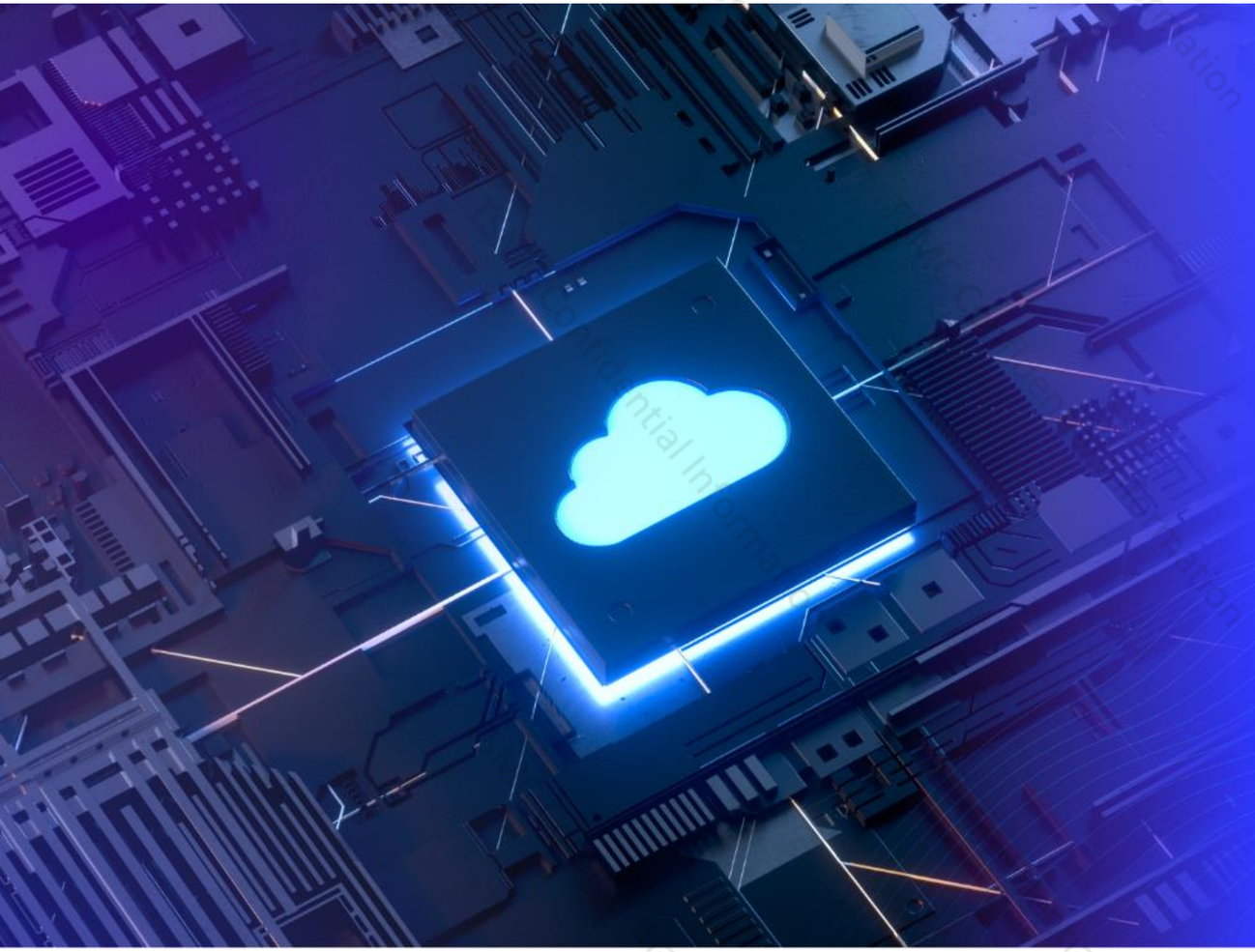
台灣積體電路製造股份有限公司
Taiwan Semiconductor Manufacturing Company, Ltd.



