คำตอบ SAD Final

2.

ไม่เหมาะสม เพราะ การที่คลาส X เป็น base class และมี method ถูกประกาศเอาไว้ และมี คลาส Y มา extend ต่อ แต่ไม่ได้ใช้งานใน method ที่คลาส x มาสร้างเอาไว้ จะ violate SOLID ในหัวข้อ SRP เนื่องจากเราสามารถแยกโค้ดที่ implement ในส่วนคลาส X ให้สามารถแยกไปอยุ่ในคลาส Y ได้ โดยคอนเซปที่ว่า "Gather together things that change for the same reason at the same times"

3.

ละเมิด SRP ในการรวมโค้ดที่ไม่เกี่ยวข้องกันไว้ด้วยกัน จะส่งผลให้ cohesion ต่ำ (คือพันธะสิ่งที่ควรจะอยู่ด้วยกัน ต่ำ [ยิ่งต่ำยิ่งไม่ดี]) และ coupling สูง (คืออะไรที่ไม่ควรจะอยู่ด้วยกัน แต่มีความเกี่ยวโยงกัน สูง [ยิ่งสูงยิ่งไม่ดี])

และการละเมิด SRP โดยแยกโค้ดออกไปหลายคลาสโดยไม่จำเป็นและแก้ไข code สม่ำเสมอ จะส่งผลให้เกิด design smel ในหัวข้อคือ needless repetition จะเห็นได้ชัดสุด เพราะเราจะเจอโค้ดซ้ำๆ หลายๆที่ วิธีแก้ก็คือ รวมโค้ดที่เหมือนกันให้อยู่ที่เดียวในรูปแบบ abstraction เวลาโค้ดอยู่ที่เดียวกัน ก็จะลดความซ้ำซ้อนไปได้

และมีโอกาสที่จะเกิด Rigidity และ Fragility ตามมา ในเรื่อง Rigidity คือ เวลาที่เราแก้ไขอะไร จะแก้ไขได้ยาก ต้องแก้โค้ดชุดเดิมๆ หลายๆครั้ง และ fragility ก็คือมีโอกาสแตกหักง่าย ในเรื่องของถ้าเรา modify บ่อยๆ ก็มีโอกาสที่เราจะแก้โค้ดชุดเดียวกันไม่เหมือนกันได้ ก็อาจจะเกิดโอกาสนี้เกิดขึ้นได้

4.

Immobility คือ ความยากที่จะเอาไปใช้งานซ้ำๆ ทั้งในโปรเจคเดียวกันหรือโปรเจคอื่นๆ อารมณ์ Hardcode มากๆ -> วิธีการแก้ มี 2 วิธีคือ แก้โดยเขียนใหม่ไปเลย ไม่ต้องเอาไปใช้ซ้ำ หรืออีกวิธีก็คือ แก้โค้ดเก่า ให้สามารถทำ repetition กับส่วนอื่นๆได้ ก็จะแก้ปัญหานี้ได้

DIP คือหลักการใน SOLID เกี่ยวกับเรื่อง High level class (คลาสที่เป็นแค่คนสั่งคร่าวๆว่าจะมีอะไรทำอยู่ในนั้นบ้าง) ไม่ควรไป dependent กับ low level class (คลาสที่มีรายละเอียดการ implement จริงๆ) ซึ่งหลักการนี้ แก้โดยการที่ ให้ สร้าง Abstraction class ขึ้นมา (interface หรือ abstraction) แล้วให้ low level กับ high level ไป dependent on abstraction class แทน โดยวิธีการนี้ จะทำให้ เวลาจะเพิ่ม low level ก็สามารถเพิ่มได้ง่ายขึ้น และ high level กับ low level ก็ไม่ผูกติดยึดโยงกันไปเหมือนเก่า

จึงสามารถแก้ปัญหาในส่วน immobility ได้ ตรงจังหวะการใช้งานซ้ำ โค้ด high level ก็จะมีการประกาศเป็น abstraction ไว้ (เป็น Polymorphism) ทำให้สามารถใช้ high level คลาสเดิม กับหลายๆ low level ที่เพิ่มเข้ามาที่หลัง ผ่านจากการ implement abstraction ไว้ได้

ในส่วน reusability จะเป็นจังหวะ abstraction ก็คือ อะไรที่เหมือนกัน ก้สามารถ รวมโค้ดในจุดๆเดียวกันได้ ทำให้ไม่เกิด repeatation โค้ด และสามารถใช้ high level กับหลายๆ low level ที่เข้ามาได้ เหมือนที่กล่าวไว้ข้างต้น

5.

