Structural Design Pattern

Parinya Ekparinya

Parinya.Ek@kmitl.ac.th

Software Architecture and Design

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/Facade_pattern
- https://en.wikipedia.org/wiki/Composite_pattern
- https://en.wikipedia.org/wiki/Decorator_pattern
- https://en.wikipedia.org/wiki/Proxy_pattern
- https://en.wikipedia.org/wiki/Flyweight_pattern

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."

Structural Design Patterns

ควบคุม ซึ่งรูปภาพหนึ่ง

- Façade ส่วนหน้า

ตัวกลาง

- Proxy

ถูกผสมรวม

- Composite

น้ำหนักเบา (ลดความหนัก)

- Flyweight

ตกแต่ง

- Decorator

ส่วนหน้า ตึกหน้า

Facade

Facade: Problem

- How to make a complex subsystem easier to use? ทำให้ระบบซับซ้อนถูกเรียกใช้ง่าย
- How can the dependencies on a subsystem be minimized? ทำให้ subsystem ลดลง (อ้างอิงน้อยที่สุด)

Applicability

ระบบซับซ้อน

- A simple interface is required to access a complex system.
- A system is very complex or difficult to understand.
- An entry point is needed to each level of layered software. สร้างจุดเข้าถึง
- The abstractions and implementations of a subsystem are tightly coupled.

Facade: Solution

Intent

- Provide a unified interface to a set of interfaces in a subsystem.

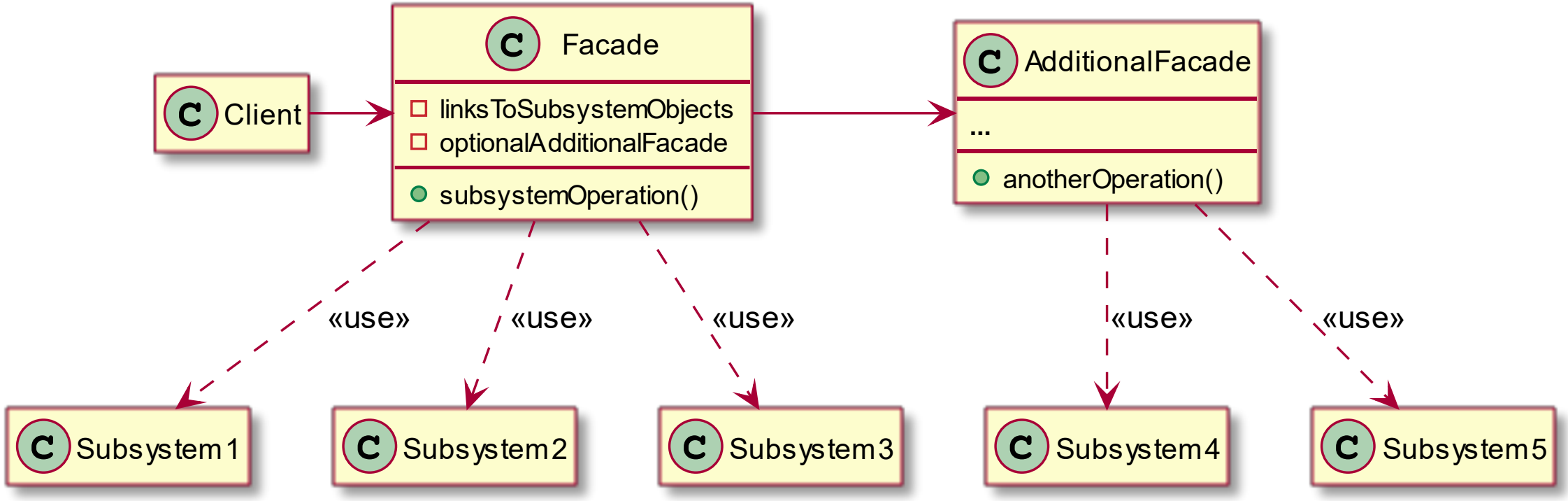
What solution does the Facade design pattern describe?

- Define a façade object that implements a simple interface in terms of (by delegating to) the interfaces in the subsystem.
- Such a façade object may perform additional functionality before/after forwarding a request.

Facade แพทเทิร์นที่ใช้งานง่าย โดยจะทำการแยก Subsystem เป็นส่วนๆ เพื่อมาประกอบกันทีเดียว

ข้อดี แยกความซับซ้อนออกจากกัน

ข้อเสีย god Facad มันจะรู้ทุกอย่าง



Facade: Consequences

+ Client code can be isolated from the complexity of a subsystem.
- A facade can become a god object!!

Facade: Examples

- Facade (refactoring.guru)
- Facade pattern - Wikipedia

Composite

Also known as:
Object Tree

Composite: Problem

- How to represent a part-whole hierarchy so that clients can treat part and whole objects uniformly?
- How to represent a part-whole hierarchy with tree structure?

Applicability

- Objects must be composed recursively.
- No distinction between individual & composed elements.
- Objects in structure can be treated uniformly.

