ที่ใช้งานจริงแน่ๆ

Software Design Pattern

Parinya Ekparinya

Parinya.Ek@kmitl.ac.th

Acknowledgement

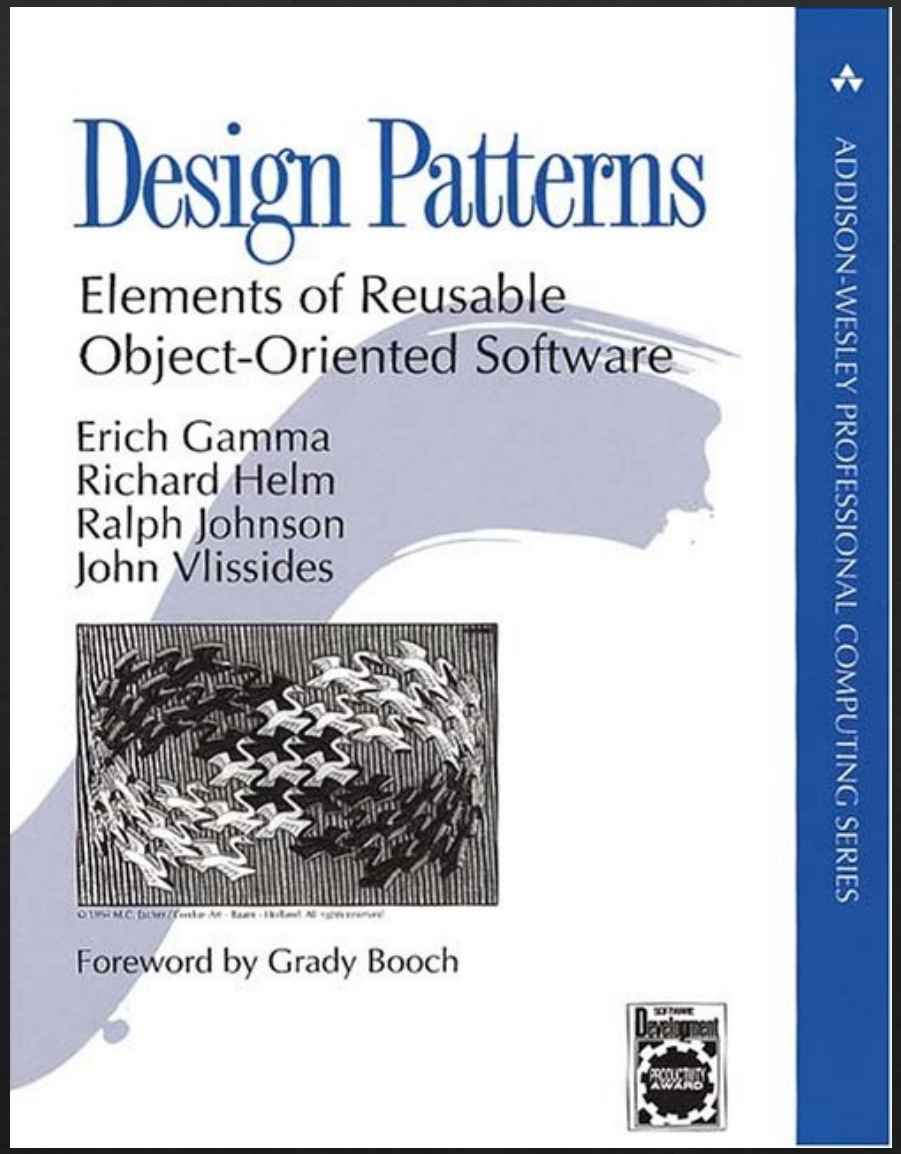
The content of the following slides are partially based on the listed material as follows:

- Object-Oriented Patterns & Frameworks by Dr. Douglas C. Schmidt
- https://en.wikipedia.org/wiki/Factory_method_pattern
- https://en.wikipedia.org/wiki/Abstract_factory_pattern
- https://en.wikipedia.org/wiki/Builder_pattern
- https://en.wikipedia.org/wiki/Prototype_pattern
- https://en.wikipedia.org/wiki/Singleton_pattern
- https://en.wikipedia.org/wiki/Dependency_injection

(Software) Design Pattern

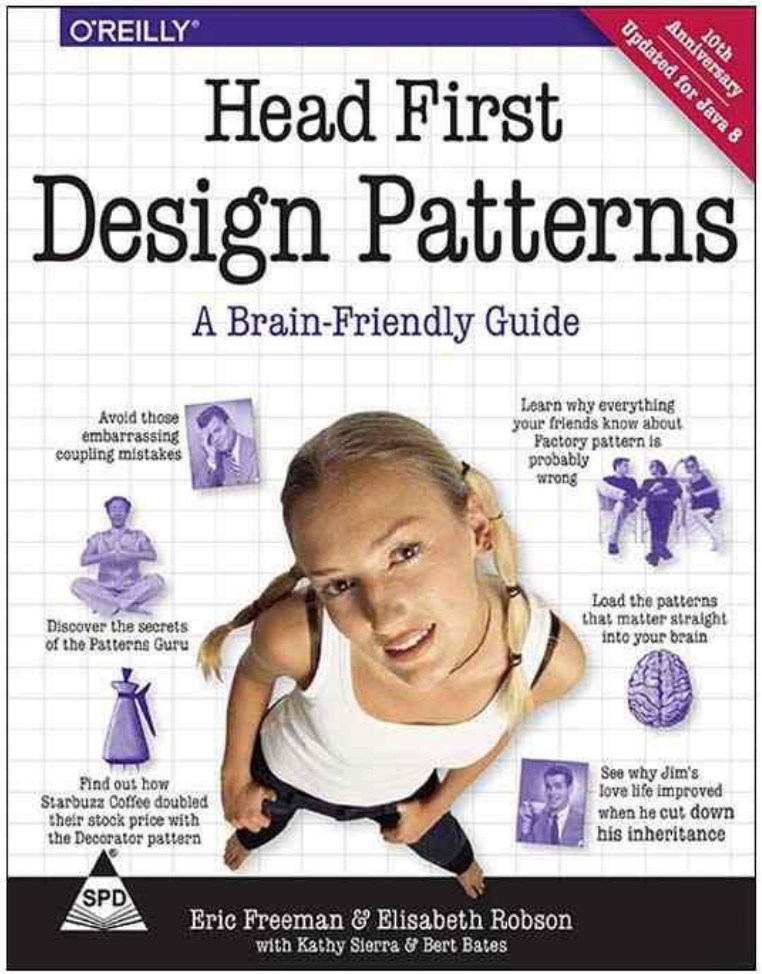
- A software design pattern is a general, reusable solution to a commonly occurring problem in the context of modular software design.
- In 1994, the concept of software design patterns for object-oriented software was published in ***"Design Patterns: Elements of Reusable Object-Oriented Software"*** by Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm.
- People often call these four authors "the Gang of Four".
- So, their book is often called "the GoF book".
- Originally, GoF design patterns covers 23 classic design patterns.
- After that people keep introducing more design patterns.

→ แนะนำให้อ่าน



The GoF Book (Smalltalk)

อ่าน 3 บทแรก



Another useful book (Java)

2nd Edition
December 2020

LSP ห้ามลบเม็ด

Relationship with Other Design Concepts

Architectural Styles

Reference Architectures / Domain Specific Architectures

reuse code , clean

Design Patterns

เฉพาะภาษา ไม่ physical

Design/Coding Idioms

Benefits & Limitations of Design Patterns

Benefits

- Design reuse
- Uniform design vocabulary
- Enhance understanding, restructuring, & team communication
- Basis for automation
- Transcends language-centric biases/myopia
- Abstracts away from many unimportant details

Limitations

- Require significant tedious & error-prone human effort to handcraft pattern implementations design reuse
- Can be deceptively simple uniform design vocabulary
- May limit design options
- Leaves important (implementation) details unresolved

WARNING:

Overuse of design patterns can lead to code that is downright over-engineered. Always go with the simplest solution that does the job and introduce patterns only where the need emerges.

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2008). *Head first design patterns*. " O'Reilly Media, Inc."

