัญหาหนัก เมื่อสมาชิกในทีมไม่มีภาษาที่ร่วมกันในการพูดคุยเกี่ยวกับโดเมน ผู้เชี่ยวชาญด้านโดเมนใช้ภาษาเฉพาะของพวกเขา ในขณะที่สมาชิกในทีมทางเทคนิคมีภาษาของตนเองที่ปรับให้เหมาะกับการพูดคุยเกี่ยวกับโดเมนจากมุมมองด้านการออกแบบ

คำศัพท์ที่ใช้ในการพูดประจำวันไม่สอดคล้องกับคำศัพท์ที่ฝังอยู่ในโค้ด (ซึ่งเป็นผลิตภัณฑ์ที่สำคัญที่สุดของโครงการซอฟต์แวร์) และแม้ว่าคนเดียวกันจะใช้ภาษาที่แตกต่างกันในการพูดและเขียน ดังนั้น แสดงให้เห็นว่าประโยคที่เจาะจงที่สุดของโดเมนบ่งชี้ในรูปแบบชั่วคราวที่ไม่เคยถูกบันทึกไว้ในโค้ดหรือแม้แต่เขียนลงกระดาษ

ในช่วงการสื่อสารเหล่านี้ การแปลถูกนำมาใช้บ่อยครั้งเพื่อให้ผู้อื่นเข้าใจเกี่ยวกับแนวคิดบางอย่าง นักพัฒนาอาจพยายามอธิบายแนวคิดอาจจะใช้ภาษาที่ไม่เชี่ยวชาญและบางครั้งก็ไม่ประสบความสำเร็จ ผู้เชี่ยวชาญด้านสาขาอาจพยายามอธิบายแนวคิดของพวกเขาโดยสร้างภาษาวิชาการใหม่ ในขั้นตอนนี้ การสื่อสารมีปัญหา และการแปลประเภทนี้ไม่ช่วยในการสร้างความรู้

เรามักใช้ภาษาโดยเฉพาะของเราในช่วงการออกแบบเหล่านี้ แต่ไม่มีภาษาโดยเฉพาะใดที่เป็นภาษาทั่วไปเนื่องจากไม่มีภาษาใดที่ตอบโจทย์ของทุกคน

เราจำเป็นต้องพูดภาษาเดียวกันเมื่อเราพบกันเพื่อพูดคุยเกี่ยวกับโมเดลและกำหนดกฎเกณฑ์ ภาษาที่จะเป็นอย่างไร? ภาษาของนักพัฒนา? ภาษาของผู้เชี่ยวชาญในด้าน? หรือบางอย่างอยู่ระหว่างทั้งสอง?

หลักการหลักของการออกแบบด้วยการใช้งาน Domain-Driven Design คือการใช้ภาษาที่เชื่อมโยงกับโมเดล โดยเนื่องจากโมเดลเป็นจุดรวมกลางที่ซอฟต์แวร์และโดเมนพบกัน ดังนั้น เป็นเหมาะที่จะใช้โมเดลนี้เป็นพื้นฐานสำหรับการสร้างภาษาในการออกแบบโปรแกรม โดยใช้โมเดลเป็นฐานในการสร้างภาษานี้

ใช้โมเดลเป็นตัวกระดูกหลักของภาษา ขอให้ทีมใช้ภาษาโดยสม่ำเสมอในการสื่อสารทั้งหมด รวมถึงในรหัสโปรแกรมด้วย ในขณะที่แบ่งปันความรู้และพัฒนาโมเดล ทีมใช้สื่อต่างๆ เช่น การพูด การเขียน และแผนภูมิ ให้แน่ใจว่าภาษานี้ปรากฏอย่างสม่ำเสมอในรูปแบบการสื่อสารที่ทีมใช้ทั้งหมด เพื่อเหตุนี้ ภาษาเรียกว่า Ubiquitous Language

ภาษาสากลเชิงกว้างเชื่อมโยงทุกส่วนของการออกแบบ และสร้างพื้นฐานให้ทีมออกแบบทำงานได้อย่างดี ในโครงการออกแบบขนาดใหญ่ อาจใช้เวลาหลายสัปดาห์หรือแม้แต่หลายเดือนในการสร้างแบบจากต้นแบบ สมาชิกในทีมค้นพบว่าบางแนวคิดเริ่มต้นไม่ถูกต้องหรือถูกใช้ไม่เหมาะสม หรือพวกเขาค้นพบองค์ประกอบใหม่ของการออกแบบที่ต้องพิจารณาและเหมาะสมกับการออกแบบโดยรวม นี้ทั้งหมดไม่สามารถทำได้โดยไม่มีภาษาที่เป็นร่วมกัน

ภาษาไม่ปรากฏขึ้นแบบครั้งใดครั้งหนึ่ง มันต้องใช้ความพยายามและการมุ่งมั่นอย่างมากเพื่อให้แน่ใจว่าองค์ประกอบสำคัญของภาษาถูกเปิดเผยขึ้นมา เราต้องหาแนวคิดสำคัญเหล่านั้นที่กำหนดโดเมนและการออกแบบ และหาคำที่เกี่ยวข้องกับพวกเขา และเริ่มใช้พวกเขา บางอย่างจะมองเห็นได้ง่าย แต่บางอย่างยากขึ้น

ปรับปรุงปัญหาโดยการทดลองใช้ส่วนประกอบทดแทน ที่สะท้อนรูปแบบทดแทนต่าง ๆ จากนั้นทำการปรับโค้ดโดยการเปลี่ยนชื่อคลาส และเมท็อด และโมดูลเพื่อให้เข้ากับรูปแบบใหม่ เรียกใช้คำศัพท์ที่สับสนในการสนทนาในลักษณะเดียวกับเราเข้าใจความหมายของคำธรรมดา

การสร้างภาษาเช่นนั้นมีผลลัพธ์ที่ชัดเจน: โมเดลและภาษาที่สร้างขึ้นมีความเชื่อมโยงกันอย่างแน่นหนา การเปลี่ยนแปลงในภาษาจะต้องเป็นการเปลี่ยนแปลงในโมเดลด้วย

**การสร้างภาษาแบบทั่วไป**

วิธีที่จะเริ่มต้นสร้างภาษาคืออะไร? นี่คือสมมติฐานการสนทนาระหว่างนักพัฒนาซอฟต์แวร์และผู้เชี่ยวชาญด้านการติดตามการจราจรทางอากาศ โปรดสังเกตคำที่ปรากฏเป็นตัวหนา

**นักพัฒนา:** พวกเราควรเริ่มต้นจากอะไรกันดีเพื่อสร้างภาษาในการติดตามการจราจรทางอากาศ?

**ผู้เชี่ยวชาญด้านสนามบิน:** เริ่มต้นกับพื้นฐานกันก่อนนะคะ การจราจรทั้งหมดนี้ประกอบด้วย**เครื่องบิน(planes)** แต่ละเครื่องบินจะขึ้นจาก**สนามบินต้นทาง(departure)**และลงที่**สนามบินปลายทาง(destination)**

**นักพัฒนา:** ง่ายมากครับ ตอนที่เครื่องบินบินได้แล้ว นักบินสามารถเลือกเส้นทางอากาศที่ต้องการได้เองหรือไม่? สามารถเป็นไปตามต้องการของพวกเขาในการตัดสินใจเส้นทางที่จะบินไปได้ เพียงแค่พวกเขาเดินทางไปถึงจุดหมายได้เท่านั้นใช่ไหมครับ?

**ผู้เชี่ยวชาญด้านสนามบิน:** โอ้ไม่ นักบินได้รับเส้นทางที่ต้องเดินทางตาม และพวกเขาควรที่จะติดตามเส้นทางนั้นให้ใกล้เคียงที่สุดเท่าที่จะเป็นไปได้

**นักพัฒนา:** ฉันกำลังคิดเส้นทางนี้เป็นเส้นทางสามมิติในอากาศ หากเราใช้ระบบพิกัดคาทีเซียน แล้วเส้นทางก็จะเป็นเพียงชุดของจุดสามมิติ

**ผู้เชี่ยวชาญด้านสนามบิน:** ฉันไม่คิดว่าเป็นเช่นนั้น เพราะเส้นทางจราจรอากาศจะเป็นการสร้างภาพลักษณ์ของเส้นทางที่คาดหวังของเครื่องบินบนพื้นดิน ซึ่งเส้นทางจะผ่านไปที่จุดต่างๆบนพื้นดินที่กำหนดโดยพิกัดละติจูดและลองจิจูดของมัน

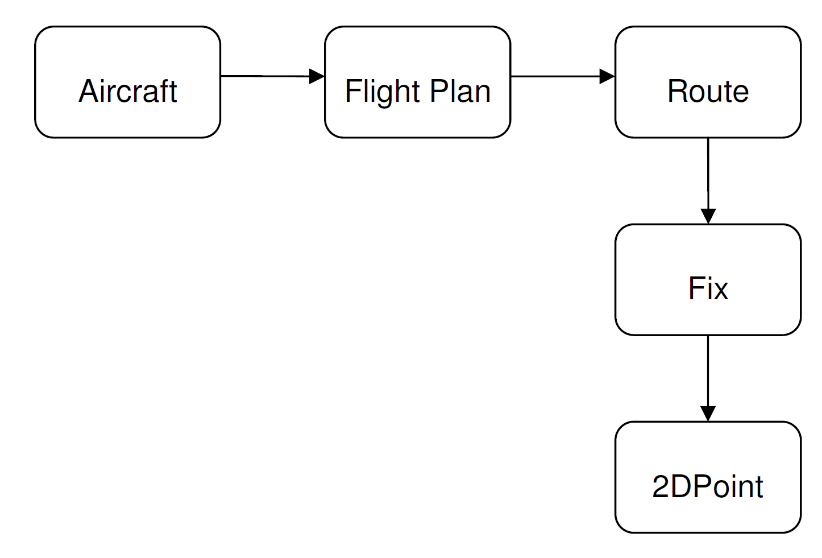
**นักพัฒนา:** โอเค แล้วเราจะเรียกจุดเหล่านั้นว่า "fix" เพราะว่าเป็นจุดที่ติดตั้งบนพื้นผิวของโลก และเราจะใช้จุด 2 มิติเพื่ออธิบายเส้นทาง นอกจากนี้ เราไม่ควรพิจารณาจุดเริ่มต้นและปลายทางเป็นแนวคิดที่แตกต่างกัน เส้นทางจะไปถึงปลายทางเหมือนกับการไปถึง fix ใด ๆ อื่น ๆ ของเส้นทาง อากาศยานต้องปฏิบัติตามเส้นทาง แต่นั่นหมายความว่ามันสามารถบินสูงหรือต่ำได้ตามที่ต้องการหรือไม่?

**ผู้เชี่ยวชาญด้านสนามบิน:** ไม่ใช่ **ความสูง(altitude)**ที่เครื่องบินจะต้องอยู่ในขณะใดขณะหนึ่งก็ถูกกำหนดไว้ในแผนการบิน**(flight plan)**

**นักพัฒนา**: **แผนการบิน(flight plan)**? มันคืออะไร?

**ผู้เชี่ยวชาญด้านสนามบิน:** ก่อนออกจากสนามบิน นักบินจะได้รับแผนการบินที่ละเอียดอ่อนซึ่งรวมถึงข้อมูลทั้งหมดเกี่ยวกับ**การบิน(flight):** **เส้นทางเที่ยวบิน(route)** **ความสูง(altitude)**ที่จะบิน **ความเร็ว(speed)** ในการบิน ประเภทของเครื่องบิน และข้อมูลเกี่ยวกับสมาชิกของลูกอากาศยาน

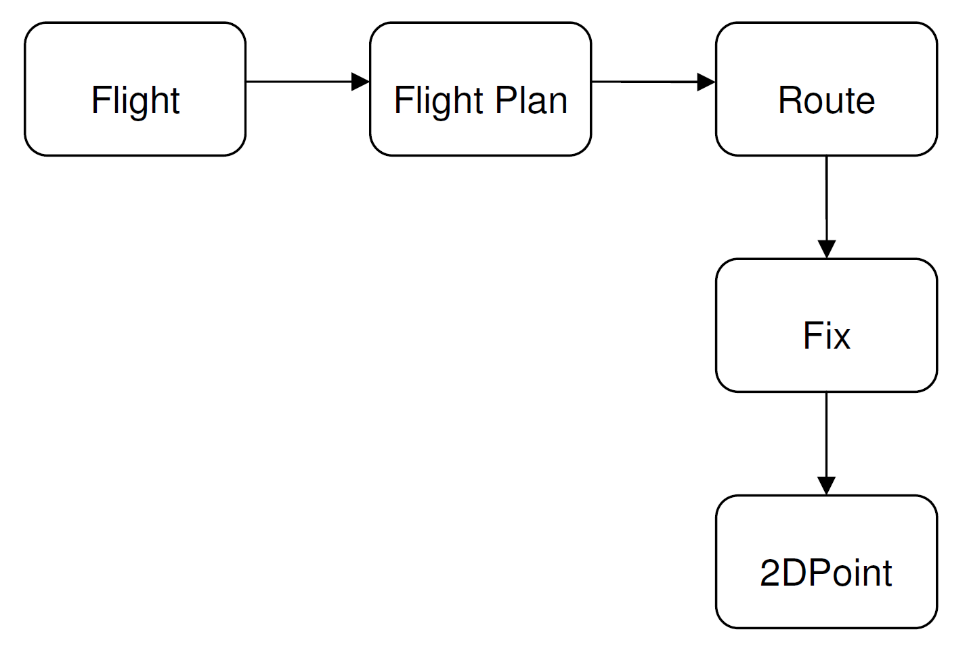
**นักพัฒนา:** ฮึ่ม **แผนการบิน(flight plan)** ดูเป็นสิ่งสำคัญอย่างมากสำหรับผม เรามาเพิ่มมันเข้าไปในโมเดลกันเถอะ



**นักพัฒนา:** ดีขึ้นแล้ว ตอนนี้ที่ฉันมองอยู่ฉันเข้าใจว่ามีสิ่งหนึ่งที่สำคัญ คือเมื่อเราตรวจสอบการจราจรทางอากาศ เราไม่ได้สนใจเครื่องบินเป็นอย่างแท้จริง ว่ามันสีขาวหรือสีน้ำเงินหรือเป็น Boeing หรือ Airbus อย่างไร แต่เราสนใจ**การบิน**ของมัน เป็นสิ่งที่เราติดตามและวัดได้อย่างแท้จริง ฉันคิดว่าเราควรเปลี่ยนโมเดลนิดหน่อยเพื่อให้มีความแม่นยำขึ้น

โปรดสังเกตว่าทีมนี้กำลังพูดถึงด้านการตรวจสอบการจราจรทางอากาศและโมเดลเริ่มต้นของพวกเขา พวกเขากำลังสร้างภาษาที่ประกอบด้วยคำที่เป็นตัวหนาอย่างช้าๆ โปรดสังเกตว่าภาษาดังกล่าวสามารถเปลี่ยนแปลงโมเดลได้!

อย่างไรก็ตามในชีวิตจริง การสนทนาแบบนี้มักจะมีคำพูดที่ยืดหยุ่นมากขึ้นและบุคคลมักพูดเกี่ยวกับเรื่องที่ไม่ตรงตามประเด็น หรือเข้ารายละเอียดมากเกินไป หรือเลือกใช้คำศัพท์ที่ไม่ถูกต้อง ซึ่งสามารถทำให้การคิดค้นภาษาเป็นเรื่องยากมากๆ อย่างไรก็ตาม เพื่อเริ่มต้นการแก้ไขปัญหานี้ สมาชิกทุกคนควรมีความตระหนักถึงความจำเป็นในการสร้างภาษาที่เหมือนกัน และควรได้รับการเตือนให้มุ่งเน้นสิ่งที่สำคัญและใช้ภาษาตลอดเวลาที่จำเป็น ในการประชุมดังกล่าว เราควรใช้ภาษาเฉพาะทางของเราให้น้อยที่สุดและควรใช้ภาษาแบบ Ubiquitous เพราะสิ่งนี้ช่วยให้เราสื่อสารได้อย่างชัดเจนและแม่นยำ



สามารถเขียนคลาสสำหรับ Route และอีกคลาสสำหรับ Fix ได้ คลาส Fix อาจสืบทอดจากคลาส 2DPoint หรือสามารถมี 2DPoint เป็นแอตทริบิวต์หลักได้ ซึ่งขึ้นอยู่กับปัจจัยอื่น ๆ ที่จะถูกพูดถึงในภายหลัง การสร้างคลาสสำหรับแนวคิดแบบนี้จะช่วยให้เราแมประหว่างแบบจำลองและโค้ด และแมพระหว่างภาษาและโค้ดได้ ซึ่งมีประโยชน์อย่างมาก เนื่องจากทำให้โค้ดอ่านง่ายขึ้น และทำให้โค้ดสามารถสะท้อนแบบจำลองได้ เมื่อโค้ดของแบบจำลองมีขนาดใหญ่ขึ้น และเมื่อมีการเปลี่ยนแปลงในโค้ดที่ออกแบบไม่เหมาะสมอาจทำให้เกิดผลกระทบที่ไม่ต้องการในโปรเจกต์ในภายหลัง

เราได้เห็นว่าภาษาถูกแชร์โดยทีมทั้งหมดแล้ว และยังช่วยสร้างความรู้และสร้างโมเดลได้อย่างไร้ข้อจำกัด แต่เราควรใช้อะไรสำหรับภาษา? เพียงแค่พูดหรือเท่านั้นหรือเราจะใช้ภาษาเขียนด้วย?

บางคนอาจพูดว่า UML เพียงพอต่อการสร้างโมเดล และในความจริงเป็นเครื่องมือที่ยอดเยี่ยมในการเขียนแนวคิดหลักเช่นคลาส และในการแสดงความสัมพันธ์ระหว่างพวกเขา คุณสามารถวาดคลาสสี่หรือห้าอย่างบนสก็อตแบบดินสอ และเขียนชื่อของพวกเขา และแสดงความสัมพันธ์ระหว่างพวกเขาได้อย่างง่ายดาย ทุกคนสามารถเข้าใจว่าคุณคิดอย่างไร และการแสดงกราฟิกของความคิดเป็นเรื่องง่ายต่อการเข้าใจ ทุกคนมีวิสัยทัศน์เดียวกันต่อเรื่องที่กำหนดไว้ และการสื่อสารกันก็ง่ายขึ้นตามนั้น โดยเมื่อมีไอเดียใหม่ๆเกิดขึ้นแล้ว แผนภาพก็จะถูกปรับเปลี่ยนให้ตรงกับการเปลี่ยนแปลงทางคอนเซปชั่นนั้นๆ

แผนผัง UML มีประโยชน์มากเมื่อจำนวนองค์ประกอบมีน้อย แต่ UML สามารถเติบโตได้เหมือนเห็ดหลังฝนฤดูร้อนที่สวยงาม คุณจะทำอย่างไรเมื่อมีร้อยๆ คลาสเต็มกระดาษเหมือนแม่น้ำมิสซิซิปี? มันยากในการอ่านแม้แต่สำหรับผู้เชี่ยวชาญซอฟต์แวร์ ไม่ต้องกล่าวถึงผู้เชี่ยวชาญด้านสาขาอื่นๆ พวกเขาจะไม่เข้าใจมันมากนักเมื่อมันใหญ่ขึ้น และมันก็จะเป็นเช่นนั้นแม้จะเป็นโปรเจกต์ขนาดกลาง

นอกจากนี้ UML ยังเหมาะสมในการแสดงคลาส แอตทริบิวต์ และความสัมพันธ์ระหว่างพวกเขา แต่พฤติกรรมของคลาสและข้อจำกัดก็ไม่ง่ายต่อการแสดงออกมา ดังนั้น UML ใช้เนื้อหาเป็นหมายเหตุที่วางไว้ในแผนภูมิเพื่อแสดงออกถึงสิ่งที่เราไม่สามารถแสดงได้ด้วย UML สองสิ่งสำคัญของโมเดล: ความหมายของแนวความคิดที่มันแทนและวัตถุที่ต้องการทำอะไร แต่นั่นไม่เป็นไร เนื่องจากเราสามารถเพิ่มเครื่องมือสื่อสารอื่น ๆ เข้าไปเพื่อทำเช่นนี้ได้

คือการสร้างภาพแผนผังเล็กๆ โดยแต่ละภาพจะประกอบด้วยส่วนย่อยของโมเดล แต่ละส่วนย่อยนั้นจะประกอบด้วยหลายคลาสและความสัมพันธ์ระหว่างพวกเขา ซึ่งรวมถึงส่วนใหญ่ของแนวคิดที่เกี่ยวข้องอยู่แล้ว จากนั้นเราสามารถเพิ่มข้อความในแผนผัง ข้อความนั้นจะอธิบายพฤติกรรมและผลของข้อจำกัดที่แผนผังไม่สามารถอธิบายได้ แต่ละส่วนย่อยพยายามอธิบายหนึ่งแง่มุมสำคัญของโดเมน โดยจุดเด่นที่เลือกจะช่วยให้เข้าใจส่วนที่สำคัญของโดเมนได้ดียิ่งขึ้น

เอกสารเหล่านั้นสามารถวาดเทียนมือได้เลย เพราะว่าการวาดนั้นสื่อถึงความรู้สึกว่าเป็นชั่วคราวและอาจเปลี่ยนแปลงได้ในอนาคตไม่กี่วันข้างหน้า ซึ่งเป็นจริง เนื่องจากโมเดลจะถูกเปลี่ยนแปลงหลายครั้งในช่วงเริ่มต้นก่อนที่จะเข้าสู่สถานะที่เสถียรมากขึ้น