Viscosity คือ ความหนืด มี 2 รูปแบบคือ Software และ environment

ในทาง software คือ ตอนแรกมีการวาง design ไว้อยู่แล้ว แต่ถ้าเกิดเราทำไปเรื่อยๆ พบว่า มีวิธีการที่ง่ายกว่า หรือเจอทางลัด แล้วถ้าเราใช้ทางลัด จะทำให้ Design ที่ออกแบบมาเสียไป viscosity จะเยอะขึ้น (ยกตัวอย่างเช่น การแก้หลายไฟล์ เลยพยายามลดจำนวนไฟล์ หรือรวมโค้ดบางอย่างให้แก้ง่ายขึ้น แต่เสีย design pattern ไป)

ในทาง environment คือทำให้ developer เกิดความยากในการทำงาน ไม่ว่าจะเรื่องช้า ไม่มีประสิทธิภาพ compile file นาน เลยเกิดความหนืด ทำให้ บางที dev ก็ไม่ได้ complie file ทั้งหมด เพราะเสียเวลา แต่ถ้าไม่ complie ทั้งหมด ก็มีโอกาสที่ software จะมีโอกาส error ได้

การละเมิด ISP ในส่วน viscosity เพราะว่า ถ้าเรามี interface และมี method ที่บังคับการ implement เยอะๆ ตามรูปแบบการ design จะทำให้ถ้าเกิดเหตุการณ์ที่เราไม่ได้ใช้ method นั้นๆ เราก็จะต้อง implement ว่างเปล่า หรือไม่มีประโยชน์ ซึ่งทำแบบนี้สุดท้าย ก็อาจจะเป็นการตัด method บางอย่างของ interface ออกไป ทำให้เสีย design เดิมที่มีอยู่ เพราะบางคลาสที่ implement อื่นๆ ก็อาจจะต้อง implement บาง method ที่จะเป็นต้องตัดทิ้งไปด้วย และในส่วนของ Environment ก็คือ ถ้าเกิดมีการแก้ตัว interface เยอะๆ ก็อาจจะทำให้เกิดการ complie ใหม่ทั้งหมดของตัวที่มา implement interface นี้ด้วยเช่นกัน

6.

จะละเมิด OCP แบบชัดมากๆ เพราะว่า OCP คือการที่เราเน้น "เพิ่ม/ต่อเติม" และ "ห้ามแก้ของเก่าที่ดีอยู่แล้ว" ใน Factory pattern ก็คือ การที่เราจะเพิ่ม type ใหม่เข้าไป จะต้องไปเพิ่มใน switch case ของ NotificationFactory (นั้นก็คือไปแก้ของเก่าที่ดีอยู่แล้ว)

วิธีแก้แก้ก็คือเปลี่ยนจาก Factory Pattern เป็น Factory Method แทน โดยการที่ เราจำเป็นต้องสร้าง abstraction method เอาไว้ในคลาส NotificationFactory และถ้าเกิดว่าจะมี type ใหม่ๆ ที่เพิ่มเข้ามา ก็จะ extend จาก factory และออกเป็น subclass ใหม่เป็นลูกๆ และ implement ส่วนที่ทำ abstraction method โดยการ return new type นั้นๆกลับไปเป็น obj ใหม่เลย จะทำให้ ไม่เกิดการละเมิด OCP เพราะจะเห็นได้ว่า ถ้าเกิดมี type เพิ่มขึ้นเข้ามาใหม่ ก็จะสามารถ subclass ออกมา โดยไม่ต้องไปแก้ไข Notification Factory เลย

7.

8.

