



01076105, 01075106

Object Oriented Programming

Object Oriented Programming Project

Design Patterns



# Design Patterns

- คือ วิธีการแก้ปัญหาที่มักจะเจอในการออกแบบซอฟต์แวร์
- จึงมีผู้รวบรวมเอาไว้ เป็นแนวทางปฏิบัติที่ดี (Best Practices)
- ประโยชน์ของการรู้จัก Design Patterns คือ
  - ได้เรียนรู้การออกแบบ - เพราะ Design Patterns แต่ละตัวจะสอนการออกแบบให้รู้ถึง ข้อดี/ข้อเสีย ว่าเหมาะสำหรับใช้กับงานอะไร ทำให้เมื่อเจอกับงานก็สามารถนำไปใช้ได้เลย
  - ประหยัดเวลาในการทำงาน เพราะแทนที่จะไปคิดแก้ปัญหา ก็สามารถนำ Pattern ที่มีผู้คิดเอาไว้ไปใช้ได้เลย
  - เพื่อให้คุยกับผู้อื่นรู้เรื่อง เพราะเป็นสิ่งที่อ้างอิงทั่วโลก



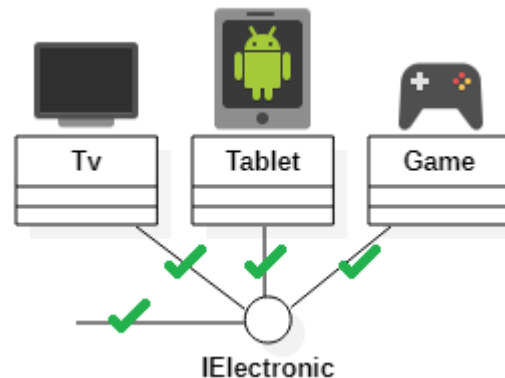
# Design Patterns

- แบ่งออกเป็น 3 กลุ่ม ตามลักษณะของการแก้ปัญหา
- Creational Patterns เป็นกลุ่มที่เกี่ยวข้องกับการกับสร้าง Object ที่ดี จะได้ไม่สร้างปัญหาในอนาคต
- Structural Patterns ช่วยในการจัดการกับโครงสร้างของ Object ที่มีความซับซ้อน ซึ่ง Design Patterns กลุ่มนี้ จะช่วยลดความซับซ้อนของโครงสร้างได้
- Behavioral patterns จะช่วยให้คลาสต่างๆ ทำงานร่วมกัน ได้ง่ายๆ เพราะทุกครั้งที่เราเพิ่มความสามารถใหม่ๆ เข้าไป หมายถึง คลาสต่างๆ ที่เรามีอยู่ ไม่ว่าจะเป็นตัวใหม่ หรือ ตัวเก่า จะเพิ่มขึ้น แต่ทำยังไงให้โครงสร้างไม่ซับซ้อน



# Creational Patterns

- เป็น Patterns ที่เกี่ยวกับการสร้าง Objects เหมาะสำหรับกรณีที่มีการสร้าง Object จำนวนมาก
- พื้นฐานหลักการ คือ **Program to an interface and not to an implementation**. แปลได้ว่า การเขียนโปรแกรม ไม่ควรไปทำงานกับระดับล่าง แต่จงทำงานกับระดับ Abstraction เท่านั้น
- จากตัวอย่างจะเห็นได้ว่าการสร้าง Abstract Class เพื่อใช้ในการเชื่อมต่อ





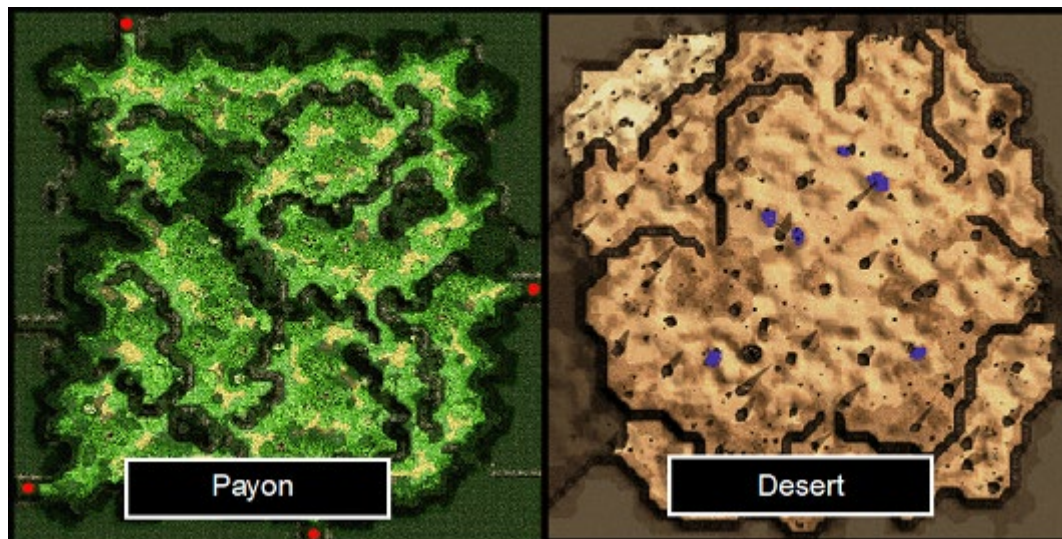
# Creational Patterns

- มีทั้งหมด 5 Pattern ได้แก่ (ในสไลด์นี้จะอธิบายไม่ครบทุกตัว)
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Singleton Pattern
  - Builder Pattern
  - Prototype Pattern



## Creational Patterns : Factory Method

- Factory Method Pattern เป็นวิธีการในการจัดการกับกรณีที่มีการสร้าง Object ที่แตกต่างกันเล็กน้อย จำนวนมาก
- ตัวอย่างปัญหา
- สมมติว่าเราเขียนเกมที่มีแผนที่ 2 แบบคือแบบ ป่า (Payon) และ ทะเลทราย (Desert) ตามรูป





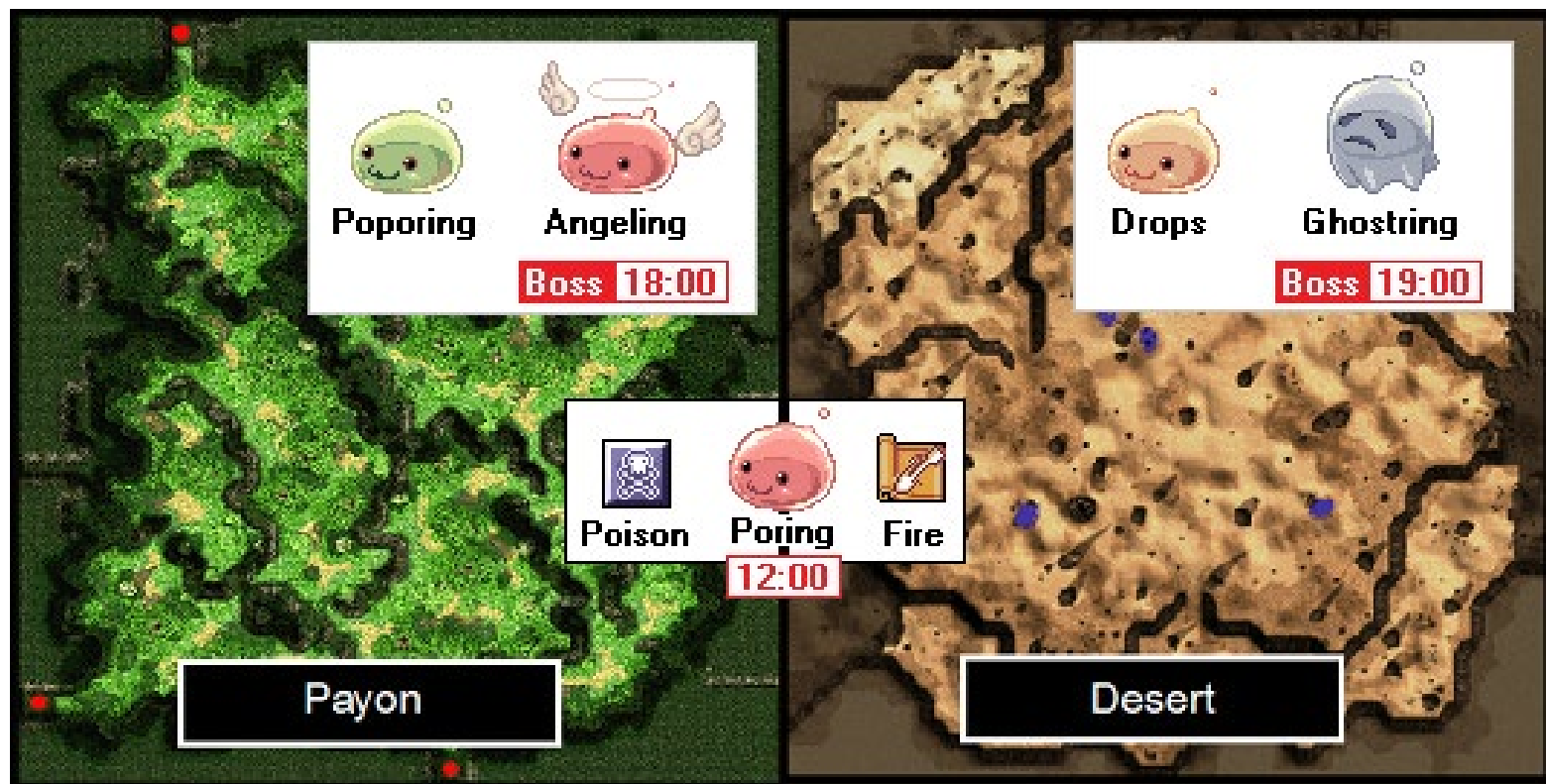
## Creational Patterns : Factory Method

- ภายในแผนที่กำหนดให้มี Slime เป็น monster เดินอยู่ภายในแผนที่เหล่านั้น แต่มีเงื่อนไขพิเศษ คือ
  - Slime ที่อยู่ใน Payon จะเป็นตัวสีเขียว ชื่อว่า Poporing สามารถปล่อยพิษ
  - Slime ที่อยู่ใน Desert จะเป็นตัวสีเหลือง ชื่อว่า Drops ที่สามารถปล่อยไฟ
  - แผนที่ Payon ถ้าเป็นเวลา 18:00 ตรงเราจะเจอบอส Slime ที่ชื่อว่า Angeling
  - แผนที่ Desert ถ้าเป็นเวลา 19:00 ตรงเราจะเจอบอส Slime ที่ชื่อว่า Ghosting
  - และแผนที่ทั้ง 2 ถ้าเป็นเวลา 12:00 เราจะได้ Slime ที่ชื่อว่า Poring ซึ่งถ้ามันเกิดในแผนที่ Payon มันจะปล่อยพิษได้ แต่ถ้ามันเกิดในแผนที่ Desert มันจะปล่อยไฟได้



# Creational Patterns : Factory Method

- สามารถสรุป monster ได้ตามภาพ

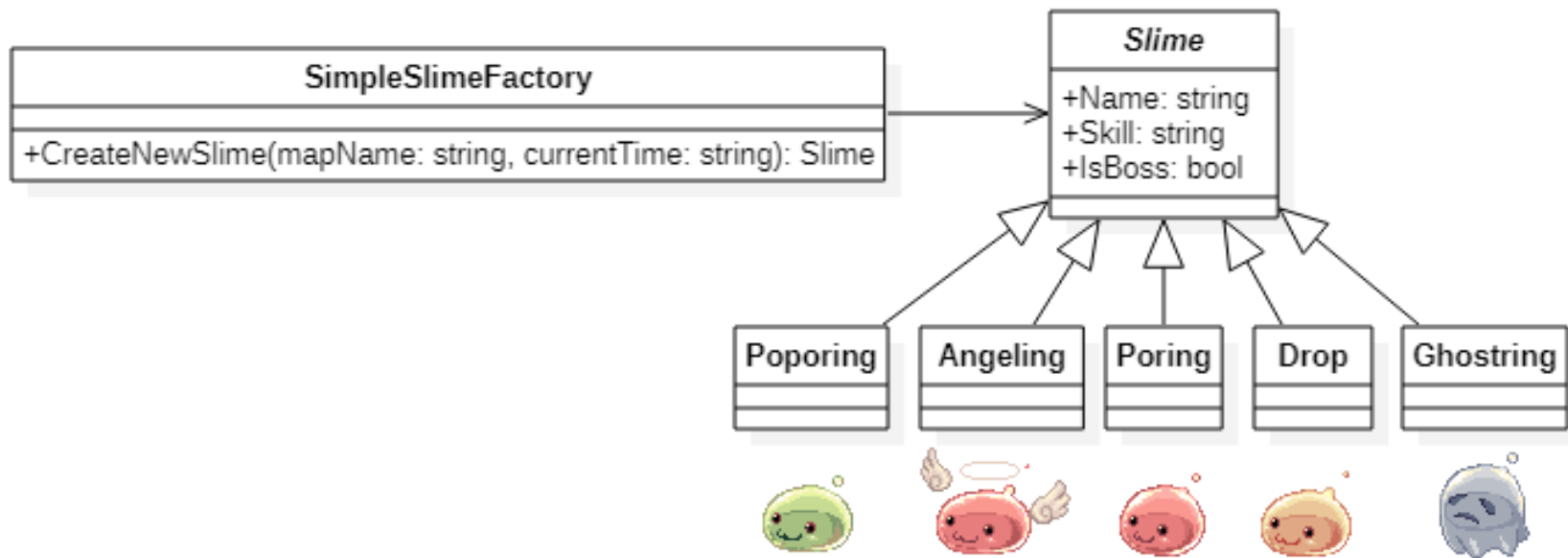






# Creational Patterns : Factory Method

- สมมติว่าเราสร้าง Class แบบนี้



- ตอนสร้าง Slime ก็แค่สร้างเงื่อนไขว่า ตอนนี้อยู่แผนที่อะไร และ เป็นเวลาเท่าไร เพียงเท่านั้นก็จะสามารถสร้าง Slime ได้ถูกต้องตามเงื่อนไข



## Creational Patterns : Factory Method

- สามารถเขียน code ได้ตามนี้

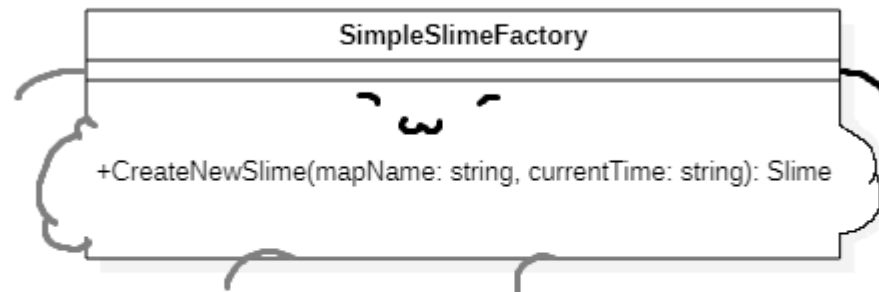
```
def create_new_slime(mapName, currentTime):  
    if mapName == "Payon":  
        if currentTime == "12:00":  
            return Poring("Poison")  
        elif currentTime == "18:00":  
            return Angeling()  
        else:  
            return Poporing()  
    else:  
        // โค้ดคล้ายๆด้านบน
```

- Code ตามตัวอย่างจะขัดกับหลัก Open & Close Principle เพราะเมื่อมีของใหม่ เช่น Monster ตัวใหม่ ก็จะต้องแก้ไข Code ไปเรื่อยๆ