อาจจะมีความต้องการที่จะพยายามสร้างแผนผังขนาดใหญ่ทั้งโมเดล อย่างไรก็ตาม ในส่วนใหญ่การสร้างแผนผังเช่นนั้นเป็นเรื่องยากมากแทบจะเป็นไปไม่ได้ และนอกจากนี้ แม้ว่าคุณจะสำเร็จในการสร้างแผนผังรวมนั้น แต่มันก็จะทับซ้อนกันมากเกินไปจนทำให้เข้าใจได้ยากและไม่ดีกว่าการใช้แผนผังขนาดเล็กหลายภาพแทนกัน

ควรระวังเอกสารที่ยาวเนื่องจากจะใช้เวลาเขียนมากและอาจกลายเป็นเอกสารที่ล้าสมัยก่อนที่จะเสร็จสมบูรณ์ ควรให้เอกสารตรงกับแบบจำลอง หากเอกสารเก่าไม่ใช้ภาษาที่ถูกต้องและไม่สอดคล้องกับแบบจำลองจะไม่เป็นประโยชน์มากนัก ควรพยายามหลีกเลี่ยงเอกสารเหล่านี้ในกรณีที่เป็นไปได้

การสื่อสารโดยใช้รหัสก็เป็นไปได้ แนวทางนี้ถูกสนับสนุนอย่างกว้างขวางโดยชุมชน XP รหัสที่เขียนได้ดีสามารถสื่อได้อย่างมากมาย แม้ว่าพฤติกรรมที่แสดงโดยวิธีการจะชัดเจน แต่ชื่อเมธอดชัดเจนเหมือนกับร่างกายของมันหรือไม่? การยืนยันของการทดสอบพูดเองแต่ตัวแปรและโครงสร้างรหัสโดยรวมเป็นอย่างไรบ้าง? พวกเขากำลังพูดเรื่องเต็มรูปแบบและชัดเจนหรือไม่? รหัสที่ทำงานตามประสิทธิภาพไม่จำเป็นต้องแสดงออกมาอย่างชัดเจนเสมอไป การเขียนโมเดลในรหัสเป็นเรื่องยากมาก

มีวิธีการสื่อสารอื่นๆ ในระหว่างการออกแบบ แต่เป็นเรื่องไม่ใช่วัตถุประสงค์ของหนังสือเล่มนี้ที่จะนำเสนอทั้งหมด อย่างไรก็ตาม สิ่งหนึ่งที่ชัดเจนอยู่แล้วคือ ทีมออกแบบที่ประกอบด้วยนักสถาปัตยกรรมซอฟต์แวร์ นักพัฒนา และผู้เชี่ยวชาญในสาขาความรู้ต่างๆ ต้องการภาษาที่รวมกันสำหรับการดำเนินการของพวกเขา และช่วยให้พวกเขาสร้างโมเดลและแสดงโมเดลนั้นด้วยโค้ดได้อย่างชัดเจน

**3 - การออกแบบโดยใช้โมเดล (Model-Driven Design)**

บทก่อนหน้าเน้นความสำคัญของการพัฒนาซอฟต์แวร์ด้วยการใช้วิธีการที่เน้นไปที่โดเมนธุรกิจ พวกเรากล่าวไว้ว่ามันเป็นสิ่งที่สำคัญอย่างพื้นฐานที่จะสร้างโมเดลที่เชื่อมโยงกับโดเมนอย่างลึกซึ้งและควรสะท้อนแนวคิดหลักของโดเมนอย่างแม่นยำ เราควรใช้ภาษาแบบกลมกล่อมไปในกระบวนการจำลองเพื่ออำนวยความสะดวกในการสื่อสารระหว่างผู้เชี่ยวชาญด้านซอฟต์แวร์และผู้เชี่ยวชาญด้านโดเมน และเพื่อค้นพบแนวคิดหลักของโดเมนที่สำคัญที่ควรใช้ในโมเดล วัตถุประสงค์ของกระบวนการจำลองนี้คือการสร้างโมเดลที่ดี ขั้นต่อไปคือการนำโมเดลนั้นไปใช้ในการเขียนโค้ด นี่เป็นขั้นตอนที่สำคัญเท่ากันของกระบวนการพัฒนาซอฟต์แวร์ หลังจากได้สร้างโมเดลที่ดี แต่ไม่สามารถโอนโมเดลนั้นเข้าสู่โค้ดได้อย่างเหมาะสม จะทำให้ซอฟต์แวร์ที่ได้มีคุณภาพที่ไม่แน่นอน

มักเกิดเหตุการณ์ที่วิเคราะห์ซอฟต์แวร์ทำงานร่วมกับผู้เชี่ยวชาญในธุรกิจเป็นเดือน ค้นพบองค์ประกอบพื้นฐานของโดเมน ให้ความสำคัญกับความสัมพันธ์ระหว่างองค์ประกอบนั้น และสร้างโมเดลที่ถูกต้องซึ่งจะจับต้องโดเมนอย่างแม่นยำ จากนั้นโมเดลจะถูกส่งต่อให้นักพัฒนาซอฟต์แวร์ นักพัฒนาอาจดูที่โมเดลและค้นพบว่าบางแนวความคิดหรือความสัมพันธ์ที่พบในโมเดลไม่สามารถแสดงอย่างถูกต้องในรหัสได้ ดังนั้นพวกเขาจะใช้โมเดลเป็นแหล่งแรงบันดาลใจต้นฉบับ แต่พวกเขาจะสร้างการออกแบบของตนเองซึ่งยืมบางไอเดียจากโมเดลและเพิ่มบางส่วนของตัวเองเข้าไป กระบวนการพัฒนาก็ยังคงดำเนินต่อไป และคลาสอื่นๆ ถูกเพิ่มเข้าไปในรหัส เพิ่มการแบ่งแยกระหว่างโมเดลต้นฉบับและการดำเนินการสุดท้าย ผลลัพธ์ที่ดีไม่ได้รับการยืนยันไว้ล่วงหน้า นักพัฒนาที่ดีอาจจะรวมกันผลิตภัณฑ์ที่ทำงานได้ แต่มันจะทนทานต่อการทดสอบเวลาหรือไม่? จะสามารถขยายตัวได้ง่ายหรือไม่? จะสามารถซ่อมบำรุงได้ง่ายหรือไม่?

โดเมนใดก็สามารถแสดงออกได้ด้วยแบบจำลองหลายรูปแบบ และแบบจำลองใดก็สามารถแสดงออกได้หลายวิธีในรหัสโปรแกรม สำหรับปัญหาที่เฉพาะเจาะจงแต่ละอันอาจมีทางเลือกมากกว่าหนึ่งวิธี จะเลือกวิธีไหนดี? การมีแบบจำลองที่ถูกต้องตามการวิเคราะห์นั้นไม่ได้หมายความว่าแบบจำลองนั้นสามารถแสดงออกเป็นรหัสโปรแกรมได้โดยตรง หรือบางครั้งการนำมันไปใช้งานอาจก่อให้เกิดความไม่เหมาะสมต่อหลักการออกแบบซอฟต์แวร์ ซึ่งไม่แนะนำให้ทำ สำคัญที่จะเลือกแบบจำลองที่สามารถแปลงเป็นรหัสโปรแกรมได้โดยง่ายและแม่นยำ คำถามพื้นฐานที่นี่คือ: เราจะเข้ามาออกแบบโค้ดจากแบบจำลองได้อย่างไร?

หนึ่งในเทคนิคการออกแบบที่แนะนำคือวิธีการออกแบบโมเดลการวิเคราะห์ที่เรียกว่า "analysis model" ซึ่งถือเป็นสิ่งที่แตกต่างจากการออกแบบโค้ดและมักจะทำโดยผู้ที่แตกต่างกัน โมเดลการวิเคราะห์นี้เป็นผลลัพธ์ของการวิเคราะห์โดเมนธุรกิจซึ่งมีผลให้เกิดโมเดลที่ไม่พิจารณาซอฟต์แวร์ที่ใช้สำหรับการนำไปสู่การปฏิสัมพันธ์ โมเดลเช่นนี้ถูกใช้เพื่อเข้าใจโดเมน มีการสร้างความรู้ระดับหนึ่งและโมเดลที่ได้รับการสร้างขึ้นอาจจะถูกต้องในเชิงวิเคราะห์ ซอฟต์แวร์ไม่ถูกพิจารณาในขั้นตอนนี้เพราะถือว่าเป็นปัจจัยที่สับสน เนื่องจากโมเดลนี้เหมาะสมกับนักพัฒนาที่ควรทำการออกแบบ โดยเนื่องจากโมเดลไม่ได้ถูกสร้างขึ้นโดยมีหลักการออกแบบอยู่แล้ว ดังนั้นมันอาจจะไม่ได้ใช้เพื่อวัตถุประสงค์นั้นได้ดี นักพัฒนาจะต้องปรับเปลี่ยนมันหรือสร้างออกแบบเป็นอย่างอื่น และไม่มีการจับคู่ระหว่างโมเดลและโค้ดอีกต่อไป ผลลัพธ์คือโมเดลการวิเคราะห์จะถูกทอดทิ้งเมื่อเริ่มต้นการเขียนโค้ด

หนึ่งในปัญหาหลักของวิธีนี้คือว่าวิเคราะห์ตัวเลขไม่สามารถทำนายบางจุดบกพร่องในโมเดลและความซับซ้อนของโดเมนได้ วิเคราะห์ตัวเลขอาจได้ลึกซึ้งเกินไปในบางส่วนของโมเดลและไม่ได้เกริ่นกับส่วนอื่นๆอย่างพอดี รายละเอียดที่สำคัญมากมักถูกค้นพบขณะกระบวนการออกแบบและการประยุกต์ใช้ โมเดลที่ถูกต้องตามโดเมนอาจมีปัญหาร้ายแรงเกี่ยวกับการเก็บออบเจกต์หรือพฤติกรรมการทำงานที่ไม่ยอมรับได้

นักพัฒนาจะถูกบังคับให้ตัดสินใจเองในบางเรื่อง และจะทำการเปลี่ยนแปลงการออกแบบเพื่อแก้ไขปัญหาจริงที่ไม่ได้ถูกพิจารณาเมื่อสร้างโมเดล พวกเขาจะสร้างการออกแบบที่หลุดออกจากโมเดล ทำให้โมเดลนั้นไม่เกี่ยวข้องมากยิ่งขึ้น

ถ้านักวิเคราะห์ทำงานแยกต่างหากกันไป พวกเขาจะสร้างโมเดลได้ในที่สุด แต่เมื่อโมเดลถูกส่งให้กับนักออกแบบ บางส่วนของความรู้เกี่ยวกับโดเมนและโมเดลของนักวิเคราะห์จะหายไป ในขณะที่โมเดลอาจถูกแสดงออกเป็นแผนภูมิและเขียนออกมา แต่โอกาสที่นักออกแบบจะเข้าใจความหมายของโมเดลทั้งหมดหรือความสัมพันธ์ระหว่างวัตถุบางอย่างหรือพฤติกรรมของพวกเขานั้นไม่สูง ยังมีรายละเอียดในโมเดลที่ไม่สามารถแสดงในแผนภูมิได้อย่างง่ายดาย และอาจไม่ได้ถูกนำเสนออย่างเต็มที่ในการเขียน นักพัฒนาจะมีความยากลำบากในการค้นหาวิธีการแก้ไขปัญหานี้ ในบางกรณีพวกเขาอาจจะสรุปสมมติฐานเกี่ยวกับพฤติกรรมที่ตั้งใจไว้ และมีโอกาสที่พวกเขาจะสร้างสมมติฐานที่ผิด ทำให้โปรแกรมทำงานผิดพลาดได้

นักวิเคราะห์มีการประชุมเป็นเอกสิทธิ์ของตนเองที่มีการพูดคุยเกี่ยวกับโดเมนและมีการแบ่งปันความรู้กันอย่างมากมาย พวกเขาสร้างโมเดลที่คาดหวังว่าจะมีข้อมูลทั้งหมดนั้นอยู่ในรูปแบบที่ย่อหน้าได้ และนักพัฒนาต้องยอมรับทั้งหมดนี้โดยการอ่านเอกสารที่ให้ไว้ให้พวกเขา จะเป็นการเพิ่มประสิทธิภาพมากขึ้นหากนักพัฒนาสามารถเข้าร่วมการประชุมของนักวิเคราะห์ได้ และมีการได้รับภาพรวมที่ชัดเจนและสมบูรณ์เกี่ยวกับโดเมนและโมเดลก่อนที่พวกเขาจะเริ่มออกแบบรหัส

วิธีการที่ดีกว่าคือการสัมพันธ์ระบบโดเมนและการออกแบบให้ใกล้ชิดกันมากขึ้น โมเดลควรถูกสร้างขึ้นโดยมีการพิจารณาถึงซอฟต์แวร์และการออกแบบด้วย นักพัฒนาควรถูกเอาไว้ในกระบวนการจัดโมเดลด้วย ความคิดหลักคือการเลือกโมเดลที่สามารถถูกแสดงอย่างเหมาะสมในซอฟต์แวร์ เพื่อให้กระบวนการออกแบบเป็นไปอย่างเรียบง่ายและขึ้นอยู่กับโมเดล เชื่อมโยงโค้ดอย่างเข้มงวดกับโมเดลพื้นฐานจะทำให้โค้ดมีความหมายและทำให้โมเดลเป็นสิ่งที่เกี่ยวข้อง

การให้ผู้พัฒนาเข้ามามีส่วนร่วมช่วยให้ได้รับคำติชมและคำแนะนำ ซึ่งทำให้มั่นใจได้ว่าโมเดลสามารถนำไปใช้งานในซอฟต์แวร์ได้จริง ถ้ามีข้อผิดพลาด ก็จะรู้ตั้งแต่ระยะต้นและสามารถแก้ไขปัญหาได้ง่ายๆ

ผู้เขียนโค้ดควรรู้จักโมเดลอย่างดีและรู้สึกต้องรับผิดชอบต่อความเป็นเชิงซื่อสัตย์ของมัน พวกเขาควรเข้าใจว่าการเปลี่ยนแปลงโค้ดแปลงผลต่อโมเดลด้วย ไม่งั้นพวกเขาจะต้องเรียกโค้ดใหม่ไปเรื่อยๆจนกว่าโค้ดจะไม่สามารถแสดงโมเดลต้นฉบับได้อีกต่อไป หากนักวิเคราะห์ถูกแยกออกจากกระบวนการดำเนินการ พวกเขาจะสูญเสียความเอาใจใส่เกี่ยวกับข้อจำกัดที่เกิดขึ้นจากการพัฒนา ผลที่เกิดขึ้นคือโมเดลที่ไม่เป็นประโยชน์

ผู้เชี่ยวชาญด้านเทคนิคใดๆที่มีส่วนร่วมในโมเดลจะต้องใช้เวลาเขียนโค้ด ไม่ว่าจะเป็นบทบาทหลักหรือรองในโครงการ ผู้รับผิดชอบการเปลี่ยนแปลงโค้ดจะต้องเรียนรู้การแสดงโมเดลผ่านโค้ด ทุกๆนักพัฒนาจะต้องมีส่วนร่วมในการพูดคุยเกี่ยวกับโมเดลในระดับใดๆ และต้องมีการติดต่อกับผู้เชี่ยวชาญในด้านที่เกี่ยวข้อง ผู้ที่มีส่วนร่วมในวิธีการต่างๆต้องมีส่วนร่วมโดยตรงกับผู้ที่เขียนโค้ดในการแลกเปลี่ยนไอเดียโมเดลผ่าน Ubiquitous Language อย่างไรก็ตาม

หากการออกแบบหรือส่วนสำคัญของการออกแบบไม่สอดคล้องกับโมเดลโดเมน โมเดลดังกล่าวจะมีค่าน้อยลง และความถูกต้องของซอฟต์แวร์นั้นไม่น่าเชื่อถือ ในเวลาเดียวกัน การทำแมปฟังก์ชั่นระหว่างโมเดลและการออกแบบที่ซับซ้อนยากต่อการเข้าใจและในการปฏิบัติไม่สามารถรักษาได้เมื่อการออกแบบเปลี่ยนแปลง เกิดการแบ่งแยกอันตรายระหว่างการวิเคราะห์และการออกแบบดังนั้นความเข้าใจที่ได้รับจากกิจกรรมแต่ละกิจกรรมจะไม่สามารถนำมาใช้กันได้

ออกแบบส่วนหนึ่งของระบบซอฟต์แวร์ให้สะท้อนโดเมนโมเดลอย่างเป็นตัวอย่างที่ชัดเจนเพื่อให้การแม่นยำในการจัดการข้อมูลของระบบดูง่ายขึ้น ตรวจสอบโมเดลอีกครั้งและปรับปรุงให้สามารถนำไปใช้งานได้อย่างสะดวกยิ่งขึ้นในซอฟต์แวร์ พยายามให้โมเดลสะท้อนความเข้าใจลึกซึ้งของโดเมนด้วย ต้องการโมเดลเดียวที่ใช้งานได้ดีทั้งสองประการ รวมถึงสนับสนุน Ubiquitous Language ที่ถูกต้องและเข้าใจง่ายด้วย

วาดภาพรวมของแบบจำลองที่ใช้ในการออกแบบและการมอบหมายภารกิจพื้นฐาน รหัสจะกลายเป็นการแสดงออกจากแบบจำลอง ดังนั้นการเปลี่ยนแปลงรหัสอาจเป็นการเปลี่ยนแปลงแบบจำลอง ผลกระทบของมันจะต้องกระจายไปยังกิจกรรมของโครงการที่เหลือด้วย ตามลำดับ

การผูกการนำไปใช้งานกับแบบจำลองอย่างเข้มงวด ต้องใช้เครื่องมือพัฒนาซอฟต์แวร์และภาษาที่รองรับแนวคิดการจำลอง เช่นโปรแกรมเชิงวัตถุ

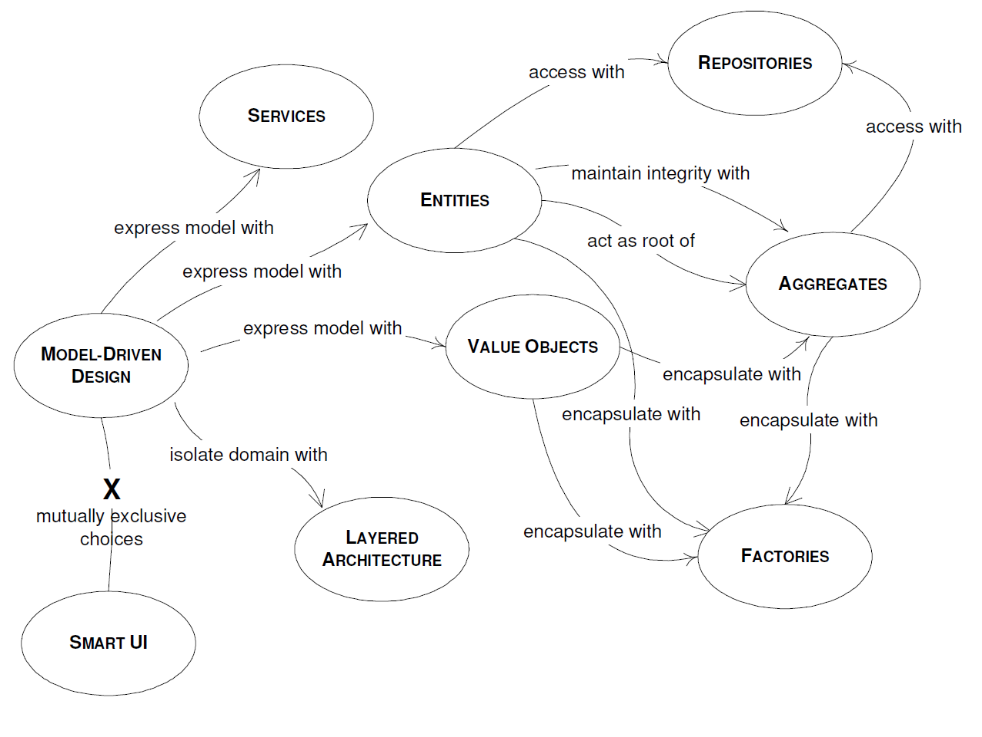
โปรแกรมเชิงวัตถุเหมาะสมสำหรับการนำมาใช้ในการสร้างโมเดลเนื่องจากทั้งสองอยู่ในแนวคิดเดียวกัน โปรแกรมเชิงวัตถุจะให้คลาสของออบเจกต์และความสัมพันธ์ระหว่างคลาส อินสแตนซ์ของออบเจกต์ และการส่งข้อความระหว่างพวกเขา ภาษาโปรแกรมเชิงวัตถุทำให้เป็นไปได้สำหรับสร้างการมอบหมายโดยตรงระหว่างวัตถุของโมเดลกับความสัมพันธ์ของพวกเขา และหน้าที่การเขียนโปรแกรมของพวกเขา

ภาษาแบบ procedural มีการสนับสนุนการออกแบบโมเดลแบบจำลองเพียงน้อยเท่านั้น ภาษาเหล่านี้ไม่มีโครงสร้างที่จำเป็นต้องใช้เพื่อการประมวลผลส่วนประกอบสำคัญของโมเดล บางคนกล่าวว่า OOP สามารถทำได้ด้วยภาษา procedural เช่น C และในความเป็นจริง บางฟังก์ชันสามารถถูกทำซ้ำได้โดยใช้วิธีนั้น วัตถุสามารถจำลองเป็นโครงสร้างข้อมูลได้ โครงสร้างเหล่านี้ไม่มีพฤติกรรมของวัตถุและจำเป็นต้องเพิ่มเป็นฟังก์ชันแยกต่างหาก ความหมายของข้อมูลเหล่านั้นมีอยู่เฉพาะในจิตของนักพัฒนาเท่านั้นเนื่องจากรหัสตนเองไม่ชัดเจน โปรแกรมที่เขียนด้วยภาษา procedural มักถูกมองเป็นชุดของฟังก์ชันที่เรียกใช้ซึ่งกันและกันเพื่อสร้างผลลัพธ์บางอย่าง โปรแกรมดังกล่าวไม่สามารถห่วงโอนความหมายได้อย่างง่ายดาย ซึ่งทำให้การแมประหว่างโดเมนและโค้ดยากต่อการตระหนักขึ้น