TSMC IT × NCTU CS 課號 5270

CLOUD NATIVE
Development Best Practice

DESIGN PRINCIPLE

TSID
LESD | 羅子威
April 20, 2022

About 羅子威

- 基隆人, 現定居 竹北
- 成功國小, 銘傳國中, 建國中學, 台大資管系, 台大資管所

2007-now @台積電 TSID ~ 15 years

- 2007-2019 TSID/F8MITD (Enterprise Mask Management System, Manufactory Process Change System, Research Department Mask Project)
- 2019-Now TSID/IESD, LESD (Litho Advanced Process Control)

興趣:

- 樂團不起眼貝斯手、股市迷途小書僮、周遊列國好奇仔

Design Principle

- Problem Solving Loop
- The 12 Factors to Cloud Success
- Design Patterns
  - Practice at Class
- 期末專題實作講解

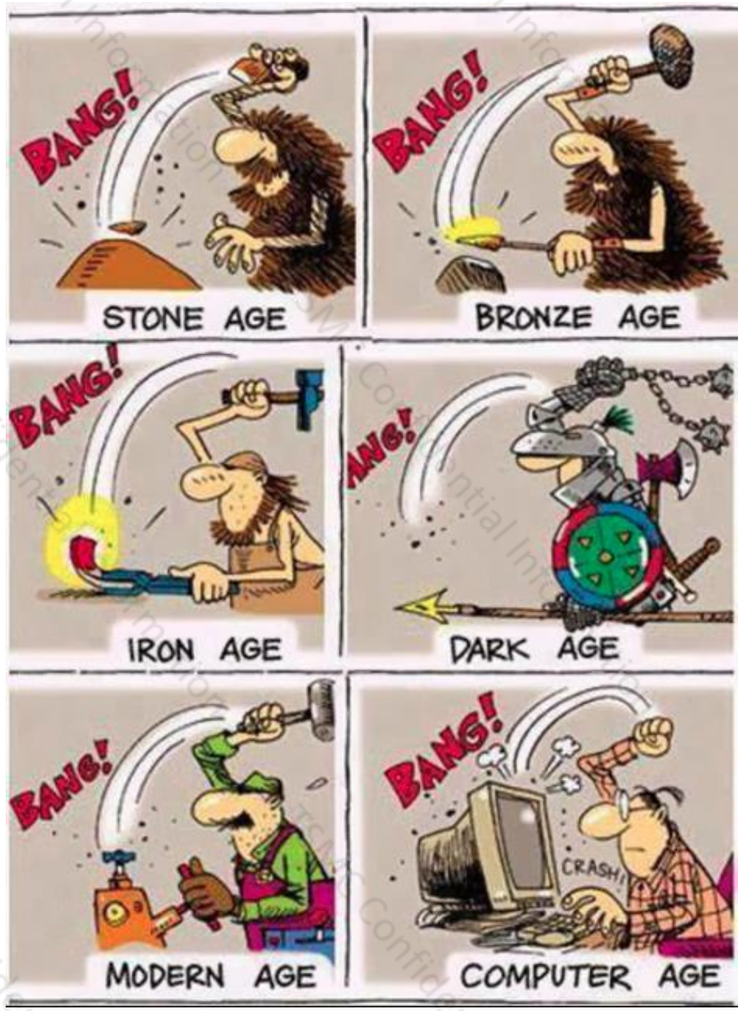
The difference between human and animals?

The difference between human and animals?

Thinking~

Thinking has taken us where no one has gone before.

Humans had been problem solvers... until they've decided to become PROGRAMMERS



Problem Solving Loop

Identify a

Analyze cause and effect

Identify the

What should we do with a solution?

Share engage people.
Together we are stronger!