# Creational Patterns : Factory Method

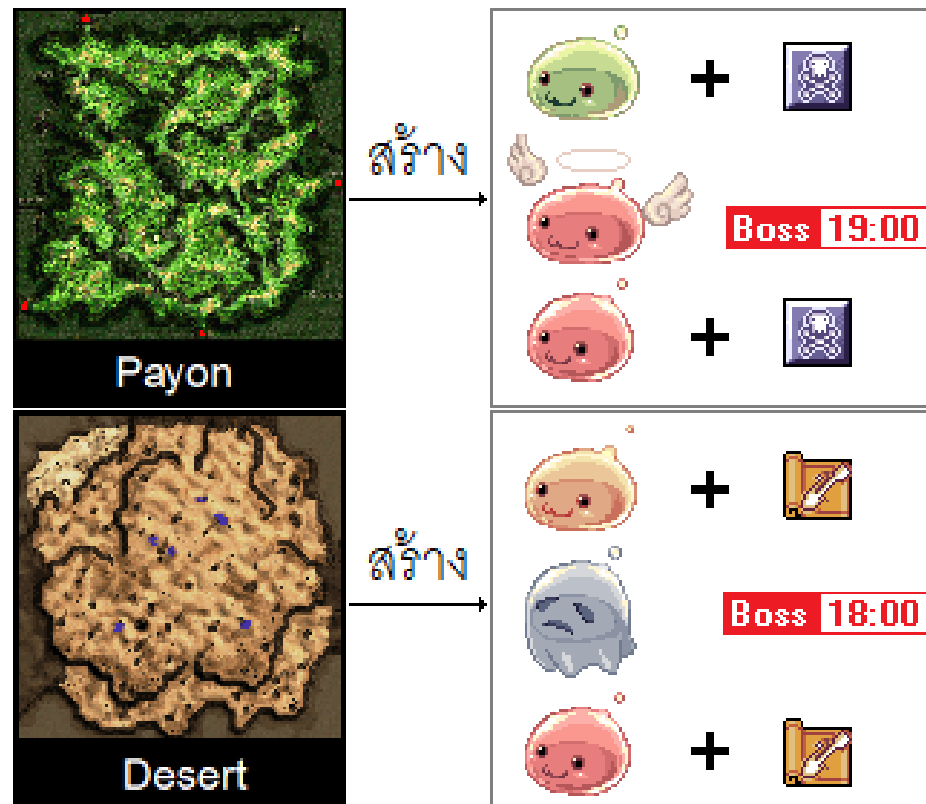
- สมมติว่ามี Map และ Monster ตามนี้ Method `create_new_slime` จะต้องแก้ไข และมีความซับซ้อนขึ้น ฟังก์ชัน `create_new_slime` ก็จะบวมขึ้น





## Creational Patterns : Factory Method

- ถ้าเราดูความสัมพันธ์ของเจ้า 2 อย่างนี้ดีๆเราจะพบว่า แผนที่เป็นตัวกำหนดว่าจะสร้าง Slime แบบไหน และรวมถึงความสามารถของ Slime ด้วย ตามรูปด้านล่าง





## Creational Patterns : Factory Method

- จากที่กล่าวมา จะดีกว่าหรือไม่ หากจะแยกให้ แผนที ไปรับผิดชอบในการสร้าง Slime ของใครของมันเอง
- ดังนั้นจะได้โรงงานผลิต Slime ออกมา 2 โรงงานตามนี้ ซึ่งเป็นไปตามหลัก Single-Responsibility Principle ด้วย



Payon

PayonSlimeFactory

+CreateNewSlime(currentTime: string): Slime



Desert

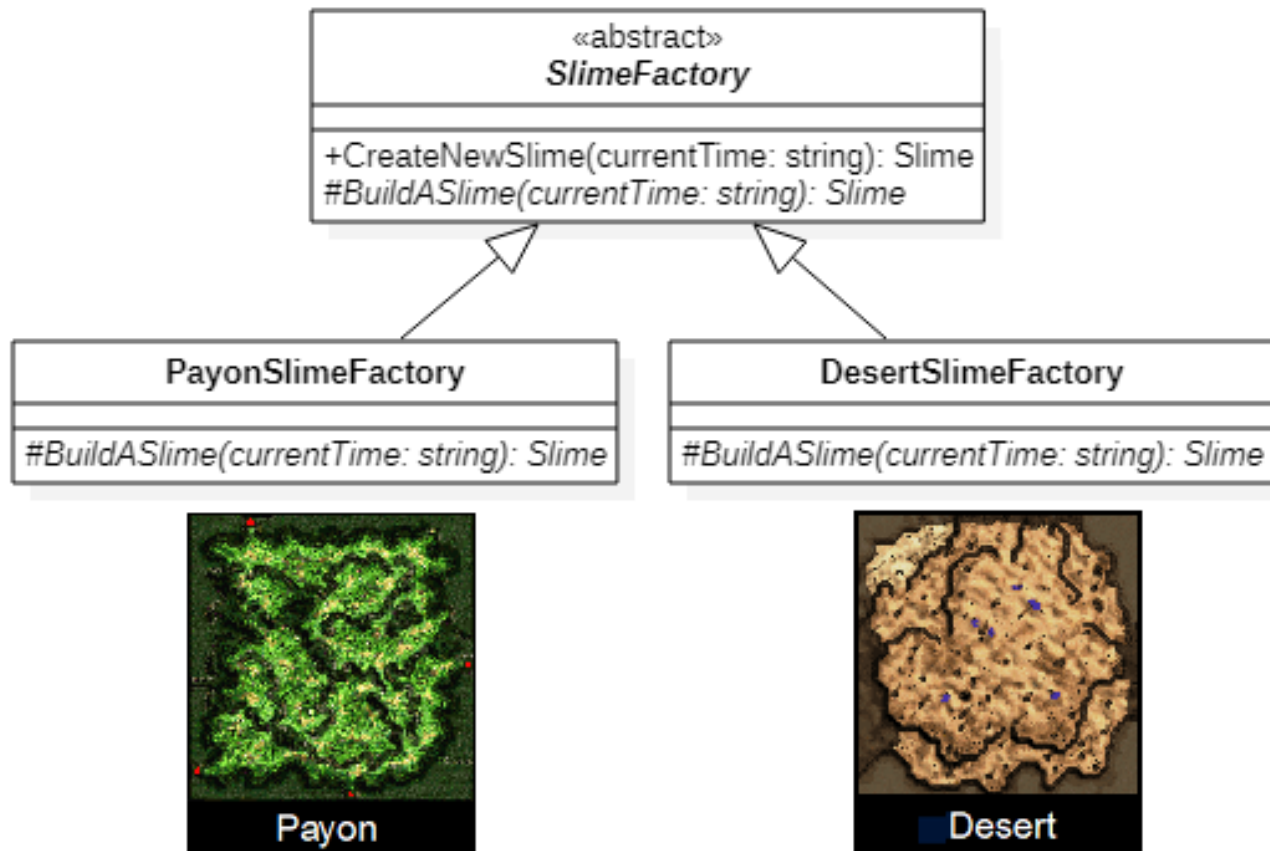
DesertSlimeFactory

+CreateNewSlime(currentTime: string): Slime



# Creational Patterns : Factory Method

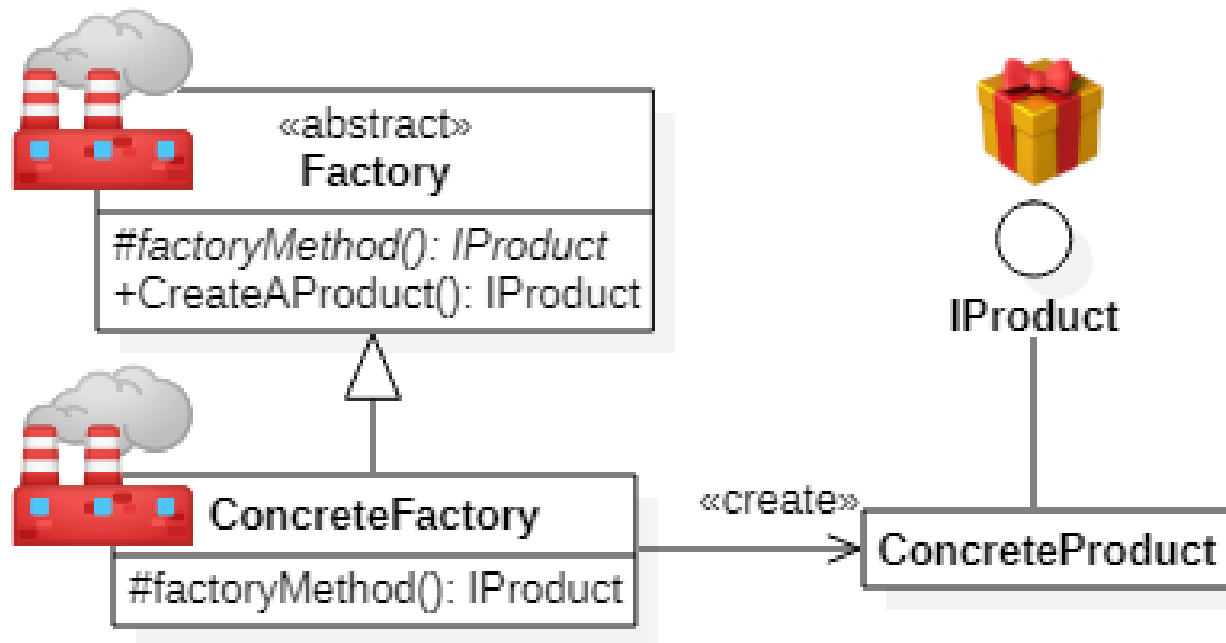
- ดังนั้น จะได้การออกแบบสุดท้ายเป็นตามรูป ซึ่งเป็นไปตามหลัก SRP, OCP และ DIP





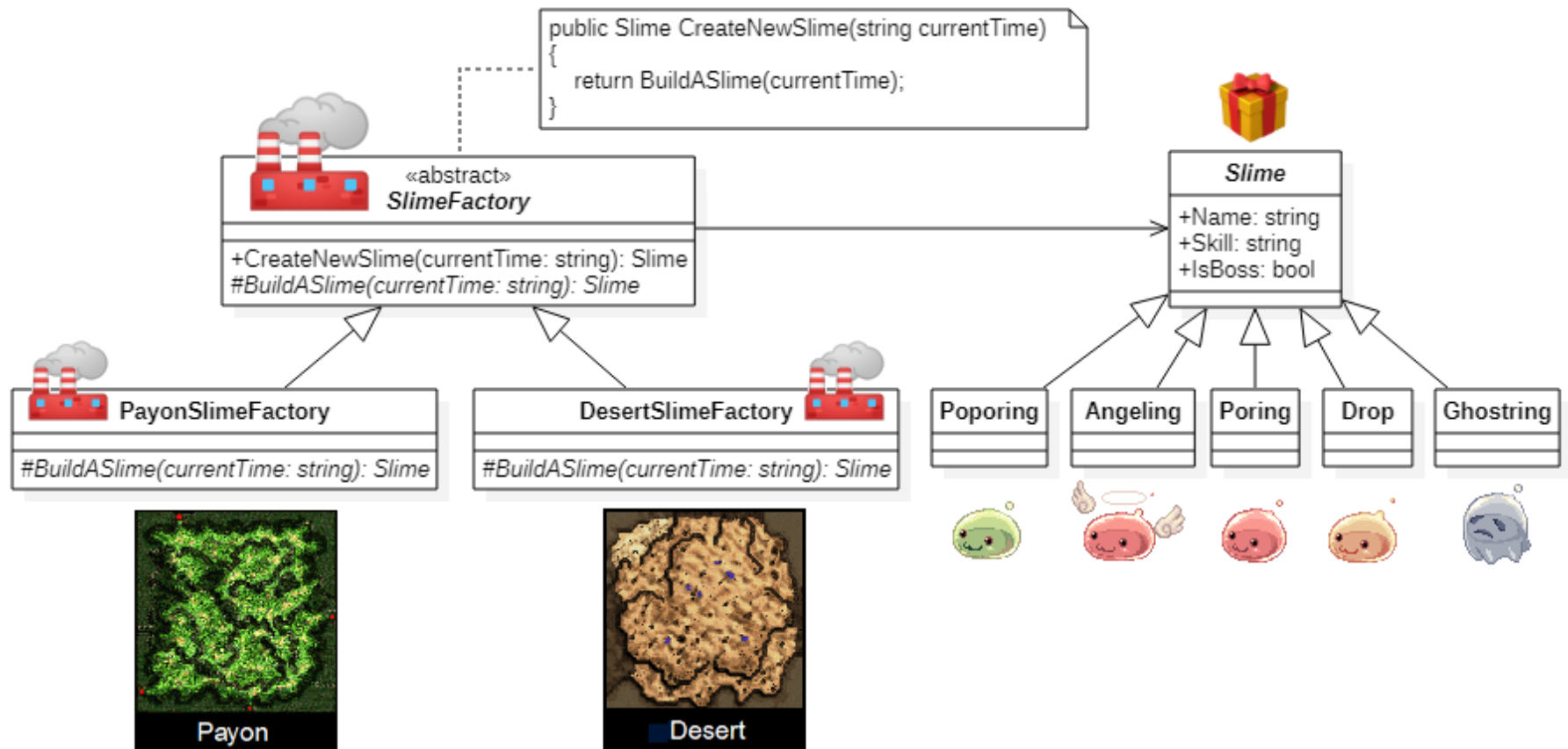
# Creational Patterns : Factory Method

- ทั้งหมดที่กล่าวมา คือ แนวทางการออกแบบที่เรียกว่า Factory Method Pattern





# Creational Patterns : Factory Method

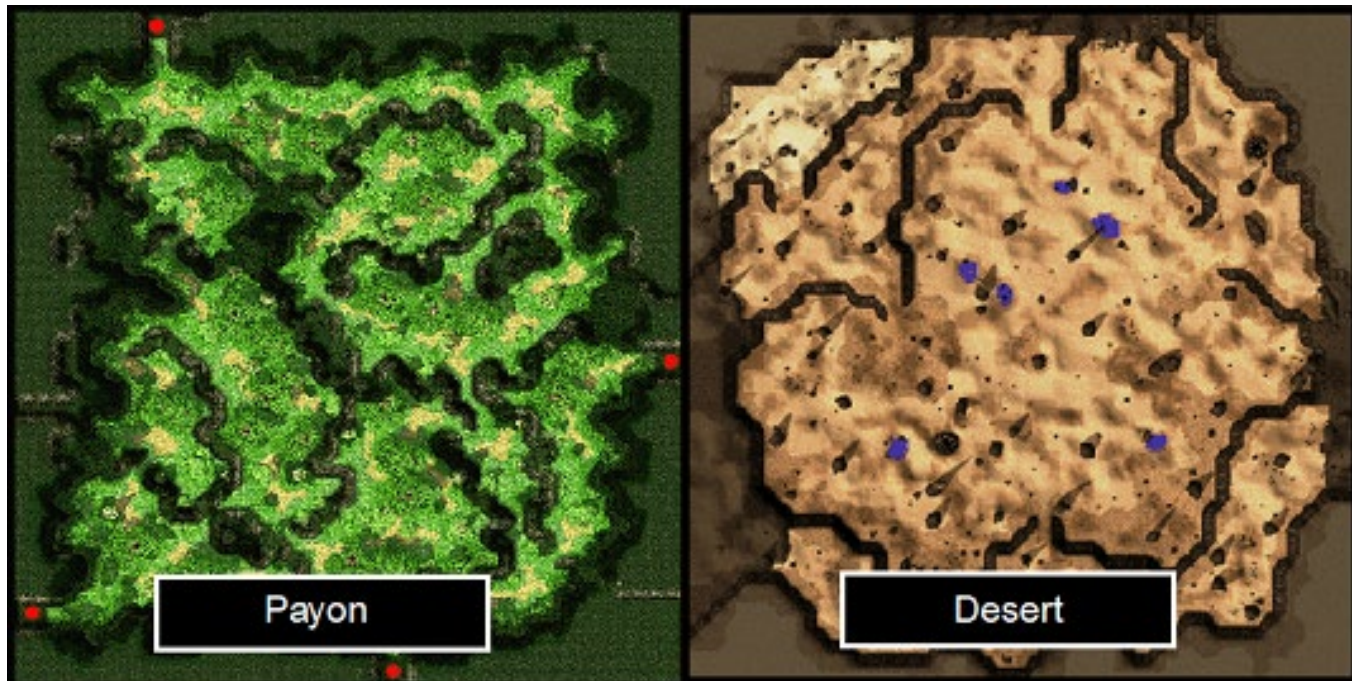






## Creational Patterns : Abstract Factory

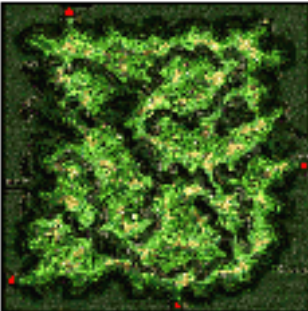



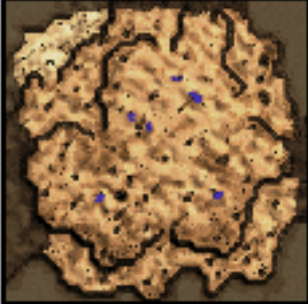



- จะคล้ายกับ Factory Method จะยกตัวอย่างด้วยเกมเหมือนเดิม
- ในเกมจะมีแผนที่ 2 แบบ เหมือนเดิม