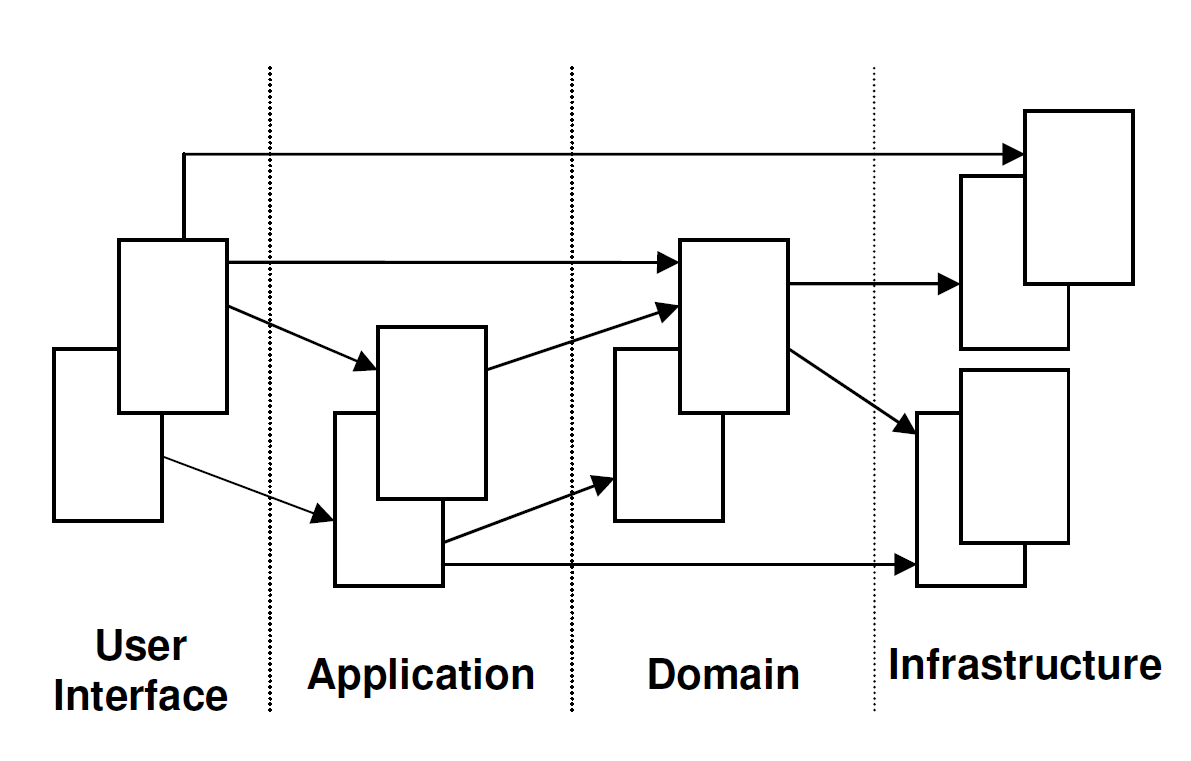
บางโดเมนที่เฉพาะเจาะจง เช่น คณิตศาสตร์ สามารถจำลองและนำไปใช้งานได้อย่างง่ายดายโดยใช้โปรแกรมแบบ procedural programming เนื่องจากหลายทฤษฎีทางคณิตศาสตร์สามารถแก้ไขได้โดยการเรียกใช้ฟังก์ชันและโครงสร้างข้อมูลเพราะมันเกี่ยวกับการคำนวณ แต่โดเมนที่ซับซ้อนมากขึ้นไม่ใช่แค่ชุดความคิดแบบนามธรรมเกี่ยวกับการคำนวณ และไม่สามารถลดลงเป็นชุดอัลกอริทึมได้ ดังนั้นภาษาโปรแกรมแบบ procedural programming ไม่สามารถตอบสนองความต้องการในการออกแบบโมเดลได้อย่างเหมาะสม เพื่อเหตุนี้ การเขียนโปรแกรมแบบ procedural programming ไม่แนะนำสำหรับการออกแบบโมเดลที่มุ่งเน้นไปที่การจำลองโดยเฉพาะ

**"Building Blocks" ในการออกแบบโมเดลที่ใช้แนวคิด Model-Driven Design**

ส่วนต่อไปของบทนี้จะนำเสนอรูปแบบสำคัญที่สุดที่จะใช้ในการออกแบบโดยใช้โมเดลเป็นฐาน จุดประสงค์ของรูปแบบเหล่านี้คือเพื่อนำเสนอบางส่วนที่สำคัญของการจัดแบ่งวัตถุและการออกแบบซอฟต์แวร์จากมุมมองของการออกแบบด้วยดีไดร์เว่น แผนภาพต่อไปนี้เป็นแผนที่ของรูปแบบที่นำเสนอและความสัมพันธ์ระหว่างพวกมัน



**Layered Architecture**



เมื่อเราสร้างแอปพลิเคชันซอฟต์แวร์ ส่วนใหญ่ของแอปพลิเคชันไม่ได้เกี่ยวข้องโดยตรงกับโดเมน แต่เป็นส่วนหนึ่งของโครงสร้างพื้นฐานหรือบริการต่างๆ ที่รองรับซอฟต์แวร์เอง มันเป็นเรื่องที่เป็นไปได้และถูกต้องที่ส่วนของโดเมนในแอปพลิเคชันจะเล็กมากเมื่อเปรียบเทียบกับส่วนที่เหลือ เนื่องจากแอปพลิเคชันทั่วไปประกอบด้วยโค้ดจำนวนมากที่เกี่ยวข้องกับการเข้าถึงฐานข้อมูล การเข้าถึงไฟล์หรือเครือข่าย อินเตอร์เฟสผู้ใช้ เป็นต้น

ในโปรแกรมที่เขียนโดยใช้การเขียนเชิงวัตถุ (Object-Oriented) บางครั้งจะมีการเขียนโค้ดสนับสนุนเช่น UI, ฐานข้อมูล และอื่น ๆ ถูกเขียนโดยตรงในอ็อบเจ็กต์ทางธุรกิจ โลจิกธุรกิจเพิ่มเติมจะถูกฝังอยู่ในพฤติกรรมของ UI widgets และสคริปต์ฐานข้อมูล ส่วนใหญ่เกิดขึ้นเพราะเป็นวิธีที่ง่ายที่สุดในการทำให้สิ่งต่าง ๆ ทำงานได้เร็วขึ้น

อย่างไรก็ตาม เมื่อรหัสที่เกี่ยวข้องกับโดเมนถูกผสมผสานกับชั้นอื่น ๆ จะเป็นสิ่งที่ยากมากที่จะมองเห็นและคิดเกี่ยวกับมัน การเปลี่ยนแปลงที่เป็นพื้นที่ของ UI อาจจะเปลี่ยนแปลงตรรกะทางธุรกิจ การเปลี่ยนกฎธุรกิจอาจจะต้องการการติดตามรหัส UI, รหัสฐานข้อมูลหรือองค์ประกอบโปรแกรมอื่น ๆ อย่างละเอียดอ่อน การสร้างวัตถุที่มีความสอดคล้องและโดยใช้โมเดลกลางกลายเป็นเรื่องไม่ทำได้ เทสอัตโนมัติจะลำบาก ด้วยเทคโนโลยีและตรรกะที่เกี่ยวข้องในกิจกรรมแต่ละอย่าง โปรแกรมต้องถูกเก็บไว้ในรูปแบบที่เข้าใจง่ายมาก ไม่งั้นจะเป็นไปไม่ได้ที่จะเข้าใจมันได้

ดังนั้น การแบ่งโปรแกรมที่ซับซ้อนออกเป็น LAYER จะเป็นสิ่งสำคัญ ในแต่ละ LAYER ควรพัฒนาการออกแบบที่สม่ำเสมอและขึ้นอยู่กับ LAYER ด้านล่างเท่านั้น โดยปฏิบัติตามแนวแบบสถาปัตยกรรมมาตรฐานเพื่อให้มีการผูกพันกันระหว่าง LAYER ด้านบน ซึ่งทำให้สามารถเน้นการเขียนโค้ดที่เกี่ยวข้องกับโมเดลด้านด้านในเพียงชั้นเดียวและแยกออกจากโค้ดของอินเตอร์เฟซผู้ใช้งานและโค้ดของโครงสร้างพื้นฐานได้ การแยกออบเจกต์ของโดเมนจากความรับผิดชอบในการแสดงผล เก็บรักษาตนเอง จัดการงานของแอพลิเคชั่นและอื่น ๆ สามารถให้โมเดลโดเมนโต้ตอบไปยังการแสดงโมเดลโดยไม่มีภาระของงานอื่น ๆ ซึ่งทำให้โมเดลได้รับการพัฒนาให้มีความหลากหลายมากพอและชัดเจนพอที่จะรวบรวมความรู้ด้านธุรกิจสำคัญได้และนำมาใช้งานได้

โดยทั่วไปแล้ว วิธีการออกแบบแนวคิดที่มุ่งเน้นไปที่โดเมน (domain-driven design) จะประกอบด้วยชั้นความคิดทางแบบ (conceptual layers) 4 ชั้น:

**User Interface (UI) (Presentation Layer)** คือ ส่วนที่รับผิดชอบในการนำเสนอข้อมูลให้แก่ผู้ใช้และแปลงคำสั่งของผู้ใช้ให้เข้าใจได้

**Application Layer** นี่เป็นชั้นบาง ๆ ที่ประสานกิจกรรมของแอปพลิเคชัน มันไม่มีตัวตนธุรกิจ (business logic) และไม่เก็บสถานะของวัตถุธุรกิจ แต่มันสามารถเก็บสถานะของความคืบหน้าของงานในแอปพลิเคชันได้

**Domain Layer** ชั้นนี้ประกอบด้วยข้อมูลเกี่ยวกับโดเมน และเป็นส่วนสำคัญของซอฟต์แวร์ธุรกิจ ภายในชั้นนี้จะเก็บข้อมูลสถานะของออบเจ็กต์ธุรกิจ การเก็บข้อมูลออบเจ็กต์ธุรกิจและสถานะของออบเจ็กต์นั้นอาจถูกมอบหมายให้กับชั้นพื้นฐาน (Infrastructure layer)

**Infrastructure Layer** ชั้นนี้เป็นห้องสมุดสนับสนุนสำหรับชั้นอื่น ๆ ทั้งหมด เป็นตัวกลางในการสื่อสารระหว่างชั้น สร้างความต่อเนื่องในการเก็บข้อมูลสำหรับวัตถุธุรกิจ รวมถึงมีห้องสมุดสนับสนุนสำหรับชั้นอินเตอร์เฟสอื่น ๆ อีกด้วย

การแบ่งแอปพลิเคชันเป็นชั้นๆ และตั้งกฎการโต้ตอบระหว่างชั้นเป็นสิ่งสำคัญ หากโค้ดไม่ได้ถูกแยกออกเป็นชั้นๆ โค้ดจะกลายเป็นหวาดกลัวที่จะเก็บรักษาการเปลี่ยนแปลงได้ การเปลี่ยนแปลงง่ายๆ ในส่วนหนึ่งของโค้ดอาจมีผลที่ไม่คาดคิดและไม่ต้องการในส่วนอื่นๆ ด้วย ชั้นของโดเมนควรให้ความสำคัญกับปัญหาสำคัญในโดเมน ไม่ควรมีการเข้าไปแทรกแซงกับกิจกรรมในพื้นฐานโครงสร้างระบบ เนื่องจาก UI ไม่ควรเชื่อมต่ออย่างเข้มงวดกับตัวตนของตัวและงานที่ต้องการตั้งแต่แรกก็อยู่ในชั้นพื้นฐาน การมีชั้นแอพพลิเคชั่นเป็นสิ่งจำเป็นในกรณีหลายๆ ครั้ง จำเป็นต้องมีผู้จัดการที่มีความรู้ความเข้าใจในตัวตนธุรกิจที่จะควบคุมและประสานกิจกรรมโดยรวมของแอพพลิเคชั่น

ตัวอย่างเช่น การปฏิสัมพันธ์ปกติของแอปพลิเคชัน โดเมน และโครงสร้างพื้นฐานอาจมีลักษณะดังนี้ ผู้ใช้ต้องการจองเส้นทางเครื่องบิน และขอให้บริการแอปพลิเคชันในชั้นของแอปพลิเคชันทำเช่นนั้น ชั้นแอปพลิเคชันดึงออบเจกต์โดเมนที่เกี่ยวข้องจากโครงสร้างพื้นฐานและเรียกใช้เมธอดที่เกี่ยวข้องกับออบเจกต์เหล่านั้น เช่นเพื่อตรวจสอบขอบเขตการรักษาความปลอดภัยสำหรับเส้นทางบินที่จองไว้แล้ว เมื่อออบเจกต์โดเมนทำการตรวจสอบทั้งหมดและอัปเดตสถานะเป็น "ตัดสินใจแล้ว" บริการแอปพลิเคชันจะเก็บรักษาออบเจกต์ในโครงสร้างพื้นฐาน

**Entities**

มีหมวดหมู่ของวัตถุที่ดูเหมือนจะมีเอกลักษณ์ที่คงเดิมตลอดสถานะของซอฟต์แวร์ สำหรับวัตถุเหล่านี้ ไม่ใช่แค่คุณลักษณะที่สำคัญ แต่เป็นเส้นด้ายของความต่อเนื่องและเอกลักษณ์ที่ขยายตัวไปในชีวิตของระบบและสามารถขยายตัวไปเกินไปได้ วัตถุเหล่านี้เรียกว่า "Entity"

ภาษา OOP เก็บอินสแตนซ์ของออบเจ็กต์ในหน่วยความจำและมีการเชื่อมโยงอ้างอิงหรือที่อยู่ของหน่วยความจำสำหรับแต่ละออบเจ็กต์ การอ้างอิงนี้เป็นเอกลักษณ์สำหรับแต่ละออบเจ็กต์ในช่วงเวลาที่กำหนดไว้ แต่ไม่มีการรับประกันว่ามันจะเป็นเช่นนั้นไปตลอดเวลาอย่างไม่จำกัด ซึ่งกลับกันก็เป็นจริง ออบเจ็กต์ถูกย้ายเข้าและออกจากหน่วยความจำอยู่เสมอ ถูกซีเรียลไลซ์และส่งผ่านเครือข่ายแล้วสร้างขึ้นใหม่ที่อีกด้านหนึ่ง หรือถูกทำลายออกจากหน่วยความจำ

อ้างอิงนี้เป็นตัวบอกตัวตนของสภาพแวดล้อมที่โปรแกรมทำงานอยู่ แต่มันไม่ใช่ตัวตนที่เรากำลังพูดถึงอยู่ หากมีคลาสที่เก็บข้อมูลสภาพอากาศ เช่น อุณหภูมิ มันเป็นไปได้อย่างมากที่จะมีอินสแตนซ์ของคลาสที่แตกต่างกันสองตัวที่มีค่าเดียวกัน วัตถุเหล่านั้นเท่ากันและสามารถสลับกันได้ แต่พวกเขามีอ้างอิงที่แตกต่างกัน พวกเขาไม่ใช่สิ่งที่มีออกแบบมาเพื่อเป็นตัวตนของมันเอง

ถ้าเราจะนำแนวคิดของบุคคลมาใช้ในโปรแกรมซอฟต์แวร์ เราจะสร้างคลาส Person พร้อมกับลักษณะบางอย่าง เช่น ชื่อ วันเกิด สถานที่เกิด เป็นต้น แต่ลักษณะใดบ้างที่เป็นเอกลักษณ์ของบุคคล? ชื่อไม่สามารถเป็นเอกลักษณ์ได้ เพราะอาจมีผู้คนที่มีชื่อเดียวกันหลายคน ถ้าเราใช้ชื่อเพียงอย่างเดียวเราจะไม่สามารถแยกแยะระหว่างบุคคลที่มีชื่อเดียวกันได้ เราไม่สามารถใช้วันเกิดได้เช่นกัน เพราะมีผู้คนที่เกิดในวันเดียวกันหลายคนเช่นกัน สิ่งเดียวกันก็เกี่ยวกับสถานที่เกิดด้วย เราต้องแยกวัตถุออกจากวัตถุอื่นๆ แม้ว่าวัตถุเหล่านั้นจะมีลักษณะเดียวกัน การแยกแยะผิดพลาดอาจส่งผลให้เกิดความเสียหายของข้อมูลได้

พิจารณาระบบบัญชีธนาคาร แต่ละบัญชีมีหมายเลขของตัวเอง บัญชีสามารถระบุได้แม่นยำด้วยหมายเลขของมัน หมายเลขนี้จะไม่เปลี่ยนแปลงตลอดการใช้งานของระบบและรับประกันความต่อเนื่อง หมายเลขบัญชีสามารถอยู่ในรูปของวัตถุในหน่วยความจำได้ หรือสามารถถูกลบออกจากหน่วยความจำและส่งไปยังฐานข้อมูลได้ และยังสามารถถูกเก็บเป็นเอกสารเมื่อบัญชีถูกปิดใช้งาน แต่หมายเลขบัญชีจะยังคงอยู่ที่ใดที่หนึ่งตลอดเวลาที่มีผู้สนใจรักษาอยู่ ไม่ว่ามันจะเป็นรูปแบบใดก็ตาม หมายเลขนี้จะยังคงเดิมไม่เปลี่ยนแปลง

มีวิธีการสร้างเอกลักษณ์ที่ไม่ซ้ำกันสำหรับแต่ละวัตถุที่แตกต่างกันได้หลายวิธี ตัวอย่างเช่น ID สามารถสร้างขึ้นโดยโมดูลอัตโนมัติและใช้ภายในซอฟต์แวร์โดยไม่ต้องแสดงให้ผู้ใช้เห็น หรืออาจเป็น primary key ในตารางฐานข้อมูลที่มั่นใจว่าจะไม่ซ้ำกันในฐานข้อมูล ทุกครั้งที่วัตถุถูกดึงขึ้นมาจากฐานข้อมูล ID ของมันจะถูกดึงขึ้นมาและสร้างในหน่วยความจำใหม่ อีกวิธีหนึ่งคือให้ผู้ใช้สร้าง ID เอง เช่นเดียวกับรหัสที่เกี่ยวข้องกับสนามบิน แต่ละสนามบินมี ID สตริงที่ไม่ซ้ำกันซึ่งได้รับการยอมรับในระดับสากลและใช้โดยสำนักงานท่องเที่ยวทั่วโลกเพื่อระบุสนามบินในกำหนดการท่องเที่ยวของพวกเขา วิธีการแก้ปัญหาอื่นคือการใช้คุณสมบัติของวัตถุเพื่อสร้าง ID และเมื่อมันไม่เพียงพอ สามารถเพิ่มคุณสมบัติอื่นเพื่อช่วยระบุวัตถุที่เกี่ยวข้องได้อีกด้วย

เมื่อวัตถุถูกแยกแยะด้วยเอกลักษณ์ของมัน ไม่ใช่คุณสมบัติของมัน ให้ทำเอกลักษณ์นี้เป็นสิ่งหลักในการกำหนดคุณลักษณะของวัตถุในโมเดล ให้นิยามคลาสอย่างเรียบง่ายและมุ่งเน้นต่อการต่อเนื่องของวัตถุและเอกลักษณ์ กำหนดวิธีการแยกแยะแต่ละวัตถุโดยไม่สนใจรูปแบบหรือประวัติศาสตร์ของมัน ต้องระมัดระวังต่อความต้องการที่ต้องการจับคู่วัตถุตามคุณสมบัติ กำหนดการดำเนินการที่สามารถรับประกันผลลัพธ์ที่ไม่ซ้ำกันสำหรับแต่ละวัตถุโดยการเชื่อมโยงสัญลักษณ์ที่รับประกันความเป็นเอกลักษณ์ วิธีการระบุตัวนี้อาจมาจากภายนอกหรืออาจเป็นตัวระบุอย่างสุ่มที่ถูกสร้างขึ้นโดยระบบหรือสำหรับระบบเท่านั้น แต่มันต้องสอดคล้องกับการแยกแยะตัวตนในโมเดล โมเดลจะต้องกำหนดความหมายของสิ่งเดียวกัน

สิ่งกิจกรรม (Entities) เป็นวัตถุสำคัญของโมเดลโดเมน และควรถูกพิจารณาตั้งแต่เริ่มต้นของกระบวนการจำลองโมเดล นอกจากนี้ยังสำคัญที่จะกำหนดว่าวัตถุใดที่จะต้องเป็นสิ่งกิจกรรมหรือไม่ ซึ่งจะถูกพูดถึงในแบบแผนต่อไป

**Value Objects**

เราได้พูดถึง entities และความสำคัญของการรู้จัก entities ในช่วงการออกแบบระบบ Entities คือวัตถุที่จำเป็นในโมเดลของโดเมน เราควรจะทำให้ทุกออบเจ็กต์เป็น entities หรือไม่? และทุกออบเจ็กต์จำเป็นต้องมี identity หรือไม่?

