Design patterns VS Anti-Pattern

**Design patterns** เป็นแนวคิดในการแก้ปัญหาที่เราเจอบ่อยๆในการออกแบบซอฟต์แวร์ ซึ่งถ้าเรามี ปัญหา แล้วปัญหานั้นมีลักษณะตรงกับ pattern ไหนก็ตาม เราก็จะสามารถนำแนวคิดของ pattern นั้นๆไปแก้ปัญหาของเราได้เลย

**Anti-Pattern** ตรงข้าม Design Pattern คือ แบบแผนที่ไม่ดี แก้ปัญหาเฉพาะหน้า เป็นการ Workaround

Functional Programming

เป็นรูปแบบ “การเขียนโปรแกรมเชิงฟังก์ชัน” การเขียนโปรแกรมแบบ functional นั้น จะเน้นไปที่การใช้ Pure Function ซึ่งหลักการของมันมีแค่ 3 ขั้นตอนง่ายๆคือ

1.รับค่า

2.ดำเนินการบางอย่างกับค่าเหล่านั้น

3.return ผลลัพธ์ออกไป

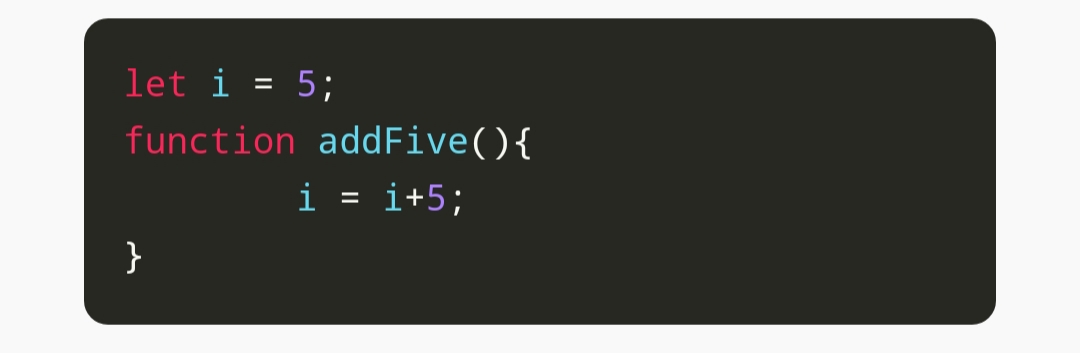
การเขียนของมันจะต้องไม่มีการยุ่งเกี่ยวหรือเปลี่ยนแปลงอะไรก็ตามภายนอก function เด็ดขาด ดังนั้นฟังก์ชันที่เขียนออกมานั้นไม่ว่าจะใส่ค่าอะไรซ้ำไปกี่รอบ ก็จะออกมาเป็นค่าเดิมเสมอ (Immutability)

ตัวอย่าง



สิ่งสำคัญที่สุดของ Functional Programming นั้น คือการลด side effect ที่เกิดจาก function เมื่อฟังก์ชันนึงมีผลกระทบกับสิ่งที่อยู่ภายนอกหรือฟังก์ชันอื่นๆที่รับค่านั้นเข้าไป ทำให้เกิดปัญหากับโปรแกรม ด้วยเหตุนี้ทำให้การใช้ functional programming ทำให้โค้ดอ่านและแก้ไขได้ง่าย

ตัวอย่าง ที่ทำให้เกิด side effect กับตัวแปร i

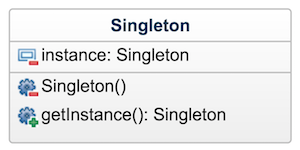


Design patterns

Singleton Pattern

เป็นรูปแบบการออกแบบซอฟต์แวร์ที่จำกัดจำนวนของ Object ที่ถูกสร้างขึ้นในระบบ ซึ่งจะเป็นประโยชน์เมื่อระบบต้องการจะมี Object นั้นเพียงตัวเดียวเพื่อป้องกันไม่ให้เกิดการทำงานซ้ำซ้อนกันเช่น class สำหรับการเก็บข้อมูล หรือเป็น Modelที่มีการเรียกใช้งานทั้งระบบ

**โครงสร้างโค้ดที่เป็นแบบ Singleton (JAVA)**



**วิธีการใช้**

ถ้าเราอยากให้คลาสไหนถูกจำกัดจำนวนในการสร้าง object เราก็แค่ ห้ามให้คนอื่นสร้าง object ได้ตามใจด้วยการทำให้คลาสนั้นเป็น private constructor ส่วนเงื่อนไขและการสร้าง object ตัวนั้นก็จะถูกจัดการอยู่ภายในคลาสตัวนั้นเอง และเปิดช่องทางให้คนอื่นเข้าถึงผ่าน static member