Composite: Solution

Intent

- To treat individual objects & multiple, recursively-composed objects uniformly.

What solution does the Composite design pattern describe?

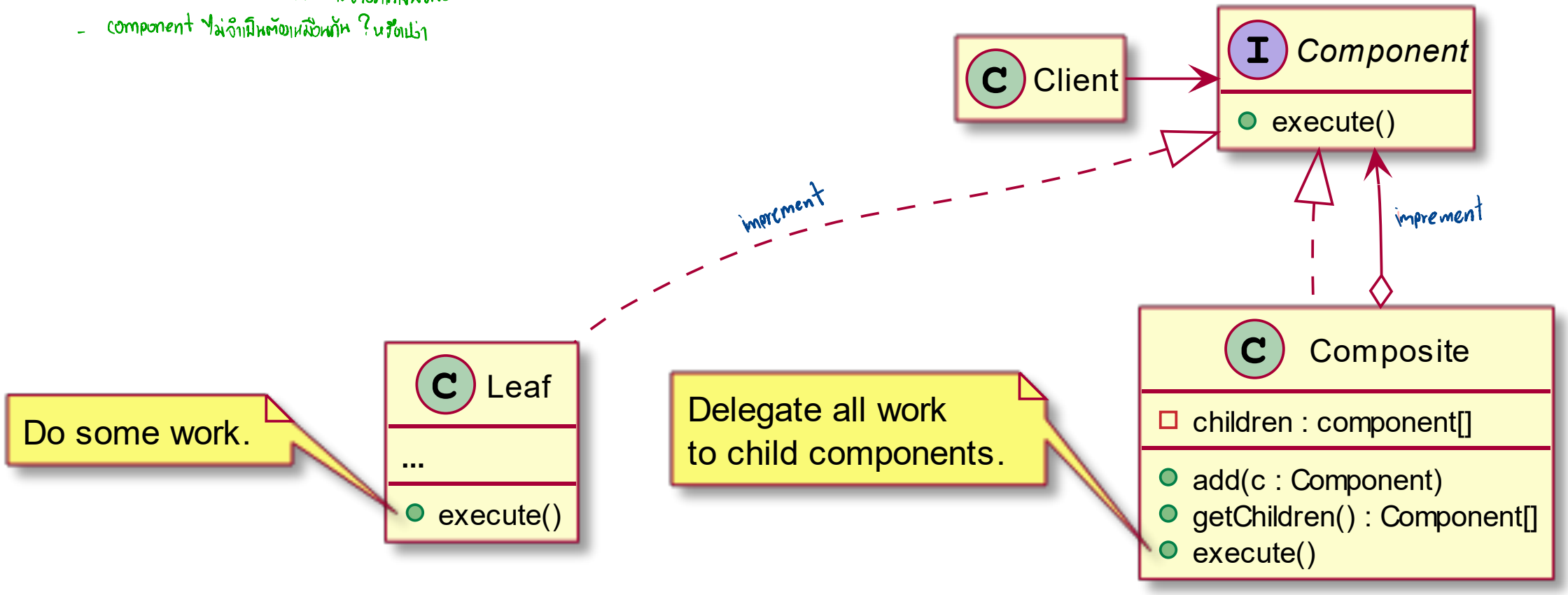
- Define a unified Component interface for both part (Leaf) objects and whole (Composite) objects.
- Individual Leaf objects implement the Component interface directly, and Composite objects forward requests to their child components.