## Creational Patterns : Abstract Factory

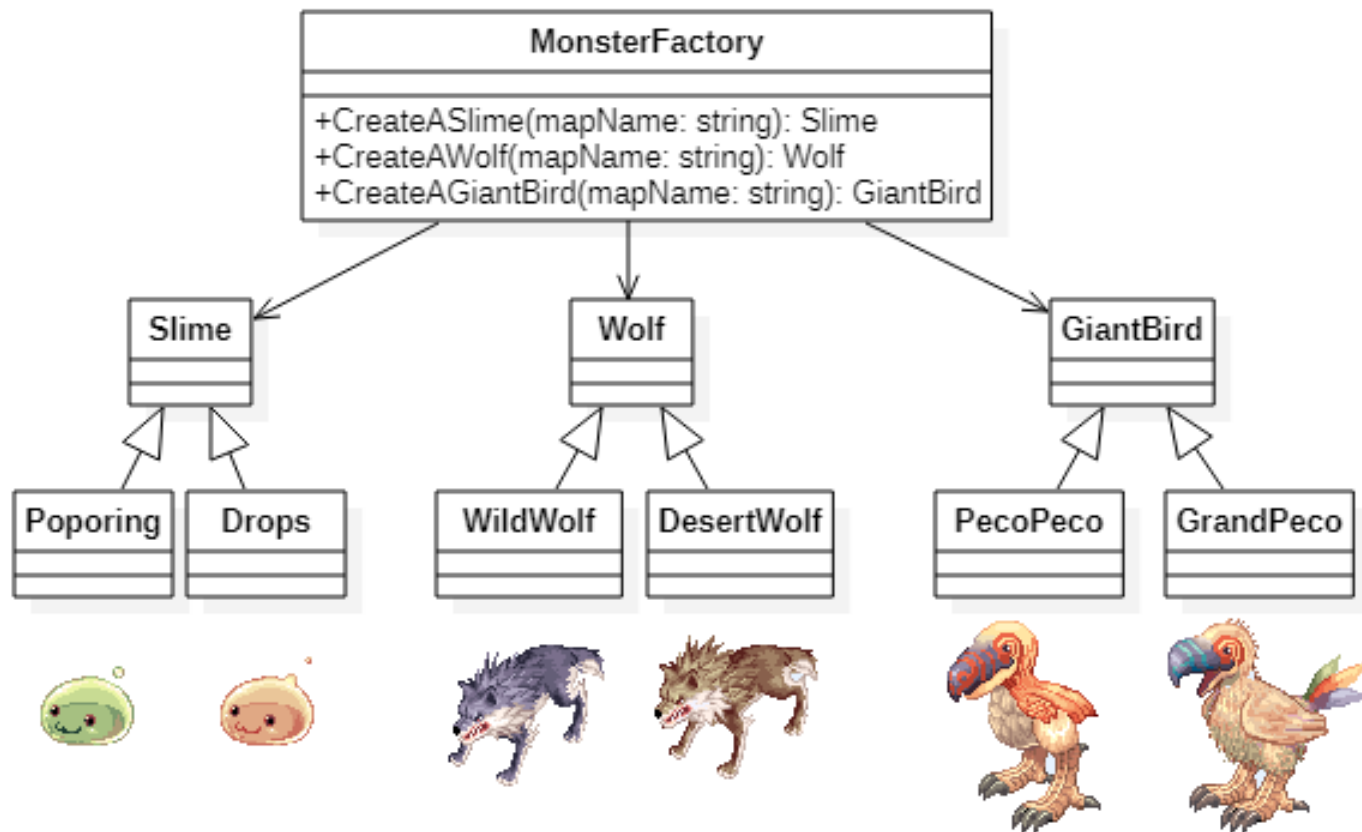
- ภายในแต่ละแผนที่จะมี monster หลายๆแบบอยู่ในนั้น เช่น สไลม์ (Slime), หมาป่า (Wolf), นกยักษ์ (Giant Bird) แต่เนื่องจากสภาพแวดล้อมต่างกันเลยทำให้ monster ที่อยู่ในนั้นมี หน้าตา กับ ชื่อเรียก ไม่เหมือนกัน ตามรูปด้านล่าง

	Slime	Wolf	Giant Bird
 Payon	 Poporing	 Wild Wolf	 Peco Peco
 Desert	 Drops	 Desert Wolf	 Grand Peco



# Creational Patterns : Abstract Factory

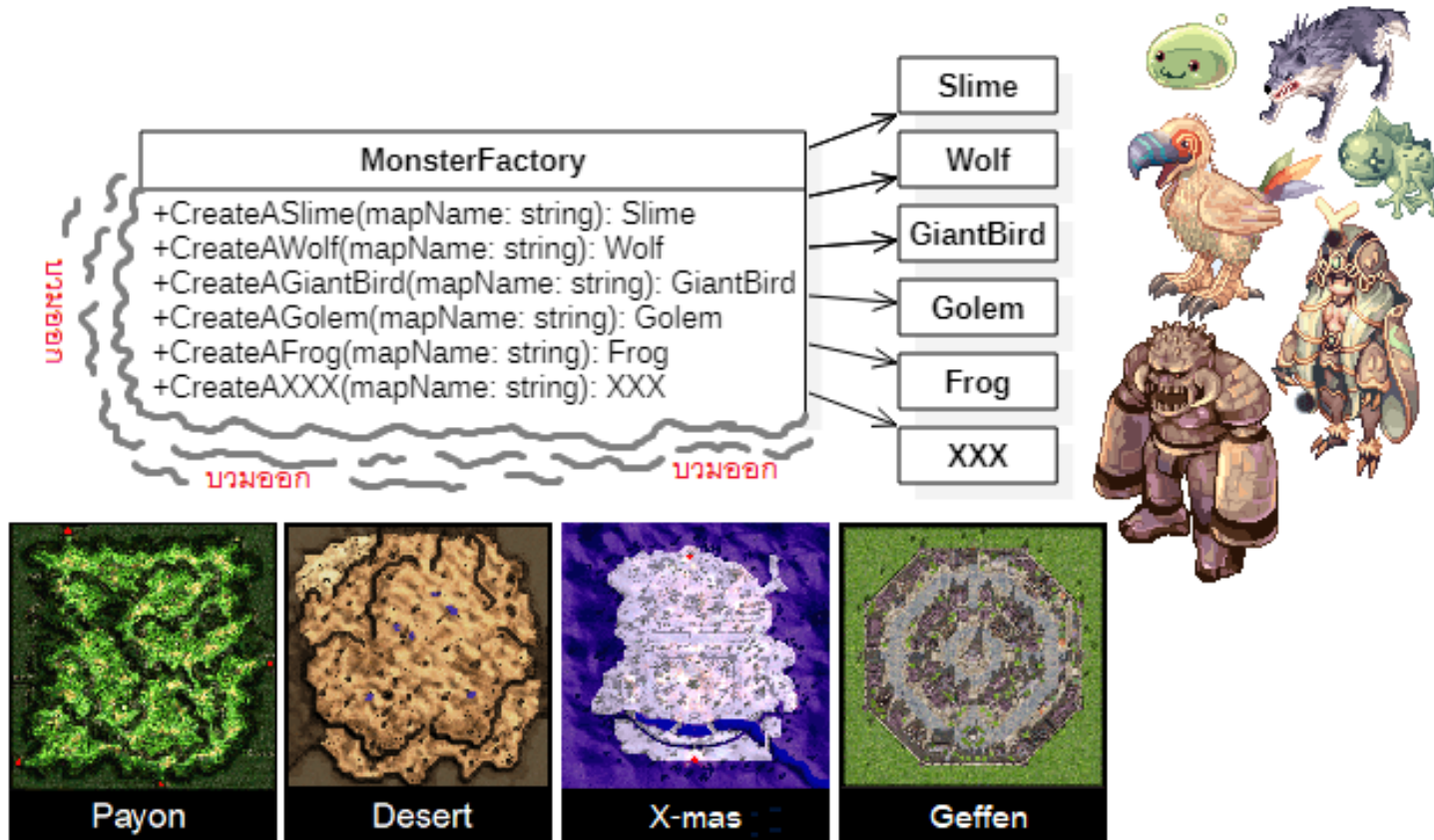
- เนื่องจากมี monster อยู่ทั้งหมด 3 ประเภท ดังนั้นก็จะแบ่งออกเป็น Model ทั้งหมด 3 กลุ่มตามรูปด้านล่าง





# Creational Patterns : Abstract Factory

- การออกแบบข้างต้นจะขัดกับหลัก Open & Close Principle เนื่องจากหากมีแผนที่ใหม่เพิ่มเข้ามา หรือ มี monster ใหม่เพิ่มเข้ามา ก็จะต้องมีการแก้ไข code

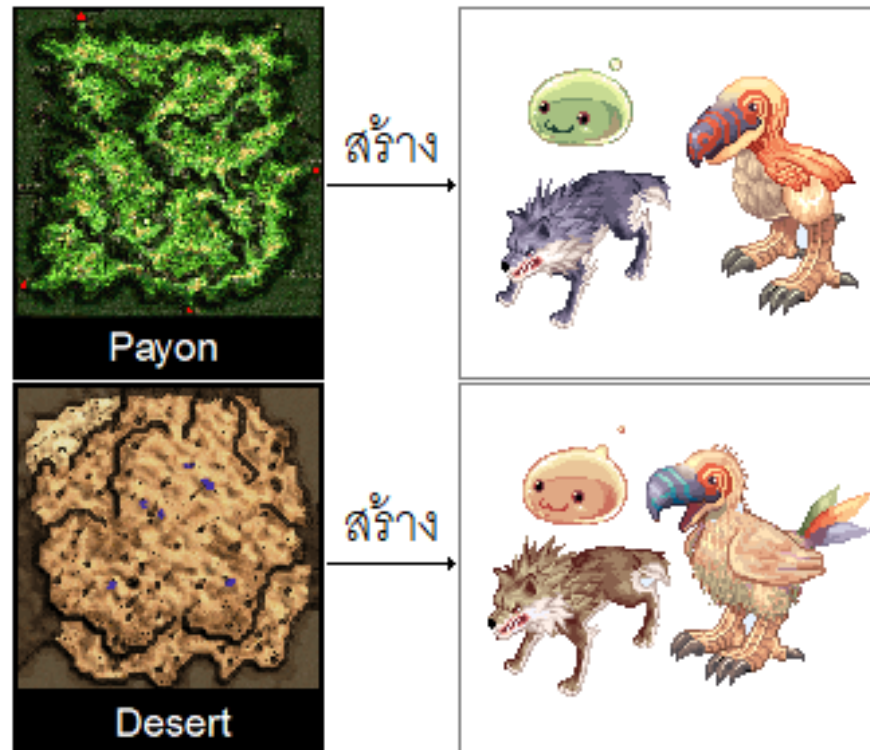






# Creational Patterns : Abstract Factory

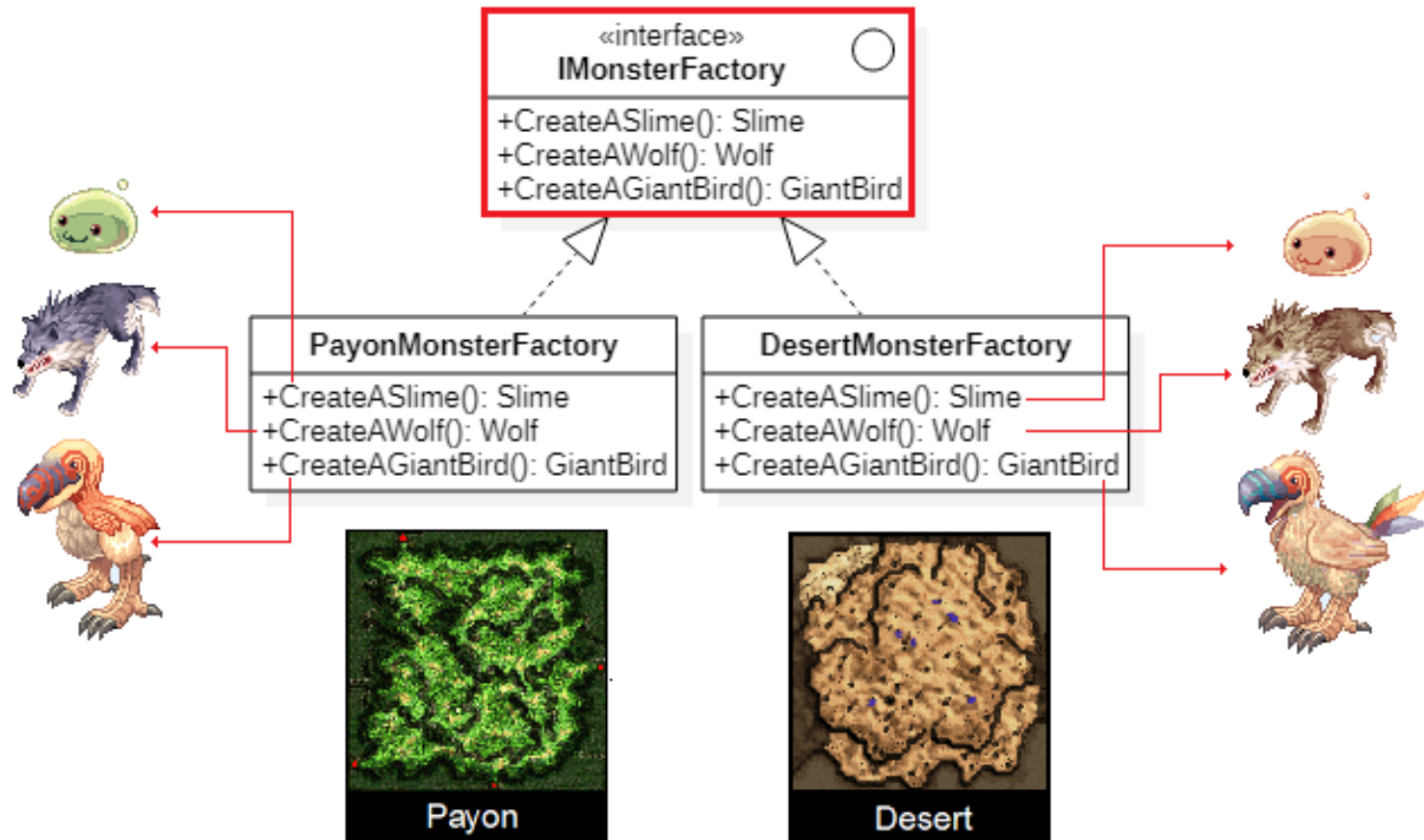
- แนวทางการแก้ไข จะคล้ายกับ Factory Method คือ แยกแผนที่กับ Monster ออกจากกัน และ ให้แผนที่รับผิดชอบในการสร้าง Monster (ตามหลัก SRP)