Four Basic Parts of a Pattern

1. Name
2. Problem (including "applicability") ปัญหา
3. Solution (both visual & textual descriptions) ทางแก้ ที่ใช้ซ้ำได้
4. Consequences (the results and trade-offs of applying the pattern) ผลที่ตามมา

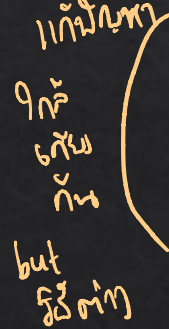
Classification of Design Patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.

มี 6

Creational Design Patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton
- Dependency Injection



สร้าง object เดียวเท่านั้น

ไม่ต้องเขียนเอง เดี๋ยวมีคนช่วยใส่

Factory Method

ปัญหา

Factory Method: Problem

- How can an object be created so that subclasses can redefine which class to instantiate?
- How can a class defer instantiation to subclasses?

Applicability

- When a class cannot anticipate the objects it must create or a class wants its subclasses to specify the objects it creates

วิธีแก้

Factory Method: Solution

Intent

Signature ของ object

- Provide an interface for creating an object, but leave choice of object's concrete type to a subclass

The Factory Method design pattern describes how to solve such problems:

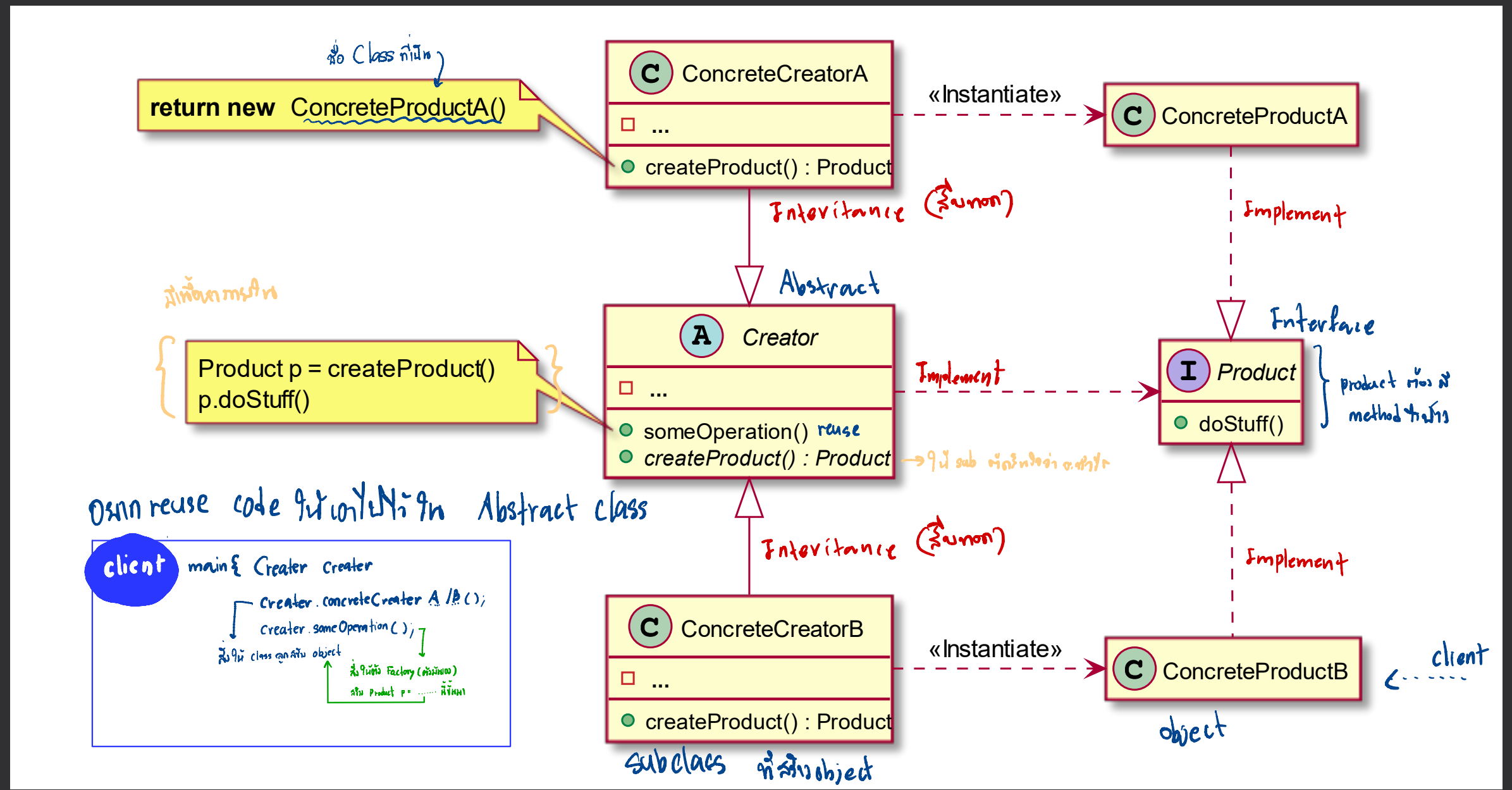
แยก (method)

- Define a separate operation (factory method) for creating an object

1 method ที่เป็น factory (class สร้าง object) ที่อยู่ใน base class

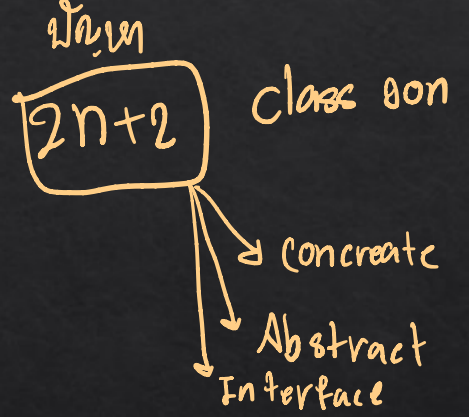
- Create an object by calling a factory method

ฐานสร้างอ้างที่ sub ใดๆ sub ไปเรียกใช้ factory method


ชื่อ Class ที่เป็น
return new ConcreteProductA()
ConcreteCreatorA
...
createProduct() : Product
«Instantiate»
ConcreteProductA
Intevitance (สืบทอด)
Implement
Abstract
Interface
มีเพื่อการใช้งาน
Product p = createProduct()
p.doStuff()
Creator
...
someOperation() reuse
createProduct() : Product
Implement
Product
doStuff()
product เพื่อ มี method ทำงาน
Intevitance (สืบทอด)
Implement
ConcreteCreatorB
...
createProduct() : Product
«Instantiate»
ConcreteProductB
client
object
subclass ที่สร้าง object
client
main{ Creater Creater
Creater.concreteCreaterA/B();
Creater.someOperation();
สั่งให้ class ลูกสร้าง object
สั่งให้ Factory (คลังผลิต)
สร้าง Product p = ... ที่ใช้งาน

Factory Method: Consequences

+ By avoiding to specify the class name of the concrete class & the details of its creation, the client code has become more flexible isolates code for construction & representation.
+ The client is only dependent on the interface.
- Construction of objects requires one additional class in some cases.



Factory Method: Examples

- Factory Method (refactoring.guru)
- Factory method pattern – Wikipedia

Abstract Factory

Abstract Factory: Problem

- How can an application be independent of how its objects are created?
- How can a class be independent of how the objects it requires are created?
- How can families of related or dependent objects be created?

Applicability

- When clients cannot anticipate groups of classes to instantiate

Abstract Factory: Solution

Intent

- Create families of related objects without specifying subclass names

The Abstract Factory design pattern describes how to solve such problems:

- Encapsulate object creation in a separate (factory) object. That is, define an interface (AbstractFactory) for creating objects, and implement the interface.
- A class delegates object creation to a factory object instead of creating objects directly.