Builder เพราะ Builder กับ Template มีความคล้ายกันมาก ในส่วน Builder ก็คือจะเป็นการสร้าง Step 1,2,3,4 เหมือน Template เลย โดย Builder จะกำหนดว่า จะมี step อะไรบ้าง และให้ concrete builder ต่างๆ ไป implement interface ตัวนี้ โดยให้มี step การทำงานที่เป็นโครงเหมือนกัน แต่สามารถเปลี่ยนไส้ในได้ (ยกตัวอย่างเช่น การสร้างบ้าน เราสามารถมี step สร้างบ้านได้เหมือนกัน โดยที่ วัสดุการทำบ้าน ไม่จำเป็นต้องเหมือนกันก็ได้ เช่น ไม้ อิฐ ปูน) กลับมาที่ Template ก็เหมือนกัน โดยมี step บอกให้ทำอะไรก่อนหลัง และมีการ implement method ในแต่ละ step โดยที่แตกต่างกันก็คือ builder จะเป็นเน้นเรื่องของการจัดการ การสร้าง obj ต่างๆ ซึ่งจะได้ outcome ออกมาหนังจากที่มีการต่อเติม step ไปเรื่อยๆในตอนสุดท้าย แต่ใน template ก็เกี่ยวกับเรื่องพฤติกรรมของโค้ด โดยที่ template จะเห็นได้ชัดคือ เราสามารถ implement abstraction หรือ step ไว้ก่อนหน้านั้นได้ (reuse code) และสามารถ override step เหล่านั้นใหม่ได้ในคลาสลูกๆ ซึ่งจะเห็นได้ชัดมากใน framework ที่เราจะต้องไป implement ใน template method เพื่อให้ framework มาเรียกใช้โค้ดที่เราเขียนลงไป

9.

Composite มีความคล้ายครึงกัน ในเรื่อง ถ้าเป็น composite ก็คือจะเป็นรูปแบบ Tree hierarchy โดยจะมี leaf node ที่เป็นตัว implement งานจริงๆ และมี compound class ที่เป็นตัวรวบรวม node เหล่านี้เพื่อ sum up result ของตัว nodeๆ ต่างๆ ให้ออกมาเป็นผลลัพท์ใหม่ เช่นเดียวกันกับ Decorator Pattern คือมีโครงสร้างเป็นการห่อหุ้ม ความสามารถเพิ่มเข้าไป โดยตอนแรก อาจจะมีแค่ base bahavior และถ้าเราต้องการให้เพิ่มความสามารถ ก็ทำการห่อความสามารถเข้าไปใหม่ได้ โดยความสามารถก็จะเพิ่มขึ้น จุดที่เหมือนกัน น่าจะเป็นตรงเพิ่มความสามารถนี่แหละ (composite จะเป็นตรง compound class และ decorator ก็จะเป็นเรื่องการห่อหุ้ม) ส่วนที่มันต่างกัน น่าจะเป็น structure ของแต่ละอัน ก็คือ composite เป็น tree (สามารถแตกแขนงได้) แต่ถ้า decorator คือเส้นๆ เดียว แล้วซ้อนความสามารถกันไปเรื่อยๆเฉยๆ

10.

Fly weight ใน Structural Design Pattern สามารถลดหน่วยความจำได้ (ลด RAM) อธิบายแนวคิด ก็คือ สมมุติ เรามีคลาสๆหนึ่ง ที่รวบรวมทุกอย่างเอาไว้ โดยคลาสนี้ เวลาสร้าง จะหนัก Memory มากๆ เราจึงทำการแยก class นี้ออกเป็น 2 อย่างหลักๆ คือ Extrinsic (Unique มีได้อันเดียวเท่านั้น) และ Intrinsic (Reuse ซ้ำได้) ยกตัวอย่างง่ายๆ เหมือนทำเกมส์แล้วกัน สมมุติ เกมๆนี้มีกระสุนเยอะมากๆ ซึ่งถ้าเรามีคลาสกระสุนอันเดียว เวลาสร้างเป็นล้านๆกระสุน จะกิน RAM เยอะ เราเลยแยก 1. ชุดรูปภาพกระสุน (เป็น Intrinsic reuse ซ้ำได้) และ 2. พิกัดกระสุน (Unique มีได้แค่อันเดียว) เวลาเราสร้าง เราก็สร้างแค่พิกัดก็พอ สร้างทุกๆพิกัดสำหรับทุกกระสุน และ ภาพกระสุน (ที่เป็นไฟล์ใหญ่ๆ) ก็จะสร้างแค่อันเดียว แต่ใช้สำหรับทุกกระสุนที่สร้างขึ้นมา แนวทางนี้จะทำให้ประหยัด RAM ได้ นี่เป็นคอนเซปแนวคิดแล้วก็วิธีการแก้เบื้องต้น

11.