**ข้อดี**

1.ไม่เสียเวลาในการสร้าง

2.ไม่เปลือง memory

**ข้อเสีย**

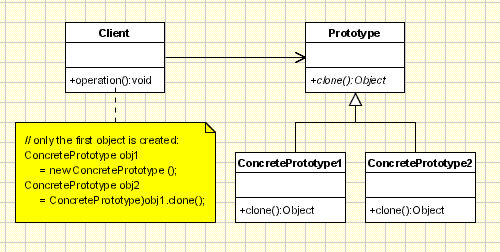
1.ตอนจะสร้างถ้ามันต้องไปทำงานนั่นนู่นนี่เยอะ มันจะทำให้โปรแกรมดูหน่วงๆหน่อยนึง จนกว่าจะสร้างเสร็จ

2.มีปัญหากับการทำงานแบบ Multi-Threading เพราะมันมีโอกาสเข้าไปสร้าง instance พร้อมกัน

Prototype Pattern

Prototype pattern มีประโยชน์ทางด้านการใช้ Object ที่มีจำนวนมาก ๆ โดยที่ว่าเราไม่ต้องไป New Object กันอีกต่อไปเรื่อย ๆ ซึ่งทำให้เปลืองทรัพยากรด้วย แต่เราก็จะต้องมีตัวต้นแบบ และก็ Clone มาเรื่อย ๆหลักการ

โครงสร้างของ pattern นี้



**อธิบาย**

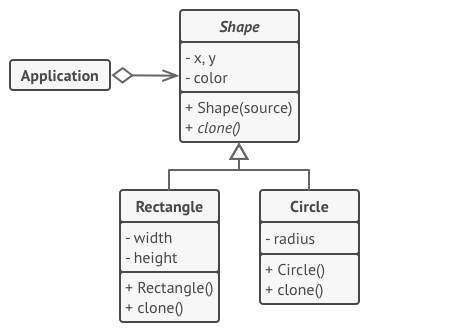
Client - จะสร้าง Object มาแล้วไปถาม Prototype Class ว่าจะคัดลอก

Prototype - กำหนด Interface class สำหรับการสร้าง Object และ Clone

ConcreatePrototype - เป็นรูปร่างหน้าตาของสิ่งที่สร้างขึ้นมาและสามารถ โคลนตัวเองได้

**ตัวอย่างการนำไปใช้งาน**

ในตัวอย่างนี้เราจะไปก๊อปปี้ object ของ class รูปทรงต่างๆ (สี่เหลี่ยม, วงกลม) โดยที่มี base เดียวกัน



**อธิบาย**

Shape มี method ชื่อ Clone ซึ่งเอาไว้สำหรับทำก๊อปปี้ object และภายใน constructor ก็รับตัวมันเองเพื่อเอาไว้กำหนดค่าเมื่อมีการเรียกใช้ method clone

Rectangle และ Circle เป็น subclass ของ Shape ดังนั้นเวลาที่มันจะก๊อปปี้ มันก็แค่สร้าง object ตัวมันเองแล้วส่งกลับให้ client ส่วน field ต่างๆที่มันไม่มีใน base class มันก็แค่ไปกำหนดไว้ใน constructor

ของมันเองก่อน

**ตัวอย่าง**

object A อ้างหา object B แล้ว object B ก็อ้างกลับไปหา object A

var a = new ObjectA();

var b = new ObjectB();

a.Ref = b;

b.Ref = a;

**ข้อดี**

สามารถก๊อปปี้/โคลน object ได้เลยโดยที่โค้ดของเราไม่ไปผูกติดกัน (coupling)