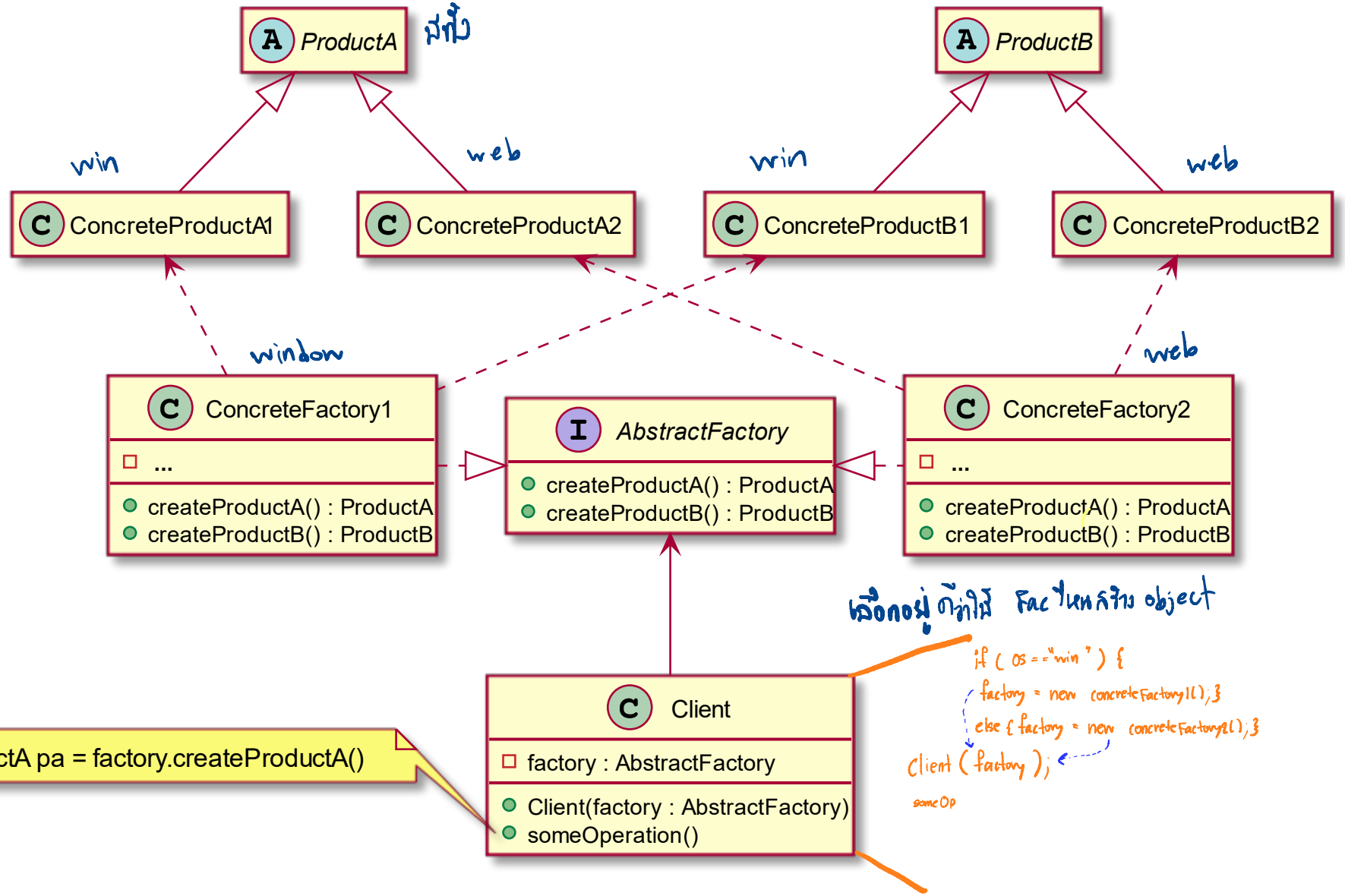
Abstract Factory

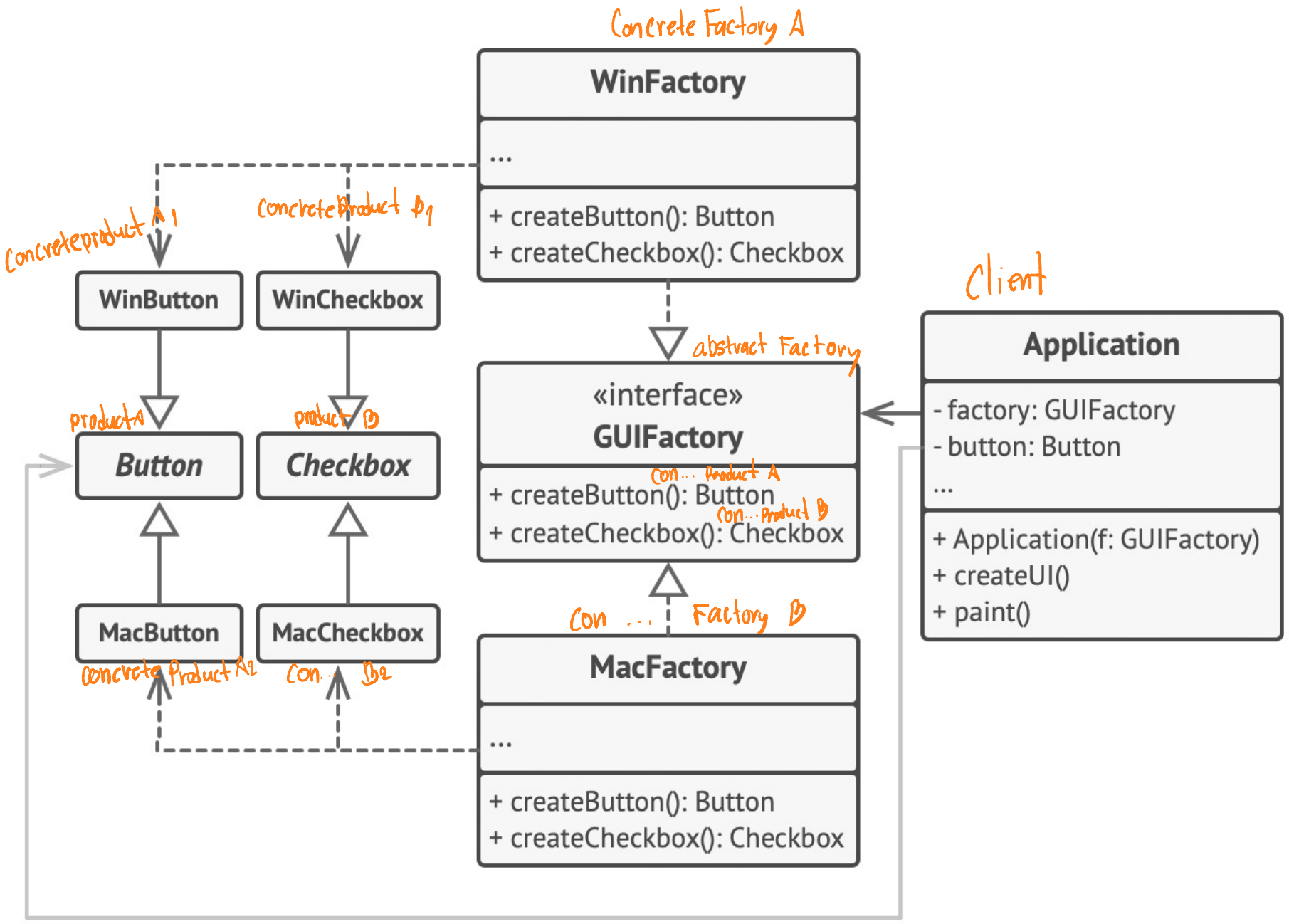
ข้อดี
1. ไม่ต้องอ้างอิง Concrete เลย
2. ตอนเปลี่ยนโค้ด มุ่งแค่ Abstraction (Interface)

ข้อเสีย
ถ้าเริ่ม สร้าง / เปลี่ยนแปลง แก้อีกหลายจุดเลย
เช่น แก้ใน Button เพิ่ม method แก้ที่ ① Factory, ② Product (Button),
แก้ที่ ③ ConcreteFactory Button

เหมาะสม: สถี type อื่นบ้าง

ภาพรวม





The cross-platform UI classes example.