# Creational Patterns : Abstract Factory

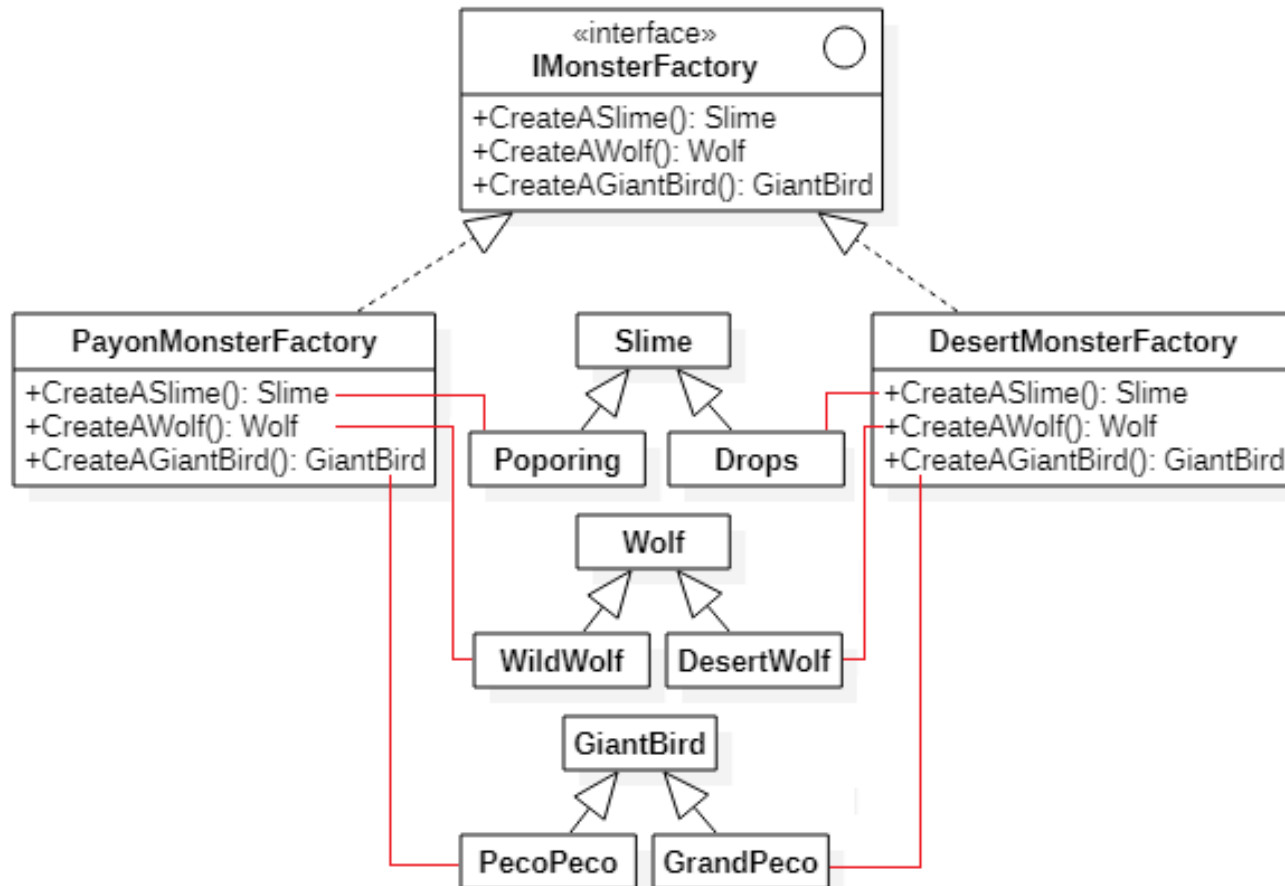
- สามารถเขียนเป็น Diagram ดังนี้





# Creational Patterns : Abstract Factory

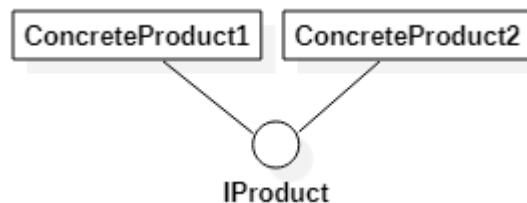
- หรือเขียนเป็น Class Diagram ได้ดังนี้





# Creational Patterns : Abstract Factory

- กรณีที่เราต้องสร้าง Object จำนวนมากๆ เราจะใช้สิ่งเรียกว่า Factory ในการสร้าง Object การทำงานควรจะเริ่มจากสร้าง Interface หรือ Abstract Class และ Inherit มาเป็น Class ที่ใช้สร้าง Product จริง ตามรูป



- ตัวอย่าง

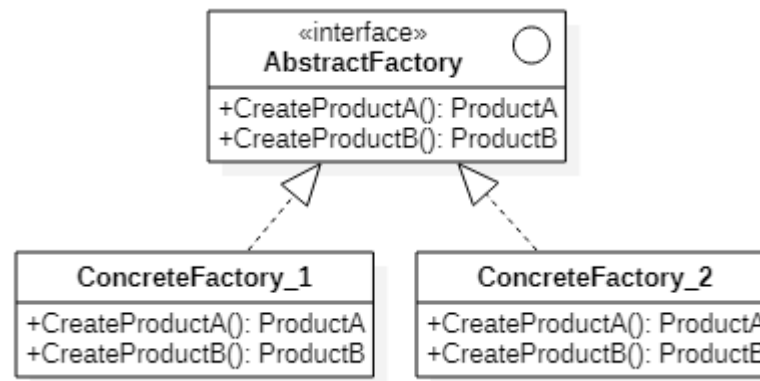






# Creational Patterns : Abstract Factory

- Abstract Factory อาจแปลได้ว่า โครงร่างโรงงานผลิต ทำหน้าที่กำหนดโครงสร้าง ส่วน Class ที่ทำหน้าที่ผลิต คือ Class ระดับล่าง (Implement Class)

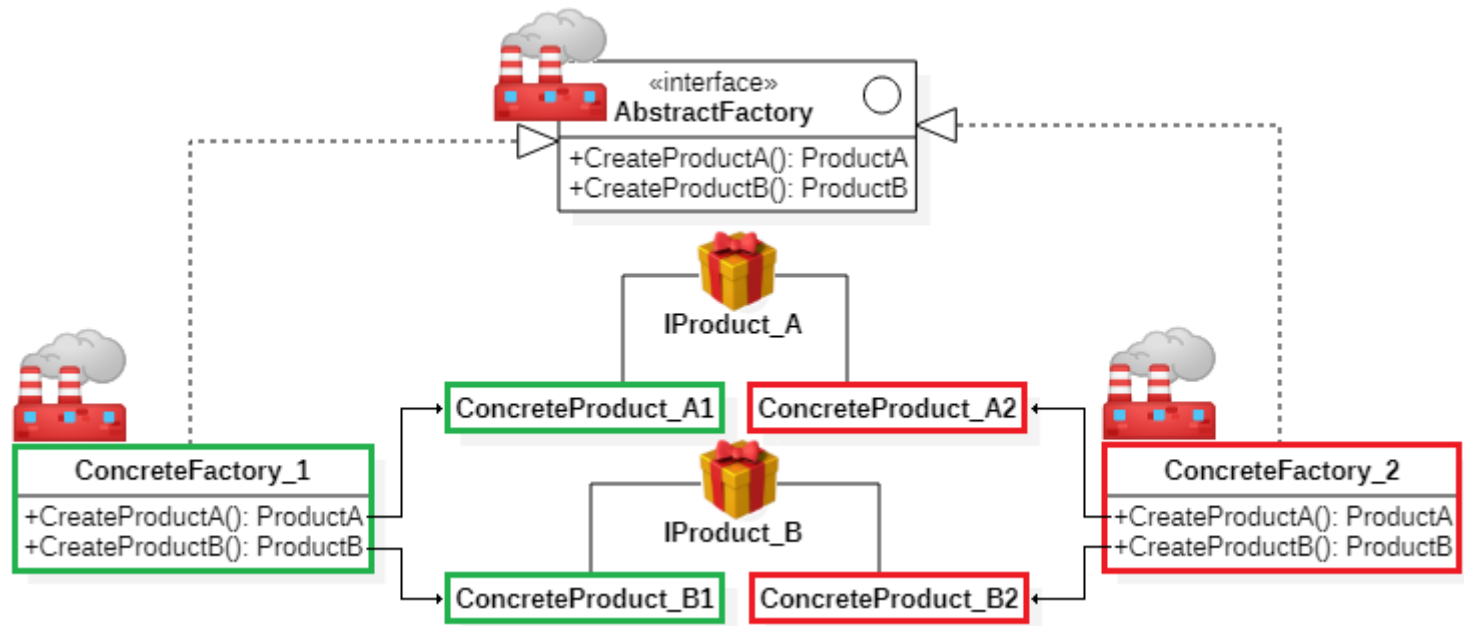


- จากรูป Product มี 2 ชนิด คือ A กับ B ดังนั้นตัว Class ระดับล่าง จะต้องสร้างทั้ง Product A และ B ด้วย



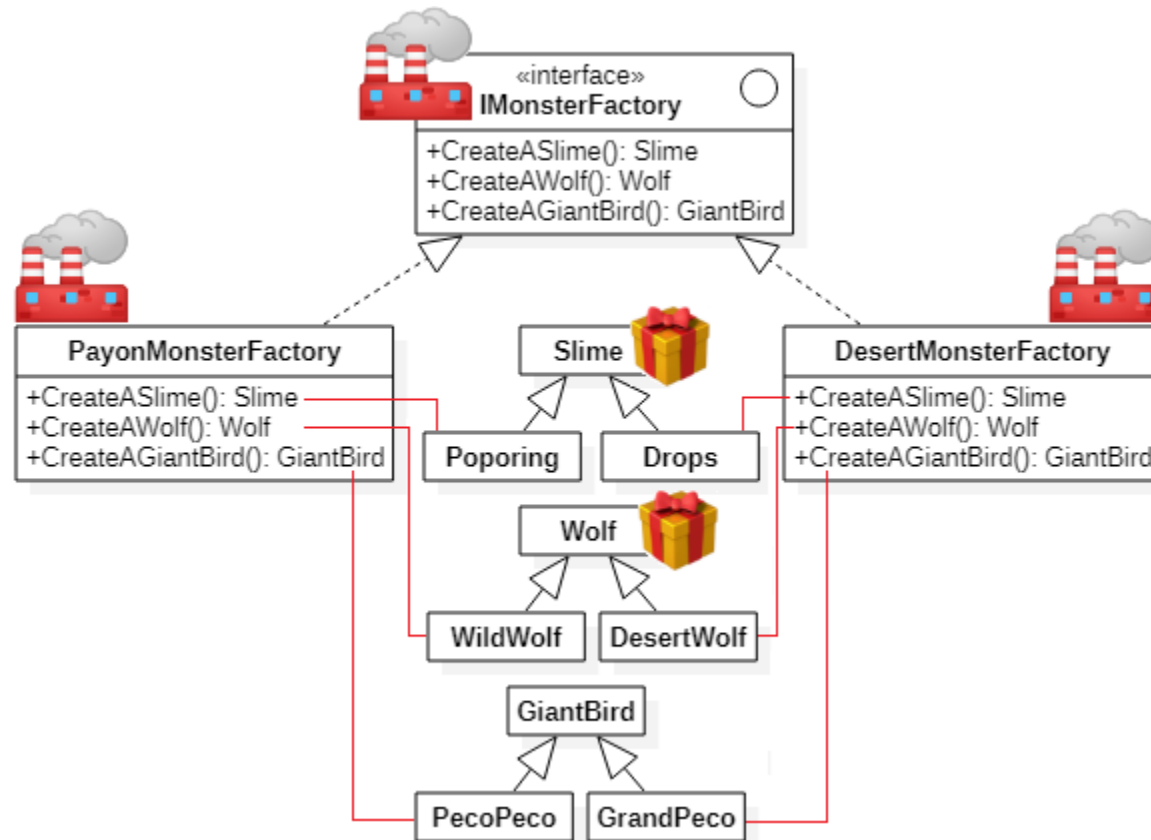
# Creational Patterns : Abstract Factory

- เขียนรูปให้ชัดเจนขึ้น





# Creational Patterns : Abstract Factory





# Creational Patterns

- ความแตกต่างระหว่าง Factory Method และ Abstract Factory
  - Factory Method จะใช้วิธี Inherit มาเป็น Subclass และให้ Subclass เป็นคนสร้าง Object
  - Abstract Factory จะใช้วิธี Composition



# Creational Patterns : Singleton Pattern

- คือ Class พิเศษ ที่นำไปสร้าง Object ได้แค่ตัวเดียว แต่ต้องเข้าถึงจากที่ไหนก็ได้
- จะยังคงใช้เกมเป็นตัวอย่าง สมมติว่าในเกมมีอีเว้นท์พิเศษ โดยจะปล่อย Boss ที่ชื่อว่า Detardeurus มาให้ผู้เล่นทุกคนช่วยกันปราบ ซึ่งถ้ามีผู้เล่นคนไหนปราบมันลงได้ ก็จะทำให้กลับมาเกิดใหม่ในทุกๆ 2 ชั่วโมง ตามรูปด้านล่าง





## Creational Patterns : Singleton Pattern

- ด้วยความพิเศษที่เป็นอีเว้นท์ จึงทำให้บอสตัวนี้จะต้อง มีได้เพียงตัวเดียวเท่านั้น และ ผู้เล่นทุกคนจะต้องเข้าถึงข้อมูลบอสตัวนี้ได้ ไม่ว่าจากที่ไหนก็ตามอีกด้วย เพราะผู้เล่นทุกคนจะสามารถโจมตีและเห็นค่าต่างๆ ของบอสตัวนี้ได้

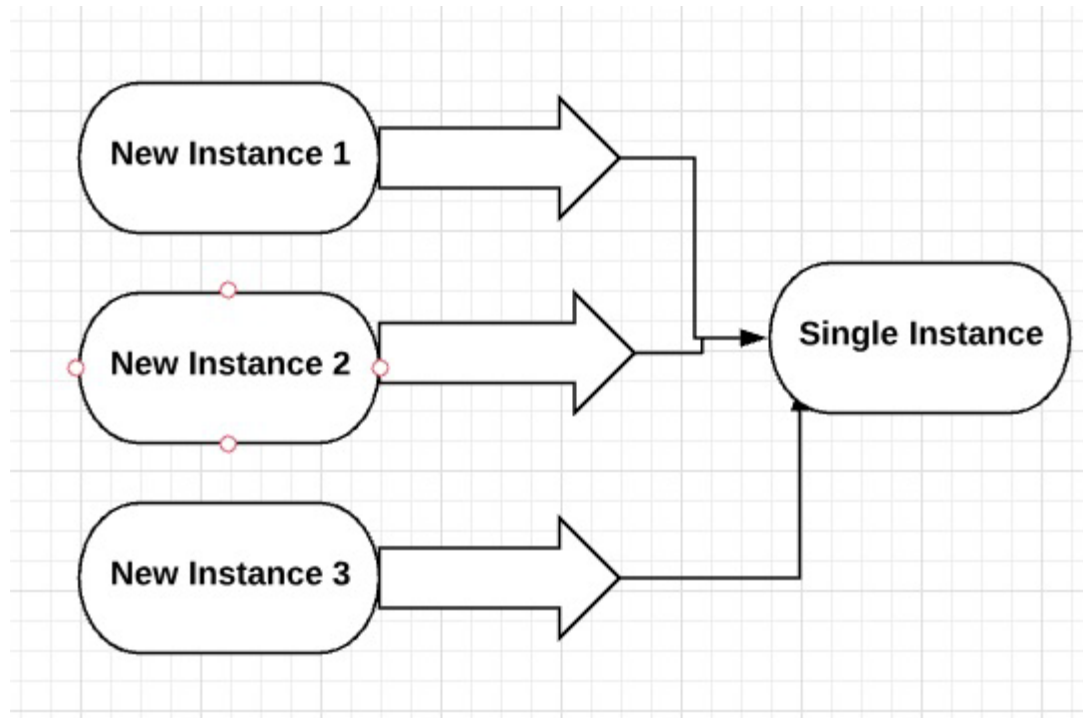


Detardeurus		
HP	1849 / 8335	55%
SP	113 / 427	26%
Respawn in: 00:00:00		



# Creational Patterns : Singleton Pattern

- รูปแบบการทำงานจะเป็นแบบนี้ คือ ไม่ว่าใครจะสร้าง Object ก็จะได้ Object เดียวกัน





# Creational Patterns : Singleton Pattern

- ปัญหาของโจทย์ข้อนี้ คือ จะทำอะไรให้สร้างได้เพียง Object เดียว เพราะปกติใครที่เรียก Constructor ก็จะได้ Instance Object คืนไปเสมอ ดังนั้นเราจะเขียนแบบนี้

```
class Detardeurus:
    __instance = None

    @staticmethod
    def getInstance():
        """ Static access method. """
        if Detardeurus.__instance == None:
            Detardeurus()
        return Detardeurus.__instance

    def __init__(self):
        """ Virtually private constructor. """
        if Detardeurus.__instance != None:
            raise Exception("This class is a singleton!")
        else:
            Detardeurus.__instance = self
```





## Creational Patterns : Singleton Pattern

- จากโปรแกรม จะเห็นว่ามี การสร้าง Class Attribute สำหรับเก็บ Instance เอาไว้ เมื่อมีการอ้างอิง
- ใน Constructor จะออกแบบให้เรียกได้ครั้งเดียวโดย หากเรียกครั้งแรกจะสร้าง Instance และเก็บตำแหน่งไว้ที่ `__instance` และหากมีการเรียกครั้งต่อไปให้ raise exception
- มีการสร้าง static method ชื่อ `getInstance` โดยหากยังไม่มี Instance ก็จะเรียก Constructor แต่หากมีแล้วก็จะส่งตำแหน่งที่เก็บไว้ที่ `__instance` ไปให้
- ในการเรียกใช้ก็จะเรียกใช้ `getInstance` เป็นหลัก

```
s = Detardeurus.getInstance()
```