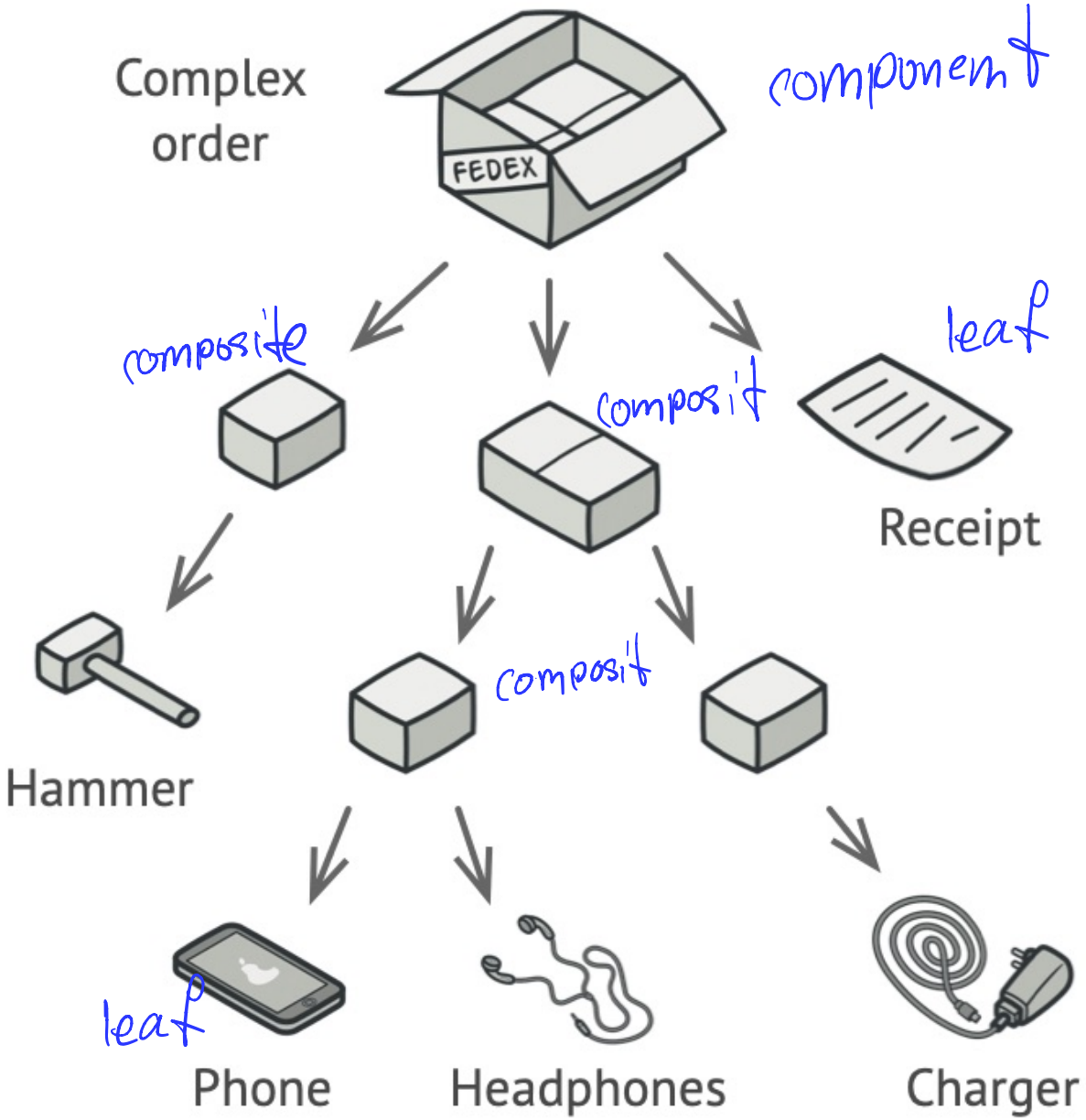
Composit

เหมาะสมกับ model แอป ที่สร้าง แผนผัง / แอป ที่ต้องใช้ องค์ประกอบ ย่อยๆมาประกอบกัน

ข้อเสีย - overhead
- พยายามฝืนทำ leaf เป็น composite

ข้อดี - เพิ่ม component subclass ไม่ต้องแก้ไขส่วนอื่น
- component ไม่จำเป็นต้องเหมือนกัน ?หรือเปล่า