Abstract Factory: Consequences

ยืดหยุด ปรับไม่ต้องอ้างอิง Concrete เลย

+ Flexibility: removes type (i.e., subclass) dependencies from clients
+ Abstraction & semantic checking: hides product's composition ตอนเปลี่ยนโค้ด มุ่งแต่ Abstraction (Interface)
– Hard to extend factory interface to create new products

สร้างใหม่ สร้าง / เปลี่ยนแปลง แก้อีกหลายจุดเลย

เหมาะสม: สร้าง type อะไรบ้าง

ข้อดี
1. ไม่ต้องอ้างอิง Concrete เลย
2. ตอนเปลี่ยนโค้ด มุ่งแต่ Abstraction (Interface)

ข้อเสีย สร้างใหม่ สร้าง / เปลี่ยนแปลง แก้อีกหลายจุดเลย
เช่น แก้ไข Button เพิ่ม method แก้ทั้ง Ⓘ Factory, Ⓘ Product (Button), แก้ทั้ง Ⓒ Concrete Factory Button

Abstract Factory: Examples

- Design Patterns: Abstract Factory in Java (refactoring.guru)
- Abstract Factory Design Pattern in Java – JournalDev

Builder

Constuctor (พารามิเตอร์ หลายตัว)

Builder: Problem

- How can a class (the same construction process) create different representations of a complex object? สองคำถามเหมือน มีออบชั่นหลาย กระบวนการสร้างมันซับซ้อน เหมือนหลัก อยาก reuse
- How can a class that includes creating a complex object be simplified?

Applicability สถานการณ์ที่เหมาะสม แยกกระบวนการสร้างออกจาก สปก

- Need to isolate knowledge of the creation of a complex object from its parts
- Need to allow different implementations/interfaces of an object's parts

Builder: Solution

Intent

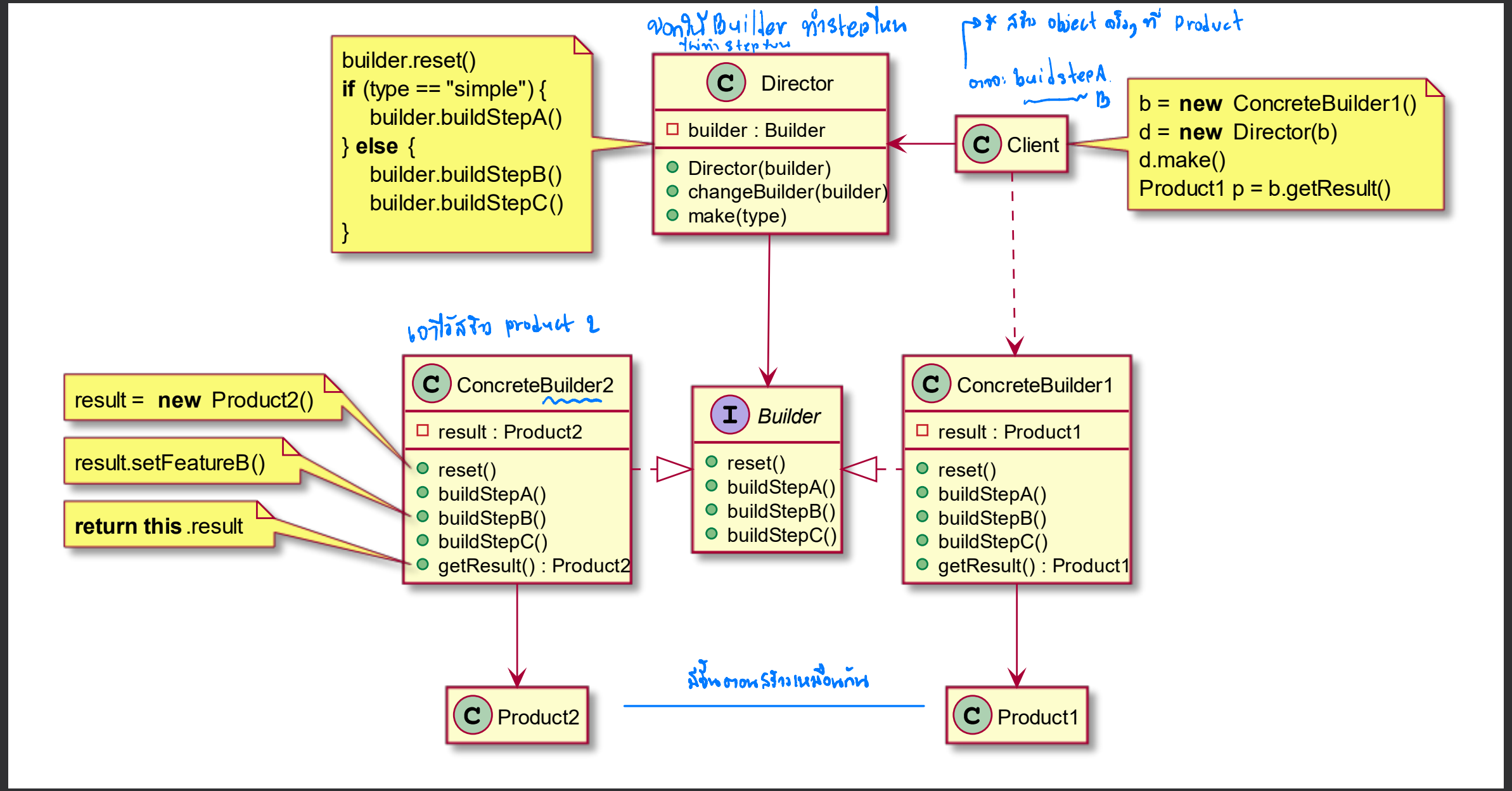
แยกการสร้าง

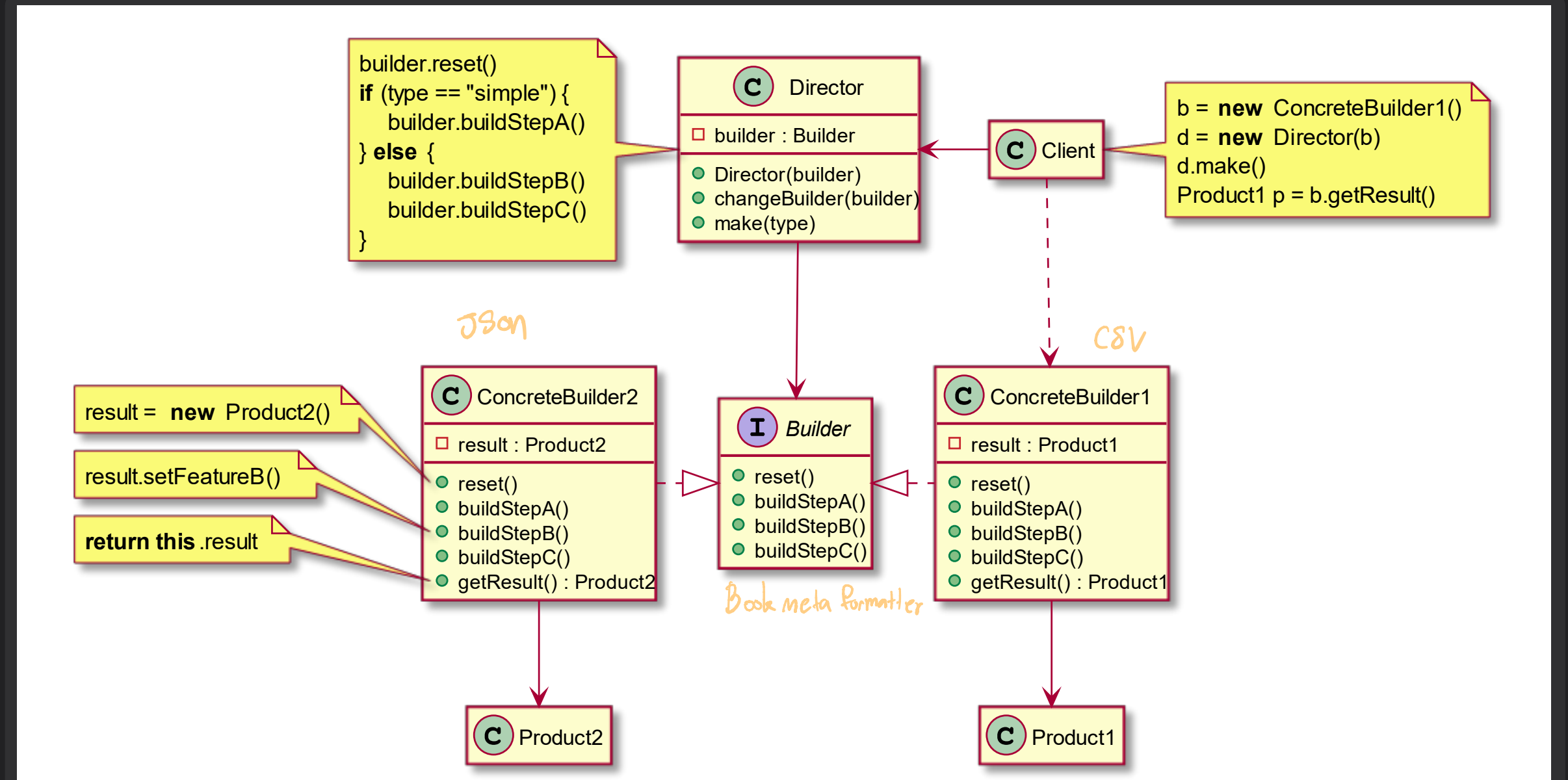
- Separate the construction of a complex object from its representation so that the same construction process can create different representations