เราอาจจะถูกผลักดันให้ทำให้ทุกๆ วัตถุกลายเป็น entity ได้ โดย entity สามารถติดตามได้ แต่การติดตามและสร้าง identity นั้นมีค่าใช้จ่าย ต้องมีการตรวจสอบให้แน่ใจว่าแต่ละ instance มี identity ที่ไม่ซ้ำซ้อน และการติดตาม identity ไม่ใช่เรื่องง่าย ต้องใช้การคิดอย่างระมัดระวังเพื่อตัดสินใจว่าอะไรทำให้มี identity และหากตัดสินใจผิด อาจทำให้เกิดวัตถุที่มี identity เดียวกัน ซึ่งไม่เป็นที่ต้องการ นอกจากนี้ยังมีผลกระทบทางประสิทธิภาพในการทำให้ทุกๆ วัตถุกลายเป็น entity จำเป็นต้องมี instance เพื่อแทนแต่ละวัตถุ ถ้า Customer เป็น entity object แล้ว ต้องมี instance เพียงหนึ่งตัวสำหรับลูกค้าแต่ละคน และไม่สามารถนำมาใช้ใหม่สำหรับการดำเนินการบัญชีที่เกี่ยวข้องกับลูกค้าอื่นได้ ผลลัพธ์คือต้องสร้าง instance เพื่อแทนแต่ละลูกค้า ซึ่งอาจทำให้ประสิทธิภาพระบบลดลงเมื่อต้องจัดการกับพัน instance

พิจารณาแอปพลิเคชันวาดรูปภาพ ผู้ใช้จะได้รับแคนวาสและเขาสามารถวาดจุดและเส้นใดๆ ที่มีความหนาแน่นสไตล์และสีอะไรก็ได้ จึงเป็นประโยชน์ที่จะสร้างคลาสออบเจกต์ชื่อ "Point" และโปรแกรมสามารถสร้างอินสแตนซ์ของคลาสนี้สำหรับแต่ละจุดบนแคนวาส จุดเช่นนั้นจะมีสองแอตทริบิวต์ที่เกี่ยวข้องกับพิกัดหน้าจอหรือแคนวาส จำเป็นต้องพิจารณาว่าแต่ละจุดมีเอกลักษณ์หรือไม่? มีความต่อเนื่องหรือไม่? ดูเหมือนว่าสิ่งเดียวที่สำคัญสำหรับออบเจกต์แบบนี้คือพิกัดของมันเท่านั้น

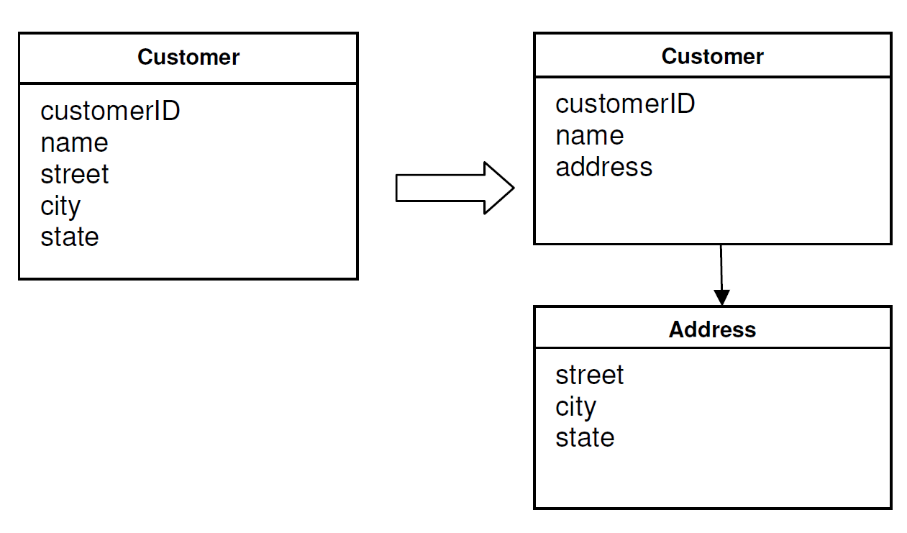
มีกรณีบางครั้งที่เราต้องการจำกัดบางแอตทริบิวต์ขององค์ประกอบโดเมน โดยเราไม่สนใจว่าเป็นวัตถุอะไร แต่เราสนใจแค่แอตทริบิวต์ที่มีอยู่ เวลาที่เราใช้วัตถุเพื่ออธิบายด้านต่างๆ ของโดเมน และวัตถุนั้นไม่มีตัวตน จะเรียกว่า Value Object.

จำเป็นต้องแยกแยะระหว่าง Entity Objects และ Value Objects โดยไม่จำเป็นต้องทำให้วัตถุทั้งหมดเป็น entity เพื่อให้เหมือนกันทั้งหมด เป็นที่แนะนำว่าควรเลือกเฉพาะวัตถุที่เป็น entity ตามนิยามของ entity และทำให้วัตถุที่เหลือเป็น Value Objects (ในส่วนต่อไปเราจะนำเสนอประเภทอื่นของวัตถุ แต่เราจะสมมติว่าเรามีเพียง entity objects และ value objects เท่านั้นในขณะนี้) นี้จะทำให้การออกแบบง่ายขึ้น และจะมีผลกระทบบางอย่างที่ดีขึ้นอีกด้วย

ไม่มีเอกลักษณ์เป็นสิ่งที่สามารถสร้างและทิ้งได้อย่างง่ายดาย ไม่มีใครสนใจในการสร้างเอกลักษณ์และเครื่องมือเก็บขยะจะดูแลวัตถุเมื่อไม่มีการอ้างอิงถึงวัตถุนี้โดยวัตถุอื่นใดอีกต่อไป นี่ทำให้การออกแบบเป็นไปได้ง่ายมาก

การแนะนำอย่างมากว่า Value Objects ควรเป็น Immutable (ไม่สามารถเปลี่ยนแปลงได้) พวกเขาถูกสร้างขึ้นด้วย constructor และไม่เคยถูกแก้ไขในช่วงเวลาที่พวกเขามีอยู่ หากคุณต้องการค่าต่างๆ สำหรับ Object นั้นๆ คุณเพียงแค่สร้าง Object ใหม่ ซึ่งมีผลลัพธ์ที่สำคัญสำหรับการออกแบบ โดยเนื่องจากพวกเขาเป็น Immutable และไม่มี Identity การใช้งาน Value Objects สามารถแชร์ได้ ซึ่งเป็นสิ่งที่สำคัญสำหรับการออกแบบบางแบบ Immutable Objects สามารถแชร์ได้โดยมีผลกระทบทางประสิทธิภาพที่สำคัญ และมีความคงสภาพของข้อมูล (data integrity) พิจารณาถึงความหมายของการแชร์ Object ที่ไม่เป็น Immutable ระบบการจองเที่ยวบินอาจสร้าง Object สำหรับแต่ละเที่ยวบิน หนึ่งใน attributes อาจเป็น flight code ลูกค้าคนหนึ่งจองเที่ยวบินสำหรับสถานที่ปลายทางบางอย่าง ลูกค้าคนอื่นต้องการจองเที่ยวบินเดียวกัน ระบบเลือกที่จะ reuse object ที่ถือ flight code เดียวกัน เพราะเป็นเที่ยวบินเดียวกัน ในขณะเดียวกันลูกค้าเปลี่ยนใจและเลือกเที่ยวบินอื่น ระบบเปลี่ยน flight code เนื่องจากไม่เป็น Immutable ผลที่เกิดขึ้นคือ flight code ของลูกค้าคนแรกเปลี่ยนแปลงด้วย

หนึ่งกฎทองคำคือ: ถ้า Value Objects สามารถแบ่งปันได้ จะต้องไม่สามารถเปลี่ยนแปลงค่าได้ (immutable) ควรเก็บ Value Objects ให้บางและเรียบง่าย เมื่อต้องการ Value Objects โดยผู้อื่น สามารถส่งผ่านค่าได้อย่างง่ายดาย หรือสร้างสำเนาและให้ได้ การสร้างสำเนาของ Value Objects เป็นเรื่องง่ายและมักไม่มีผลกระทบใดๆ หากไม่มี Identity คุณสามารถสร้างสำเนาได้เท่าที่ต้องการและลบทั้งหมดเมื่อจำเป็น



Value Objects สามารถมี Value Objects อื่น ๆ และอาจมีการอ้างอิง Entity เข้าไปด้วยได้ แม้ว่า Value Objects จะถูกใช้เพื่อใช้ในการเก็บ attribute ของ domain object อย่างเดียว แต่นั้นไม่ได้หมายความว่าจะต้องมีรายการ attribute ยาวนานทั้งหมดอยู่ใน Value Objects เดียว สามารถจัดกลุ่ม attribute ใน Object ต่าง ๆ ได้ Attribute ที่ถูกเลือกมาเพื่อเป็น Value Object ควรเป็นส่วนประกอบที่เกี่ยวข้องกันอย่างสมบูรณ์ ลูกค้าสัมพันธ์กับชื่อ ถนน เมือง และรัฐ ดังนั้นจึงดีกว่าจะเก็บข้อมูลที่อยู่ใน Object ต่าง ๆ แยกต่างหาก และ Object ของลูกค้าจะเก็บการอ้างอิงไปยัง Object เหล่านั้น ถนน เมือง และรัฐควรจะมี Object ของตัวเอง เช่น Address เพราะว่าพวกเขาเป็นส่วนประกอบที่เกี่ยวข้องกันแน่นอน ไม่ใช่ attribute ที่แยกออกจากกันของลูกค้า ตามแสดงในแผนภาพด้านล่าง

**Services**

เมื่อเราวิเคราะห์โดเมนและพยายามจะกำหนดวัตถุหลักที่ประกอบด้วยโมเดล เราพบว่ามีบางด้านของโดเมนที่ไม่สามารถแมปไปยังวัตถุได้ง่าย ๆ วัตถุทั่วไปจะถูกพิจารณาว่ามีแอตทริบิวต์ภายในซึ่งจัดการโดยวัตถุและแสดงพฤติกรรม ตอนที่เราพัฒนาภาษาแพร่หลาย แนวคิดสำคัญของโดเมนจะถูกนำเสนอในภาษา และคำนามของภาษาจะแมปไปยังวัตถุได้ง่าย คำกริยาของภาษาที่เกี่ยวข้องกับนามที่เหมือนกันกลายเป็นส่วนหนึ่งของพฤติกรรมของวัตถุเหล่านั้น แต่มีบางการกระทำในโดเมนบางอย่าง บางคำกริยาที่ดูเหมือนว่าไม่เหมาะสมกับวัตถุใด ๆ พวกเขาแทนพฤติกรรมที่สำคัญของโดเมนดังนั้นไม่สามารถละเว้นหรือรวมไปในบาง Entity หรือ Value Objects ได้เพียงแค่เพิ่มพฤติกรรมเช่นนี้ในออบเจกต์ก็จะทำให้ออบเจกต์เสียหาย ทำให้ออบเจกต์เป็นฟังก์ชั่นที่ไม่เกี่ยวข้องกัน แต่อย่างไรก็ตาม ในการใช้ภาษาเชิงวัตถุเราต้องใช้ออบเจกต์สำหรับวัตถุประสงค์นี้ เราไม่สามารถมีฟังก์ชันแยกต่างหากได้เพียงอย่างเดียว มันต้องถูกผูกกับออบเจกต์ใดออบเจกต์หนึ่ง บ่อยครั้งพฤติกรรมประเภทนี้ทำงานข้ามออบเจกต์หลายออบเจกต์บางครั้งอาจเป็นออบเจกต์คลาสที่แตกต่างกัน ตัวอย่างเช่นการโอนเงินจากบัญชีหนึ่งไปยังอีกบัญชีหนึ่ง ฟังก์ชั่นดังกล่าวควรจะอยู่ในบัญชีส่งหรือบัญชีผู้รับ? มันดูเหมือนว่าไม่เหมาะสมกับทั้งสองบัญชี

เมื่อพฤติกรรมเช่นนี้ได้รับการรับรู้ในโดเมน วิธีการที่ดีที่สุดคือการประกาศเป็นบริการ (Service) วัตถุดังกล่าวไม่มีสถานะภายในและจุดประสงค์ของมันคือเพียงการให้ความสามารถสำหรับโดเมน การช่วยเหลือที่ให้โดยบริการอาจเป็นสิ่งที่สำคัญมาก และบริการสามารถรวมความสัมพันธ์ของความสามารถที่ให้บริการแก่ Entity และ Value Object ได้ การประกาศ Service โดยชัดเจนเป็นการสร้างการแยกแยะที่ชัดเจนในโดเมน มันแยกแนวคิดออกมา การนำฟังก์ชันเช่นนี้มาประกอบกับ Entity หรือ Value Object จะทำให้สับสนว่าวัตถุเหล่านั้นแสดงอะไรขึ้นอยู่กับอะไร

บริการเป็นอินเทอร์เฟซที่ให้การดำเนินการ บริการเป็นสิ่งที่พบได้บ่อยในกรอบเทคนิค แต่ก็สามารถใช้ในชั้นโดเมนได้อีกด้วย บริการไม่ได้เกี่ยวกับวัตถุที่ทำบริการ แต่เกี่ยวข้องกับวัตถุที่การดำเนินการถูกดำเนินการ ต่อ/ บน ด้วยวิธีนี้บริการมักจะเป็นจุดเชื่อมต่อสำหรับหลาย ๆ วัตถุ นี่เป็นหนึ่งในเหตุผลที่พฤติกรรมที่เป็นธรรมชาติของบริการไม่ควรรวมอยู่ในวัตถุโดเมน หากความสามารถดังกล่าวถูกรวมอยู่ในวัตถุโดเมน จะสร้างเครือข่ายแบบกองทัพของความสัมพันธ์ระหว่างวัตถุและวัตถุที่ได้รับผลประโยชน์จากการดำเนินการ ความเชื่อมโยงที่มีความสูงระหว่างวัตถุหลาย ๆ อันเป็นสัญญาณของการออกแบบที่แย่เพราะทำให้รหัสยากต่อการอ่านและเข้าใจ และสิ่งที่สำคัญกว่าคือ มันทำให้เปลี่ยนแปลงได้ยาก

บริการไม่ควรแทนการดำเนินการที่สัมพันธ์กับวัตถุดิบของโดเมน ไม่ควรสร้างบริการสำหรับทุกการดำเนินการที่จำเป็น แต่เมื่อการดำเนินการดังกล่าวเด่นชัดเป็นแนวคิดสำคัญในโดเมน ต้องสร้างบริการสำหรับมัน มีลักษณะของบริการทั้งสามอย่างดังนี้:

1. การดำเนินการที่ดำเนินการโดยบริการอ้างอิงถึงแนวความคิดของโดเมนซึ่งไม่สัมพันธ์กับ Entity หรือ Value Object อย่างเป็นธรรมชาติ

2. การดำเนินการที่ดำเนินการอ้างถึงอ็อบเจกต์อื่นในโดเมน

3. การดำเนินการเป็น Stateless (ไม่มีสถานะ)

เมื่อมีกระบวนการหรือการเปลี่ยนแปลงที่สำคัญในโดเมนที่ไม่ใช่ความรับผิดชอบธรรมชาติของ Entity หรือ Value Object ให้เพิ่ม operation ลงในโมเดลเป็น standalone interface ที่ประกาศเป็น Service กำหนด interface ในเชิงภาษาของโมเดลและตรวจสอบให้แน่ใจว่าชื่อ operation เป็นส่วนหนึ่งของ Ubiquitous Language ทำให้ Service เป็น stateless ด้วย

ในขณะที่ใช้บริการ สิ่งที่สำคัญคือการเก็บชั้นด้านโดเมน (domain layer) ไว้ในสภาพแวดล้อมที่ไม่มีการเชื่อมต่อกับชั้นอื่น ๆ การสับสนอาจเกิดขึ้นระหว่างการใช้บริการที่เป็นส่วนของชั้นโดเมนและบริการที่เป็นส่วนของโครงสร้างพื้นฐาน (infrastructure) อาจมีบริการในชั้นแอปพลิเคชัน (application layer) ที่เพิ่มระดับความซับซ้อน เหล่าบริการเหล่านั้นยิ่งยากขึ้นในการแยกจากคู่หูของพวกเขาที่อยู่ในชั้นโดเมน ในขณะที่ทำงานกับโมเดลและในช่วงการออกแบบ เราต้องตรวจสอบให้แน่ใจว่าชั้นโดเมนยังคงเป็นตัวกระจายตัวจากชั้นอื่น ๆ

บริการแอปพลิเคชันและโดเมนโดยปกติถูกสร้างขึ้นบน Entities และ Values ของโดเมนเพื่อให้บริการฟังก์ชันที่เกี่ยวข้องกับวัตถุเหล่านั้นโดยตรง การตัดสินใจว่า Service จะอยู่ในชั้นใดเป็นเรื่องยาก หากการดำเนินการที่ดำเนินการในแนวคิดของชั้นแอปพลิเคชัน แล้ว Service ควรอยู่ที่นั่น แต่หากการดำเนินการเกี่ยวกับวัตถุโดเมนและเกี่ยวข้องอย่างเครื่องคงกับโดเมนในการตอบสนองความต้องการ จะต้องอยู่ในชั้นโดเมน

พิจารณาตัวอย่างที่เป็นปฏิบัติจริงกันดีกว่า โปรแกรมรายงานผ่านเว็บ รายงานนั้นใช้ข้อมูลที่เก็บไว้ในฐานข้อมูลและสร้างขึ้นโดยใช้เทมเพลต ผลลัพธ์สุดท้ายคือหน้า HTML ที่แสดงให้ผู้ใช้เห็นในเว็บเบราว์เซอร์

ชั้นหน้าที่ใช้ในการจัดการกับ UI ถูกนำเข้าไว้ในหน้าเว็บและอนุญาตให้ผู้ใช้เข้าสู่ระบบ เลือกรายงานที่ต้องการ และคลิกปุ่มเพื่อขอรายงาน ชั้นแอปพลิเคชันเป็นชั้นที่บางเพียงในการเชื่อมต่อระหว่างหน้าต่าง UI, โดเมน และโครงสร้างพื้นฐาน มันจะปฏิสัมพันธ์กับโครงสร้างพื้นฐานฐานข้อมูลในขณะที่ผู้ใช้เข้าสู่ระบบ และจะปฏิสัมพันธ์กับโดเมนเลเยอร์เมื่อต้องการสร้างรายงาน ชั้นโดเมนจะมีส่วนสำคัญของโดเมนคือวัตถุที่เกี่ยวข้องโดยตรงกับรายงาน สองอย่างของวัตถุเหล่านั้นคือรายงานและเทมเพลตซึ่งรายงานจะมีพื้นฐานอยู่บนเทมเพลต ชั้นโครงสร้างพื้นฐานจะสนับสนุนการเข้าถึงฐานข้อมูลและการเข้าถึงไฟล์

เมื่อผู้ใช้เลือกรายงานที่ต้องการสร้าง จริงๆแล้วเขาจะเลือกชื่อของรายงานจากรายการชื่อ ซึ่งเป็น reportID หรือสตริง บางพารามิเตอร์อื่นๆถูกส่งผ่านเช่น รายการที่แสดงในรายงานและช่วงเวลาของข้อมูลที่รวมอยู่ในรายงาน แต่เราจะกล่าวถึง reportID เท่านั้นเพื่อความง่ายดังนี้ ชื่อนี้จะถูกส่งผ่านชั้นประยุกต์ไปยังชั้นโดเมน ชั้นโดเมนรับผิดชอบในการสร้างและส่งคืนรายงานโดยใช้ชื่อเหล่านี้ เนื่องจากรายงานจะต้องใช้เทมเพลตเป็นพื้นฐาน จึงสามารถสร้าง Service เพื่อที่จะได้รับเทมเพลตที่สอดคล้องกับ reportID โดยเทมเพลตนี้จะถูกเก็บไว้ในไฟล์หรือฐานข้อมูล ไม่เหมาะสมที่จะใส่การดำเนินการเช่นนี้ในอ็อบเจกต์ Report เอง และไม่เหมาะกับ Object Template ด้วย เราจึงสร้าง Service แยกต่างหากขึ้นมาเพื่อเตรียมเอกสารรายงานโดยใช้ ID ของรายงานนั้น ๆ ซึ่งจะเป็น Service ที่ตั้งอยู่ใน domain layer และจะใช้โครงสร้างไฟล์เพื่อดึงเทมเพลตจากดิสก์

**Modules**

สำหรับแอปพลิเคชันที่ใหญ่และซับซ้อน โมเดลมักจะเพิ่มขนาดขึ้นเรื่อย ๆ โมเดลก็จะมาถึงจุดที่ยากต่อการพูดถึงเป็นรวมๆ และเข้าใจความสัมพันธ์และปฏิสัมพันธ์ระหว่างส่วนต่าง ๆ ก็จะเป็นเรื่องที่ยากขึ้น ด้วยเหตุนี้จึงจำเป็นต้องจัดองค์ประกอบของโมเดลเป็นโมดูล โมดูลนั้นใช้เป็นวิธีการจัดระบบแนวคิดและงานที่เกี่ยวข้องกันเพื่อลดความซับซ้อน

โมดูลถูกนำมาใช้กันอย่างแพร่หลายในโครงการหลายๆ อัน การมองภาพรวมของโมเดลใหญ่จะง่ายกว่า ถ้าเรามองไปที่โมดูลที่มีอยู่ภายใน แล้วพิจารณาความสัมพันธ์ระหว่างโมดูลเหล่านั้น หลังจากเข้าใจการเชื่อมต่อระหว่างโมดูลแล้ว เราจะสามารถเริ่มหารายละเอียดภายในโมดูลได้เร็วขึ้น วิธีการจัดการความซับซ้อนที่เรียบง่ายและมีประสิทธิภาพ