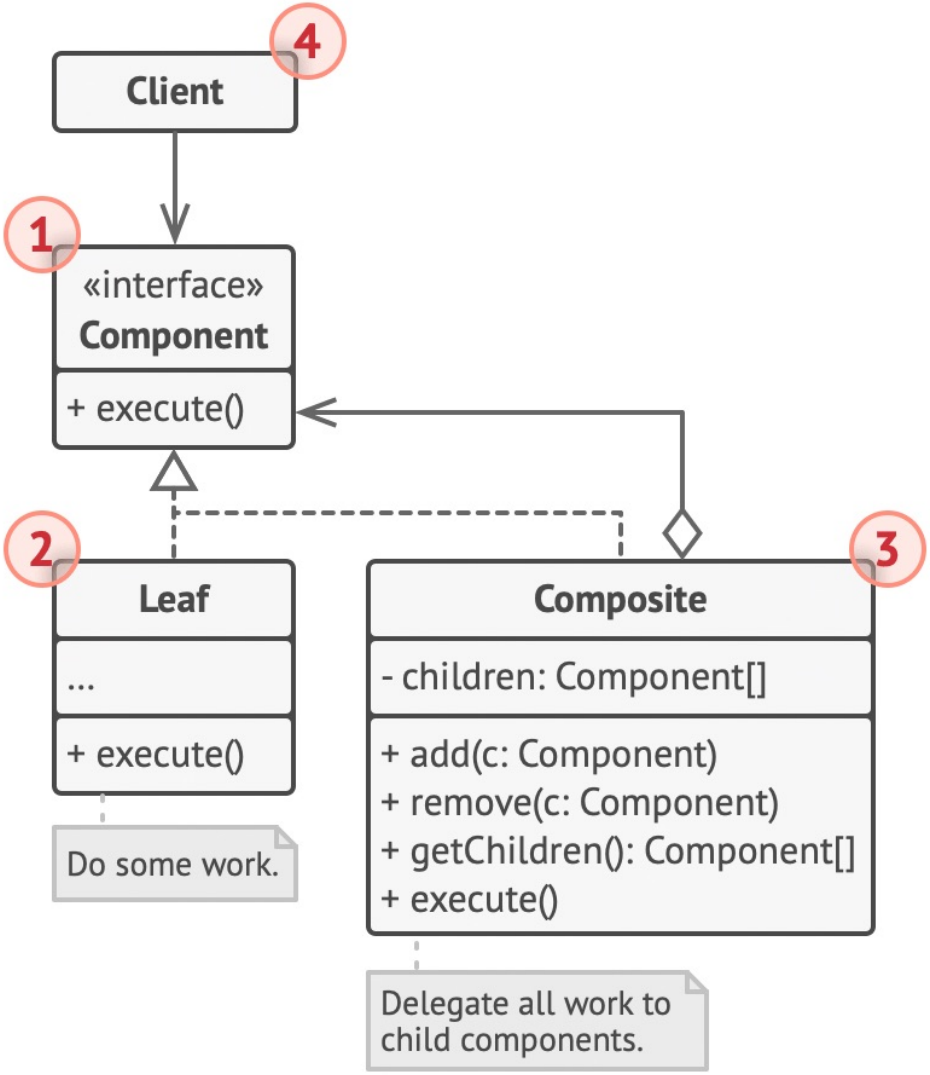



Complex order
FEDEX
component
composite
leaf
composit
Receipt
composit
Hammer
leaf
Phone
Headphones
Charger

Structure

Composite

code มีหลายอันเหมือนกัน ทำหลายอันก็ได้ method เหมือนกัน

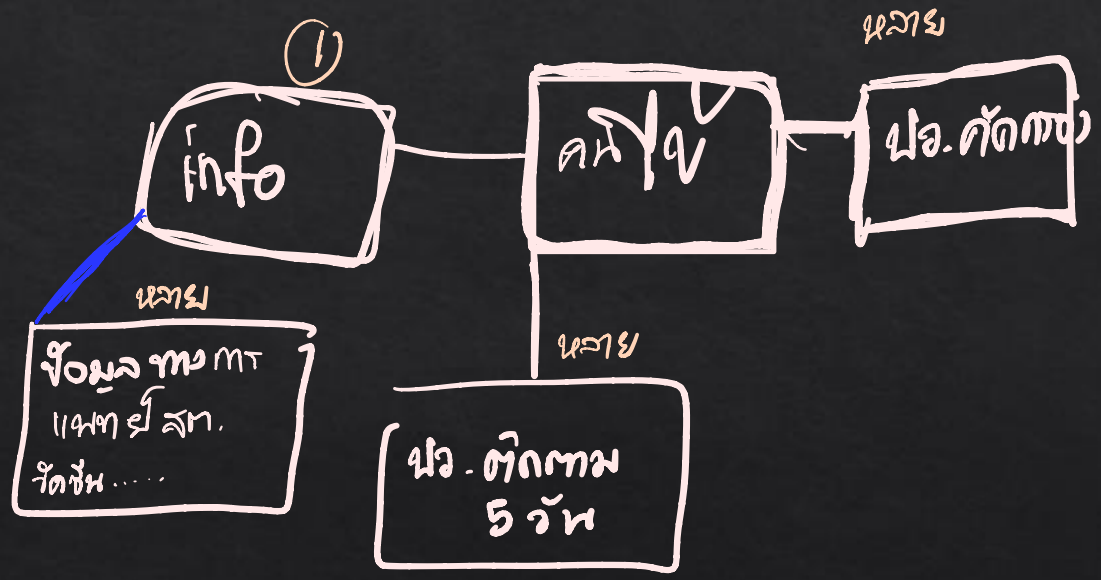


Composite: Consequences

+ Uniformity: treat components the same regardless of complexity.
+ Extensibility: new Component subclasses work wherever old ones do.
– Overhead: might need prohibitive numbers of objects.
– Awkward designs: may need to treat leaves as lobotomized composites in some cases

Composite: Examples

- Composite (refactoring.guru)
- Composite pattern - Wikipedia



ดูเล่นๆ

Decorator

Also known as:
Wrapper

หรือเรียก

Decorator: Problem

- How to add responsibilities to (and removed from) an object dynamically at run-time?
- How to provide a flexible alternative to subclassing for extending functionality?

Applicability

- When responsibilities are needed to add to individual objects dynamically and transparently.
- When such responsibilities can be withdrawn from objects.
- When extension by subclassing is impractical.