The Builder design pattern describes how to solve such problems:

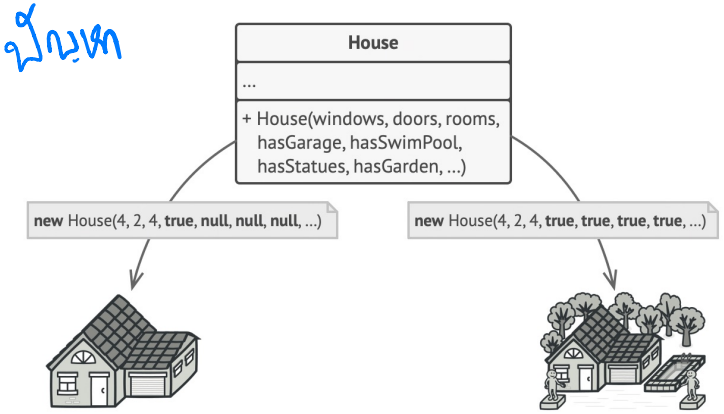
แยกส่วนประกอบ ทับ สิ่งที่สร้าง

- Encapsulate creating and assembling the parts of a complex object in a separate Builder object
- A class delegates object creation to a Builder object instead of creating the objects directly.


builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepC()
}
Director
builder : Builder
Director(builder)
changeBuilder(builder)
make(type)
Client
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
result = new Product2()
result.setFeatureB()
return this .result
ConcreteBuilder2
result : Product2
reset()
buildStepA()
buildStepB()
buildStepC()
getResult() : Product2
Builder
reset()
buildStepA()
buildStepB()
buildStepC()
ConcreteBuilder1
result : Product1
reset()
buildStepA()
buildStepB()
buildStepC()
getResult() : Product1
Product2
Product1

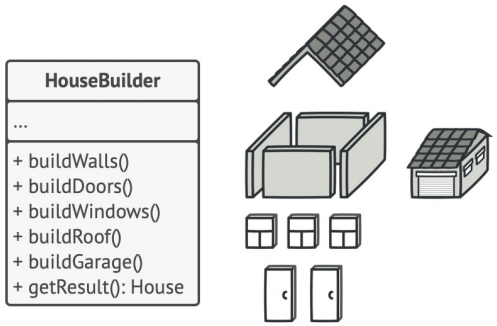

builder.reset()
if (type == "simple") {
builder.buildStepA()
} else {
builder.buildStepB()
builder.buildStepC()
}
Director
builder : Builder
Director(builder)
changeBuilder(builder)
make(type)
Client
b = new ConcreteBuilder1()
d = new Director(b)
d.make()
Product1 p = b.getResult()
JSon
CSV
result = new Product2()
result.setFeatureB()
return this .result
ConcreteBuilder2
result : Product2
reset()
buildStepA()
buildStepB()
buildStepC()
getResult() : Product2
Builder
reset()
buildStepA()
buildStepB()
buildStepC()
ConcreteBuilder1
result : Product1
reset()
buildStepA()
buildStepB()
buildStepC()
getResult() : Product1
Book meta formatter
Product2
Product1

บ้าน

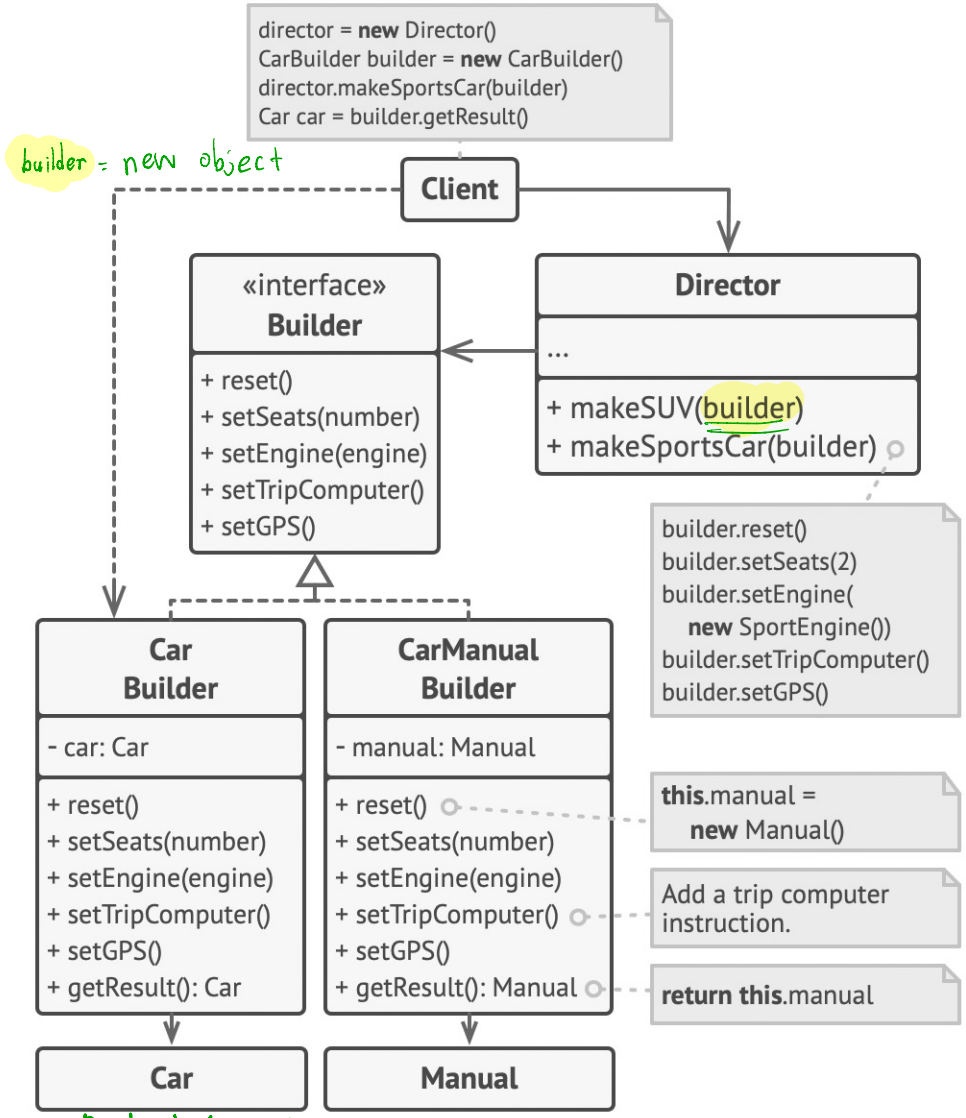


The constructor with lots of parameters has its downside: not all the parameters are needed at all times.