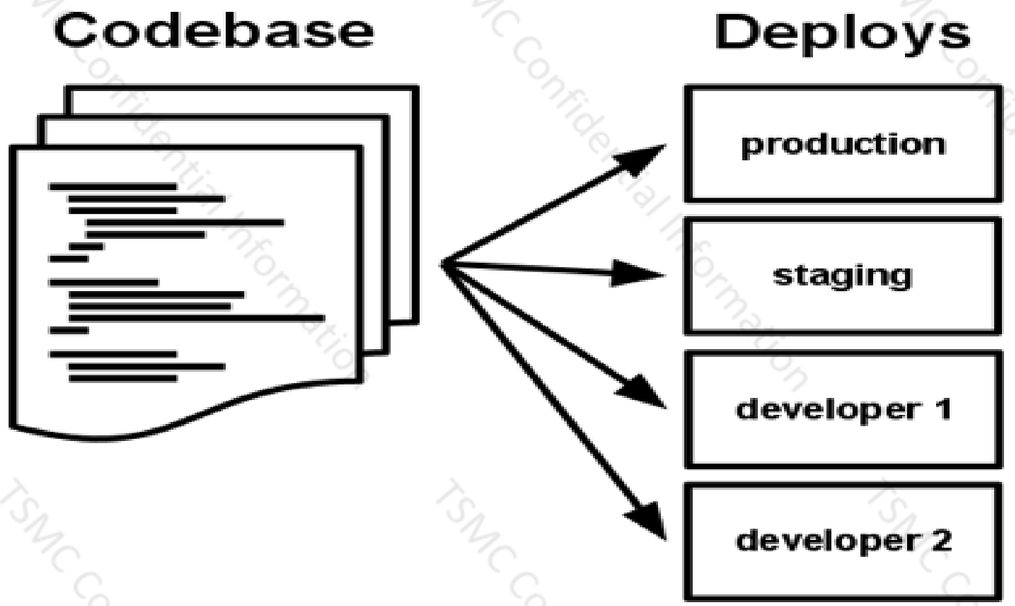
Design Principle

- Problem Solving Loop
- The 12 Factors to Cloud Success
- Design Patterns
  - Practice at Class (TSMC APC case study)
- 期末實作專題講解

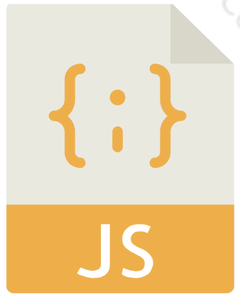
1. Codebase
2. Dependencies
3. Config
4. Backing services
5. Build, release, run
6. Processes
7. Port binding
8. Concurrency
9. Disposability
10. Dev/prod parity
11. Logs
12. Admin processes

1 - Codebase

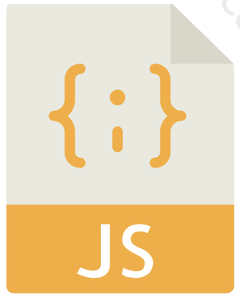
One codebase tracked in revision control, many deploys