Decorator: Solution

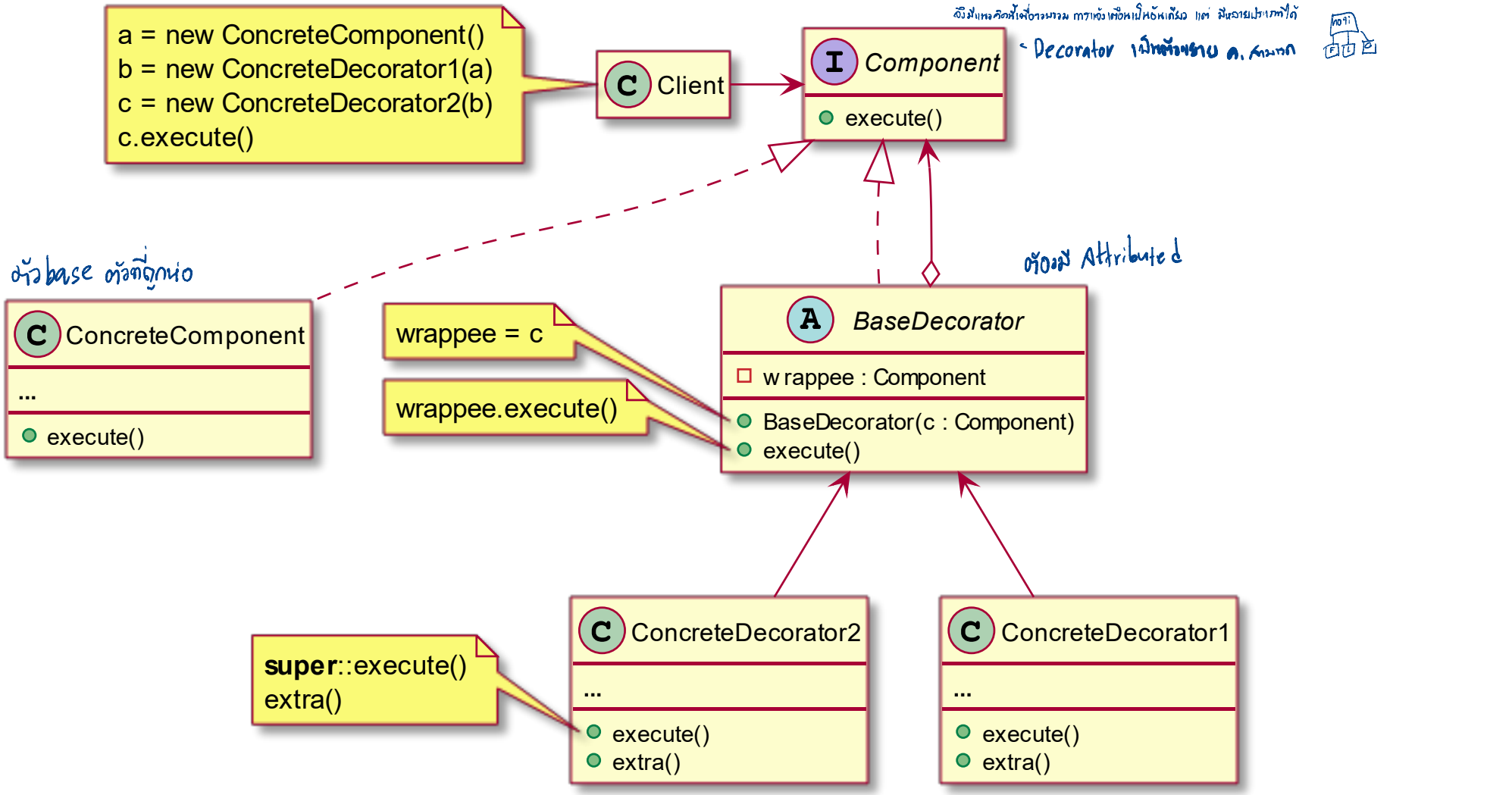
Intent

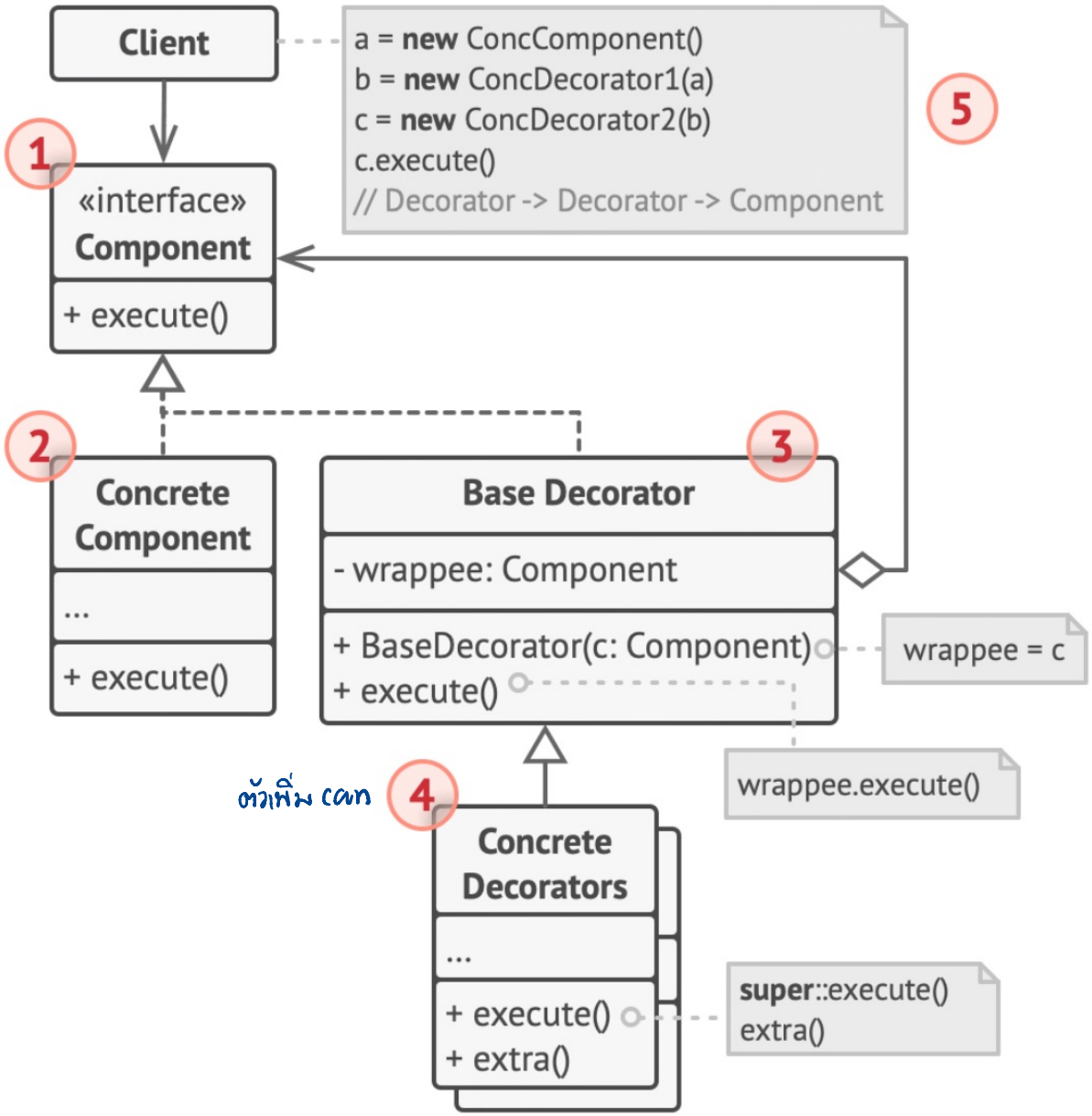
- To attach additional