วิธีแก้



Ex. Car



- Product /concrete

The example of step-by-step construction of cars and the user guides that fit those car models.

- Abstraction /Interface ใช้

Builder: Consequences

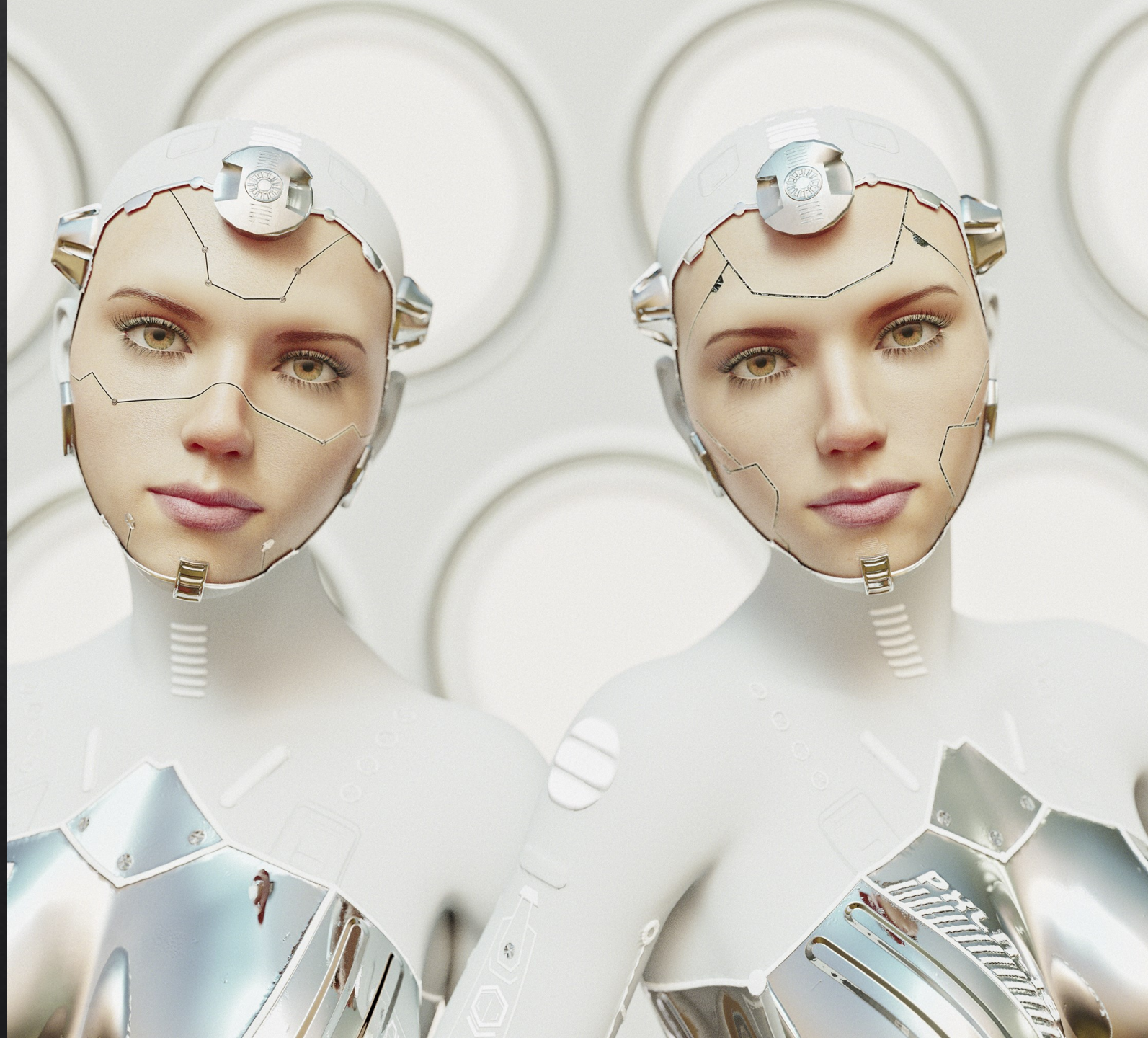
+ Can vary a product's internal representation ปรับเปลี่ยน ข้างในได้ง่ายขึ้น
+ Isolates code for construction & representation แยกกระบวนการสร้าง ออกจากส่วนที่ต้องการ
+ Finer control over the construction process การควบคุมที่ปรับเปลี่ยนได้ เช่น get product มาจาก Builder
– May involve a lot of classes เพิ่มเยอะมาก

Interface
Direc
Builder

Builder: Examples

- Design Patterns: Builder in Java (refactoring.guru)
- Builder Design Pattern in Java - JournalDev
- Implementing the builder pattern in Java 8 - Tutorial (vogella.com)

Prototype



Prototype: Problem

สร้าง object ยังไง : เราจะสร้าง object อะไรก็ไม่รู้จนกว่าจะ run time

- How can objects be created so that which objects to create can be specified at run-time?
- How can dynamically loaded classes be instantiated?

Applicability

ใช้กับกรณีที่ไม่ต้องการสร้าง

- When a system should be independent of how its products are created, composed, & represented
- When the classes to instantiate are specified at run-time