Code

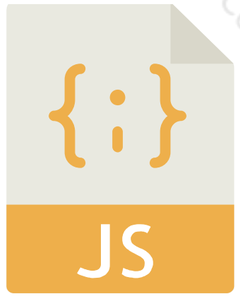


Code



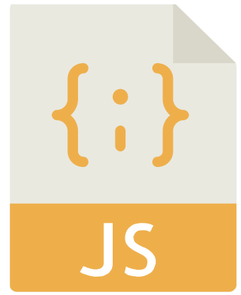
Version Control

Code



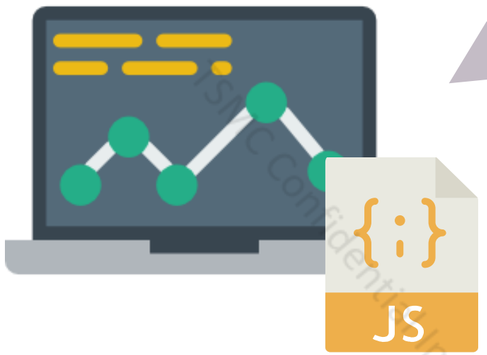
Version Control

Deployed Version



Dev #1

Dev #2

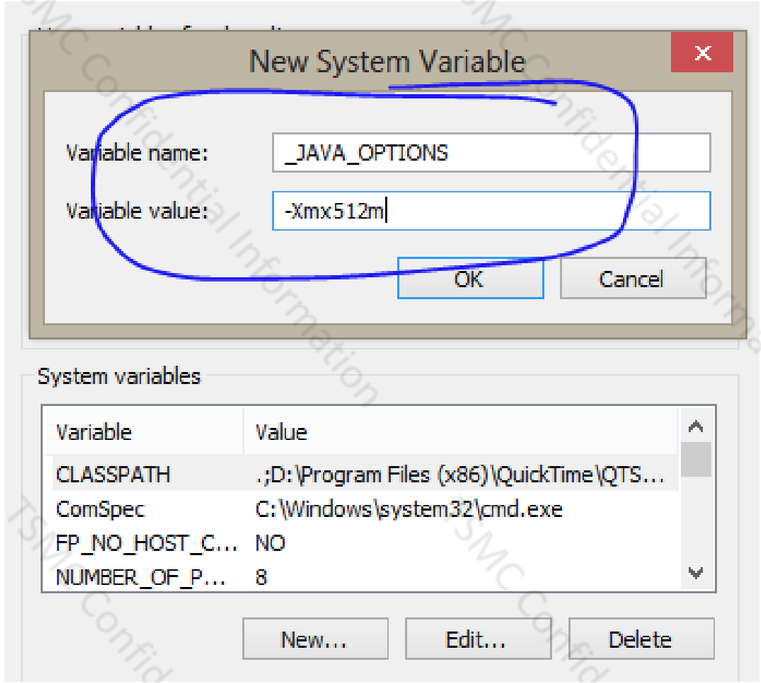


Staging / QA

Production

3 - Config

Store config in the environment

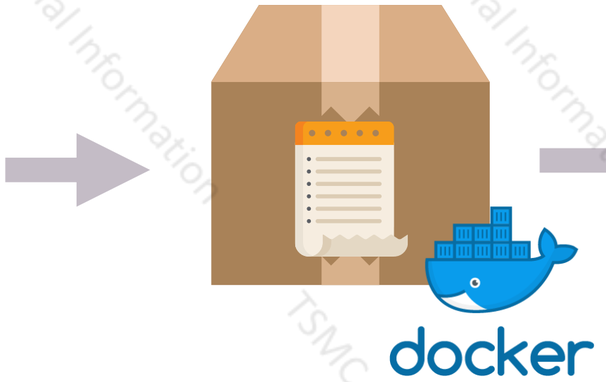


3 - Config (What does it mean?)

If you have to repackage your application, you're doing it wrong!

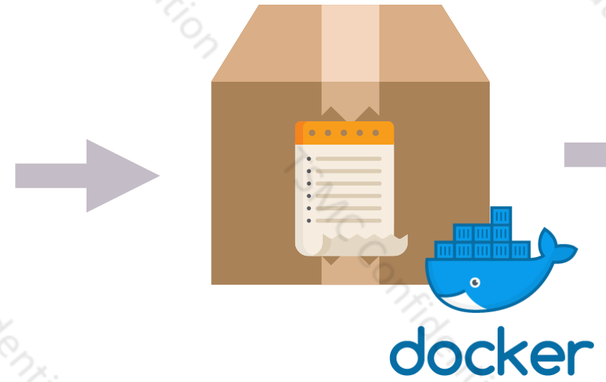
Prefer to store the config outside your application