responsibilities to an object dynamically.

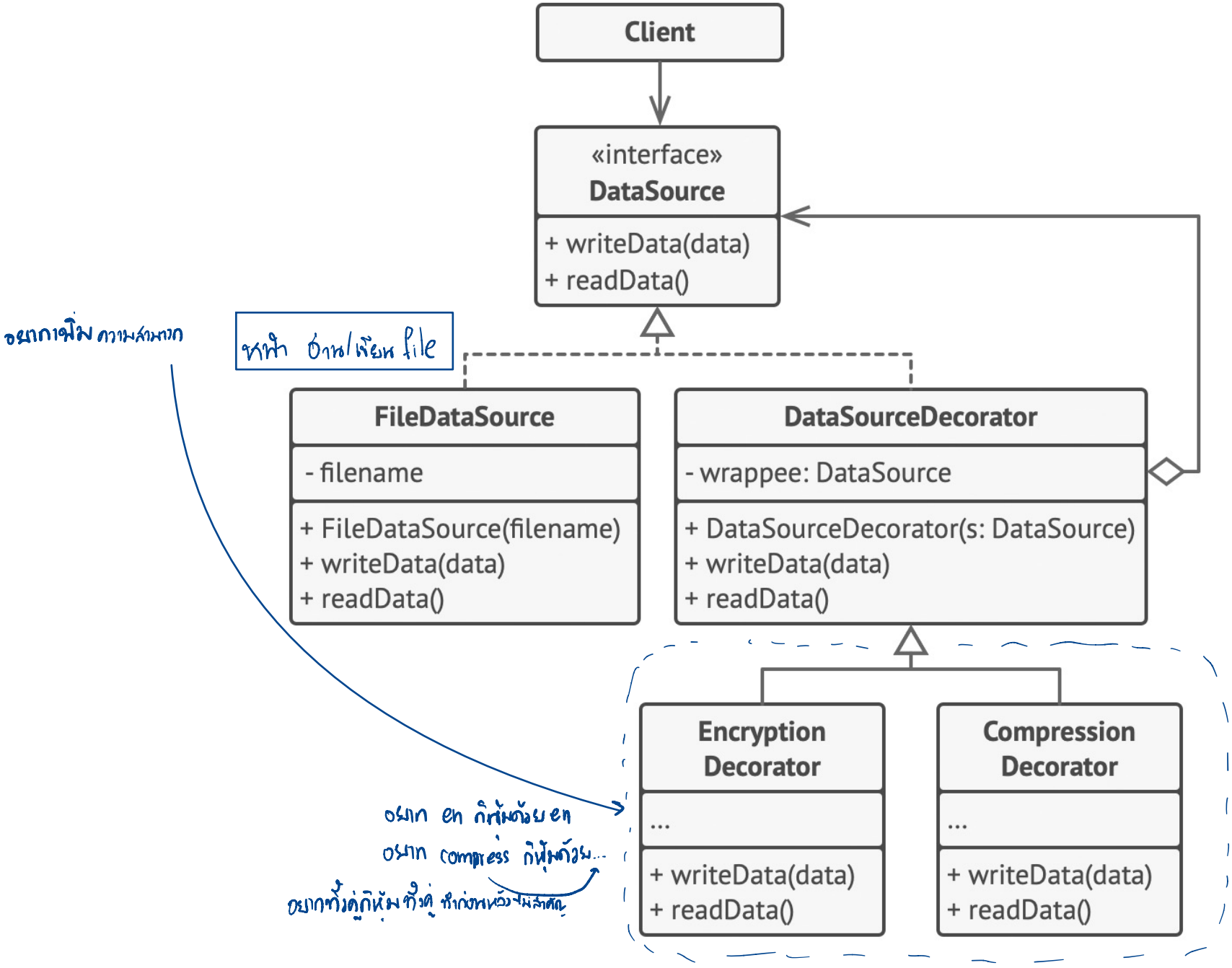
What solution does the Decorator design pattern describe?