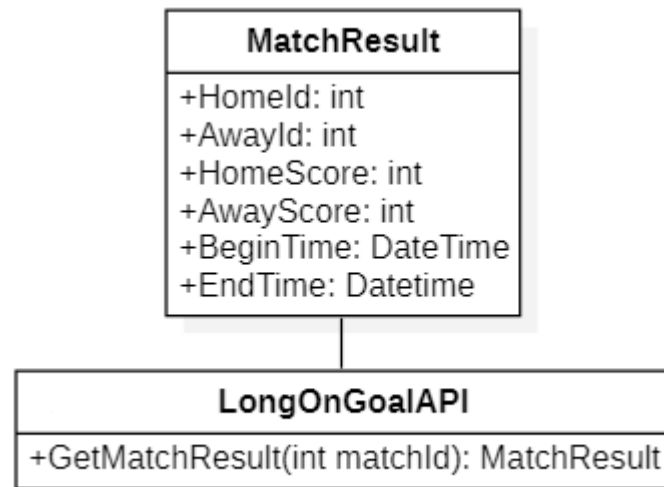
# Structural Patterns

- เป็นการออกแบบที่ช่วยบอกว่าควรมีคลาสอะไรบ้าง? แต่ละคลาสควรมีโครงสร้างยังไงถึงจะช่วยให้ทำงานได้เร็วขึ้น ไม่ซับซ้อนจนเกินไป
- มีทั้งหมด 7 Pattern
  - Adaptor Pattern
  - Proxy Pattern
  - Bridge Pattern
  - Composite Pattern
  - Facade Pattern
  - Flyweight Pattern



## Structural Patterns : Adapter Pattern

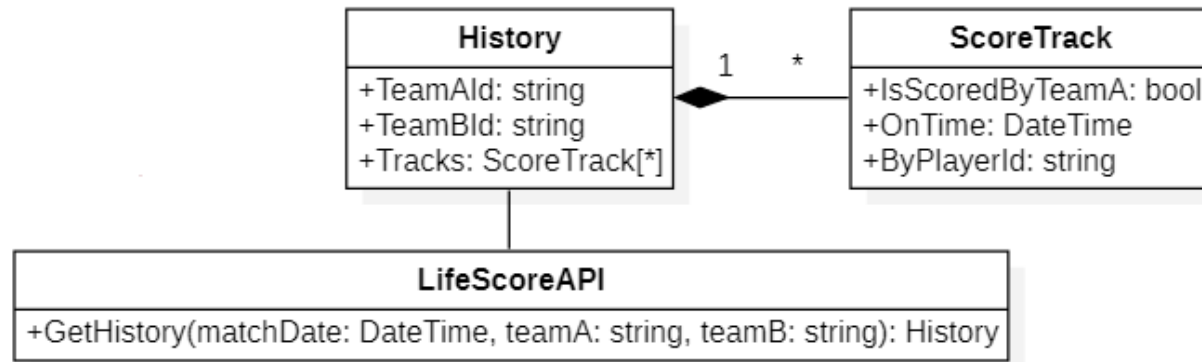
- ทำหน้าที่เชื่อมการทำงาน 2 อย่างที่แตกต่างกัน
- สมมติว่าเราจะทำเว็บรายงานผลฟุตบอล แต่เราไม่อยากจะเก็บข้อมูลเอง จึงไปดึงข้อมูลจาก API ของคนอื่น สมมติว่าไปดึงมาจาก 2 เว็บ คือ LongOnGoal และ LifeScore (เพราะแต่ละเว็บก็มีข้อมูลไม่ครบ จึงต้องใช้หลายเว็บประกอบกัน)
- ปัญหา คือ รูปแบบ API ของทั้ง 2 เว็บไม่ตรงกัน โดย LoginOnGoalAPI มีดังนี้



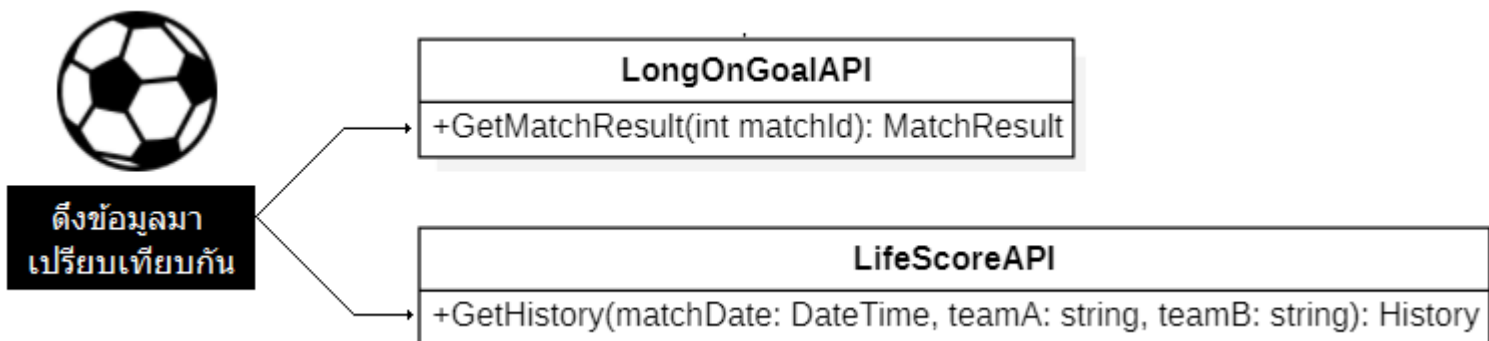


# Structural Patterns : Adapter Pattern

- LifeScore มี API ดังนี้



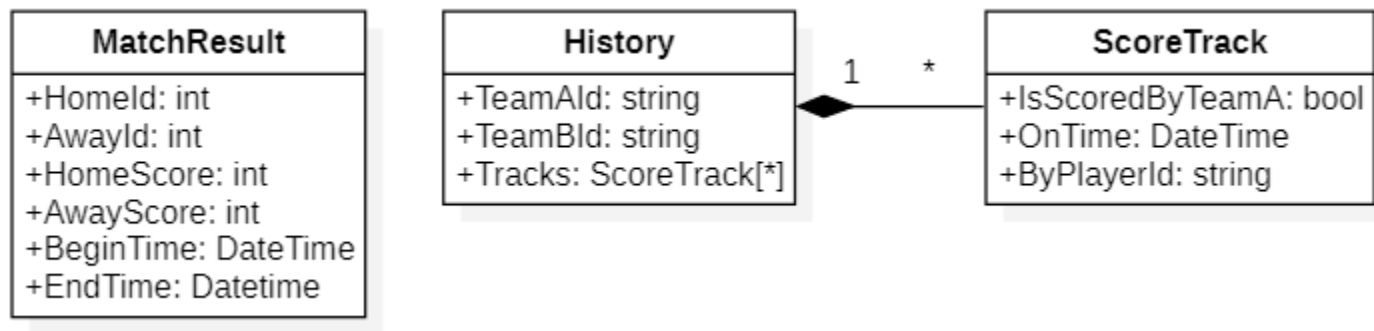
- เราอาจจะคิดง่ายๆ ก็เอาผลมารวมกันตามรูป





## Structural Patterns : Adapter Pattern

- แต่ปัญหา คือ มันรวมไม่ง่าย เพราะรูปแบบ API ไม่เหมือนกัน ตามรูป

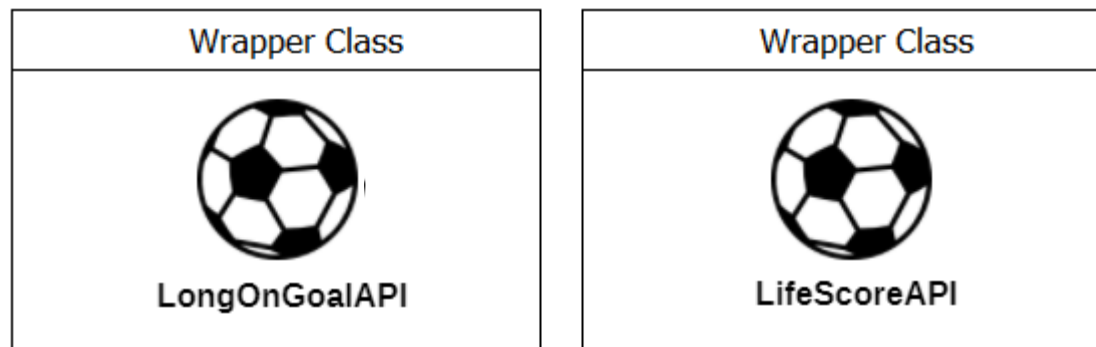


- จะเขียนโปรแกรมแปลง แล้วค่อยเอามารวมกัน ก็ทำได้ไม่ผิดอะไร
- แต่ปัญหา คือ หากเราเพิ่ม Web API ตัวอื่นๆ เข้ามาอีกละ โปรแกรมที่เขียนก็จะซับซ้อนขึ้น

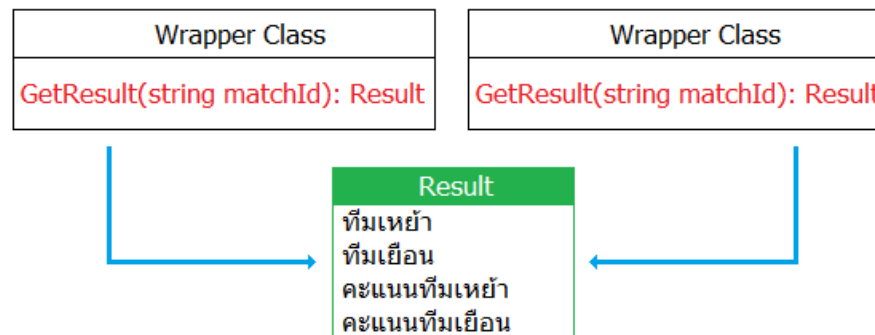


# Structural Patterns : Adapter Pattern

- วิธีแก้โดยใช้ Adapter Pattern ทำได้โดยการสร้าง Class ขึ้นมาครอบ เรียกว่า Wrapper Class



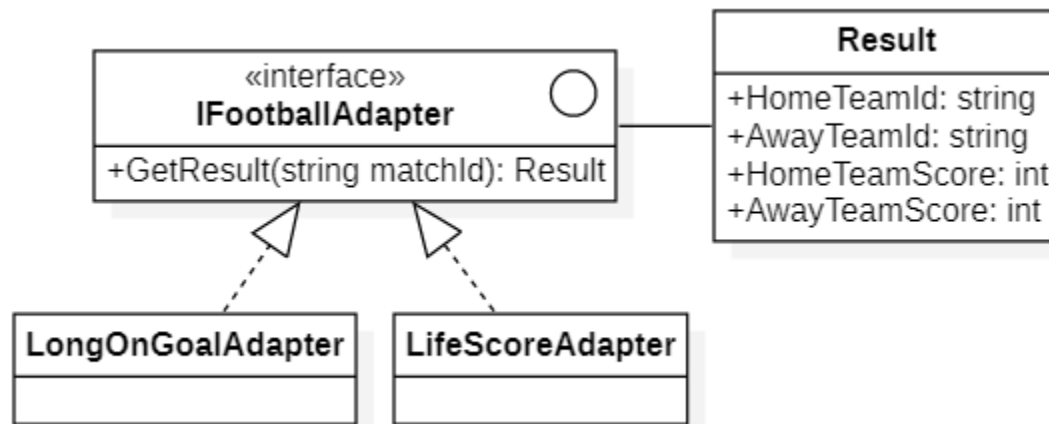
- โดยไม่ว่าข้อมูลของ API จากเว็บทั้ง 2 จะเป็นอะไร ตัว Wrapper มีหน้าที่แปลงให้อยู่ในรูปแบบที่เหมือนกัน ถ้าทำแบบนี้จะมีก็ API ก็ไม่เป็นปัญหาอีกต่อไป





# Structural Patterns : Adapter Pattern

- สามารถเขียนในรูปแบบของ Class Diagram ได้ดังนี้

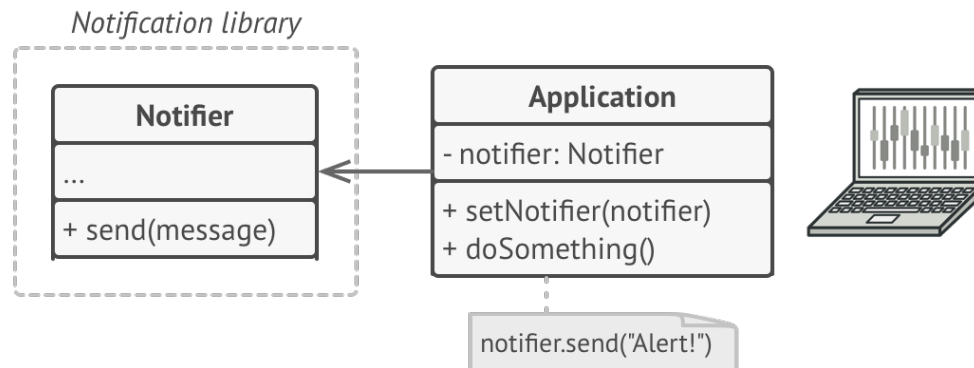


- การทำแบบนี้จะมีข้อดีอีกอย่าง คือ กรณีที่ API ของแต่ละเว็บมีการเปลี่ยนแปลง เราจะแก้ไขเฉพาะ Wrapper Class ของ API นั้นๆ เพียง Class เดียว และ ไม่มีผลกระทบกับ Class อื่นๆ เลย



# Structural Patterns : Decorator Pattern

- เป้าหมายของ Pattern นี้ คือ ทำหน้าที่เพิ่มความสามารถของ Class โดยไม่ต้องไปแก้ไข Class เดิม โดยการสร้าง Wrapper ครอบเข้าไป
- สมมติว่าเรากำลังพัฒนา library ที่ใช้ส่งข้อความออกไปตัวหนึ่ง ตามรูป



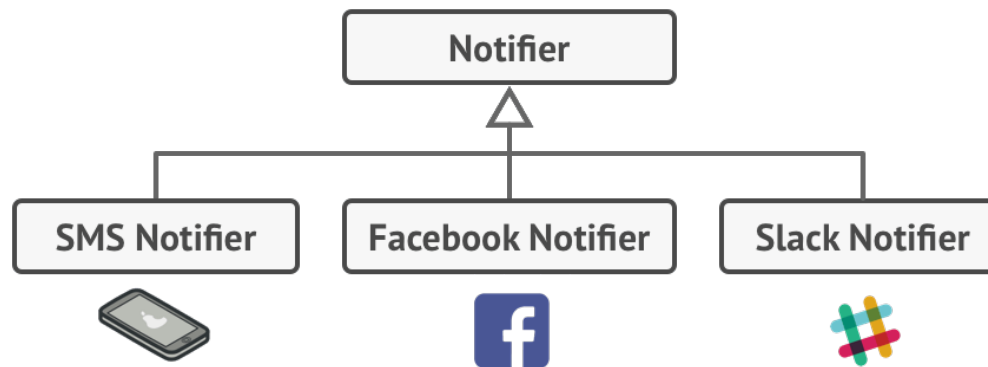
- เมื่อใครต้องการส่งข้อความออกไป เขาก็จะเรียกใช้ Send method เพื่อส่งอีเมลออกไป





# Structural Patterns : Decorator Pattern

- หลังจากนั้น ก็มีบางคนอยากส่งข้อความออกไปทางอื่นที่ไม่ใช้อีเมลบ้าง เช่น SMS, Facebook หรือ Slack
- ปัญหา คือ จะทำยังไงดี จะทำเป็น Subclass ตามรูปเดิมๆ มันก็ใช้ได้แต่ปัญหาที่ตามมา คือ หากต้องการส่งออกทั้ง 3 ทาง จะต้องสร้าง Object ของทั้ง 3 Class จึงจะส่งได้