อีกเหตุผลหนึ่งในการใช้โมดูลเกี่ยวข้องกับคุณภาพของโค้ด การใช้โมดูลช่วยให้โค้ดมีความสอดคล้องกันสูงและมีการเชื่อมโยงต่ำ จะเริ่มต้นด้วยความสอดคล้องกันของคลาสและเมธอดแต่สามารถใช้กับระดับโมดูลได้ด้วย แนะนำให้จัดกลุ่มคลาสที่เกี่ยวข้องกันมากๆเข้าด้วยกันเป็นโมดูลเพื่อให้มีความสอดคล้องกันสูงสุดที่เป็นไปได้ มีหลายประเภทของความสอดคล้องกัน แต่สองประเภทที่นิยมใช้มากที่สุดคือ ความสอดคล้องกันในการสื่อสาร และ ความสอดคล้องกันในการทำงานฟังก์ชัน ความสอดคล้องกันในการสื่อสารเกิดขึ้นเมื่อส่วนต่างๆของโมดูลทำงานกับข้อมูลเดียวกัน มีความสัมพันธ์กันอย่างมาก จึงเหมาะที่จะกลุ่มเข้าด้วยกัน เนื่องจากมีความสัมพันธ์กันอย่างแข็งแรง ส่วนความสอดคล้องกันในการทำงานฟังก์ชันเกิดขึ้นเมื่อส่วนต่างๆของโมดูลทำงานร่วมกันเพื่อประมวลผลงานที่กำหนดไว้ ซึ่งถือว่าเป็นประเภทของความสอดคล้องกันที่ดีที่สุด

การใช้โมดูลในการออกแบบเป็นวิธีหนึ่งในการเพิ่มความสมดุลและลดการผูกมัดของระบบ โมดูลควรประกอบด้วยองค์ประกอบที่สัมพันธ์กันทางฟังก์ชันหรือตรรกะ เพื่อให้มีความสมดุลภายในโมดูล โมดูลควรมีอินเตอร์เฟสที่ถูกกำหนดอย่างชัดเจนซึ่งถูกเรียกใช้โดยโมดูลอื่น ๆ แทนที่จะเรียกใช้วัตถุในโมดูลสามตัว เพราะจะช่วยลดการผูกมัดของระบบ การลดการผูกมัดของระบบจะช่วยลดความซับซ้อนและเพิ่มความสามารถในการบำรุงรักษา การเข้าใจว่าระบบทำงานอย่างไรจะง่ายขึ้นเมื่อมีการเชื่อมต่อน้อยระหว่างโมดูลที่ทำงานที่กำหนดไว้ชัดเจน โดยเปรียบเทียบกับการมีการเชื่อมต่อมากจากโมดูลอื่น ๆ

เลือกโมดูลที่เล่าเรื่องราวของระบบและมีชุดคอนเซปต์ที่สอดคล้องกันอย่างสม่ำเสมอ นี้จะส่งผลให้โมดูลไม่มีการผูกติดกันมากนัก แต่หากไม่สามารถทำได้ให้มองหาวิธีเปลี่ยนแปลงโมเดลเพื่อแยกความสับสนของคอนเซปต์ หรือคอนเซปต์ที่ถูกละเลยซึ่งอาจเป็นพื้นฐานของโมดูลที่จะนำองค์ประกอบมาเชื่อมต่อกันในทางที่สมเหตุสมผล ค้นหาความผูกติดที่ต่ำในแง่ของคอนเซปต์ที่สามารถเข้าใจและวิเคราะห์ได้อย่างอิสระจากกัน ปรับปรุงโมเดลจนแบ่งออกเป็นคอนเซปต์ระดับสูงและรหัสที่เกี่ยวข้องได้ถูกตัดการผูกติดกันด้วย

ให้ตั้งชื่อโมดูลที่เป็นส่วนหนึ่งของภาษาแบบอัตโนมัติ (Ubiquitous Language) โดยชื่อโมดูลและชื่อของแต่ละโมดูลควรสะท้อนความเข้าใจเกี่ยวกับโดเมนนั้นๆ

นักออกแบบเคยใช้งานการสร้างโมดูลตั้งแต่เริ่มต้น โมดูลเป็นส่วนที่พบบ่อยในการออกแบบของเรา หลังจากตัดสินใจบทบาทของโมดูลแล้ว มักจะไม่เปลี่ยนแปลงแต่เนื้อหาภายในของโมดูลอาจมีการเปลี่ยนแปลงได้มาก แนะนำว่าควรมีความยืดหยุ่นและอนุญาตให้โมดูลเปลี่ยนแปลงไปกับโครงการได้และไม่ควรติดตัวไว้เสมอ จริงๆ แล้วการปรับปรุงโมดูลอาจจะมีค่าใช้จ่ายสูงกว่าการปรับปรุงคลาส แต่เมื่อพบข้อผิดพลาดในการออกแบบโมดูล จะดีกว่าที่จะแก้ไขโดยการหาวิธีการรอบรู้แทน การแก้ไขโมดูลจะเป็นวิธีที่ดีกว่า

**Aggregates**

รูปแบบที่สามสุดท้ายในบทนี้จะเกี่ยวกับความท้าทายในการจำลองแบบต่าง ๆ ที่เกี่ยวข้องกับวงจรชีวิตของวัตถุในโดเมน วัตถุในโดเมนจะผ่านการเปลี่ยนแปลงสถานะตามรอบชีวิตของพวกเขา พวกเขาถูกสร้างขึ้น จัดเก็บในหน่วยความจำและนำมาใช้ในการคำนวณ และถูกทำลาย ในบางกรณีพวกเขาถูกบันทึกไว้ในตำแหน่งถาวร เช่นฐานข้อมูล ซึ่งสามารถเรียกกลับมาใช้งานได้ในภายหลัง หรือพวกเขาสามารถถูกเก็บถาวรได้ ในบางจุดพวกเขาอาจถูกลบออกจากระบบทั้งหมด รวมถึงฐานข้อมูลและพื้นที่เก็บถาวร

การจัดการช่วงชีวิตของวัตถุโดเมนเป็นที่ท้าทายเองและหากไม่ทำให้เหมาะสมอาจมีผลกระทบลบต่อโมเดลโดเมน ในบทความนี้เราจะนำเสนอรูปแบบที่ช่วยให้เราจัดการกับปัญหานี้ได้ 3 แบบแพทเทิร์นที่ช่วยให้เราจัดการกับการสร้างวัตถุ และการเก็บรักษาวัตถุ การรวมกลุ่ม (Aggregate) เป็นรูปแบบของโดเมนที่ใช้กำหนดการเป็นเจ้าของของวัตถุและขอบเขตของวัตถุ โรงงาน (Factories) และเก็บคลัง (Repositories) เป็นแบบแพทเทิร์นสองรูปแบบที่ช่วยให้เราจัดการกับการสร้างวัตถุและการเก็บรักษาวัตถุ เราจะเริ่มต้นโดยพูดถึง Aggregates กันก่อน

โมเดลสามารถมีวัตถุโดเมนจำนวนมากได้ ไม่ว่าจะใส่ความสำคัญเท่าไหร่ในการออกแบบ บางครั้งก็จะมีวัตถุหลายอันที่เกี่ยวข้องกัน สร้างเครือข่ายความสัมพันธ์ที่ซับซ้อน มีหลายประเภทของการเชื่อมโยงกัน สำหรับทุกการเชื่อมโยงที่สามารถไล่ไปมาได้ในโมเดล จะต้องมีกลไกซอฟต์แวร์ที่สอดคล้องกันเพื่อให้ทำงานได้ การเชื่อมโยงจริงระหว่างวัตถุโดเมนก็จะถูกแสดงในรหัส บ่อยครั้งที่ก็จะแสดงในฐานข้อมูลด้วย ความสัมพันธ์แบบหนึ่งต่อหนึ่งระหว่างลูกค้าและบัญชีธนาคารที่เปิดในชื่อของเขาจะถูกแสดงเป็นการอ้างอิงระหว่างวัตถุสองอัน และนั่นแปลว่ามีความสัมพันธ์ระหว่างตารางฐานข้อมูลสองตาราง คือตารางที่เก็บข้อมูลลูกค้าและตารางที่เก็บบัญชี

ความท้าทายของโมเดลมักไม่ใช่การทำให้มันสมบูรณ์พอแต่มากกว่านั้นก็คือการทำให้มันเรียบง่ายและเข้าใจง่ายที่สุดเท่าที่จะทำได้ เป็นเวลาส่วนใหญ่จะช่วยให้เราลดหรือบูรณาการความสัมพันธ์ในโมเดล นั่นคือ หากมันไม่ได้ส่งผลต่อความเข้าใจลึกซึ้งในโดเมน โดยทั่วไปแล้วจะมีประโยชน์ที่จะทำลายหรือทำให้ความสัมพันธ์เหล่านั้นเรียบง่ายขึ้นจากโมเดล

ความสัมพันธ์แบบ one-to-many มีความซับซ้อนมากกว่าเนื่องจากมีการเชื่อมโยงหลายออบเจกต์เข้าด้วยกัน ความสัมพันธ์นี้สามารถทำให้เรียบง่ายขึ้นได้โดยการแปลงมันเป็นความสัมพันธ์ระหว่างออบเจกต์หนึ่งกับกลุ่มของออบเจกต์อื่น ๆ แม้ว่าจะไม่สามารถทำได้เสมอไปทุกกรณี

มีการเชื่อมโยงแบบหลายต่อหลายและมีจำนวนมากที่เป็นแบบสองทิศทาง สิ่งนี้ทำให้ความซับซ้อนเพิ่มขึ้นอย่างมาก ทำให้การจัดการวงจรชีวิตของวัตถุเหล่านี้ยากมากๆ จำนวนการเชื่อมโยงควรลดลงเท่าที่เป็นไปได้ โดยเริ่มต้นด้วยการลบการเชื่อมโยงที่ไม่จำเป็นสำหรับโมเดลออกก่อน อาจจะมีอยู่ในโดเมน แต่ไม่จำเป็นต่อโมเดลของเรา เราควรเอาออกไป นอกจากนี้ เราสามารถลดความซับซ้อนของจำนวนการเชื่อมโยงได้โดยการเพิ่มเงื่อนไข หากวัตถุหลายๆ อยู่ในความสัมพันธ์เดียวกัน อาจจะมีเพียงตัวเดียวที่ทำได้หากมีการบังคับเงื่อนไขที่เหมาะสมบนความสัมพันธ์นั้นๆ เมื่อมีการเชื่อมโยงแบบสองทิศทาง บางครั้งสามารถแปลงเป็นแบบเดียวทิศทางได้ รถทุกคันมีเครื่องยนต์และทุกเครื่องยนต์มีรถที่มันทำงานอยู่ ความสัมพันธ์เป็นแบบสองทิศทาง แต่มันสามารถที่จะตกแต่งให้เป็นแบบเดียวทิศทางได้อย่างง่ายดาย โดยพิจารณาว่ารถมีเครื่องยนต์ แต่ไม่ได้พิจารณาเครื่องยนต์ว่ามีรถอยู่ด้วย

หลังจากที่เราลดลงและทำให้การเชื่อมโยงระหว่างวัตถุเรียบง่ายลง เราอาจยังต้องมีความสัมพันธ์อยู่หลายอย่าง ระบบธนาคารเก็บและประมวลผลข้อมูลลูกค้า ซึ่งประกอบด้วยข้อมูลส่วนตัวของลูกค้า เช่น ชื่อ ที่อยู่ เบอร์โทรศัพท์ ลักษณะงาน และข้อมูลบัญชี เช่น หมายเลขบัญชี ยอดเงินคงเหลือ การดำเนินการ เป็นต้น เมื่อระบบเก็บข้อมูลลูกค้าเข้าถึงข้อมูลเก่าหรือลบข้อมูลออกจากระบบ ระบบจะต้องตรวจสอบให้แน่ใจว่ามีการลบการอ้างอิงทั้งหมดออกไป หากมีวัตถุหลายอย่างที่เก็บการอ้างอิงเช่นนี้ จะยากที่จะตรวจสอบว่าทั้งหมดถูกลบออกหรือไม่ นอกจากนี้ เมื่อข้อมูลบางอย่างของลูกค้าเปลี่ยนแปลง ระบบต้องตรวจสอบให้แน่ใจว่ามันถูกปรับปรุงอย่างเหมาะสมทั้งระบบ และความสอดคล้องของข้อมูลได้รับการรับประกัน ซึ่งมักถูกเว้นไว้ให้ทำการจัดการที่ระดับฐานข้อมูล การดำเนินการจะถูกใช้เพื่อให้มั่นใจว่าความสอดคล้องของข้อมูลถูกทำให้ตรงกัน การดำเนินการดังกล่าวเรียกว่า "การทำธุรกรรม" (Transactions)

แต่ถ้าโมเดลไม่ได้ออกแบบอย่างระมัดระวัง จะเกิดการแข่งขันฐานข้อมูลที่สูง ทำให้ประสิทธิภาพไม่ดี ในขณะที่ธุรกรรมของฐานข้อมูลมีบทบาทสำคัญในการดำเนินการเช่นนั้น มีความต้องการที่จะแก้ไขบางปัญหาที่เกี่ยวข้องกับความสอดคล้องของข้อมูลโดยตรงในโมเดล

การใช้งานเป็นสิ่งจำเป็นที่ต้องทำได้ในการใช้กฎที่ต้องการรักษาอยู่เสมอเมื่อข้อมูลเปลี่ยนแปลง กฎเหล่านี้เป็นสิ่งที่ยากต่อการตระหนักเมื่อมีอ็อบเจกต์หลายอ็อบเจกต์ที่ถือการอ้างอิงไปยังวัตถุข้อมูลที่เปลี่ยนแปลง

การรักษาความสอดคล้องของการเปลี่ยนแปลงที่เกิดขึ้นกับวัตถุในโมเดลที่มีการผูกพันซับซ้อนเป็นเรื่องยาก บ่อยครั้งอินวาเรียนต์จะถูกใช้กับวัตถุที่เกี่ยวข้องกันอย่างใกล้ชิดไม่ใช่แค่วัตถุแยกต่างหาก เว้นแต่การล็อคที่ระมัดระวังอาจทำให้ผู้ใช้หลายคนมีส่วนประสานงานกันอย่างไร้ประโยชน์และทำให้ระบบไม่สามารถใช้งานได้

ดังนั้นให้ใช้ Aggregates โดย Aggregate คือกลุ่มของวัตถุที่เกี่ยวข้องกันซึ่งถือว่าเป็นหน่วยเดียวกันเมื่อพิจารณาเกี่ยวกับการเปลี่ยนแปลงของข้อมูล โดย Aggregate จะมีขอบเขตที่แบ่งแยกวัตถุภายในจากวัตถุภายนอก และแต่ละ Aggregate จะมี Root อย่างเดียว โดย Root คือ Entity และเป็นวัตถุที่สามารถเข้าถึงได้จากภายนอกเพียงอย่างเดียว โดย Root สามารถเก็บอ้างอิงไปยังวัตถุใดก็ได้ภายใน Aggregate และวัตถุอื่น ๆ สามารถเก็บอ้างอิงกันได้ แต่วัตถุภายนอกสามารถเก็บอ้างอิงไปยัง Root object เท่านั้น หากมี Entities อื่น ๆ อยู่ภายในขอบเขต จะต้องมี identity ที่เป็นท้องถิ่นเท่านั้น และมีความหมายเพียงแค่อยู่ภายใน Aggregate เท่านั้น

Aggregate ทำการรักษาความสมบูรณ์ของข้อมูลและบังคับใช้ invariant อย่างไร? เนื่องจากว่าอ็อบเจกต์อื่นๆ สามารถเก็บ reference ไว้ได้เพียงแค่ที่ root เท่านั้น นั่นหมายความว่าพวกเขาไม่สามารถเปลี่ยนแปลงอ็อบเจกต์อื่นๆ ใน aggregate ได้โดยตรง พวกเขาสามารถเปลี่ยนแปลง root หรือขอให้ root ดำเนินการบางอย่างได้เท่านั้น และ root จะสามารถเปลี่ยนแปลงอ็อบเจกต์อื่นๆ ได้ แต่นั่นเป็นการดำเนินการที่มีอยู่ภายใน aggregate เท่านั้น และมีควบคุมได้ หาก root ถูกลบและถูกนำออกจากหน่วยความจำ อ็อบเจกต์อื่นๆ ใน aggregate จะถูกลบไปด้วย เนื่องจากไม่มีอ็อบเจกต์อื่นๆ ที่จะเก็บ reference ไว้ในทุกๆ ตัว ในกรณีที่มีการเปลี่ยนแปลงใดๆ ที่ทำโดย root และส่งผลต่ออ็อบเจกต์อื่นๆ ใน aggregate นั้น การบังคับใช้ invariant จะง่ายเพราะ root จะทำการนั้น การบังคับใช้ invariant ในกรณีที่อ็อบเจกต์ภายนอกมีการเข้าถึงอ็อบเจกต์ภายในโดยตรงและเปลี่ยนแปลงได้นั้นจะยากกว่านั้น การบังคับใช้ invariant ในสถานการณ์เช่นนี้ต้องมีการเขียนโค้ดเพิ่มเติมในอ็อบเจกต์ภายนอกเพื่อจัดการกับสถานการณ์ดังกล่าวซึ่งไม่เป็นสิ่งที่ต้องการ

เป็นไปได้ว่ารากสามารถส่งอ้างอิงชั่วคราวของวัตถุภายในไปยังวัตถุภายนอก โดยมีเงื่อนไขว่าวัตถุภายนอกไม่คงอ้างอิงหลังจากที่การดำเนินการเสร็จสิ้นแล้ว วิธีที่ง่ายที่สุดคือส่งสำเนาของวัตถุค่า (Value Objects) ไปยังวัตถุภายนอก ไม่ว่าวัตถุเหล่านั้นจะเกิดอะไรขึ้นก็ตาม เนื่องจากมันจะไม่มีผลต่อความสมบูรณ์ของสิ่งรวมใด ๆ

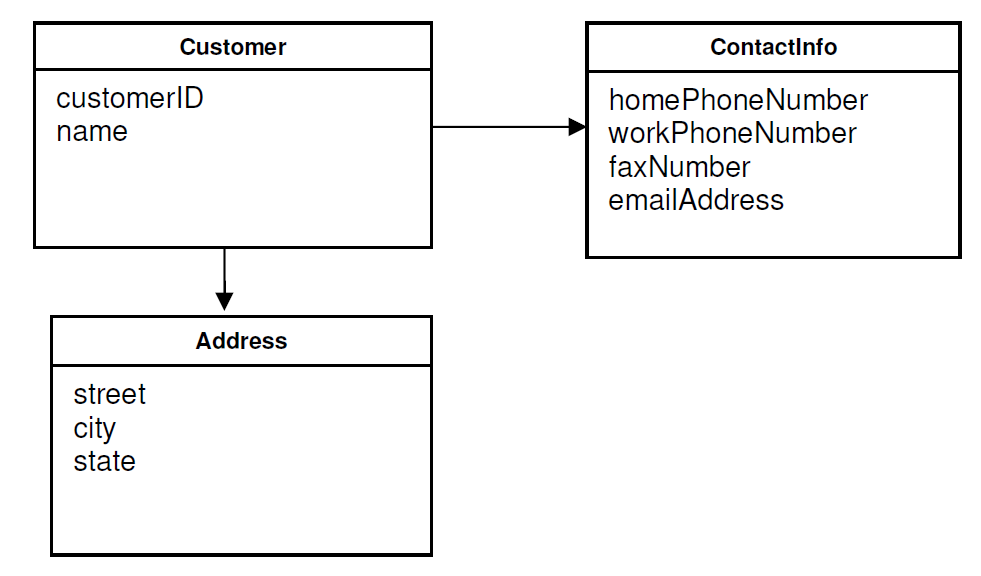
หากวัตถุของ Aggregate ถูกเก็บไว้ในฐานข้อมูล ต้องการเพียงรากเท่านั้นที่สามารถเข้าถึงได้ผ่านการสอบถาม (queries) วัตถุอื่นๆควรได้รับการเข้าถึงผ่านความสัมพันธ์การเดินทาง (traversal associations)

วัตถุภายในการรวมกลุ่ม (Aggregate) ควรได้รับอนุญาตให้เก็บอ้างอิงไปยังรากของ Aggregate อื่น ๆ

ตัวแทนรากมีเอกลักษณ์ระดับทั้งหมดและรับผิดชอบในการรักษาค่าคงที่ เนื้อหาภายใน (Internal Entities) มีเอกลักษณ์ระดับท้องถิ่น

จัดกลุ่ม Entities และ Value Objects เข้าไปใน Aggregates และกำหนดขอบเขตรอบแต่ละอัน โดยเลือก Entity 1 ตัวเป็น Root ของแต่ละ Aggregate และควบคุมการเข้าถึง Object ภายในของขอบเขตผ่าน Root เท่านั้น อนุญาตให้ Object ภายนอกเก็บ reference ไปที่ Root เท่านั้น ส่วน Transient reference ไปยัง internal members สามารถส่งออกไปใช้งานภายใน single operation เท่านั้น เนื่องจาก Root ควบคุมการเข้าถึง ดังนั้นไม่สามารถมองข้ามการเปลี่ยนแปลงใน internals ได้ การจัดเรียงนี้ทำให้เป็นไปได้ต่อการบังคับ invariant สำหรับ objects ภายใน Aggregate และสำหรับ Aggregate ทั้งหมดในการเปลี่ยนสถานะใดๆ

ตัวอย่างง่ายๆของ Aggregation แสดงในแผนภูมิด้านล่าง ลูกค้าเป็นรากของ Aggregate และวัตถุอื่น ๆ ทั้งหมดเป็นภายใน หากต้องการที่อยู่ สามารถส่งสำเนาของมันไปยังวัตถุภายนอกได้



**Factories**

Entities และ Aggregates สามารถมีขนาดใหญ่และซับซ้อนได้บ่อยครั้ง - มันซับซ้อนเกินไปที่จะสร้างใน constructor ของ root entity ในความเป็นจริงการพยายามสร้าง aggregate ที่ซับซ้อนใน constructor เป็นการขัดแย้งกับสิ่งที่เกิดขึ้นในโดเมนเอง โดยสิ่งที่สร้างขึ้นจะเป็นผลมาจากสิ่งอื่น ๆ (เช่นอิเล็กทรอนิกส์ถูกสร้างขึ้นบนสายงานการประกอบ) มันเหมือนกับมีเครื่องพิมพ์สร้างตนเอง

เมื่อวัตถุลูกค้าต้องการสร้างวัตถุอื่น ๆ เข้ามา จะเรียก constructor ของมันและอาจส่งพารามิเตอร์บางอย่างไปด้วย แต่เมื่อการสร้างวัตถุเป็นกระบวนการที่ยุ่งยาก การสร้างวัตถุนั้นเกี่ยวข้องกับความรู้เกี่ยวกับโครงสร้างภายในของวัตถุ ความสัมพันธ์ระหว่างวัตถุที่มีอยู่ภายใน และกฎที่ใช้กับวัตถุเหล่านั้น นั่นหมายความว่าแต่ละลูกค้าของวัตถุจะถือความรู้เฉพาะเกี่ยวกับวัตถุที่สร้างขึ้น ซึ่งทำให้การปกคลุมของวัตถุโดเมนและ Aggregate ถูกขัดขวาง หากไคลเอนต์เป็นส่วนหนึ่งของเลเยอร์แอปพลิเคชัน เลเยอร์โดเมนบางส่วนถูกย้ายไปภายนอกและทำให้การออกแบบทั้งหมดเกิดความสับสน ในชีวิตจริง เหมือนว่าเราได้รับพลาสติก ยาง โลหะ ซิลิโคน และกำลังสร้างเครื่องพิมพ์ของเราเอง มันไม่ได้เป็นไปไม่ได้ แต่มันจะคุ้มค่าจริง ๆ หรือไม่?