ไม่รู้ว่าจะสร้าง object ว่า class อะไร รู้ตอน run

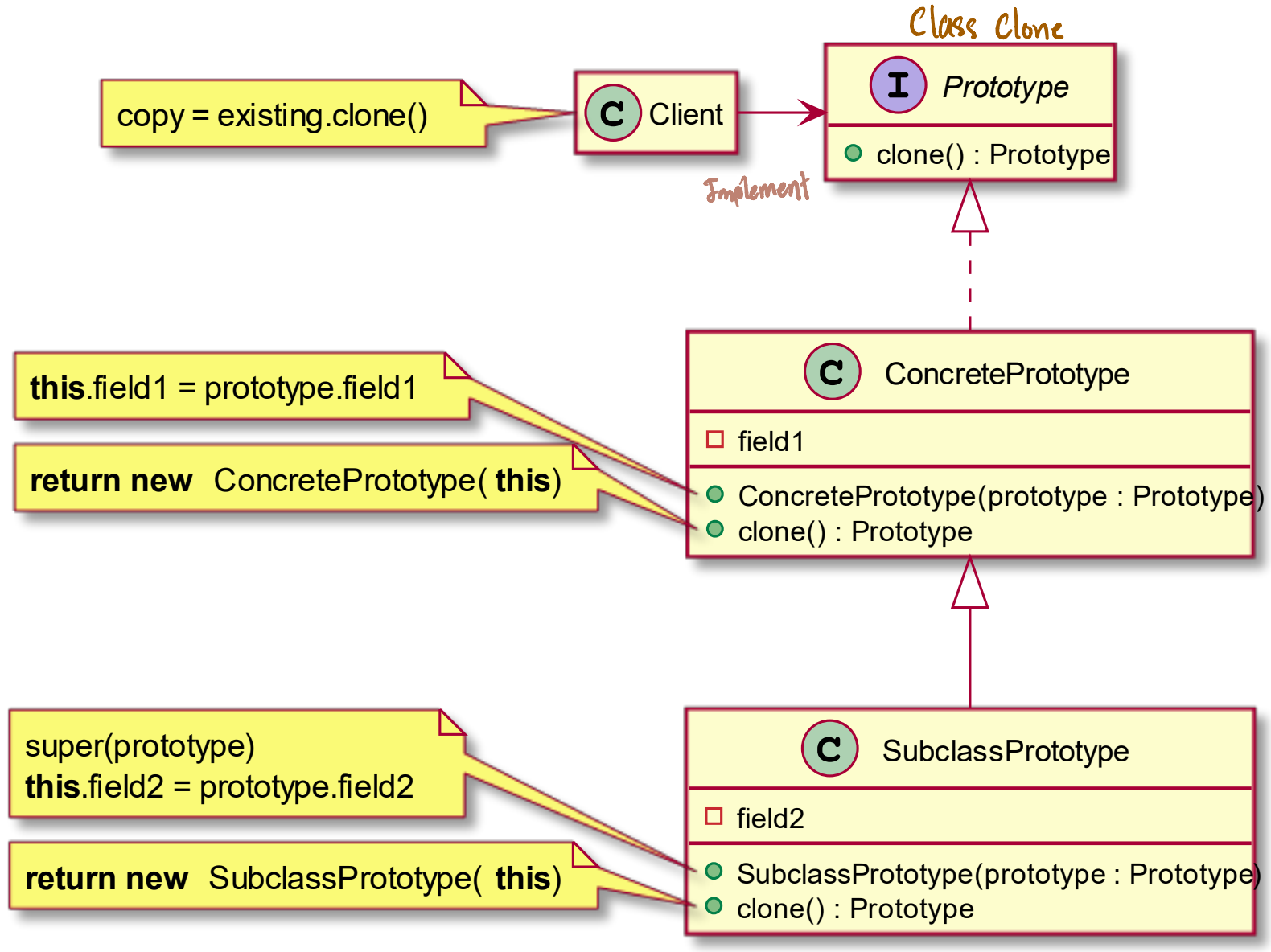
Prototype: Solution

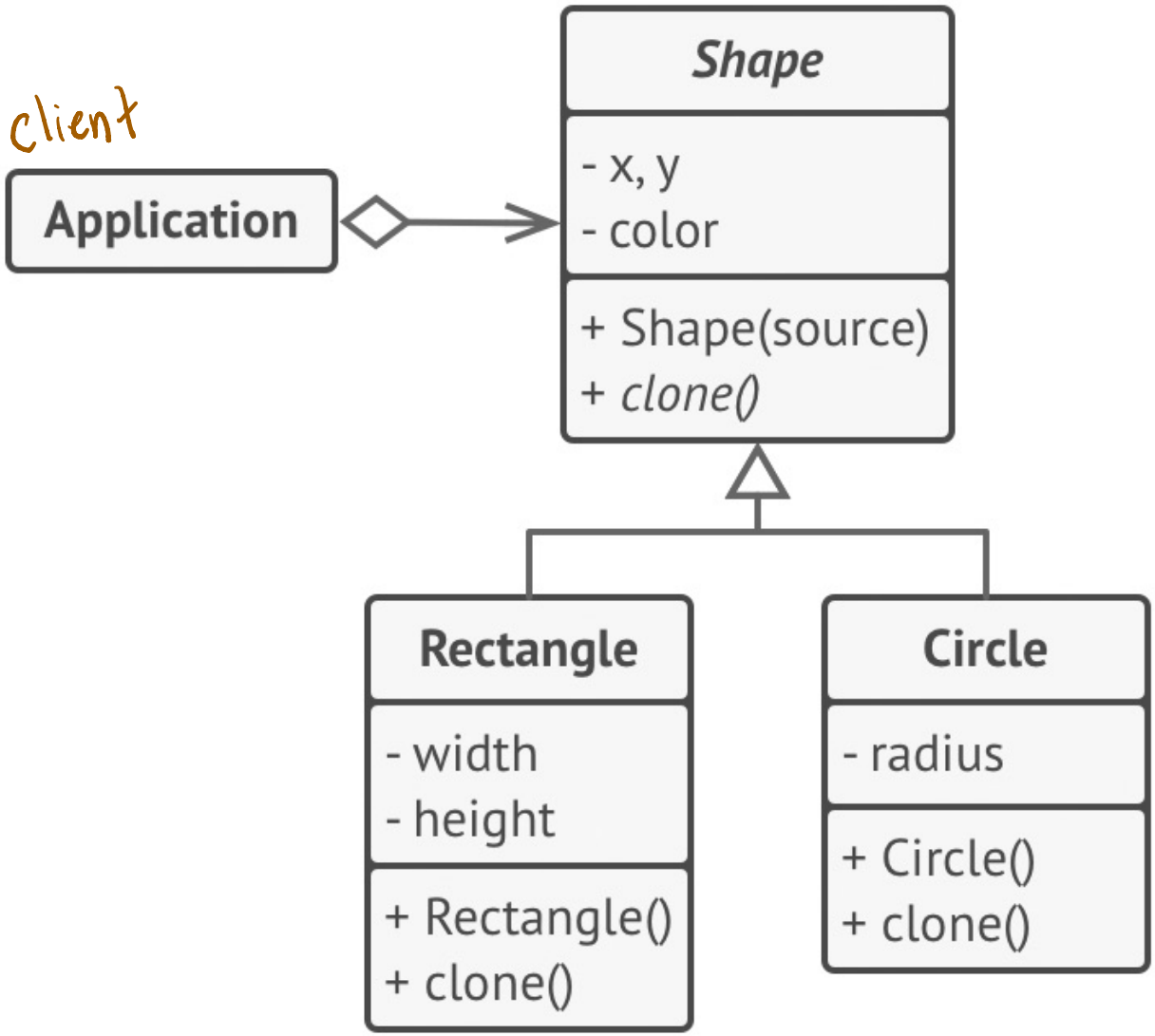
Intent

- Specify the kinds of objects to create using a prototypical instance & create new objects by copying this prototype

The prototype design pattern describes how to solve such problems:

- Define a ***Prototype*** object that returns a copy of itself.
- Create new objects by copying a ***Prototype*** object.


Class Clone
copy = existing.clone()
Client
Prototype
clone() : Prototype
Implement
this.field1 = prototype.field1
return new ConcretePrototype(this)
ConcretePrototype
field1
ConcretePrototype(prototype : Prototype)
clone() : Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)
SubclassPrototype
field2
SubclassPrototype(prototype : Prototype)
clone() : Prototype



Cloning a set of objects that belong to a class hierarchy.

Prototype: Consequences

+ Can add & remove classes at runtime by cloning them as needed
+ Reduced subclassing minimizes/eliminates need for lexical dependencies at run-time
– Every class that used as a prototype must itself be instantiated
– Classes that have circular references to other classes cannot really be cloned

Prototype: Examples

- Design Patterns: Prototype in Java (refactoring.guru)
- Prototype Design Pattern in Java - JournalDev

Singleton

Singleton: Problem

How to:

- Ensure that a class only has one instance
- Easily access the sole instance of a class
- Control its instantiation
- Restrict the number of instances

Applicability

- When there must be exactly one instance of a class & it must be accessible from a well-known access point
- When the sole instance should be extensible by subclassing & clients should be able to use an extended instance without modifying their code

Singleton: Solution

Intent

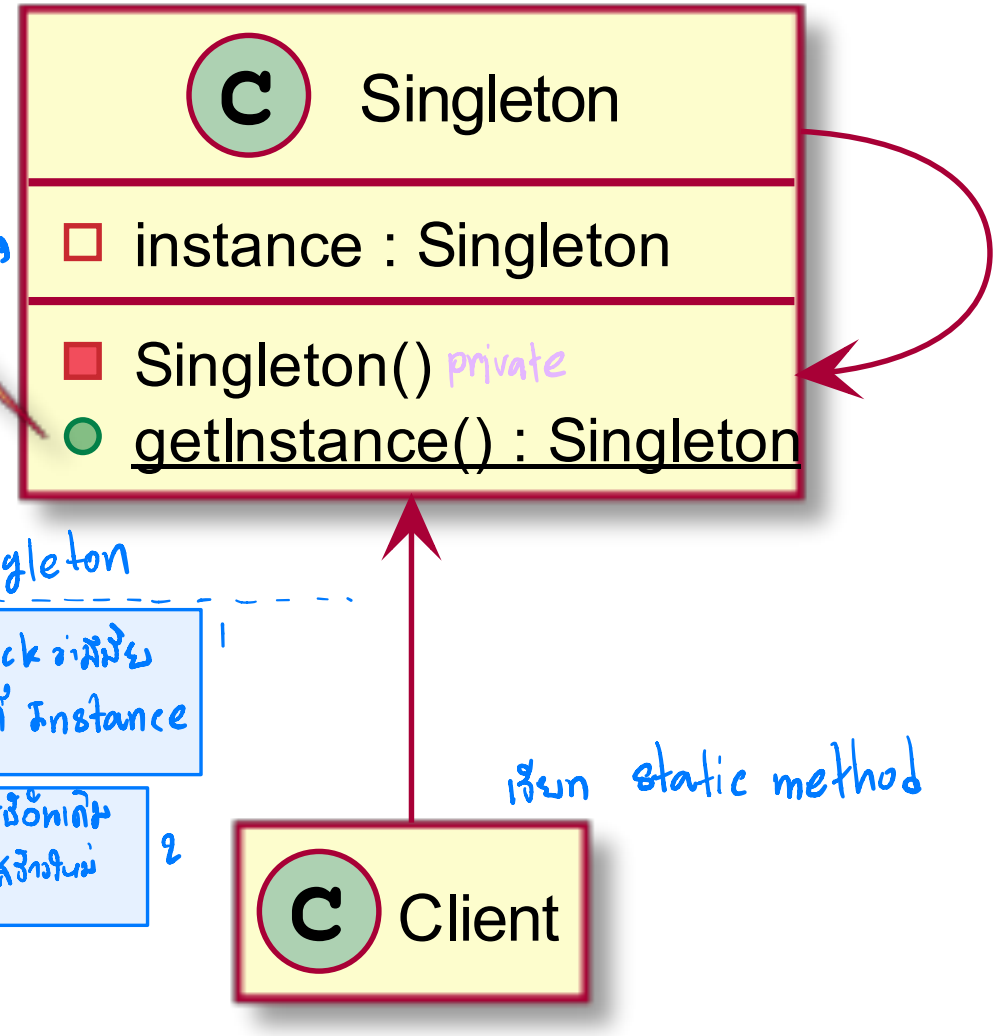
- Ensure a class only ever has one instance & provide a global point of access