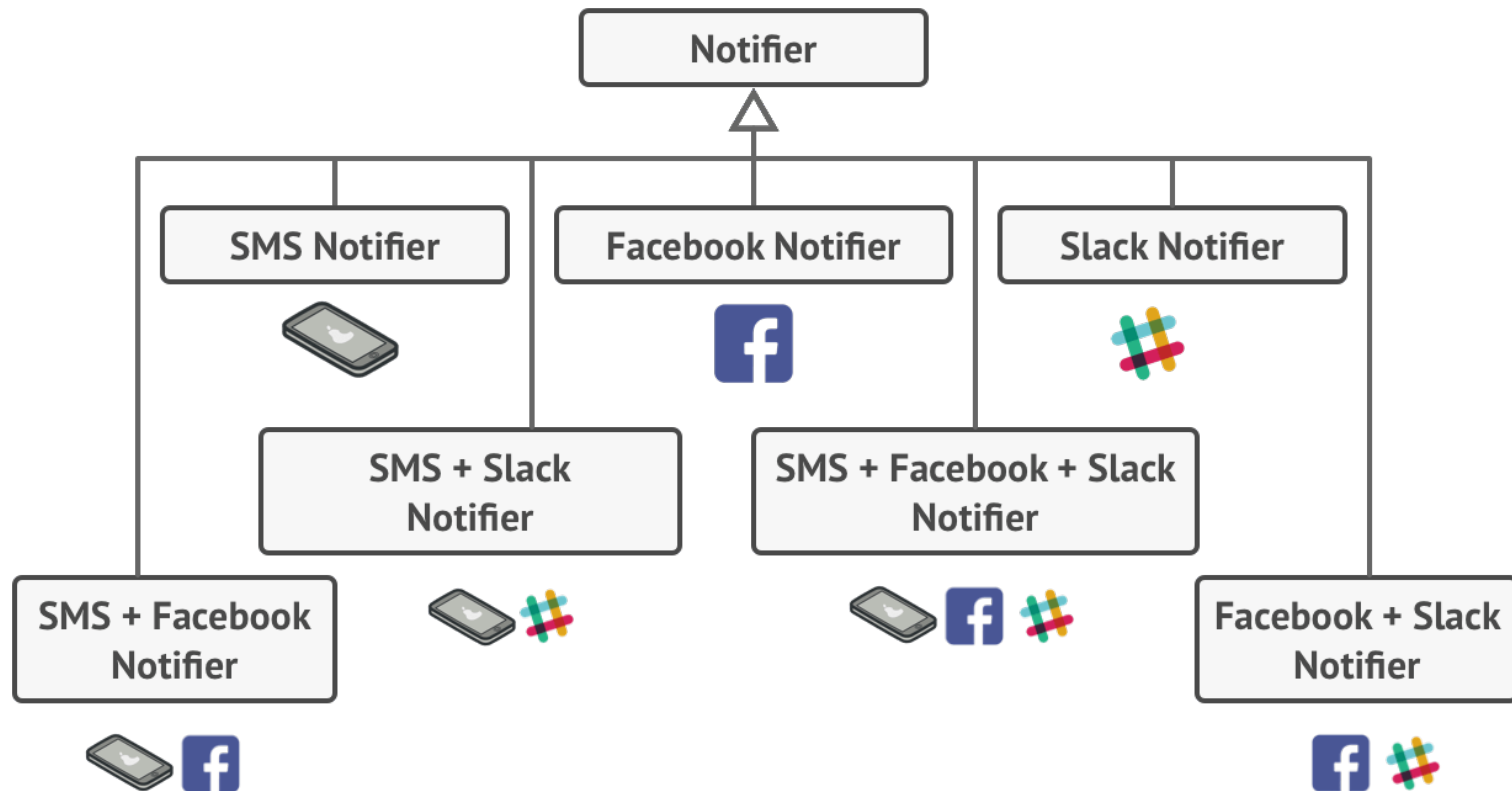


- ก็เป็นเรื่องที่ค่อนข้างยุ่งยาก



# Structural Patterns : Decorator Pattern

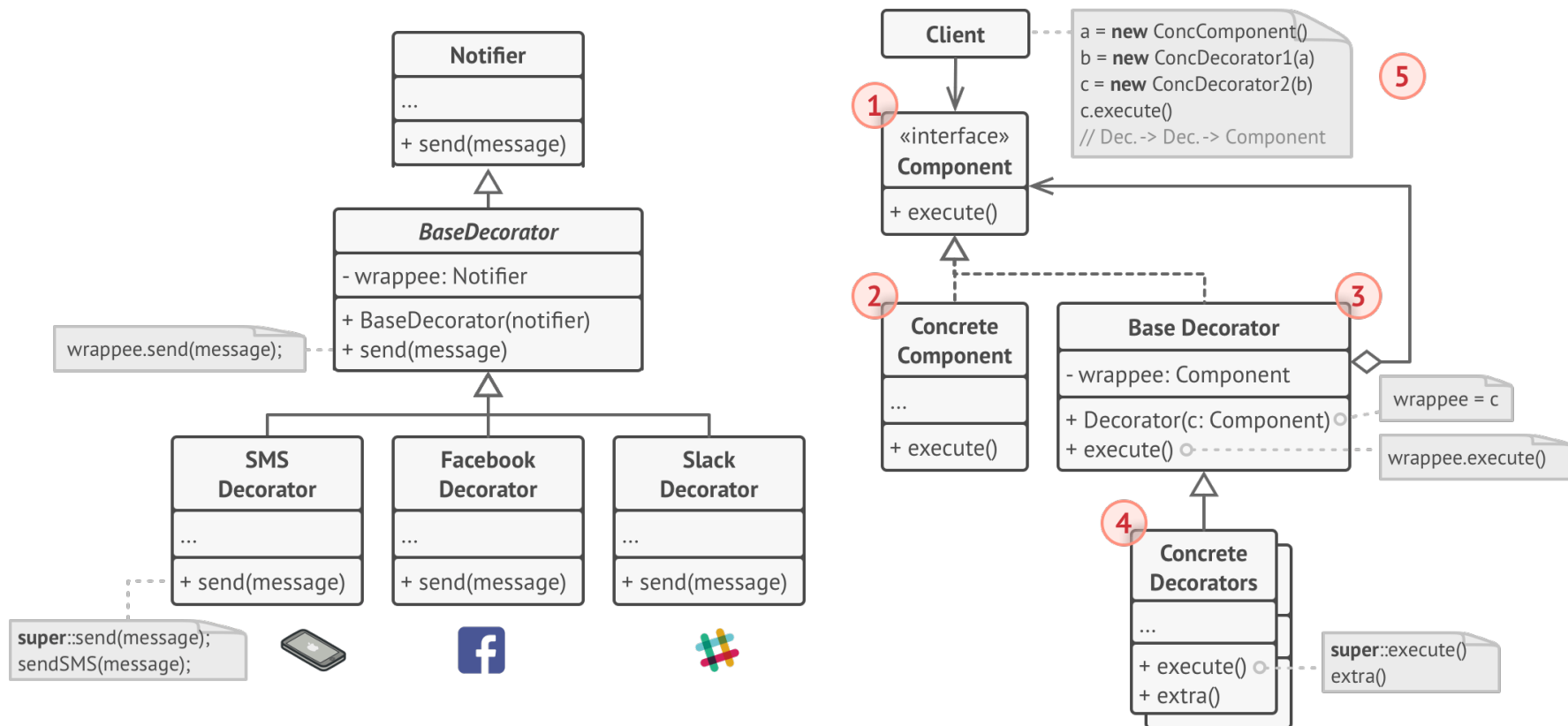
- หรือจะสร้างเป็น Class แบบนี้ ก็แก้ปัญหาก็ได้ แต่ไม่ค่อยเข้าท่าเท่าไร



# Structural Patterns : Decorator Pattern



- เราจะใช้ Decorator Pattern มาช่วย โดยแยก Notifier ออกมา แล้วสร้าง Decorator มาครอบ โดยให้ SMS, FB และ Slack เป็น Subclass ของ Decorator





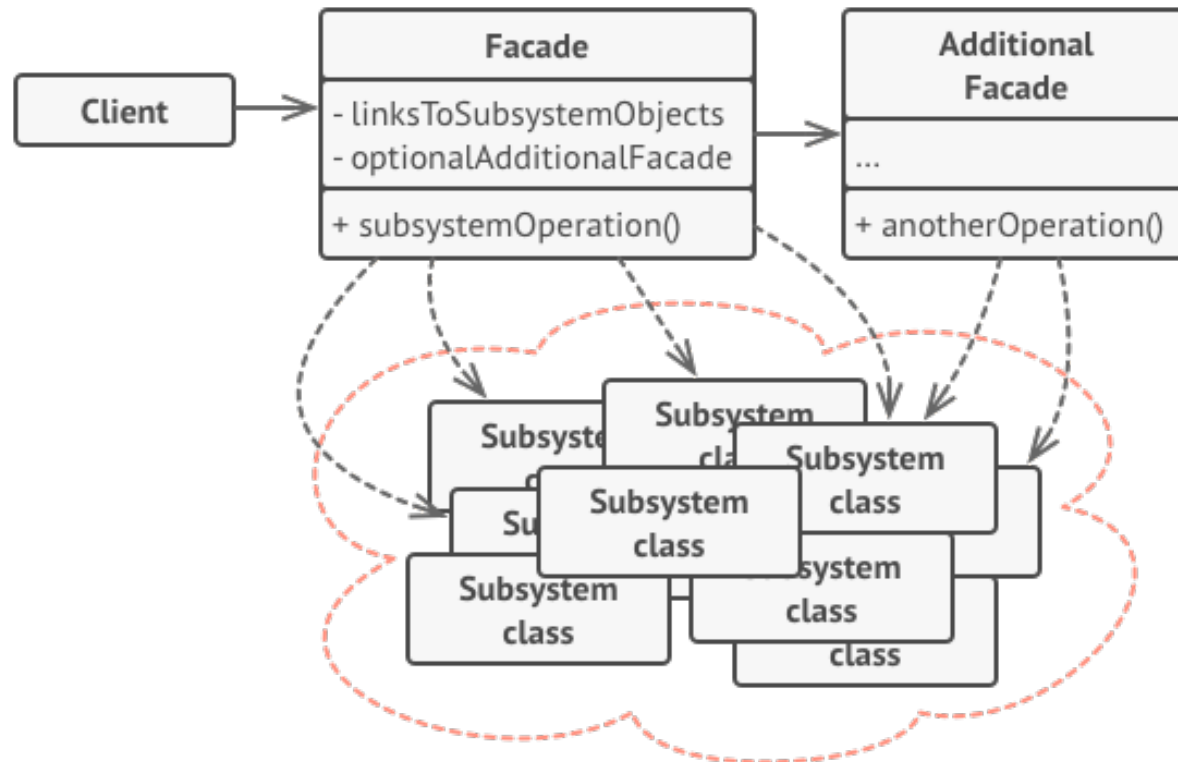
# Structural Patterns : Facade Pattern

- จุดประสงค์ของ Pattern คือ ทำให้ของที่ใช้งานยาก ใช้งานได้ง่ายขึ้น
- สมมติว่าเรากำลังเขียนโปรแกรม เพื่อต่อกับ library ที่ใช้ในการควบคุมระบบไฟฟ้า พลังงานนิวเคลียร์ ซึ่งภายใน library นี้มีโครงสร้างที่ค่อนข้างซับซ้อน เช่น ระบบหล่อเย็น ระบบความดัน และระบบควบคุมอื่นๆอีกมากมายที่ทำงานร่วมกันอยู่
- ถ้าเราจะสั่งให้ระบบแต่ละตัวทำงาน ระบบนั้นๆจะต้องตรงเงื่อนไขของมันก่อนถึงจะสั่งให้มันทำงานได้ และการสั่งในแต่ละขั้นตอนต้องเป็นตามลำดับที่ถูกกำหนดไว้ ไม่สามารถข้ามขั้นตอนได้ เช่นก่อนจะสตาร์ทระบบความดัน ระบบหล่อเย็นจะต้องถูกสตาร์ทก่อน ไม่งั้นอาจเกิดความเสียหายต่อระบบนั้นๆได้



# Structural Patterns : Facade Pattern

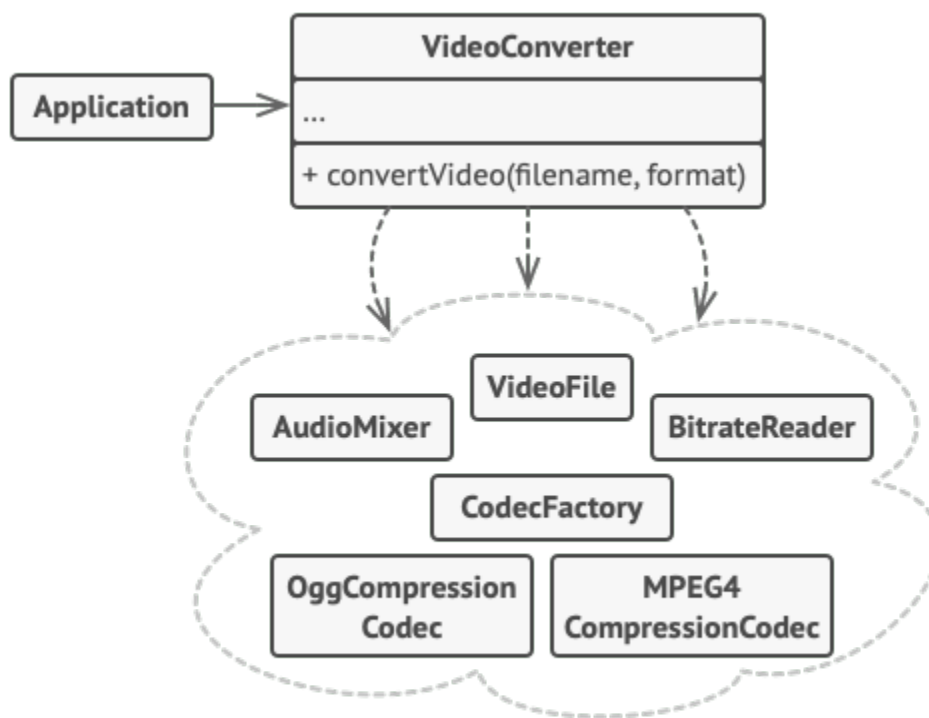
- โครงสร้างของ Facade คือ Class ที่รวมการทำงานของ Subsystem ต่างๆ เอาไว้ เพื่อให้การเรียกใช้สามารถทำได้ง่าย และอาจจะมีหลาย Façade ก็ได้



# Structural Patterns : Facade Pattern



- ตัวอย่าง สมมติว่าการจะ Upload VDO ประกอบด้วยหลายขั้นตอน เช่น ปรับขนาดไฟล์ให้เหมาะสม, เข้ารหัสให้ถูก Format ที่เว็บนั้นใช้, จัดการคุณภาพเสียง แล้วจึงอัปโหลดขึ้นเซิร์ฟเวอร์





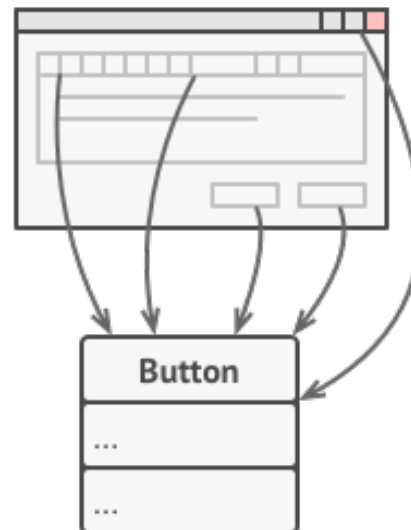
# Behavioral patterns

- เป็น Design Patterns ที่ช่วยให้ Class ต่างๆ ทำงานร่วมกัน ประกอบด้วย
  - Command Pattern
  - Observer Pattern
  - Iterator Pattern
  - State Pattern
  - Strategy Pattern
  - Template Pattern
  - Visitor Pattern
  - etc



## Behavioral patterns : Command Pattern

- เปลี่ยน Action ต่างๆให้กลายเป็น object ทำให้โปรแกรมจัดการ action ที่เข้ามาได้หลายรูปแบบ
- สมมติว่าเขียนโปรแกรม Text editor ตัวนี้ ซึ่งภายในโปรแกรมจะมี Toolbar อยู่ด้านบน ซึ่งภายในนั้นจะมีปุ่มต่างๆให้กดมากมาย ซึ่งปุ่มทุกปุ่มสร้างมาจาก Button class ตามรูป