การสร้างวัตถุอาจเป็นการดำเนินการที่สำคัญเอง แต่การประกอบของวัตถุที่ซับซ้อนไม่สอดคล้องกับความรับผิดชอบของวัตถุที่ถูกสร้างขึ้น การรวมความรับผิดชอบเช่นนี้อาจสร้างการออกแบบที่ซับซ้อนและยากในการเข้าใจ

ดังนั้น จึงจำเป็นต้องมีแนวคิดใหม่ที่จะช่วยในการกลบกลุ่มกระบวนการสร้างวัตถุที่ซับซ้อน นั่นคือ Factory โรงงาน (Factory) ถูกใช้เพื่อห่อหุ้มความรู้ที่จำเป็นสำหรับการสร้างวัตถุ และมันมีประโยชน์อย่างมากในการสร้าง Aggregate โดยเมื่อสร้าง Root ของ Aggregate ทั้งหมด วัตถุทั้งหมดที่อยู่ใน Aggregate จะถูกสร้างพร้อมกันด้วย และความตรงเวลาของวัตถุทั้งหมดจะถูกบังคับใช้งาน

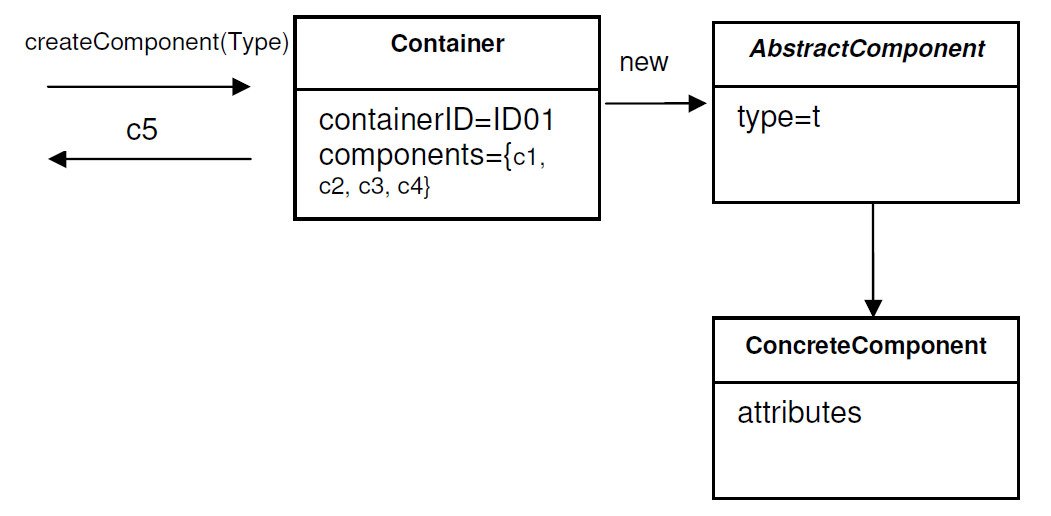
การสร้างกระบวนการจะต้องเป็นอะตอมิกส์มีความสำคัญ หากไม่ได้เช่นนั้น จะมีโอกาสที่กระบวนการสร้างจะเสร็จไม่สมบูรณ์สำหรับบางอ็อบเจกต์ ทำให้เหลืออยู่ในสถานะที่ไม่ได้กำหนดไว้ นี่เป็นเรื่องที่จริงมากขึ้นสำหรับอะแกร็กเกต โดยเมื่อสร้างรูท (root) จะต้องมีการสร้างวัตถุทั้งหมดที่มีอยู่ในสิ่งที่ไม่เปลี่ยนแปลง (invariants) ด้วย มิฉะนั้นสิ่งที่ไม่เปลี่ยนแปลงไม่สามารถปฏิบัติได้ สำหรับอ็อบเจกต์ที่มีค่าที่ไม่เปลี่ยนแปลงแบบไม่สามารถเปลี่ยนแปลงได้ (immutable Value Objects) นั่นหมายความว่าทุกคุณสมบัติถูกกำหนดค่าเพื่อให้เป็นสถานะที่ถูกต้อง หากไม่สามารถสร้างวัตถุได้อย่างถูกต้อง จะต้องเรียกใช้ข้อยกเว้น (exception) เพื่อให้มั่นใจว่าจะไม่มีค่าที่ไม่ถูกต้องถูกส่งกลับ

ดังนั้น ให้ย้ายความรับผิดชอบในการสร้างออบเจกต์ที่ซับซ้อนและเชิงรวมไปยังออบเจกต์ที่แยกต่างหากซึ่งอาจไม่มีความรับผิดชอบในโดเมนโมเดล แต่ยังเป็นส่วนหนึ่งของการออกแบบโดเมน ให้มีอินเทอร์เฟซที่ห่อหุ้มการประกอบที่ซับซ้อนทั้งหมดโดยไม่ต้องให้ผู้ใช้งานอ้างอิงถึงคลาสที่เป็นที่สร้างของออบเจกต์ สร้างออบเจกต์ทั้งหมดของ Aggregates ในรูปแบบเดียวกันเพื่อบังคับเงื่อนไขคงที่ของพวกเขา

มีรูปแบบการออกแบบหลายแบบที่ใช้ในการนำไปประยุกต์ใช้กับโรงงาน (Factories) หนังสือ Design Patterns โดย Gamma และผู้อื่นอธิบายรายละเอียดและนำเสนอแพทเทิร์นสองแบบระหว่างอื่น: Factory Method, Abstract Factory จะไม่พยายามนำเสนอแพทเทิร์นจากมุมมองด้านการออกแบบ แต่จะนำเสนอจากมุมมองการจัดแบบโดเมน (domain modeling)

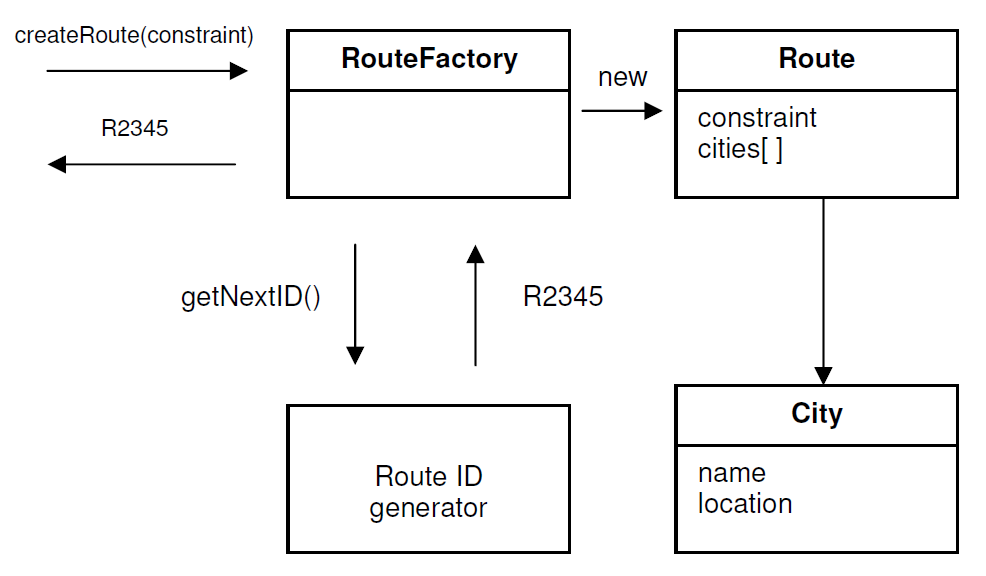
มีรูปแบบการออกแบบหลายแบบที่ใช้ในการนำไปประยุกต์ใช้กับโรงงาน (Factories) หนังสือ Design Patterns โดย Gamma และผู้อื่นอธิบายรายละเอียดและนำเสนอแพทเทิร์นสองแบบระหว่างอื่น: Factory Method, Abstract Factory จะไม่พยายามนำเสนอแพทเทิร์นจากมุมมองด้านการออกแบบ แต่จะนำเสนอจากมุมมองการจัดแบบโดเมน (domain modeling)

Factory Method คือ เมธอดของออบเจ็กต์ที่มีความรู้และซ่อนเร้นที่จำเป็นในการสร้างออบเจ็กต์อื่น ๆ ซึ่งมีประโยชน์อย่างมากเมื่อไคลเอนต์ต้องการสร้างออบเจ็กต์ที่เป็นส่วนประกอบ วิธีการแก้ไขปัญหานี้คือการเพิ่มเมธอดในรากของส่วนประกอบ (Aggregate) ซึ่งจะดูแลการสร้างออบเจ็กต์ บังคับเงื่อนไขสมมูลทั้งหมด และส่งออบเจ็กต์หรือสำเนาออกมาให้.



คอนเทนเนอร์ประกอบด้วยส่วนประกอบและมีประเภทเฉพาะตามนั้น จำเป็นต้องทำให้เมื่อสร้างส่วนประกอบดังกล่าวแล้ว จะต้องเป็นส่วนประกอบของคอนเทนเนอร์โดยอัตโนมัติ ลูกค้าจะเรียกใช้วิธี createComponent(Type t) ของคอนเทนเนอร์ คอนเทนเนอร์จะสร้างองค์ประกอบใหม่ ชนิดขององค์ประกอบจะถูกกำหนดขึ้นอยู่กับประเภทของมัน หลังจากสร้างแล้ว องค์ประกอบจะถูกเพิ่มเข้าไปในคอลเล็กชันขององค์ประกอบที่อยู่ในคอนเทนเนอร์ และส่งกลับไปยังลูกค้าด้วยการคัดลอกองค์ประกอบนั้น

มีช่วงเวลาบางครั้งที่การสร้างวัตถุมีความซับซ้อนมากขึ้น หรือเมื่อการสร้างวัตถุเป็นการสร้างชุดของวัตถุ เช่นการสร้าง Aggregate การซ่อนความต้องการสร้างภายในของ Aggregate สามารถทำได้ด้วยวัตถุ Factory ที่มีการจัดสรรสำหรับงานนี้ไว้ในตัวอย่างนี้ เราจะพิจารณาโมดูลโปรแกรมที่คำนวณเส้นทางที่รถยนต์สามารถตามได้จากจุดเริ่มต้นถึงปลายทางโดยให้มีการกำหนดเงื่อนไขต่าง ๆ ผู้ใช้เข้าสู่เว็บไซต์ที่เรียกใช้งานแอปพลิเคชันและระบุเงื่อนไขต่อไปนี้: เส้นทางสั้นที่สุด เส้นทางที่เร็วที่สุด หรือเส้นทางที่ถูกที่สุด เส้นทางที่สร้างขึ้นสามารถประกอบไปด้วยข้อมูลผู้ใช้งานที่ต้องการบันทึกไว้ เพื่อให้สามารถดึงข้อมูลเหล่านี้มาใช้งานได้ในภายหลังเมื่อลูกค้าเข้าสู่ระบบอีกครั้ง



ตัวสร้างรหัสเส้นทาง (Route ID generator) ถูกใช้สร้างตัวตนที่ไม่ซ้ำกันสำหรับเส้นทางแต่ละเส้น ซึ่งจำเป็นสำหรับฟังก์ชันการดำเนินการ (Entity) ใดๆ

เมื่อสร้าง Factory จะต้องละเมิดการซ่อนข้อมูลของ Object (encapsulation) ซึ่งต้องทำด้วยความระมัดระวัง ทุกครั้งที่มีการเปลี่ยนแปลงใน Object ที่มีผลต่อกฎการสร้างหรือสิ่งที่เกี่ยวข้องกับเงื่อนไขต่าง ๆ ของ Object ต้องตรวจสอบให้แน่ใจว่า Factory ได้รับการอัพเดตเพื่อรองรับเงื่อนไขใหม่ ๆ ได้

Factory เป็นสิ่งที่เกี่ยวข้องกับ Object ที่ถูกสร้างขึ้น ซึ่งอาจเป็นจุดอ่อน แต่ก็อาจเป็นจุดแข็งของระบบด้วย เนื่องจาก Aggregate ประกอบด้วย Object หลายตัวที่เกี่ยวข้องกันอย่างใกล้ชิด การสร้าง Root Object มีความสัมพันธ์กับการสร้าง Object อื่น ๆ ใน Aggregate ต้องมีตรรกะที่นำเข้าด้วยกัน เนื่องจากตรรกะนั้นไม่สามารถอยู่ใน Object ใด ๆ ได้ เนื่องจากมันเกี่ยวข้องกับการสร้าง Object อื่น ๆ ด้วย ดังนั้นการใช้คลาส Factory เพื่อสร้าง Aggregate ทั้งหมดและประกอบด้วยกฎการสร้าง การจำกัดและเงื่อนไขต่าง ๆ ที่ต้องปฏิบัติให้ถูกต้องเพื่อให้ Aggregate เป็นไปตามเงื่อนไขที่ถูกต้อง จะเหลือ Object ซึ่งง่ายและทำงานในวัตถุประสงค์เฉพาะของตัวเองโดยไม่มีการปนเปื้อนของตรรกะการสร้าง Object ที่ซับซ้อน

Entity Factories และ Value Object Factories ต่างกันอย่างชัดเจน ค่า (Values) มักจะเป็นออบเจ็กต์ที่ไม่สามารถเปลี่ยนแปลงได้ และจะต้องมีแอตทริบิวต์ที่จำเป็นทั้งหมดที่ต้องสร้างขึ้นตอนสร้างออบเจ็กต์ ในการสร้างออบเจ็กต์ จะต้องเป็นไปตามเงื่อนไขและความสมบูรณ์แบบและไม่สามารถเปลี่ยนแปลงได้อีกต่อไป แต่ Entity ไม่เป็นออบเจ็กต์ที่ไม่สามารถเปลี่ยนแปลงได้ เพราะออบเจ็กต์นี้สามารถเปลี่ยนแปลงได้ในภายหลัง โดยการตั้งค่าบางอย่างจากแอตทริบิวต์ โดยมีการตรวจสอบเงื่อนไขของการเปลี่ยนแปลง อีกทั้งความแตกต่างที่มาจากตัว Identity ซึ่งจำเป็นต่อ Entity แต่ไม่จำเป็นต่อ Value Object

มีช่วงเวลาที่ไม่จำเป็นต้องใช้ Factory และ Constructor ที่เรียบง่ายเพียงอย่างเดียวก็เพียงพอแล้ว ใช้ Constructor เมื่อ:

• การสร้างไม่ซับซ้อน

• การสร้างวัตถุไม่รวมการสร้างวัตถุอื่นๆ และมีแอตทริบิวต์ทั้งหมดที่จำเป็นผ่านทางคอนสตรักเตอร์

• ลูกค้าสนใจในการดำเนินการสร้างภายใน และบางครั้งอาจต้องการเลือกใช้กลยุทธ์ที่ใช้งาน

• คลาสเป็นชนิด ไม่มีลำดับชั้นที่เกี่ยวข้อง ดังนั้นไม่จำเป็นต้องเลือกตัวอย่างการใช้งานที่เป็นขั้นตอนเดียวกัน.

ความสังเกตอีกอย่างคือ โรงงานต้องสร้างวัตถุใหม่โดยสิ้นเชิง หรือต้องสร้างวัตถุใหม่จากการประกอบด้วยวัตถุที่มีอยู่ก่อนหน้านี้แต่อาจถูกเก็บไว้ในฐานข้อมูล การนำ Entity กลับมาในหน่วยความจำจากที่พักในฐานข้อมูลนั้นเกี่ยวข้องกับกระบวนการที่แตกต่างกันอย่างสิ้นเชิงเมื่อสร้างวัตถุใหม่ หนึ่งในความแตกต่างที่ชัดเจนคือ วัตถุใหม่ไม่ต้องการตัวระบุใหม่ เพราะวัตถุมีตัวระบุอยู่แล้ว การละเว้น invariant จะถูกจัดการต่างหาก เมื่อสร้างวัตถุใหม่จากสิ่งที่ไม่มีอยู่มาก่อน การละเว้น invariant อันไหนก็ตามจะสิ้นสุดลงใน exception แต่เราไม่สามารถทำเช่นนั้นกับวัตถุที่สร้างใหม่จากฐานข้อมูลได้ วัตถุต้องได้รับการซ่อมแซมให้เป็นปกติเพื่อให้สามารถทำงานได้ มิฉะนั้นจะเกิดการสูญเสียข้อมูล

**Repositories**

ในการออกแบบด้วยแบบจำลองขับเคลื่อนด้วยโมเดล (Model-driven design) วัตถุมีชีวิตขึ้นตั้งแต่ขั้นตอนการสร้างและสิ้นสุดลงด้วยการลบหรือเก็บถาวร (deletion or archiving) คอนสตรัคเตอร์หรือแฟคทอรี่จะดูแลการสร้างวัตถุ จุดประสงค์ทั้งหมดของการสร้างวัตถุคือเพื่อใช้งาน ในภาษาเชิงวัตถุ ผู้ใช้งานจะต้องเก็บอ้างอิงวัตถุเพื่อใช้งาน ในการเก็บอ้างอิงวัตถุนั้น ผู้ใช้งานจะต้องสร้างวัตถุเองหรือได้รับวัตถุจากอื่นๆ โดยการเดินทางผ่านสัมพันธ์ที่มีอยู่แล้ว ตัวอย่างเช่น หากต้องการรับวัตถุค่าตัวแปรของ Aggregate ผู้ใช้งานต้องขอจากตัวรากของ Aggregate ปัญหาตอนนี้คือลูกค้าต้องมีการอ้างอิงถึงรูท (root) สำหรับแอปพลิเคชันขนาดใหญ่นั้นเป็นปัญหาเนื่องจากต้องมั่นใจว่าลูกค้ามีการอ้างอิงถึงวัตถุที่ต้องการเสมอ หรือถึงวัตถุอื่นที่มีการอ้างอิงถึงวัตถุที่เกี่ยวข้อง เมื่อใช้กฎนี้ในการออกแบบจะทำให้วัตถุต้องเก็บการอ้างอิงเป็นชุดของวัตถุที่พวกเขาอาจจะไม่จำเป็นต้องเก็บไว้เนื่องจากเพิ่มความเชื่อมโยง (coupling) และสร้างความสัมพันธ์ระหว่างวัตถุที่ไม่จำเป็นเกินไป

การใช้วัตถุหมายถึงว่าวัตถุนั้นถูกสร้างขึ้นแล้ว หากวัตถุเป็นรากของ Aggregate แสดงว่าเป็น Entity และมีโอกาสที่จะถูกเก็บไว้ในสถานะที่ต่อเนื่องในฐานข้อมูลหรือช่องทางการเก็บข้อมูลอื่น ๆ หากเป็น Value Object อาจพบได้จาก Entity โดยการเดินทางผ่านการเชื่อมโยง ด้วยความเป็นจริงแล้วมีจำนวนมากของวัตถุที่สามารถได้รับโดยตรงจากฐานข้อมูลได้ ซึ่งแก้ไขปัญหาการอ้างอิงวัตถุได้ ในกรณีที่ไคลเอนต์ต้องการใช้วัตถุ จะเข้าถึงฐานข้อมูล ดึงข้อมูลวัตถุจากฐานข้อมูลแล้วนำมาใช้ ดูเหมือนว่าเป็นวิธีที่รวดเร็วและง่าย แต่มีผลกระทบลบต่อการออกแบบ