Development Configuration

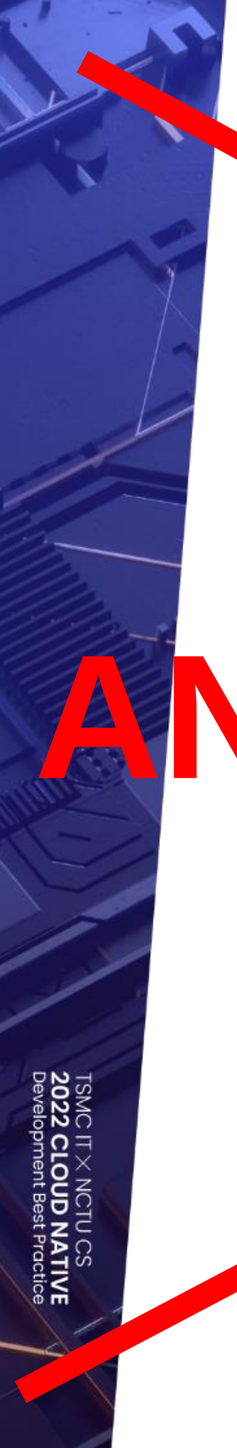


Development

Production Configuration



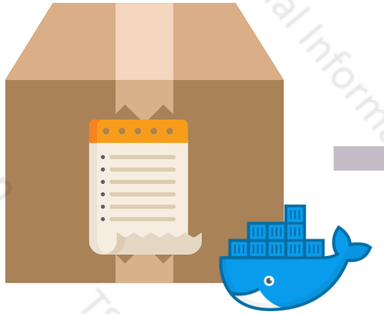
Production

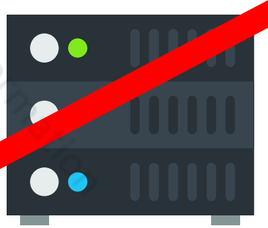


TSMC IT × NCTU CS
2022 CLOUD NATIVE
Development Best Practice

ANTIPATTERN

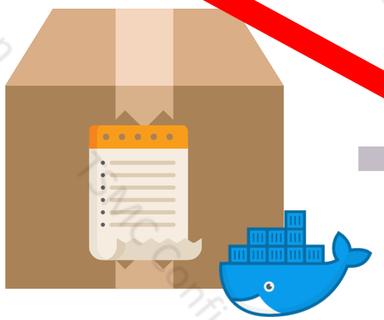
Development Configuration





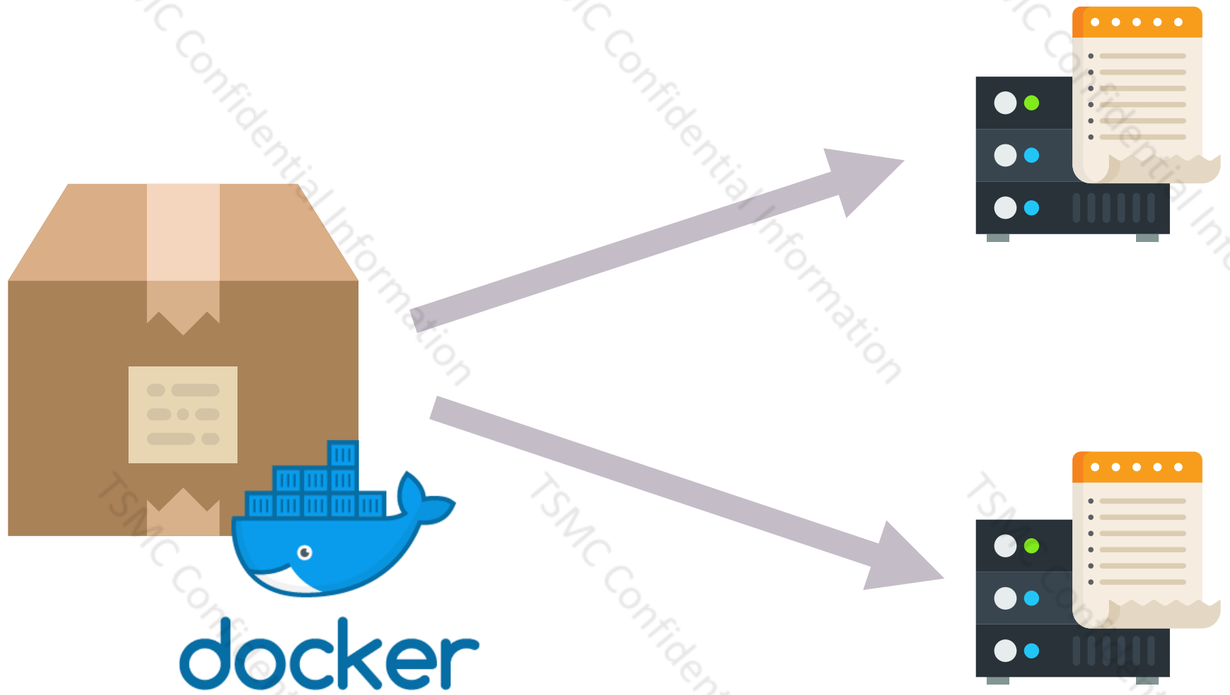
Development

Production Configuration



Production

Same container deployed to both environments. Configuration is part of the environment on the host.