สามารถก๊อปปี้/โคลน object ที่มีความซับซ้อนได้ง่ายขึ้น

ลดการซ้ำซ้อนของโค้ดตอนทำการก๊อปปี้

**ข้อเสีย**

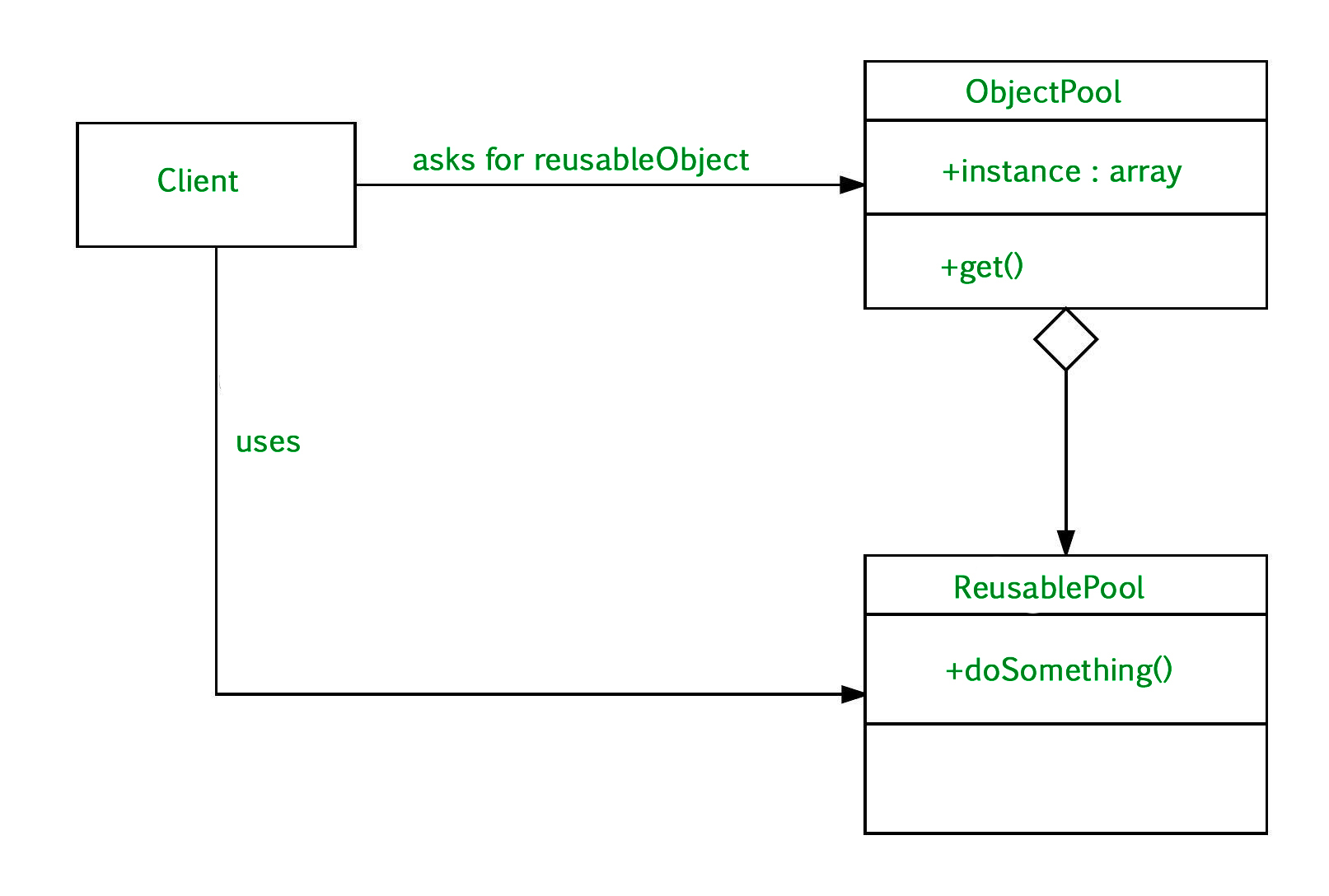
ถ้า object มีการอ้างกันแบบงูกินหาง (circular references) จะทำได้ยาก และอาจเกิดปัญหาตามมาภายหลัง

Object Pool

Object pool ก็คือการ มีวัตถุจำนวนหนึ่ง อยู่ใน pool และหากต้องการใช้อีกครั้งก็จะมีการนำกลับมา

ลองนึกภาพถึงห้องสมุด จำนวนคนที่ไปยืมหนังสือกับจำนวนหนังสือนั้นไม่เท่ากัน แต่จะมีช่วงๆนึงเช่นช่วงจะสอบไฟนอล คนกลุ่มๆนึงต้องการใช้หนังสือเล่มเดียวกัน ทำให้ห้องสมุดนั้นต้องมีหนังสือจำนวนเพียงพอต่อการใช้งาน แล้วพอใช้งานเสร็จ ก็นำหนังสือมาคืน นี้ก็คือการใช้งาน Object pool pattern

เหมาะสำหรับระบบที่มีการสร้าง Object ชนิดเดียวกันจำนวนมากๆ เพราะจะสามารถ Utilize การใช้ object pool ได้อย่างเต็มที่



**ข้อดี**

มีการนำกลับมาใช้ซ้ำ

ประหยัดเวลา

ลดต้นทุน

**ข้อเสีย**

อาจจะเปลืองพื้นที่มากกว่า