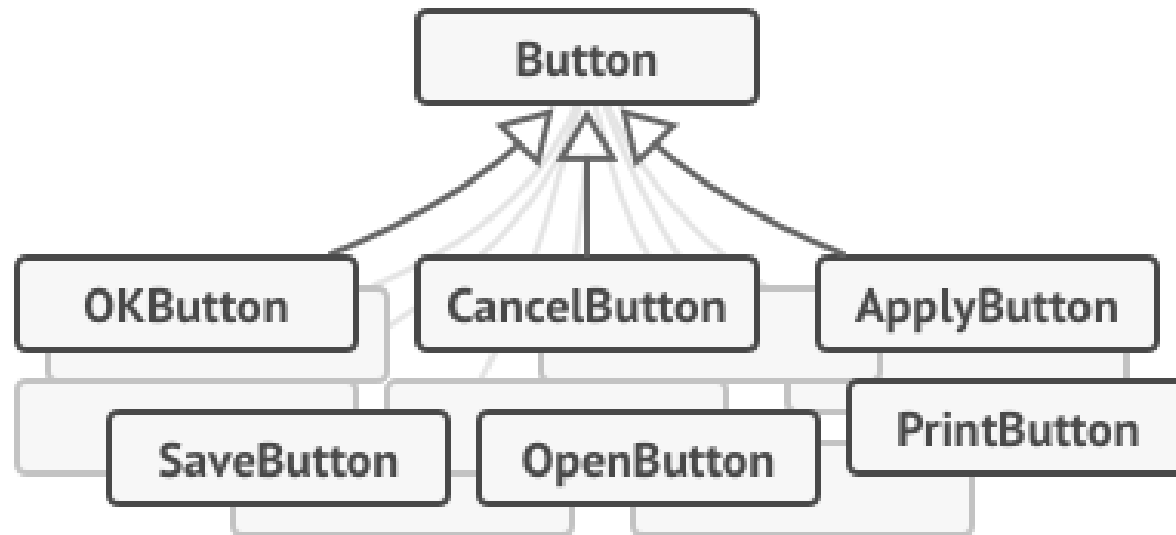






## Behavioral patterns : Command Pattern

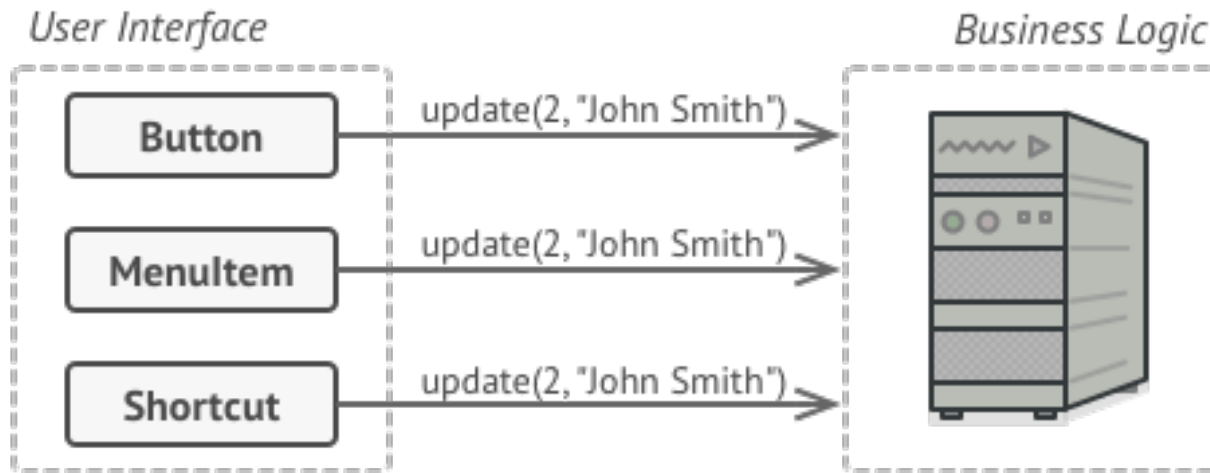
- เนื่องจากปุ่มที่อยู่บน toolbar มีการทำงานหลายแบบซึ่งไม่เหมือนกันเลย จึงต้องสร้าง subclass ขึ้นมา เพื่อให้ทำงานในรูปแบบต่างๆ กันได้ เช่น ปุ่มตกลง, ปุ่มยกเลิก, ปุ่มบันทึก ตามรูปด้านล่าง





## Behavioral patterns : Command Pattern

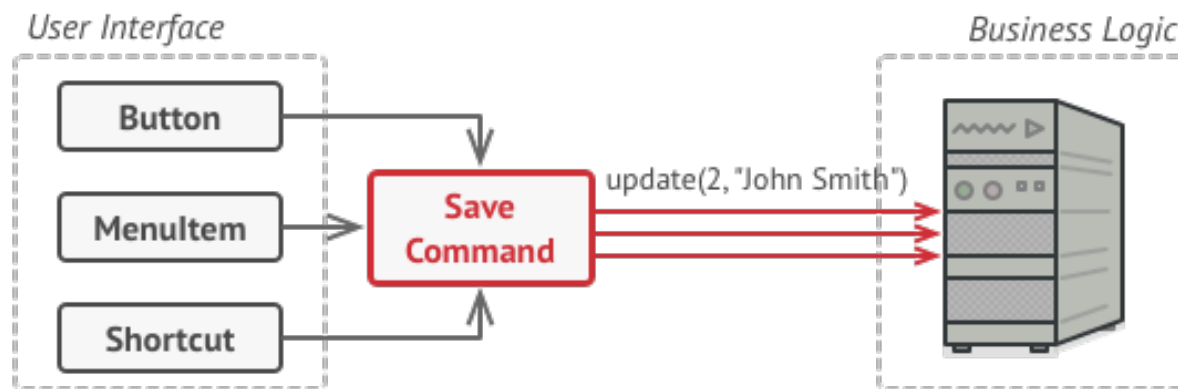
- ปัญหาของวิธีการข้างต้น คือ Subclass จะเยอะมาก
- นอกจากนั้นจะมีปัญหาอีกอย่าง คือ นอกจากปุ่ม save แล้ว ยังมีคำสั่ง save อีกต่างหาก ตามรูป





## Behavioral patterns : Command Pattern

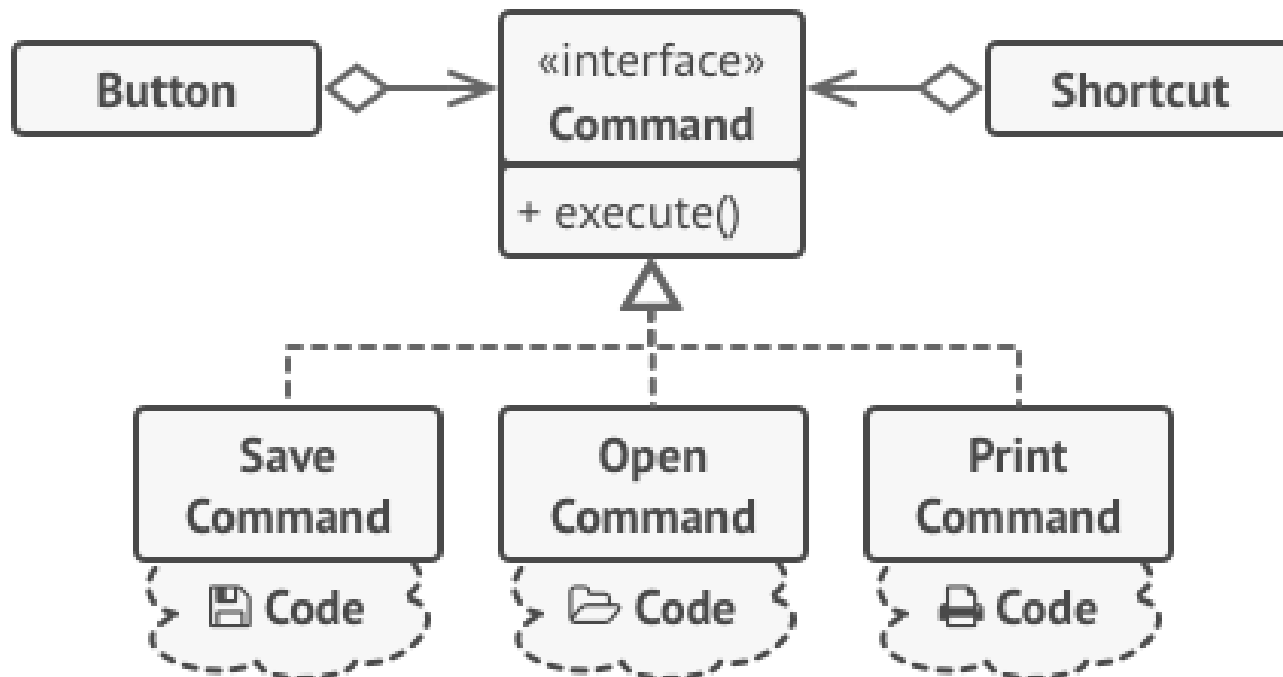
- ในกรณีเช่นนี้ จึงมีข้อเสนอว่า ให้เปลี่ยนสิ่งที่ผู้ใช้กระทำ (Action) มาเป็น object และถ้ามีข้อมูลอะไรที่เกี่ยวข้อง ก็ให้รวมใน object นั้นเลย ซึ่ง object นี้จะเรียกว่า Command นั่นเอง
- ดังนั้นปุ่ม save ถูกกด จะไม่บันทึกข้อมูล แต่จะไปสร้าง command object ขึ้นมา ซึ่งเจ้าตัว command object จะเป็นเสมือนตัวเชื่อมระหว่าง UI layer กับ Business logic layer





## Behavioral patterns : Command Pattern

- ตัว Command ทุกตัวจะ implement Interface เดียวกัน ซึ่ง interface นั้นจะมีแค่ Execute() method เท่านั้น ทำให้คนที่ส่ง/รับ command ไม่ต้องผูกติด (coupling) กับ concrete command ใดๆ





## Behavioral patterns : Observer Pattern

- ทำหน้าที่สร้างกลไกในการติดตามเหตุการณ์
- สมมติว่าเราเขียนแอปขายของบนมือถือตัวนี้อยู่ ซึ่งมี class อยู่ 2 ตัวคือ Customer และ Store
- เมื่อมีสินค้าใหม่ๆเข้ามา ก็จะอัปเดตสินค้าลงใน database ถ้า Customer อยากรู้ว่ามีสินค้าอะไรใหม่เข้ามาบ้าง ก็จะต้องไปถามจาก Store
- แต่ทุกครั้งที่ Customer เข้ามาดึงข้อมูลจาก Store ก็ไม่ได้หมายความว่าจะมีสินค้าใหม่เข้ามาทุกครั้ง ทำให้ในหลายๆ ครั้งที่ไปดึงข้อมูลมันจะทำงานฟรี ทั้ง Customer และ Store เลย
- แล้วถ้าทำตรงข้าม คือ ถ้ามีสินค้าใหม่ ก็ให้ Store แจ้งไปที่ Customer จะดีมั้ย ปัญหา คือ ไม่ใช่ Customer ทุกคนที่อยากรู้



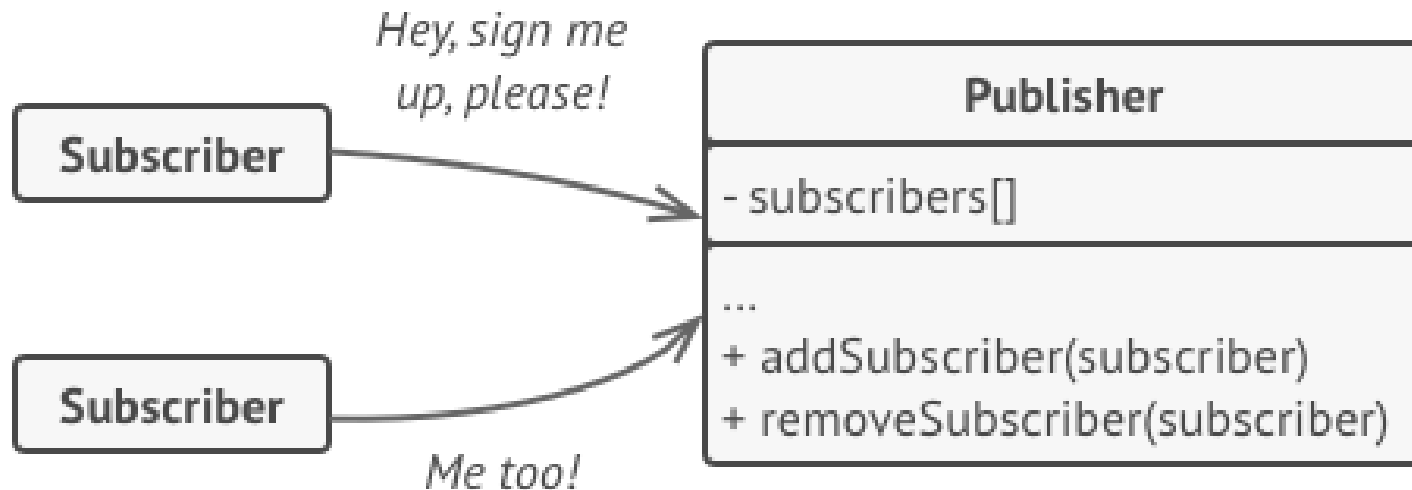
## Behavioral patterns : Observer Pattern

- วิธีการแก้ก็คือใช้ Observer Pattern โดยวิธีการมีดังนี้
- อะไรก็แล้วแต่ที่มีคนสนใจอยากรู้สถานะของมัน เราจะเรียกมันว่า Subject (ในตัวอย่างนี้คือ สถานะว่ามีสินค้าใหม่หรือเปล่า)
- เมื่อไหร่ก็ตามที่ Subject มีการเปลี่ยนแปลง จะทำให้ต้องส่งข้อความแจ้งเตือนออกไป จะเรียกตัวที่ส่งการแจ้งเตือนออกไปว่า Publisher (ในตัวอย่างนี้คือ Store)
- ใครก็ตามที่สนใจอยากรู้ว่า Subject มีการเปลี่ยนแปลงไหม เราจะเรียกว่า Subscriber (ในตัวอย่างนี้คือ Customer)



## Behavioral patterns : Observer Pattern

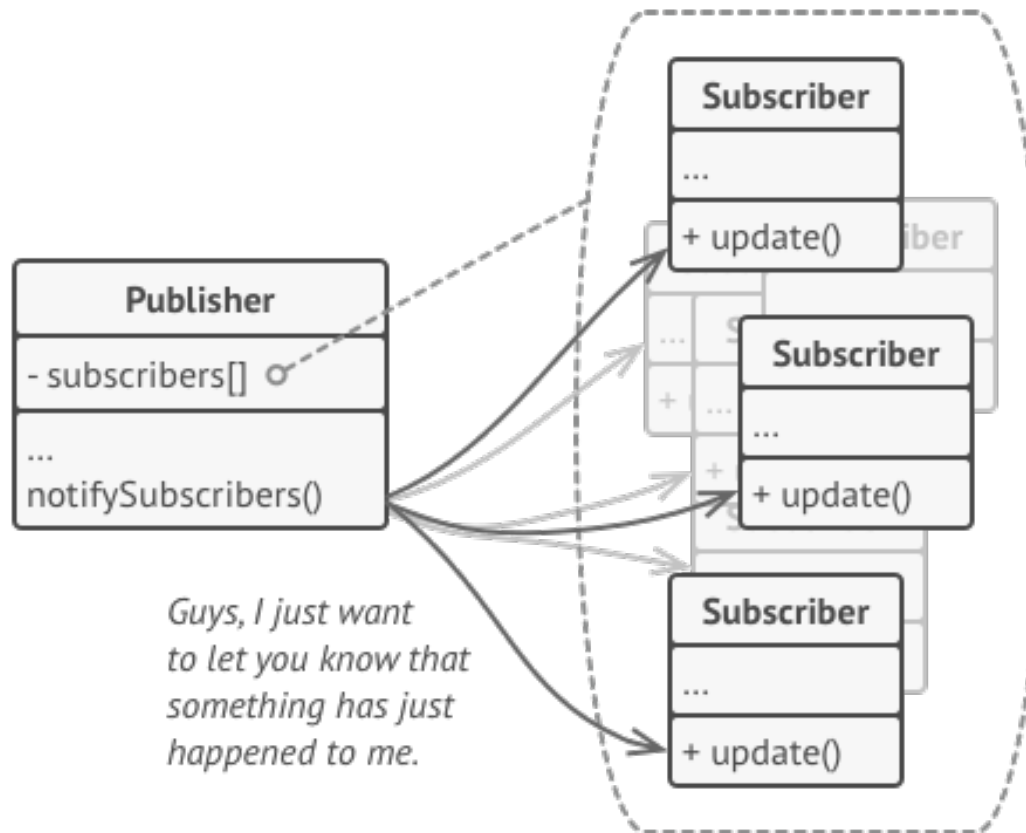
- Subscriber มีหน้าที่มาขอติดตาม (Subscribe) ข้อมูลที่ตัวเองสนใจ (หรือยกเลิกการติดตาม) กับ Publisher (ตามรูป)





## Behavioral patterns : Observer Pattern

- คราวนี้เมื่อ Subject มีการอัปเดตเกิดขึ้น Publisher ก็จะส่งการแจ้งเตือนให้กับ Subscriber ที่ได้ขอเข้ามาติดตามข่าวสารนั่นเอง ตามรูป