ปัญหาก็คือ ถ้าเกิดเราสร้าง Button หลายๆอัน หรือ shortcut หลายๆอัน (UI หรือการเข้าถึงปุ่ม) และเรามีโค้ด logic ชุดเดียวกัน (business logic) ถ้าเราทำแบบปกติ ก็คือ ตัว UI จะมีการยึดโยงกับ business logic มากเกินไป โดยจะเห็นได้ว่า ทุกๆ UI จะต้องชี้ไปที่ business logic เหล่านี้เพื่อเรึยกใช้ จึงเกิดปัญหา เพราะจะเกิดความซ้ำซ้อนของโค้ด เนื่องจาก business logic ก็ควรมีโค้ดชุดเดียว ที่ไม่ว่าจะเข้าถึงผ่าน ปุ่ม หรือ shortcut ก็ยังต้องใช้ logic ชุดนี้อยู่

วิธีการปรับแก้คือ ใช้ Command Design Pattern แนวคิดก็คือ ถ้า UI กับ ตัว business logic มันยึดโยงกันมากเกินไป ก็หาจุดร่วมกัน โดยการสร้าง Obj หนึ่งขึ้นมา (เรียกว่า Command Object) เพื่อที่จะให้ UI หรือการเข้าถึงปุ่มทุกตัว เรียกใช้ตัว Command Obj ขึ้นมาก่อน และตัว Command Obj จึงค่อยไปเรียกใช้โค้ดส่วน Business Logic อีกทีหนึ่ง การทำแบบนี้จะลดการยึดโยงกันระหว่างส่วน UI และ business logic โค้ดได้ ทำให้เวลาแก้ไขก็จะแก้ไขได้ง่าย ไม่เกี่ยวโยงกัน

12.

ใช้ Composite Pattern จุดสังเกตคือ ในคลาส CursorList จะเป็นเหมือน Compound Class ที่เป็นตัวรวบรวม Cursor ต่างๆ เช่น EntryFilteringCursorImpl เป็นต้น โดยจะมีจุดอย่างเช่น การเก็บ list: List<EntryFilteringCursor> ไว้ในคลาส CursorList และมีการ addEntryFilter รวบรวมอยู่ใน list เพื่อเอาไปประมวลผลอะไรบางอย่างต่อไป ซึ่งนี้จะเป็นจุดสังเกตของ composite Pattern

ส่วนการใช้เพื่ออะไร โดย EntryFilteringCursor คาดว่าเป็นการเก็บแบบเป็น Tree Hierarchy โดยการใช้ครั้งนี้ ก็ใช้เพื่อรวบรวม EntryfilterCursorImpl มารวมกันเพื่อเอาไปประมวลผลอะไรบางอย่างต่อไป

13

Observer เพื่ออะไร ในการสร้างครั้งนี้ จะมี publisher เป็น Scheduler Engine ที่จะเป็นตัว notify ให้เหล่า listener เพื่อให้ listener update เช่น Transform Task, RollupJobTask เป็นต้น จุดสังเกตุก็คือมี register ใน SchedulerEngine เพื่อการสมัครสมาชิกสำหรับเหล่า listener ที่จะ subscribe หรือติดตาม publisher และในคลาส SchedulerEngine มีการทำ NotifyListeners ซึ่งจะเป็นการแจ้งเตือนให้ listener รู้เมื่อมีการ update ที่ตัว publisher