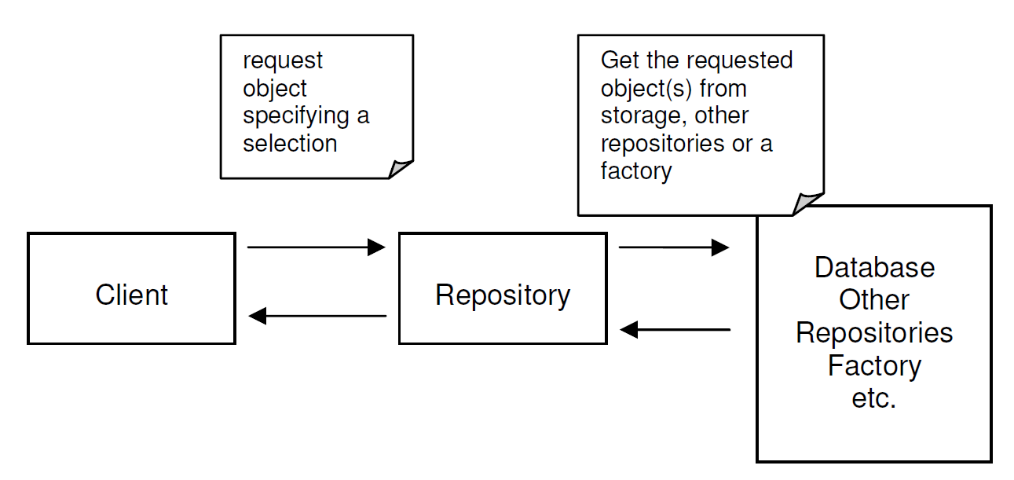
ฐานข้อมูลเป็นส่วนหนึ่งของโครงสร้างพื้นฐาน (Infrastructure) การแก้ปัญหาที่ไม่ดีคือให้ลูกค้าต้องรู้เกี่ยวกับรายละเอียดที่ต้องการเพื่อเข้าถึงฐานข้อมูล เช่น ลูกค้าจะต้องสร้างคำสั่ง SQL เพื่อเรียกดูข้อมูลที่ต้องการ การคิวรี่ฐานข้อมูลอาจคืนค่าเซตของเรคคอร์ดออกมา ซึ่งอาจเปิดเผยรายละเอียดภายในของฐานข้อมูลได้อีกด้วย เมื่อมีลูกค้าหลายคนต้องสร้างออบเจ็กต์โดยตรงจากฐานข้อมูล จะพบว่าโค้ดเช่นนี้กระจายไปทั่วทั้งโดเมน ณ จุดนั้นโมเดลโดเมนก็เสียหายไป ต้องจัดการกับรายละเอียดของสถาปัตยกรรมมากมายแทนที่จะจัดการกับแนวคิดของโดเมน หากมีการตัดสินใจเปลี่ยนฐานข้อมูลในภายหลัง โค้ดที่กระจายอยู่ทั่วไปนั้นจะต้องเปลี่ยนแปลงเพื่อเข้าถึงการจัดเก็บข้อมูลใหม่ หากโค้ดของลูกค้าเข้าถึงฐานข้อมูลโดยตรง อาจเป็นไปได้ว่ามันจะกู้คืนออบเจ็กต์ภายใน Aggregate ซึ่งจะทำลายการห่วงความปลอดภัยของ Aggregate และอาจมีผลที่ไม่คาดคิด

ลูกค้าต้องการวิธีการที่สามารถเข้าถึงอ็อบเจกต์โดเมนที่มีอยู่แล้วได้อย่างมีประสิทธิภาพ หากโครงสร้างพื้นฐานทำให้สามารถทำได้ง่าย นักพัฒนาของลูกค้าอาจเพิ่มการเชื่อมโยงที่สามารถเดินทางได้มากขึ้น ทำให้โมเดลเป็นความสับสน อย่างไรก็ตาม พวกเขาอาจใช้คิวรีเพื่อดึงข้อมูลที่ต้องการจากฐานข้อมูล หรือดึงอ็อบเจกต์บางอย่างที่ต้องการโดยตรง โดยไม่ต้องนำเสนอจาก Aggregate roots ตรรกะโดเมน (Domain logic) ถูกย้ายไปยังคำสั่งคิวรีและโค้ดของไคลเอ็นต์ (client code) และ Entity และ Value Object กลายเป็นเพียงคอนเทนเนอร์ข้อมูลเท่านั้น ความซับซ้อนทางเทคนิคของการเข้าถึงฐานข้อมูลส่วนใหญ่รวดเร็วเกินไป จึงส่งผลให้โค้ดของไคลเอ็นต์แย่ลง ซึ่งนำมาซึ่งการลดความซับซ้อนของโดเมนเลเยอร์ (domain layer) ซึ่งทำให้โมเดลไม่สำคัญและส่งผลให้การออกแบบเสียหาย โดยทั่วไปแล้วจะเสียไปทั้งโฟกัสด้านโดเมนและการออกแบบ

ดังนั้นให้ใช้ Repository ซึ่งจุดประสงค์ของมันคือการห่อหุ้มโลจิกทั้งหมดที่จำเป็นในการได้รับการอ้างอิงของวัตถุ วัตถุในโดเมนจะไม่ต้องจัดการโครงสร้างพื้นฐานเพื่อรับการอ้างอิงที่จำเป็นกับวัตถุโดเมนอื่น ๆ พวกเขาจะได้รับจาก Repository เท่านั้นและโมเดลก็กลับคืนความชัดเจนและโฟกัสของมัน

ที่เก็บข้อมูลอาจเก็บอ้างอิงไปยังวัตถุบางอย่าง เมื่อสร้างวัตถุขึ้นมา อาจจะถูกบันทึกไว้ในที่เก็บข้อมูล และเรียกดึงข้อมูลวัตถุนั้นมาใช้งานในภายหลัง หากผู้ใช้ร้องขอวัตถุจากที่เก็บข้อมูล และที่เก็บข้อมูลไม่มีวัตถุนั้น ที่เก็บข้อมูลอาจจะไปเรียกดึงวัตถุจากที่จัดเก็บอยู่ โดยวิธีใดวิธีหนึ่ง ที่เก็บข้อมูลเป็นสถานที่จัดเก็บวัตถุที่สามารถเข้าถึงได้โดยรวม

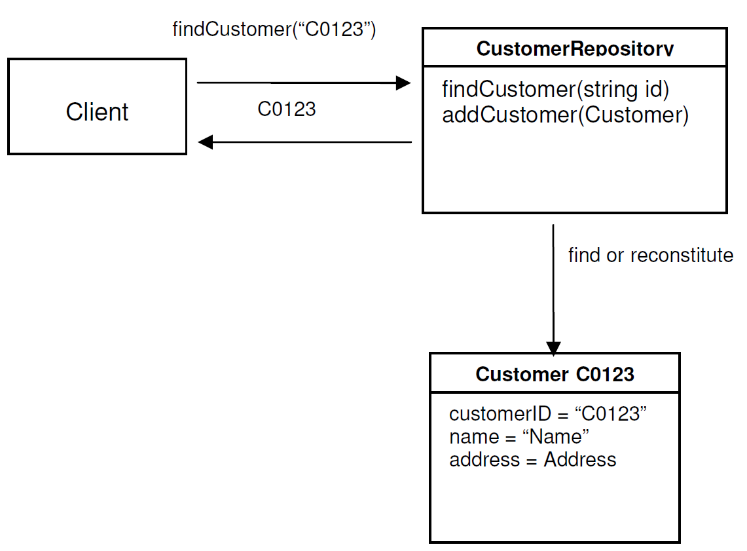
คลังข้อมูล (Repository) อาจมีกลยุทธ์ (Strategy) ร่วมด้วย โดยอาจเข้าถึงที่เก็บข้อมูลต่าง ๆ ได้ตามกลยุทธ์ที่ระบุไว้ และอาจใช้ตำแหน่งที่เก็บข้อมูลต่าง ๆ สำหรับวัตถุประเภทต่าง ๆ โดยผลลัพธ์ที่ได้คือโมเดลโดเมนจะไม่ได้ผูกพันกับการจัดเก็บวัตถุหรือการอ้างอิงถึงวัตถุ และการเข้าถึงโครงสร้างพื้นฐานของการเก็บข้อมูลที่อยู่ภายใต้ระบบจัดเก็บข้อมูลก็จะไม่เกี่ยวข้องกับโมเดลโดเมนอีกต่อไป

****

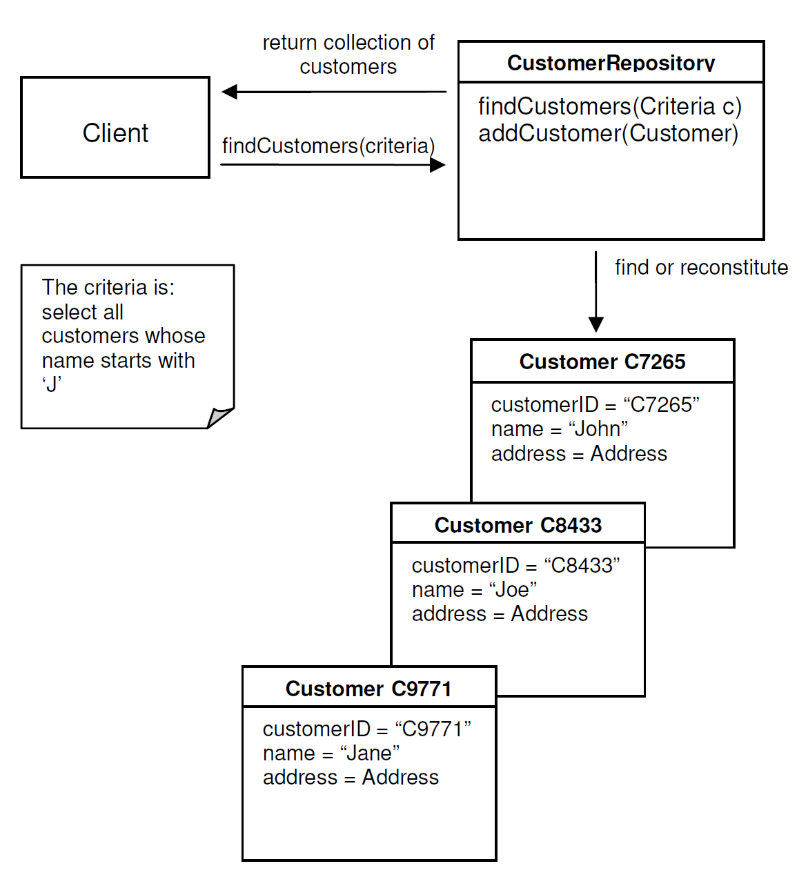
สร้างวัตถุสำหรับแต่ละประเภทของวัตถุที่ต้องการเข้าถึงทั่วโลก โดยวัตถุเหล่านี้จะสามารถให้ความเป็นจริงของการเก็บรวบรวมวัตถุทั้งหมดของประเภทนั้นๆในหน่วยความจำ ติดตั้งการเข้าถึงผ่านอินเทอร์เฟซโกลบอลที่รู้จักกันดี ให้เมทอดเพื่อเพิ่มและลบวัตถุ ซึ่งจะแยกแยะการแทรกหรือลบข้อมูลจริงๆในเก็บข้อมูล ให้เมทอดที่เลือกวัตถุตามเงื่อนไขใดๆ และส่งกลับวัตถุที่ถูกสร้างขึ้นเต็มรูปแบบหรือคอลเล็กชันของวัตถุที่มีค่าเฉพาะตามเงื่อนไข ซึ่งจะแยกแยะเทคโนโลยีการเก็บรักษาและค้นหาข้อมูลจริงๆ ให้เป็นส่วนที่แยกต่างหาก ให้ทำ Repository เฉพาะสำหรับ Aggregate roots เท่านั้นที่จริงๆต้องการเข้าถึงโดยตรง ให้ Client มุ่งเน้นไปที่โมเดล โดยมอบหมดหน้าที่ในการเก็บรักษาและเข้าถึงวัตถุให้กับ Repository

การจัดเก็บข้อมูล (Repository) อาจมีข้อมูลที่ละเอียดอ่อนเพื่อใช้ในการเข้าถึงโครงสร้างพื้นฐาน แต่อินเทอร์เฟซของการจัดเก็บข้อมูล ควรเป็นอย่างง่าย การจัดเก็บข้อมูล ควรมีชุดของเมธอดที่ใช้ในการดึงวัตถุออกมา ลูกค้าเรียกใช้เมธอดดังกล่าวและส่งพารามิเตอร์หนึ่งหรือมากกว่าซึ่งแทนเกณฑ์การเลือกวัตถุหรือชุดของวัตถุที่ตรงกับเกณฑ์การเลือก การระบุเอนทิตี้ (Entity) สามารถทำได้ง่ายโดยการส่งตัวระบุของมัน แต่เกณฑ์การเลือกอื่นๆ สามารถประกอบด้วยชุดของแอตทริบิวต์ของวัตถุได้ การจัดเก็บข้อมูล จะเปรียบเทียบวัตถุทั้งหมดกับชุดนั้นและจะส่งกลับเฉพาะวัตถุที่ตรงกับเกณฑ์การเลือก อินเทอร์เฟซของการจัดเก็บข้อมูลอาจมีเมธอดที่ใช้ในการดำเนินการคำนวณเสริมเช่นจำนวนวัตถุของประเภทที่กำหนดไว้

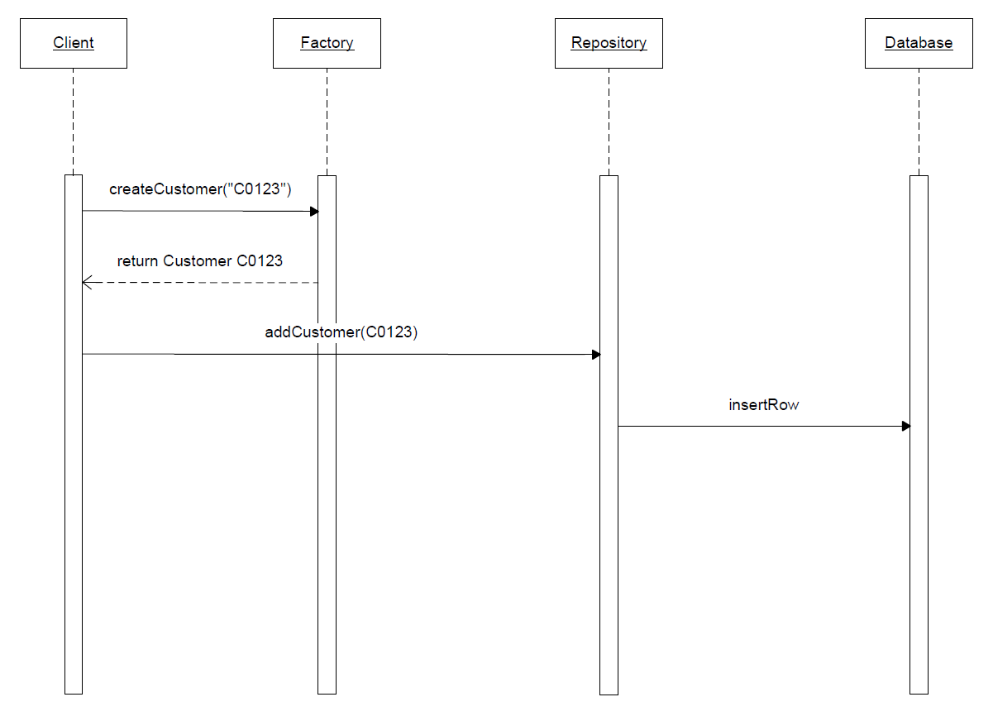
สามารถสังเกตได้ว่าการนำไปใช้งานของระบบจัดเก็บข้อมูล (repository) สามารถเชื่อมโยงกับโครงสร้างพื้นฐานได้อย่างใกล้ชิด แต่อินเตอร์เฟซของ repository จะเป็น domain model แบบบริสุทธิ์

****

ตัวเลือกอื่น ๆ คือระบุเกณฑ์การเลือกเป็น Specification โดย Specification ช่วยกำหนดเกณฑ์ที่ซับซ้อนมากขึ้น เช่นตัวอย่างต่อไปนี้:

****

มีความสัมพันธ์ระหว่าง Factory และ Repository ทั้งสองเป็นแบบแผนการออกแบบตามแบบจำลองและช่วยให้เราจัดการวัตถุโดเมนในระบบได้ดีขึ้น ซึ่ง Factory เน้นการสร้างวัตถุ ในขณะที่ Repository เป็นผู้ดูแลวัตถุที่มีอยู่แล้ว Repository อาจจะเก็บแคชวัตถุไว้ในระบบภายใน แต่บ่อยครั้งจะต้องดึงวัตถุมาจากการเก็บรักษาที่ถาวร วัตถุถูกสร้างขึ้นโดยใช้ constructor หรือถูกส่งไปยัง Factory เพื่อให้สร้างขึ้น ด้วยเหตุนี้ Repository อาจถูกมองเป็น Factory เพราะมันสร้างวัตถุ แต่มันไม่ใช่การสร้างจากศูนย์ แต่เป็นการนำวัตถุที่มีอยู่มาสร้างใหม่ เราไม่ควรผสม Repository กับ Factory โดย Factory ควรสร้างวัตถุใหม่ในขณะที่ Repository ควรค้นหาวัตถุที่มีอยู่แล้ว หากต้องการเพิ่มวัตถุใหม่ใน Repository จะต้องสร้างใหม่โดยใช้ Factory ก่อนแล้วส่งไปยัง Repository เพื่อจะเก็บเกี่ยวเป็นตัวอย่างด้านล่าง



วิธีหนึ่งที่สังเกตได้คือว่าโรงงาน ("Factories") เป็น "โดเมนบริสุทธิ์" ("pure domain") แต่ที่เก็บข้อมูลเกี่ยวข้องกับพื้นฐาน (Infrastructure) ได้แก่ฐานข้อมูล เป็นต้นอยู่ใน Repositories ได้

**4 - การ Refactoring สู่การมองเห็นที่ลึกซึ้งขึ้น**

**การปรับปรุงโค้ดอย่างต่อเนื่อง (Continuous Refactoring)**

จนถึงขณะนี้เราได้พูดถึงโดเมนและความสำคัญของการสร้างโมเดลที่แสดงถึงโดเมนนั้น และเราก็ให้คำแนะนำเกี่ยวกับเทคนิคที่จะใช้ในการสร้างโมเดลที่มีประโยชน์ โมเดลจะต้องเชื่อมโยงอย่างมั่นคงกับโดเมนที่มาจาก นอกจากนี้เรายังกล่าวไว้ว่าการออกแบบโค้ดต้องทำโดยฉะนั้นที่โมเดล และโมเดลเองควรได้รับการปรับปรุงตามการตัดสินใจในการออกแบบ การออกแบบโดยไม่มีโมเดลสามารถนำไปสู่ซอฟต์แวร์ที่ไม่ตรงตามโดเมนที่มันรับผิดชอบ และอาจไม่มีพฤติกรรมที่คาดหวัง การโมเดลโดยไม่มีการตอบกลับจากการออกแบบและโดยไม่มีผู้พัฒนาเข้าร่วมนั้นอาจทำให้โมเดลนั้นไม่เข้าใจได้อย่างดีโดยผู้ที่ต้องนำมันไปใช้ และอาจไม่เหมาะสมกับเทคโนโลยีที่ใช้งานอยู่ด้วย

ในขั้นตอนการออกแบบและพัฒนา จะต้องหยุดเป็นระยะๆ เพื่อมองไปที่โค้ดบ้าง อาจจะเป็นเวลาสำหรับการทำ Refactoring โค้ด การ Refactoring คือ กระบวนการออกแบบโค้ดใหม่เพื่อทำให้ดีขึ้นโดยไม่เปลี่ยนแปลงพฤติกรรมของแอปพลิเคชัน การ Refactoring มักจะทำด้วยขั้นตอนเล็กๆ ที่ควบคุมได้ง่าย โดยใช้ความระมัดระวังอย่างมากเพื่อไม่ทำลายฟังก์ชันหรือบั๊กที่เกิดขึ้น หลังจากทั้งหมดการทำ Refactoring เพื่อทำให้โค้ดดีขึ้น ไม่ใช่ทำให้แย่ลง การทดสอบอัตโนมัติเป็นประโยชน์อย่างมากเพื่อให้เรามั่นใจได้ว่าเราไม่ได้ทำให้บางสิ่งเสียหาย

มีหลายวิธีในการทำการ Refactoring โค้ด โดยมี Refactoring patterns อีกด้วย แบบแพตเทิร์นเหล่านี้แทนการทำ Refactoring โดยอัตโนมัติ มีเครื่องมือที่พัฒนาขึ้นบนแพตเทิร์นเหล่านี้ทำให้ชีวิตของนักพัฒนาง่ายขึ้นมากกว่าเดิม หากไม่มีเครื่องมือเหล่านี้ การ Refactoring อาจจะยากมาก การ Refactoring แบบนี้เกี่ยวข้องมากกับโค้ดและคุณภาพของมัน.