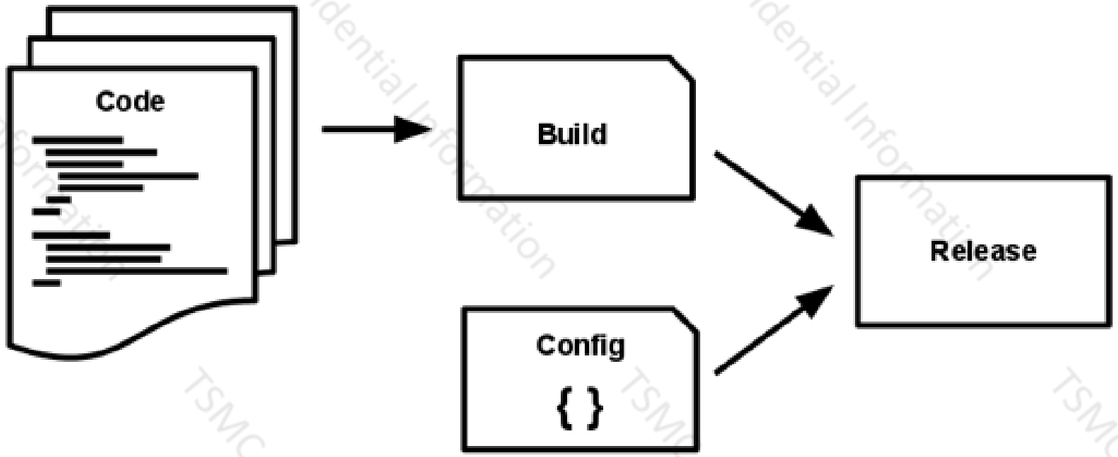


Development

Production

5 - Build, release, run

Strictly separate build and run stages



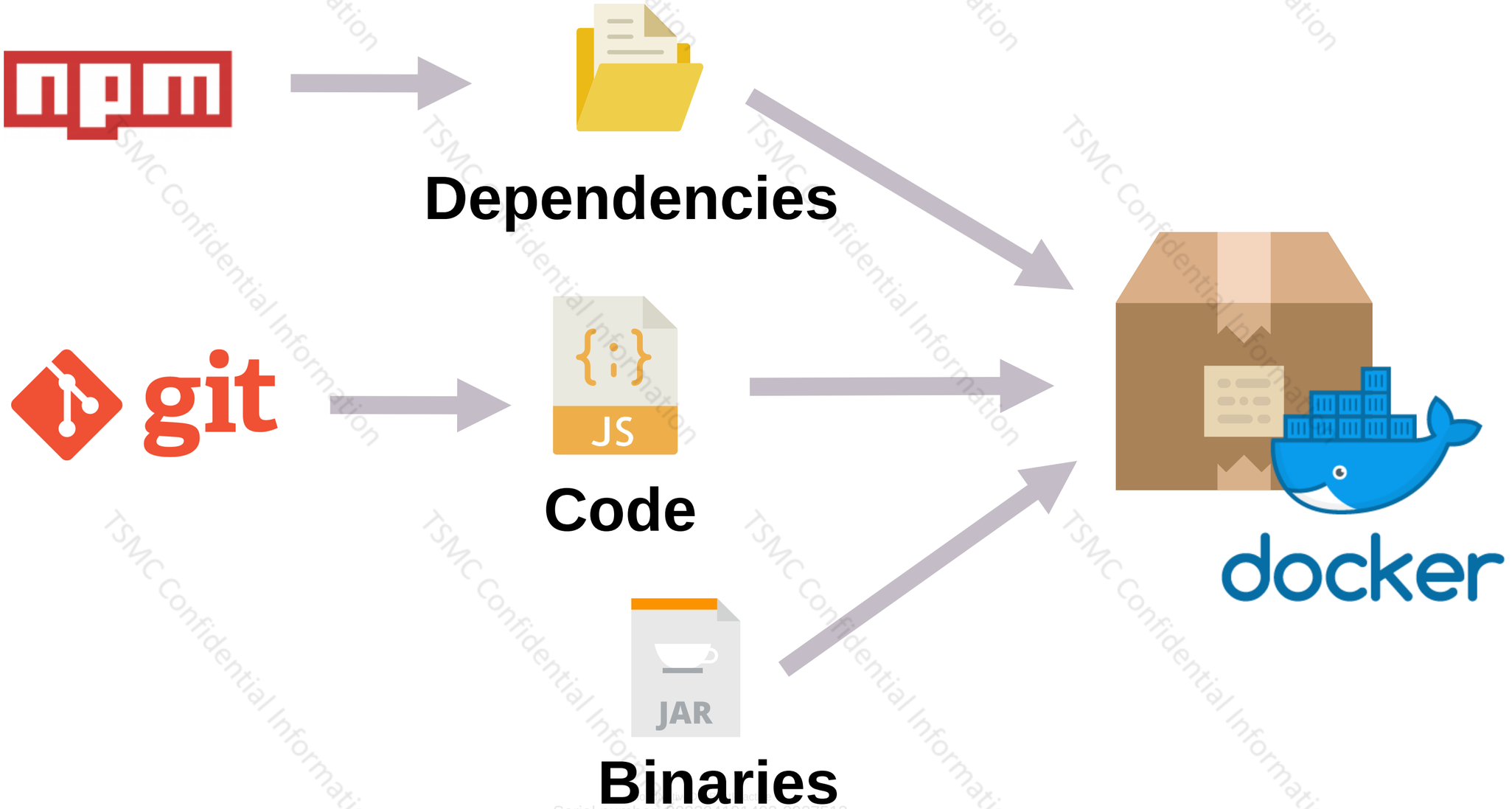
5 - Build, release, run (What does it mean?)