14

Singleton เพื่อจำกัดการสร้างจำนวน obj ให้มีแค่ obj อันเดียวเท่านั้น และสามารถเข้าถึงได้จากทุกไฟล์ จุดสังเกต บรรทัด 95-100 คือ เป็น public method ที่สามารถเรียกใช้ได้จากคลาสอื่นๆได้ (เป็นการเข้าถึงแบบ global) และการเช็คว่ามีการสร้างหรือยัง ถ้าไม่มีให้สร้าง ถ้ามี ก็ให้ใช้ obj ที่เคยสร้างแล้ว return กลับไป, 79 ประกาศตัวแปรเป็น private เพื่อเก็บ obj singleton ไว้, 108-116 คือการประกาศ constructor แบบ Private เพื่อให้ไม่สามารถสร้างได้จากภายนอกหรือคลาสที่ไม่เกี่ยวข้อง (\*หมายเหตุ obj = object)

15

Decorator เพื่อเป็นการเพิ่มความสามารถของตัว object ที่เราสร้างขึ้นมา จุดสังเกตก็คือ ตอนนี้มี 3 คลาสหลักๆ อยากจะให้ focus ที่ 2 class คือ DelegatingHandler และ NioIPFilter DelegatingHandler เป็นเหมือน Decorator ที่เป็นตัวรวม Abstraction และมี NiolPFilter ที่ไป extend มาจาก Decorator อีกที ซึ่งจะเป็นการเพิ่มความสามารถให้เวลาเรียกใช้งานคลาสลูกๆ ใน่ส่วนบรรทัด เช่น ในไฟล์ DelegatingHandler , บรรทัด 17 จะมีการเก็บตัว object ที่ต้องการเพิ่มความสามารถไว้ และจะสังเกตุได้ว่า ในคลาสนี้จะมี method ต่างๆมากมาย ซึ่งจะ implement ไว้เป็น default ทุกส่วนอยู่แล้ว บรรทัด 19-56 (ไว้สำหรับถ้าเกิด ตัวลูกๆไม่ได้มีการเพิ่มความสามารถ method นี้ก็สามารถนำไปใช้ได้) และในไฟล์ NioIPFilter นี่แหละ จะเป็นตัวเพิ่มความสามารถขึ้น จะเห็นได้ว่า จะมี method บางอันเท่านั้น ที่จะเพิ่มความสามารถบางอย่างให้ เช่นในบรรทัดที่ 32-38 เป็นต้น

16

Template Method เพื่อเป็นการสร้างโครงสร้างเพื่อให้เราไป implement เพิ่มเติมได้ (โดยอาจจะมีการ implement ไว้ก่อนหน้านี้อยู่แล้ว) จุดสังเกตคือ SqlExecutable.java บรรทัด 16-19 มีการสร้าง method execute() เป็น default และมีการ implement โค้ดไว้ส่วนหนึ่งแล้ว ซึ่งพอมาใน EmptyExecutable.java บรรทัดที่ 31-33 จะมีการ Override execute() ใหม่อีกหนึ่งที จะเห็นได้ว่านี่เป็นการนำ default ที่ implement ไว้อยู่แล้ว มา implement เพิ่มเติมจาก method เดิมด้วยการ override ก็จะเป็นหนึ่งใน template method design pattern

17

ในมุมมองของ Flow of control Framework แลพ Library ต่างกันชัดเจนมากในเรื่อง IoC หรือ Inversion of Control โดย Framework เป็นตัวที่เลือกใช้ Inversion of Control จุดที่แตกต่างก็คือ ถ้าเป็น Library ก็คือเราไปเรียกใช้ Library มาใช้ โดยเราจะเป็นคนคลุม Flow ทั้งหมด นั้นก็คือ สามารถเลือกได้ว่าจะให้ทำทำงานตอนไหน ยังไง เมื่อไหร่ ในขณะที่ Framework ก็คือ เราเขียนโค้ด implement function ที่ Framework ทำขึ้นมาไว้เป็น Skeleton ให้แล้ว แล้วก็ให้ Framework มาเรียกใช้โค้ดเราอีกทีนึง ซึ่งในกรณีนี้ Framework จะเป็นคนคลุม Flow แทนเรา หรือจะเป็นคนที่จะเลือกว่า จะเอาโค้ดเราไปใช้ตอนไหน เมื่อไหร่ อย่างไหร่ อย่างตัวอย่างเห็นได้ชัดคือ Template Method ใน Framework

18.