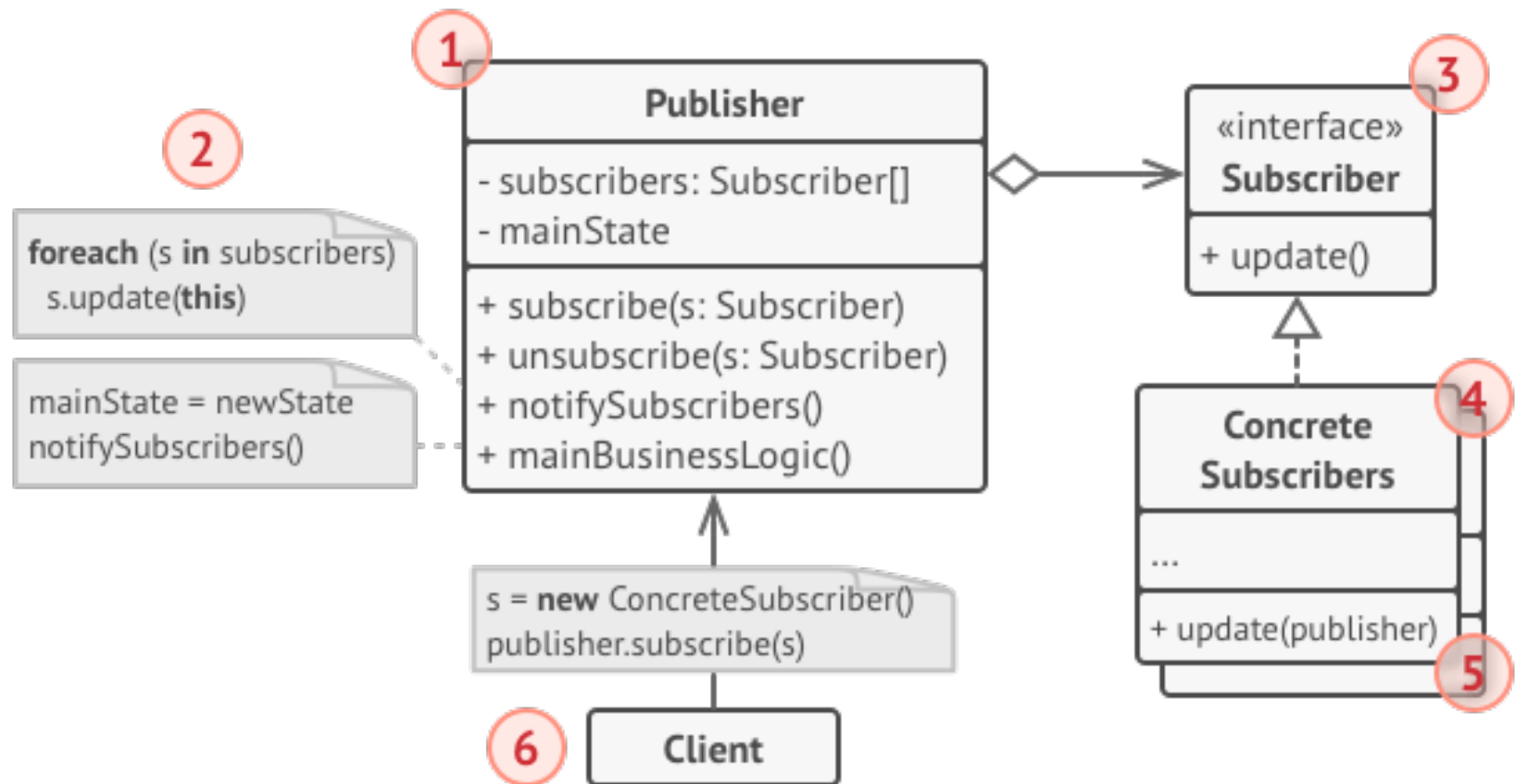






# Behavioral patterns : Observer Pattern

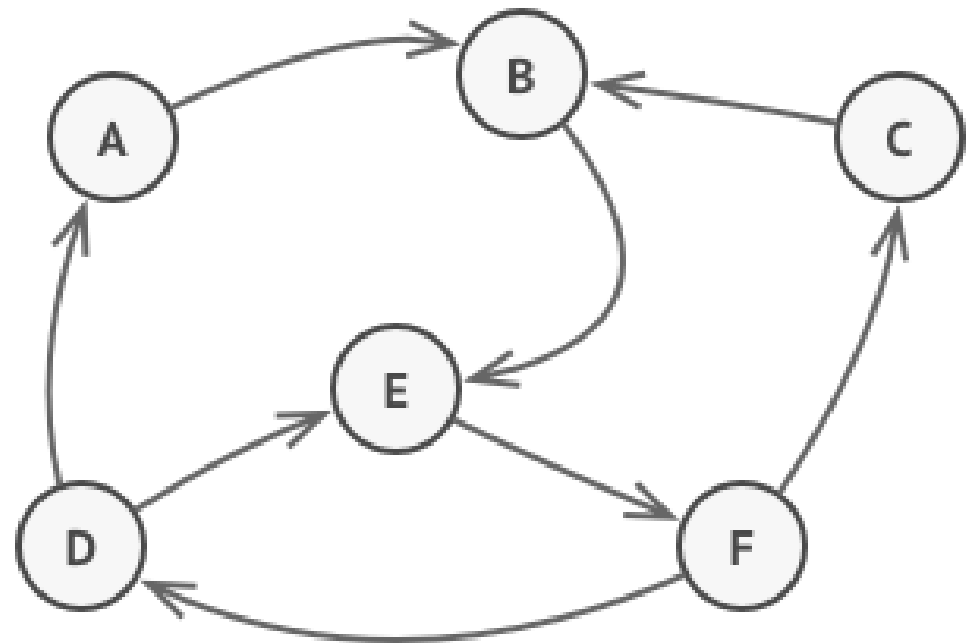
- โครงสร้างของ Pattern





## Behavioral patterns : State Pattern

- เป้าหมายของ Pattern คือ Object ที่สามารถเปลี่ยนการทำงานตาม State
- จะคล้ายกับ Finite State Machine คือ Object จะอยู่ใน State ใด State หนึ่ง ไม่สามารถเป็น 2 State ได้
- Object จะเปลี่ยน State ได้ ถ้าตรงกับเงื่อนไขที่กำหนด
- จำนวน State ต้องมีจำนวนจำกัด





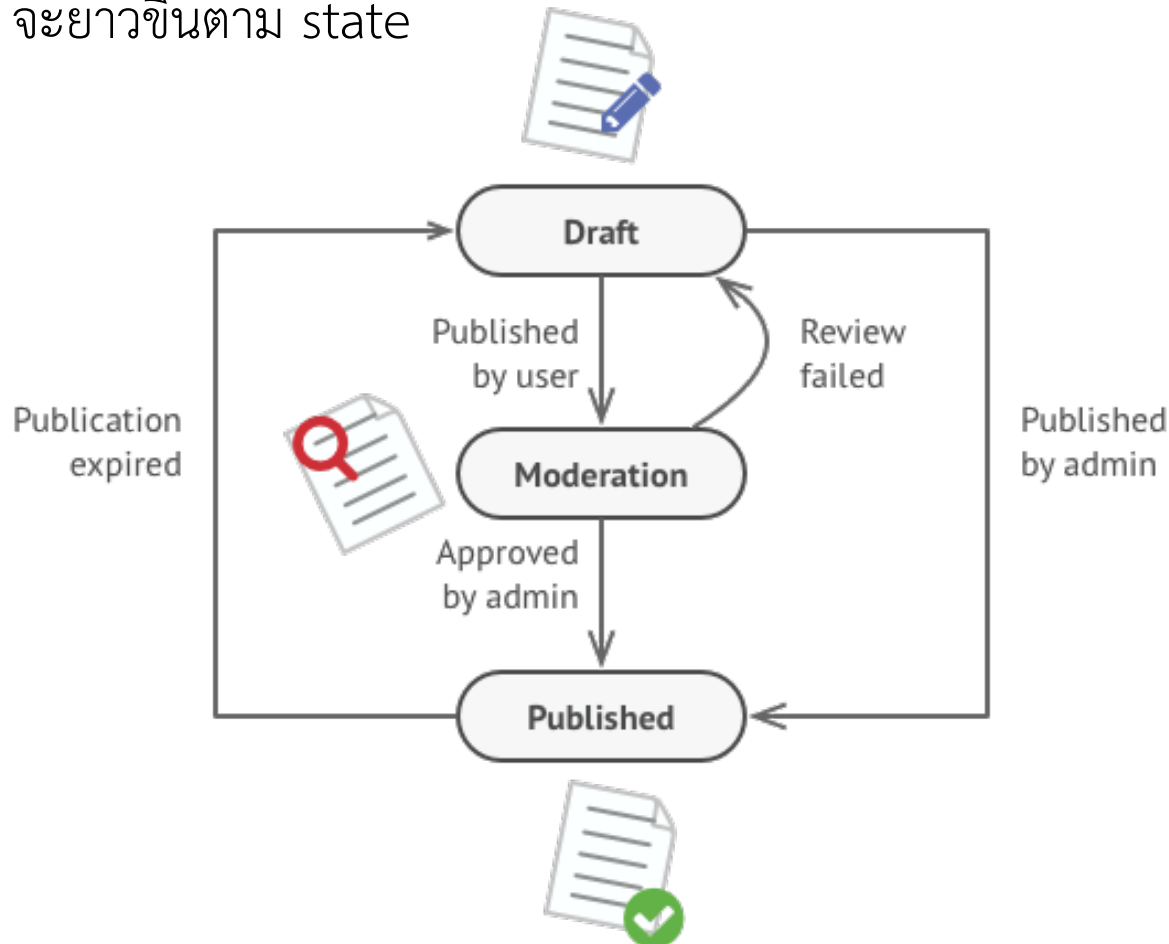
## Behavioral patterns : State Pattern

- สมมุติว่ามี Class ที่ชื่อ Document ซึ่ง Object ของมันก็สามารถเปลี่ยนเป็น state ต่างๆได้ เช่น กำลังร่างเอกสาร (Draft) กำลังตรวจสอบ (Moderation) ตีพิมพ์แล้ว (Published)
- สมมุติอีกว่า Document มี method 1 ตัวที่ชื่อว่า Publish มีหน้าที่ไว้เผยแพร่เอกสาร ซึ่งถ้าเรียกใช้งาน method นี้ มันจะทำงานยังไงขึ้นอยู่กับว่าตอนนี้เป็น State อะไรตามนี้
  - ถ้าอยู่ใน state กำลังร่างเอกสาร (Draft) จะเปลี่ยน state เป็น กำลังตรวจสอบ (Moderation) ถ้าคนที่กดเป็น Admin จะเปลี่ยน state เป็น ถูกตีพิมพ์แล้ว (Published)
  - ถ้าอยู่ใน state กำลังร่างเอกสาร (Moderation) ถ้าคนที่กดจะต้องเป็น Admin จะเปลี่ยน state เป็น ถูกตีพิมพ์แล้ว (Published) แต่ถ้าไม่ใช่ จะไม่ทำอะไรเลย
  - ถ้าอยู่ใน state ถูกตีพิมพ์แล้ว (Published) มันจะไม่ทำอะไรเลย



## Behavioral patterns : State Pattern

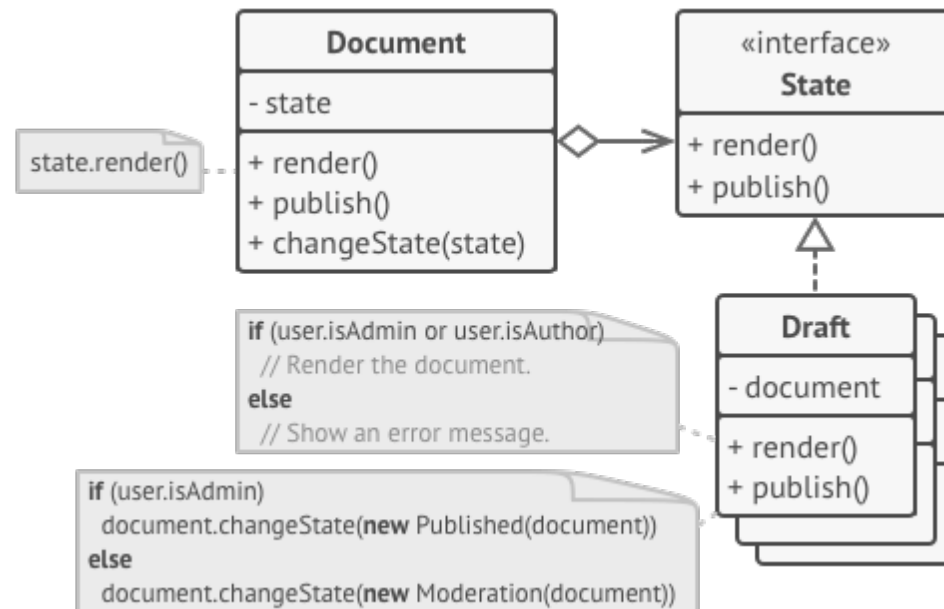
- เขียนเป็น state ได้ตามรูป ซึ่งเราอาจใช้ if else ในการกำหนดเงื่อนไขก็ได้ แต่ปัญหาคือ code จะยาวขึ้นตาม state





# Behavioral patterns : State Pattern

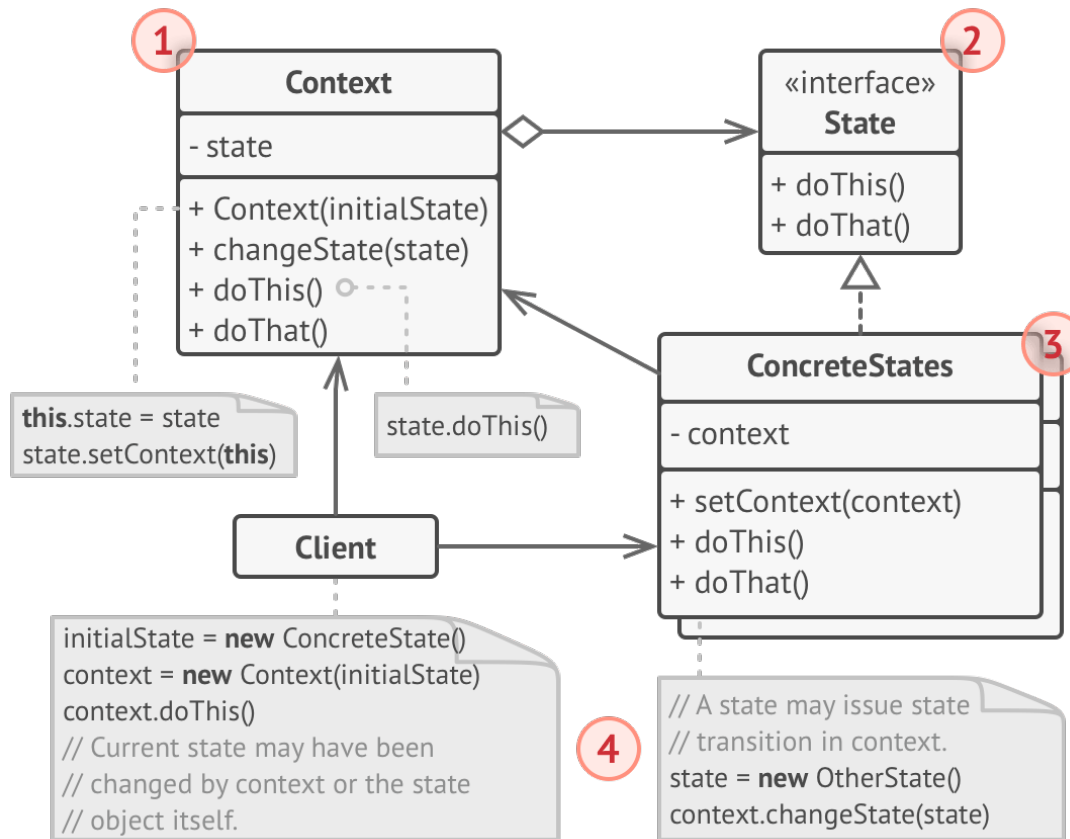
- ใน State Pattern จะแก้ปัญหาดังนี้
  - แยก state ทั้งหมดภายใน object ออกไปเป็น class ของมันเอง และย้ายเงื่อนไขต่างๆของมันไปอยู่ภายใน class ใหม่ด้วย
  - จากนั้นให้ object หลัก reference ไปยัง state class เพื่อจะได้ไม่มีเงื่อนไขไปอยู่ใน object หลัก





# Behavioral patterns : State Pattern

- การทำงานจะเป็นไปตามรูป





# Design Pattern : Conclusion

- สรุปว่า การเรียนรู้ Design Pattern จะเป็นประโยชน์ในการออกแบบ Software โดยมีเป้าหมายให้ Software ง่ายต่อการดูแลรักษา สามารถขยายความสามารถได้ง่าย



*For your attention*