The singleton design pattern describes how to solve such problems:

- Hide the constructors of the class.
- Define a public static operation (getInstance()) that returns the sole instance of the class.

Singleton อยากให้มี Instance เดียว, ไม่อยากให้ concrete ควบคุมมัน

```
if (instance ==  null){
    instance =  new Singleton()
}
return instance
```



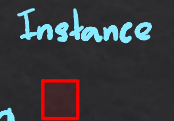
* Concept คือ - Constructor
  + getInstance() เป็น static method

ข้อดี ไม่ต้องมี part package เยอะ: Instance
อยากให้มีมากกว่า 1 เปลี่ยน
can sub class ได้ ไม่ต้องแก้ไฟ base class

ข้อเสีย 1. overhead
2. เหมือน global
3. ใช้แทน global อาจมีประสิทธิภาพมากกว่า การใช้พื้นที่มากกว่า

Singleton: Consequences

+ Reduces namespace pollution ไม่ต้องมี part package เยอะ
+ Makes it easy to change your mind & allow more than one instance อยากให้มีมากกว่า 1 เปลี่ยนง่าย
+ Allow extension by subclassing can sub class ได้ ไม่ต้องแก้ไฟล์ base class
– Same drawbacks of a global if misused เหมือน global
– Implementation may be less efficient than a global ไม่ประหยัดกว่า heap
– Concurrency/cache pitfalls & communication overhead



Singleton: Examples

- Design Patterns: Singleton in Java (refactoring.guru)
- Java Singleton Design Pattern Example Best Practices – JournalDev

Dependency Injection

Dependency Injection: Problem

- How can a class be independent of how the objects on which it depends are created?
- How can the way objects are created be specified in separate configuration files?
- How can an application support different configurations?

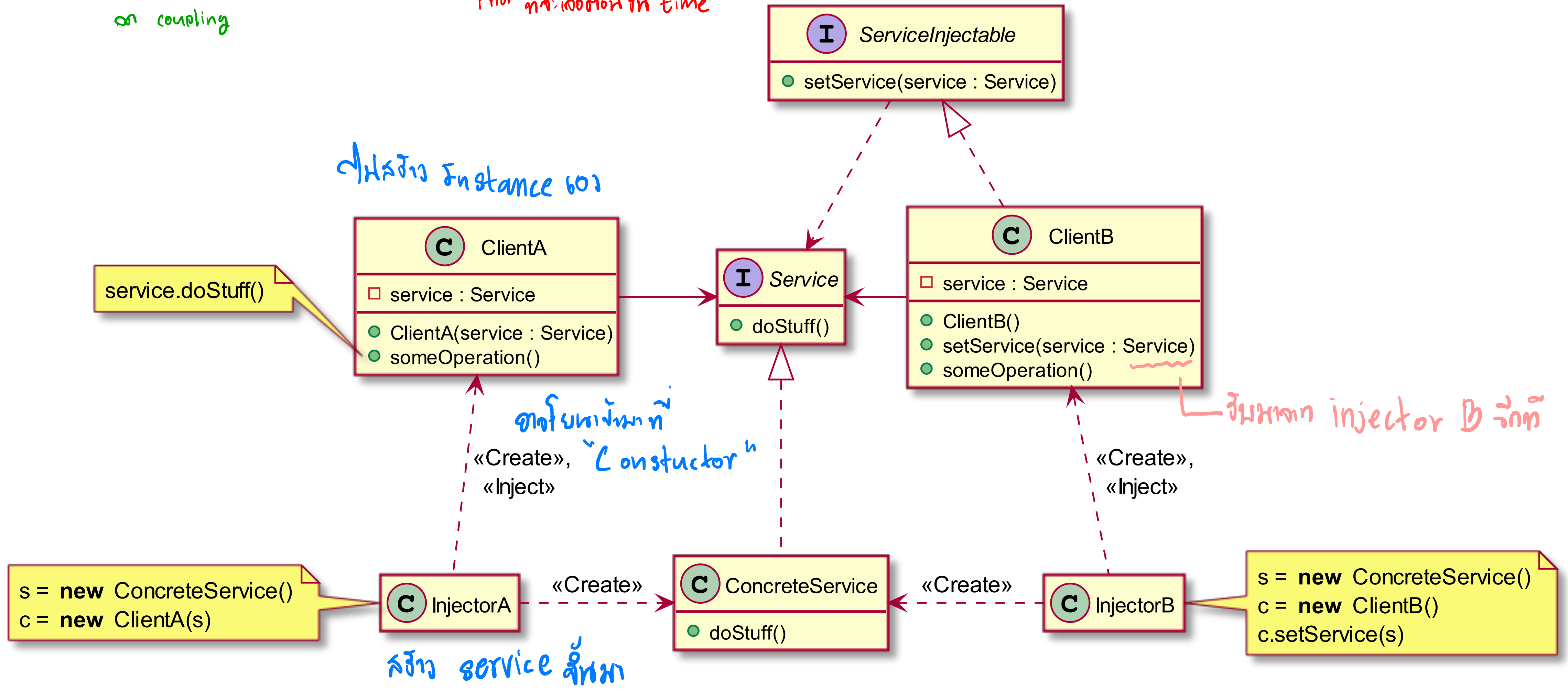
Dependency Injection: Solution

- Dependency injection separates the creation of a client's dependencies from the client's behavior, which promotes loosely coupled programs and the dependency inversion and single responsibility principles.
- Fundamentally, dependency injection is based on passing parameters to a method.
- Dependency injection is an example of the more general concept of inversion of control.
- For details: Inversion of Control Containers and the Dependency Injection pattern (martinfowler.com)

Dependency Injector   ลดการเกิด coupling

ข้อดี
- ช่วย unit testing
- กระจายการงานที่ injector ตัวเดียว
- ลด coupling

ข้อเสีย
- ลด coupling
- มันซับซ้อน , ถ้ามีคลาสใช้มากเกินไป จะมีปัญหาอื่นๆตามมา
- Error ที่จะเจอตอน Run time



Dependency Injection: Consequences

- \+ Helps in Unit testing.
- \+ Boiler plate code is reduced, as initializing of dependencies is done by the injector component.
- \+ Helps to enable loose coupling, which is important in application programming.
- – It's a bit complex to learn, and if overused can lead to management issues and other problems.
- – Many compile time errors are pushed to run-time.

Dependency Injection: Examples

- Using dependency injection in Java - Introduction - Tutorial (vogella.com)
- Java Dependency Injection - DI Design Pattern Example Tutorial – JournalDev
- A quick intro to Dependency Injection: what it is, and when to use it (freecodecamp.org)