- Define decorator objects that implement the interface of the extended (decorated) object (Component) transparently by forwarding all requests to it.
- Such decorator objects may perform additional functionality before/after forwarding a request.

Decorator โดย




Client
a = new ConcComponent()
b = new ConcDecorator1(a)
c = new ConcDecorator2(b)
c.execute()
// Decorator -> Decorator -> Component
5
1
«interface»
Component
+ execute()
2
Concrete
Component
...
+ execute()
3
Base Decorator
- wrappee: Component
+ BaseDecorator(c: Component)
+ execute()
wrappee = c
wrappee.execute()
ตัวเพิ่ม can
4
Concrete
Decorators
...
+ execute()
+ extra()
super::execute()
extra()



The encryption and compression decorators example.

Decorator: Consequences

เพิ่ม ลด function ที่ obj ตอน runtime

+ Add or remove responsibilities from an object at runtime.

Single Responsibility ส่งเสริม SRP เพราะ แยก class เล็ก Decorator ทำงานแยก

+ SRP: many variants of behavior can be implemented with several smaller classes.

– Hard to remove a specific wrapper from the wrappers stack. ถอดยากเพราะ ต้องถอดออก แค่ข้างนอกก่อน

– Certain variants of behavior may depend on the order of the decorators stack.

มากสุ่ม ทางเพิ่มความสามารถ ต้อง คำนึง ลำดับก่อนหลัง ในทางบ่อ

Decorator: Examples

- Decorator (refactoring.guru)
- Decorator pattern – Wikipedia

Proxy

Proxy: Problem

ควบคุมการเข้าถึง code นั่นๆ

- How to control the access to an object?.
- How to provide additional functionality when accessing an object?

เพิ่มความ can ก้ายDecorator

Applicability

- There are dozen ways to use this pattern.
  See https://refactoring.guru/design-patterns/proxy

Proxy: Solution

Intent

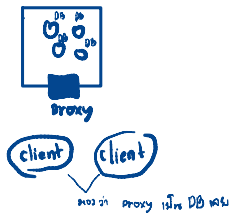
- Provide a surrogate or placeholder for another object to control access to it.

What solution does the Proxy design pattern describe?

- Define a separate proxy object that can be used as substitute for another object (Subject).
- Such a proxy object implements additional functionality to control the access to this subject.

Proxy ตัวมันเอง เข้ามาใช้ (ผ่าน) สิ่งที่ต้องการนั่นเอง

มี base เหมือน Decorator ใช่ไหม



ประพฤติตัวเหมือน obj ที่กันไว้

```
realService = s
```

```
if ( checkAccess() ) {
    realService.operation()
}
```

Proxy: Consequences

ถูกควบคุมได้โดยที่ client ไม่รู้เลย

+ Service object can be controlled without clients' awareness.

ผล proxy type ที่ไม่ต้องเปลี่ยน service or client

+ OCP: introduction new proxies without change service or clients.

– Overhead: the response from the service might get delayed.

+ อาจจะที่ต้องโหลด เช่น map เกม, proxy เอาไว้โหลดแผนที่ตรงนั้นไปก่อน นอกเหนือนั้นจะทิ้ง

Proxy: Examples

- Proxy (refactoring.guru)
- Proxy pattern – Wikipedia

ลดปริมาณ หน้าที่ ตอบเรื่อง
ช่วย performance ลดการใช้งาน mem

Flyweight

+ใช้พ่วงกับ เกม หลายที่ๆมีsprite เยอะๆ

Flyweight: Problem

- How to minimizes memory usage by sharing some of object data with other similar objects?