มีแบบอื่นของการ refactoring ที่เกี่ยวข้องกับโดเมนและโมเดลของมันด้วย บางครั้งมีความเข้าใจใหม่ในโดเมน สิ่งที่เป็นไปได้ก็คือมีความชัดเจนขึ้น หรือมีความสัมพันธ์ระหว่างองค์ประกอบสองตัวที่ถูกค้นพบ ทุกสิ่งที่เกี่ยวข้องควรถูกนำเข้าไปในการออกแบบโดยการ refactoring นั่นเป็นสิ่งสำคัญอย่างมากที่ต้องมีโค้ดที่ตั้งค่าได้ง่ายและอ่านง่าย จากการอ่านโค้ด ควรจะสามารถบอกได้ว่าโค้ดทำอะไร แต่ยังสามารถบอกได้ว่าทำไมโค้ดถึงทำเช่นนั้นด้วย เพียงเท่านั้นที่โค้ดจะสามารถจับสาระสำคัญของโมเดลได้อย่างแท้จริง

การ Refactoring เทคนิคที่ใช้รูปแบบ (patterns) เป็นหนึ่งในวิธีการจัดการและกำหนดโครงสร้างได้อย่างชัดเจน แต่การ Refactoring โดยการเข้าใจลึกลงไปในเนื้อหาไม่สามารถทำได้ด้วยวิธีเดียวกัน เราไม่สามารถสร้างรูปแบบสำหรับมันได้ เนื่องจากความซับซ้อนของแบบจำลองและความหลากหลายของแบบจำลองที่ไม่ให้เรามีโอกาสในการจัดการแบบกลโกง (mechanistic) โมเดลที่ดีเกิดจากการคิดอย่างลึกซึ้ง มีความเข้าใจลึกลง ประสบการณ์ และความชำนาญในการสร้าง ณ ที่สุด

หนึ่งในสิ่งแรกที่เราถูกสอนเกี่ยวกับการจำลองคือการอ่านข้อกำหนดธุรกิจและค้นหาคำนามและคำกริยา คำนามจะถูกแปลงเป็นคลาสในขณะที่คำกริยากลายเป็นเมธอด นี่เป็นกระบวนการที่เรียบง่ายและจะนำไปสู่โมเดลที่มีความเป็นพื้นที่เพียงพอ โมเดลทุกตัวขาดความลึกในเริ่มต้น แต่เราควรปรับโมเดลให้เข้าใจลึกขึ้นไปเรื่อยๆ

การออกแบบต้องเป็นอย่างยืดหยุ่น การออกแบบที่แข็งแรงจะต้านการทำรหัสใหม่ โค้ดที่ไม่ได้ถูกสร้างขึ้นโดยคิดถึงความยืดหยุ่นจะทำให้มันยากต่อการใช้งาน ในกรณีที่ต้องการเปลี่ยนแปลง คุณจะเห็นว่าโค้ดกำลังต่อสู้กับคุณ และสิ่งที่ควรถูกทำรหัสใหม่ได้อย่างง่ายดายกลับใช้เวลามาก

การใช้ชุดของบล็อกพื้นฐานที่ได้รับการพิสูจน์แล้วพร้อมกับการใช้ภาษาที่สม่ำเสมอ จะช่วยเพิ่มความเป็นสมดุลในการพัฒนา แต่ก็ยังมีความท้าทายในการค้นหาแบบจำลองที่มีความฉลาดเฉียบพลัน ซึ่งสามารถจับประเด็นที่ซับซ้อนของผู้เชี่ยวชาญด้านและสามารถนำไปสู่การออกแบบที่เหมาะสมได้ แบบจำลองที่สามารถละเลยสิ่งที่ไม่จำเป็นและจับต้องสิ่งที่สำคัญเป็นแบบลึก นั่นจะทำให้ซอฟต์แวร์มีความเข้ากันได้กับวิธีการคิดของผู้เชี่ยวชาญด้านมากขึ้นและตอบสนองความต้องการของผู้ใช้ได้ดียิ่งขึ้น

ตามแบบฉบับดั้งเดิม การ Refactoring จะถูกอธิบายในเชิงของการเปลี่ยนแปลงรหัสด้วยแรงบันดาลใจทางเทคนิค อย่างไรก็ตาม การ Refactoring ยังสามารถได้รับแรงบันดาลใจจากความเข้าใจในโดเมนและการเขียนรหัสตามนั้น และการปรับปรุงโมเดลหรือแสดงในรหัสอาจเกิดขึ้นได้ตามได้ยิ่งขึ้นกับความเข้าใจในโดเมนนั้นๆ

โมเดลโดเมนที่ซับซ้อนนัก จะไม่พัฒนาขึ้นแบบเดียวกัน ยกเว้นผ่านกระบวนการซ้ำไปซ้ำมาในการปรับปรุงโครงสร้าง ซึ่งรวมถึงการมีความสัมพันธ์ใกล้ชิดระหว่างผู้เชี่ยวชาญในด้านนั้น และนักพัฒนาที่สนใจเรียนรู้เกี่ยวกับด้านนั้นด้วย

**นำแนวคิดหลักมาเปิดเผย**

การ Refactoring ทำด้วยขั้นตอนเล็ก ๆ รวมกันเป็นการปรับปรุงขนาดเล็ก ๆ ตามลำดับ มีช่วงเวลาที่การเปลี่ยนแปลงขนาดเล็กมากๆ ไม่เพิ่มคุณค่าใด ๆ ในการออกแบบ และมีช่วงเวลาที่การเปลี่ยนแปลงขนาดเล็กน้อยก็สามารถทำให้เกิดความแตกต่างได้มาก ๆ เป็นการฝ่าอุปสรรค(Breakthrough)

เราเริ่มต้นด้วยโมเดลที่เป็นรูปพื้นผิว จากนั้นเราปรับปรุงและออกแบบใหม่โดยใช้ความรู้ลึกลงเกี่ยวกับด้านต่างๆ และเข้าใจปัญหาได้ดียิ่งขึ้น เราเพิ่มแนวคิดใหม่และการนำมาใช้เพื่อสร้างความสะดวกสบาย และเรียกใช้การออกแบบอีกครั้ง การปรับปรุงแต่ละครั้งจะช่วยเพิ่มความชัดเจนให้กับการออกแบบ และสร้างพื้นฐานสำหรับการฝ่าอุปสรรคใหม่ๆอีกต่อไป

การฝ่าอุปสรรค (Breakthrough) มักเกิดจากการเปลี่ยนแปลงวิธีการคิด วิธีการมองโมเดลที่เรามีอยู่แล้ว นี่เป็นแหล่งของความคืบหน้าที่ใหญ่โตในโครงการ แต่มันยังมีข้อเสียบ้าง เมื่อมี Breakthrough เกิดขึ้น มันอาจแสดงให้เห็นว่าต้องมีการเรียกใช้ทรัพยากรและเวลาในการปรับปรุงใหม่นั่นหมายความว่าเราไม่เคยมีเวลาเพียงพอเสมอไป นอกจากนี้ยังเสี่ยงต่อความไม่สมดุลย์เนื่องจากการปรับปรุงใหม่อาจเปลี่ยนแปลงพฤติกรรมของแอปพลิเคชันได้

เพื่อที่จะได้ถึงจุดที่เรียกว่า "Breakthrough" เราต้องทำให้แนวคิดที่ถูกเลี่ยงไปนั้นเป็นแนวคิดที่ชัดเจนขึ้น เมื่อเราพูดคุยกับผู้เชี่ยวชาญในสาขานั้น ๆ เราจะแลกเปลี่ยนความคิดและความรู้กันอย่างมากมาย บางคำศัพท์อาจจะถูกนำเข้าไปในภาษาสากล(Ubiquitous Language)แล้ว แต่บางคำอาจจะยังไม่ได้รับการสังเกตเห็นเลยตอนเริ่มต้น คำเหล่านี้เป็นแนวคิดที่ถูกเลี่ยงไปและถูกใช้เพื่ออธิบายแนวคิดอื่น ๆ ที่อยู่ในโมเดลแล้ว ในขั้นตอนการออกแบบที่เรียบร้อยแล้ว บางคำเหล่านี้อาจจะดึงดูดความสนใจของเรา และเราพบว่าบางคำเหล่านี้เป็นส่วนสำคัญในการออกแบบ ในจุดนั้นเราควรทำให้แนวคิดดังกล่าวเป็นแนวคิดที่ชัดเจนขึ้น โดยการสร้างคลาสและความสัมพันธ์สำหรับคำเหล่านี้ เมื่อเกิดเหตุการณ์ดังกล่าวขึ้น เราอาจจะมีโอกาสได้ถึงจุด "Breakthrough" ก็เป็นได้

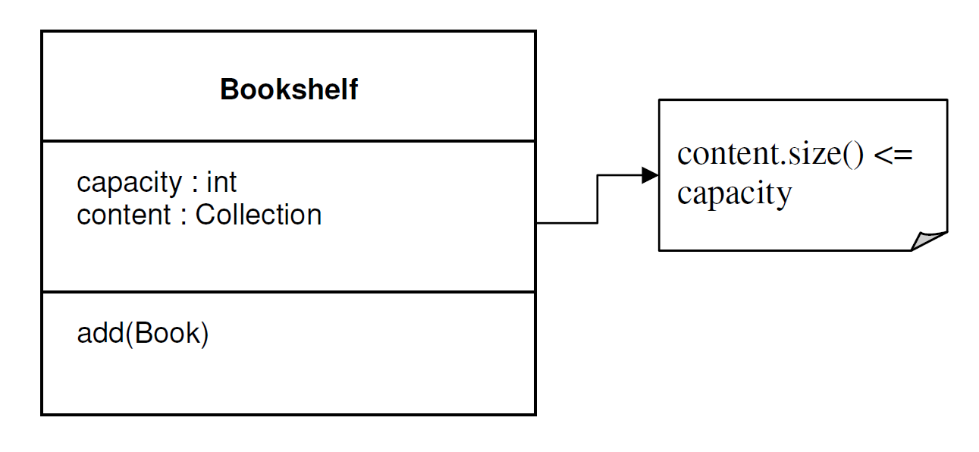
แนวคิดที่ไม่ได้ระบุโดยชัดเจนไม่ควรเป็นเช่นนั้นต่อไป หากเป็นแนวคิดในโดเมนที่เกี่ยวข้อง ควรมีอยู่ในโมเดลและการออกแบบ วิธีการรู้จักแนวคิดที่ไม่ได้ระบุโดยชัดเจนคืออะไร? วิธีแรกในการค้นพบแนวคิดที่ไม่ได้ระบุชัดเจนคือการฟังภาษา ภาษาที่เราใช้ในการจำลองและออกแบบมีข้อมูลมากมายเกี่ยวกับโดเมน อย่างไรก็ตามเริ่มต้นอาจจะไม่มีมากนักหรือบางส่วนของข้อมูลอาจจะไม่ถูกใช้ถูกต้อง บางแนวคิดอาจจะไม่เข้าใจอย่างเต็มที่หรืออาจจะเข้าใจผิดพลาดอย่างสมบูรณ์ นี่เป็นส่วนหนึ่งของการเรียนรู้โดเมนใหม่ แต่เมื่อเราสร้าง Ubiquitous Language ของเราแล้ว แนวคิดสำคัญจะเข้ามาอยู่ในนั้น นี่คือสิ่งที่เราควรเริ่มต้นมองหาแนวคิดที่ไม่ได้ระบุโดยชัดเจน

บางส่วนของการออกแบบอาจจะไม่ชัดเจน มีชุดความสัมพันธ์ที่ทำให้เส้นทางการคำนวณยากต่อการติดตาม หรือกระบวนการที่ทำอยู่ซับซ้อนและยากในการเข้าใจ นี่เป็นความลำบากในการออกแบบ นี่เป็นสถานที่ที่ดีที่จะหาแนวคิดที่ซ่อนอยู่ อาจจะขาดบางอย่าง หากแนวคิดสำคัญขาดไปจากปริศนา อย่างอื่นจะต้องแทนที่ฟังก์ชันของมัน นี้จะทำให้เพิ่มความหนาของวัตถุบางอย่างโดยเพิ่มพฤติกรรมที่ไม่ควรมีอยู่ ความชัดเจนในการออกแบบจะเสียหาย พยายามดูว่ามีแนวคิดที่ขาดหายไปหรือไม่ หากพบแนวคิดที่ขาดหายไปให้ทำให้มันเป็นชัดเจน แก้ไขการออกแบบให้เป็นรูปแบบที่ง่ายและยืดหยุ่นมากขึ้น

ในการสร้างความรู้อาจพบเจอการขัดแย้งกันได้ สิ่งที่ผู้เชี่ยวชาญด้านหนึ่งกล่าวอาจจะขัดแย้งกับสิ่งที่ผู้เชี่ยวชาญด้านอื่นยืนยันอยู่ ความต้องการบางอย่างอาจจะดูเหมือนขัดแย้งกัน บางข้อขัดแย้งนั้นอาจไม่ใช่ขัดแย้งจริง ๆ แต่เป็นวิธีการมองเห็นสิ่งเดียวกันในมุมต่าง ๆ หรืออาจเป็นเพราะขาดความแม่นยำในการอธิบาย พวกเราควรพยายามประสานความขัดแย้งกัน บางครั้งการทำเช่นนี้อาจนำเสนอแนวคิดที่สำคัญ แม้ว่าจะไม่ได้เป็นเช่นนั้น แต่ยังคงเป็นสิ่งสำคัญที่จะเก็บรักษาความชัดเจนในทุก ๆ เรื่อง

วิธีหนึ่งที่ชัดเจนในการค้นหาแนวคิดของโมเดลคือใช้วรรณกรรมด้านสาขาวิชา เพราะมีหนังสือที่เขียนเกี่ยวกับหัวข้อเกือบทุกรูปแบบ และเนื้อหาในหนังสือนั้นมีความรู้มากเกี่ยวกับสาขาวิชาที่เกี่ยวข้อง อย่างไรก็ตาม หนังสือไม่มีโมเดลสำหรับสาขาวิชาที่นำเสนออยู่ ต้องมีการดำเนินการประมวลผล เหลี่ยมเล็กเพื่อนำข้อมูลที่มีอยู่ไปกรอบคิดและปรับปรุง แต่ข้อมูลที่พบในหนังสือมีค่าเป็นที่สูง และให้มุมมองลึกซึ้งเกี่ยวกับสาขาวิชานั้นๆ

มีแนวคิดอื่นๆ ที่มีประโยชน์มากเมื่อถูกกำหนดโดยชัดเจน: Constraint, Process และ Specification โดย Constraint เป็นวิธีที่ง่ายๆ ในการแสดงอินวารีแบบไม่เปลี่ยนแปลง ไม่ว่าเกิดอะไรขึ้นกับข้อมูลวัตถุ อินวารียังคงเป็นไปตามปกติ การทำงานนี้สามารถทำได้โดยการนำโลจิกของอินวารีเข้าไปใน Constraint ต่อไปเป็นตัวอย่างที่ง่าย วัตถุกำหนดให้เป็นจำนวนเต็มและต้องไม่เป็นลบ ดังนั้น Constraint สามารถกำหนดได้ว่าค่าของวัตถุจะต้องมีค่าไม่ต่ำกว่าศูนย์ และ Process ก็เป็นวิธีการเพื่อให้มั่นใจได้ว่าโปรแกรมทำงานตามที่ต้องการ ส่วน Specification เป็นวิธีการใช้คำกำหนดที่ชัดเจนเพื่อกำหนดการทำงานของระบบให้เป็นไปตามที่ต้องการโดยไม่ต้องคำนึงถึงวิธีการทำงานดังกล่าวของระบบ การอธิบายนี้เพื่ออธิบายแนวคิด ไม่ได้แสดงถึงวิธีการทำในกรณีที่คล้ายกัน

****

เราสามารถเพิ่มหนังสือลงในชั้นหนังสือได้ แต่เราไม่ควรเพิ่มมากกว่าความจุของมัน สามารถมองเห็นได้ว่าเป็นส่วนหนึ่งของพฤติกรรมของชั้นหนังสือ เช่นเดียวกับในโค้ด C# ต่อไปนี้

public class Bookshelf

{

private int capacity = 20;

private IList<Book> content;

public void Add(Book book)

{

if (content.Count() + 1 <= capacity)

{

content.Add(book);

}

else

{

throw new InvalidOperationException("The bookshelf has reached its limit.");

}

}

}

เราสามารถ Refactor code โดยการแยก constraint ออกเป็นเมธอดที่แยกต่างหากได้

public class Bookshelf

{

private int capacity = 20;

private IList<Book> content;

public void Add(Book book)

{

if (!isSpaceAvailable())

throw new InvalidOperationException("The bookshelf has reached its limit.");

content.Add(book);

}

private bool isSpaceAvailable => content.Count < capacity;

}

การเรียกใช้เงื่อนไขโดยแยกออกเป็นเมธอดที่แตกต่างกัน มีข้อดีที่ทำให้มันเป็นเรื่องชัดเจน ง่ายต่อการอ่านและทุกคนจะสังเกตได้ว่าเมธอด Add() อยู่ภายใต้เงื่อนไขนี้ ยังมีพื้นที่สำหรับการเติบโตโดยเพิ่มตัวตนของเมธอดหากเงื่อนไขกลายเป็นซับซ้อนมากขึ้นด้วย

กระบวนการจะถูกแสดงออกมาในรูปแบบของโค้ดแบบ procedural โดยปกติแล้ว แต่เนื่องจากเรากำลังใช้ภาษาเชิงวัตถุ ดังนั้นเราจะต้องเลือกวัตถุสำหรับกระบวนการและเพิ่มพฤติกรรมให้กับวัตถุนั้น วิธีการที่ดีที่สุดในการดำเนินกระบวนการคือการใช้บริการ (Service) หากมีวิธีการดำเนินกระบวนการที่แตกต่างกัน เราสามารถกลบกลุ่มอัลกอริทึมในวัตถุและใช้กลยุทธ์ (Strategy) ไม่ใช่กระบวนการทั้งหมดที่ควรทำให้เป็นโปรแกรมอย่างชัดเจน หากภาษาแบบ Ubiquitous กล่าวถึงกระบวนการที่เกี่ยวข้องอย่างชัดเจน จะถึงเวลาที่จะต้องมีการนำเสนอกระบวนการโดยชัดเจน

วิธีสุดท้ายในการทำให้แนวคิดโดดเด่นที่เรากำลังพูดถึงที่นี่คือการกำหนดข้อกำหนด กล่าวง่ายๆ กำหนดข้อกำหนดถูกใช้เพื่อทดสอบวัตถุเพื่อดูว่ามันประสบความสำเร็จตามเกณฑ์บางอย่างหรือไม่

ชั้นข้อมูล (domain layer) ประกอบด้วยกฎธุรกิจที่นำไปใช้กับ Entity และ Value Object กฎเหล่านั้นมักถูกรวมเข้าไว้ในวัตถุที่นำไปใช้งาน บางกฎเป็นชุดคำถามที่ต้องตอบ "ใช่" หรือ "ไม่ใช่" กฎเหล่านี้สามารถแสดงผ่านชุดของการดำเนินการตรรกะบนค่าบูลีน และผลลัพธ์สุดท้ายก็เป็นค่าบูลีนด้วย ตัวอย่างหนึ่งคือการทดสอบบนวัตถุลูกค้าเพื่อดูว่ามีสิทธิ์ได้รับเครดิตบางประเภทหรือไม่ กฎนี้สามารถแสดงผลเป็นเมธอดที่มีชื่อว่า IsEligible() และสามารถแนบไปยังวัตถุลูกค้าได้ แต่กฎนี้ไม่ใช่เพียงเมธอดที่ดำเนินการอย่างเดียวบนข้อมูลของลูกค้า มันยังได้ดำเนินการอื่นๆอีกด้วย การประเมินกฎนี้เกี่ยวข้องกับการตรวจสอบข้อมูลประวัติของลูกค้า เช่นการตรวจสอบว่าเคยผ่อนชำระหนี้ในอดีตหรือไม่ การตรวจสอบว่ามียอดค้างชำระหรือไม่ เป็นต้น กฎธุรกิจเช่นนี้อาจมีขนาดใหญ่และซับซ้อนมากขึ้นจนถึงจุดที่ไม่สามารถใช้งานในการแก้ปัญหาเดิมได้อีกต่อไป ในขณะนี้เราอาจจะลองย้ายกฎทั้งหมดไปยังระดับแอปพลิเคชัน เนื่องจากดูเหมือนว่ามันเกินขอบเขตของโดเมน แต่จริงๆ แล้ว นั่นคือเวลาสำหรับการ Refactoring แล้ว

กฎควรถูกบรรจุในวัตถุของตัวเองที่เรียกว่า Specification ของลูกค้า และควรถูกเก็บไว้ในชั้นด้านโดเมน (domain layer) วัตถุใหม่นี้จะมีเมธอดบูลีนต่างๆ ที่ทดสอบว่าวัตถุลูกค้าบางอย่างมีสิทธิ์ได้รับเครดิตหรือไม่ แต่ละเมธอดจะเป็นการทดสอบเล็กๆ และการรวมกันของทุกเมธอดจะให้คำตอบกับคำถามเดิม หากกฎธุรกิจไม่ได้รวมอยู่ในวัตถุ Specification เดียวกัน รหัสที่เกี่ยวข้องก็จะแพร่กระจายอยู่ในหลายวัตถุ ซึ่งจะทำให้ไม่สอดคล้องกัน

ข้อกำหนด (Specification) ถูกใช้สำหรับทดสอบวัตถุเพื่อดูว่ามันตอบโจทย์ความต้องการหรือพร้อมสำหรับการใช้งานในวัตถุประสงค์ใดๆ นอกจากนี้ยังสามารถใช้เพื่อเลือกวัตถุบางอย่างจากคอลเล็กชัน หรือเป็นเงื่อนไขในระหว่างการสร้างวัตถุด้วย

บ่อยครั้งแล้ว Specification เดียวจะตรวจสอบว่ากฎง่าย ๆ ถูกต้องหรือไม่ แล้วจึงรวม Specification เหล่านั้นเข้าด้วยกันเป็นหนึ่งเพื่อแสดงกฎที่ซับซ้อนขึ้น อย่างเช่นนี้:

Customer customer = customerRepository.FindCustomer(customerIdentiy);

…

Specification customerEligibleForRefund = new Specification(

new CustomerPaidHisDebtsInThePast(),

new CustomerHasNoOutstandingBalances());

if (customerEligibleForRefund.IsSatisfiedBy(customer)

{

refundService.IssueRefundTo(customer);

}

การทดสอบกฏง่ายๆ เป็นเรื่องที่ง่ายกว่า และเพียงแค่อ่านโค้ดนี้ ก็จะเข้าใจได้ชัดเจนว่าลูกค้ามีสิทธิ์ได้รับการคืนเงินคืนหรือไม่

**5 – การสงวนความถูกต้องของโมเดล**