Applicability

- An application needs to spawn a huge number of similar objects.
- The objects contain duplicate states which can be extracted and shared between multiple objects.

Flyweight: Solution

Intent

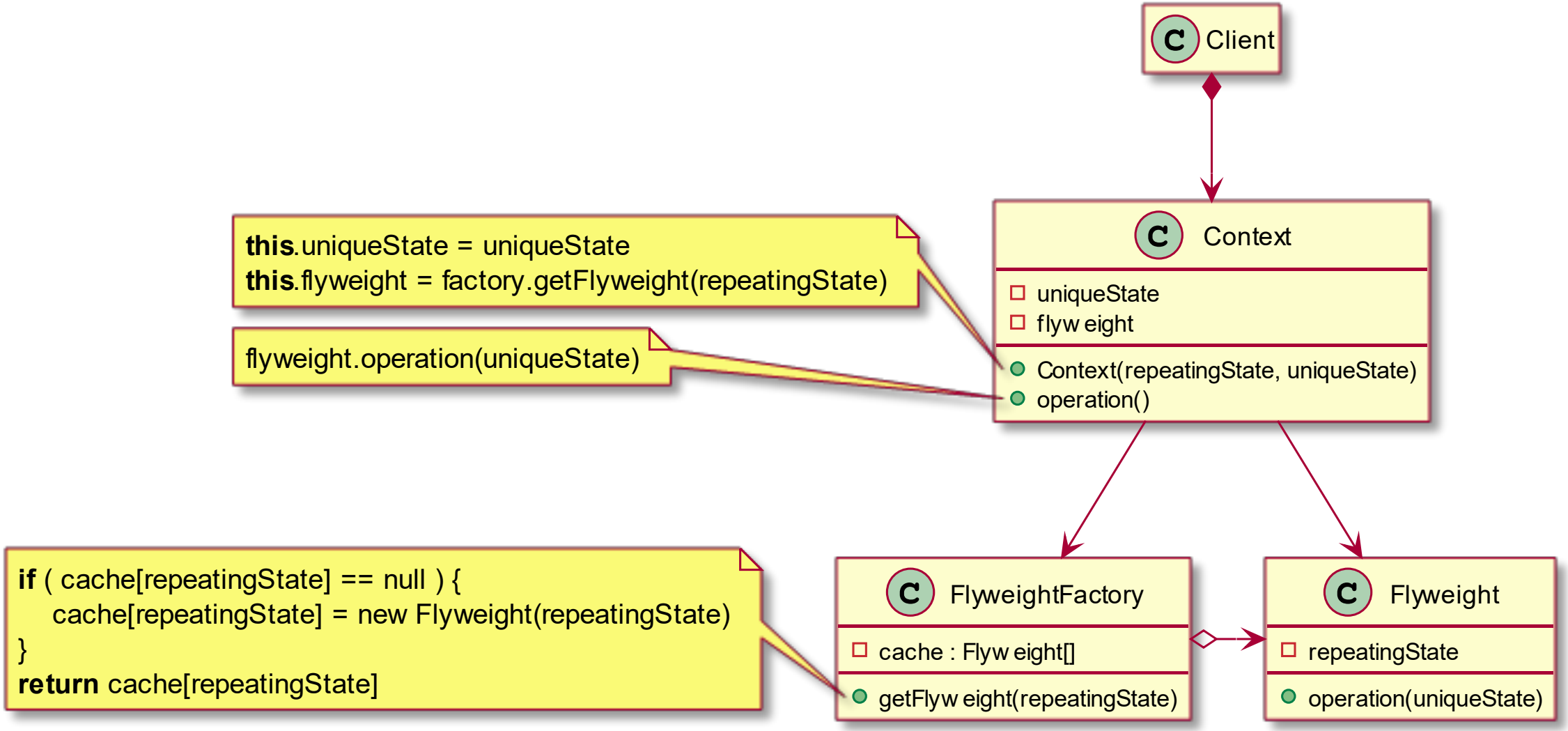
- To use sharing to support large numbers of fine-grained objects efficiently.

What solution does the Flyweight design pattern describes?

- Define a flyweight object to store shared *intrinsic state*, constant data of objects.
- *Extrinsic state*, the rest of the object's state that can be altered, should be stored in another object with a reference to a flyweight object.

Flyweight

ตอบโจทย์ เรื่อง performance , ลดการใช้งาน mem แทน share data ส่วนที่เหมือน Ex. ต้นไม้ในเกม
สังเกต มีลักษณะเหมือนกัน แต่จุดไม่เหมือนกัน object แยก ที่ไม่เหมือนได้



Flyweight: Consequences

+ Save lots of RAM!!
- Trading RAM over CPU cycles in some cases.
- The code can become much more complicated.

Flyweight: Examples

- Flyweight (refactoring.guru)
- Flyweight pattern – Wikipedia