The twelve-factor app uses strict separation between the build, release, and run stages.

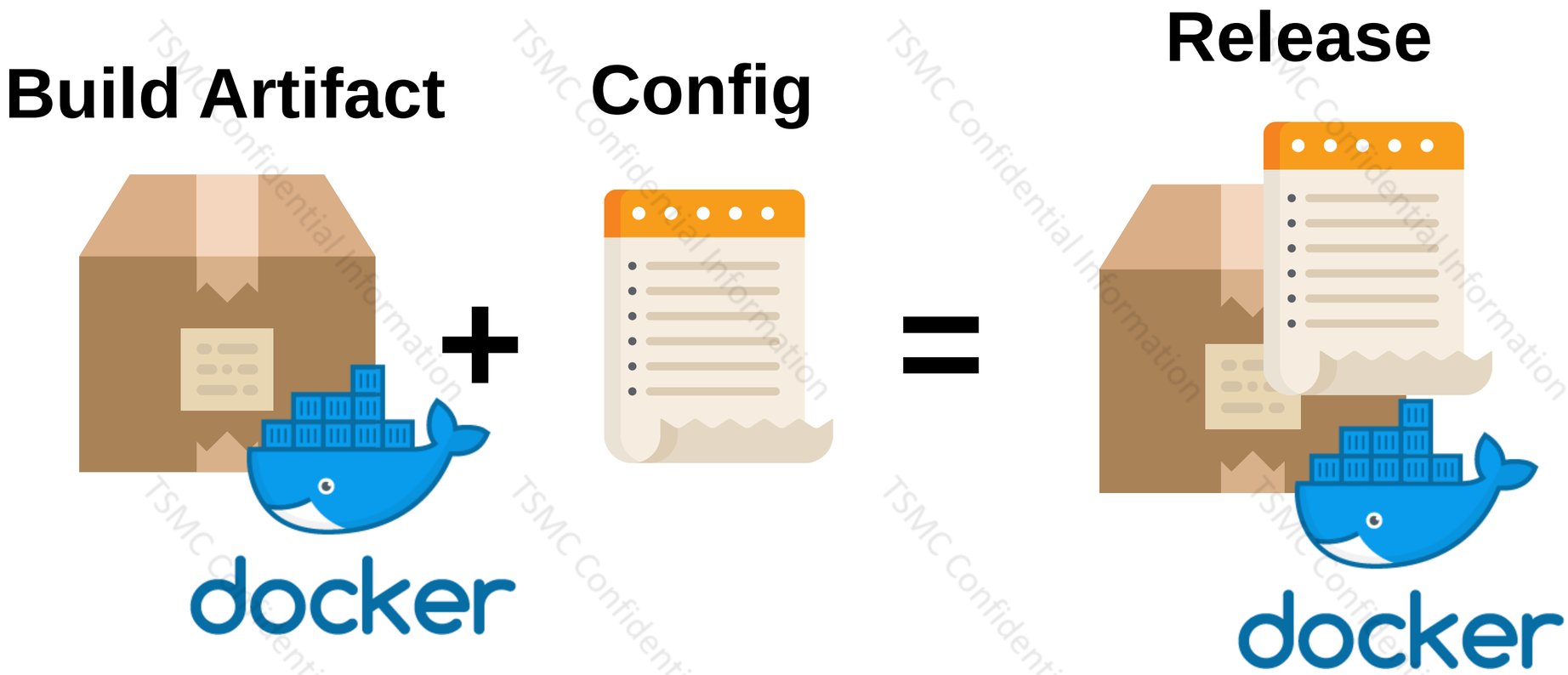
- **Every release should always have a unique release ID**
- **Releases should allow rollbacks**

Stage	Who?	What?	Why?
Build	CI	WAR / JAR / etc	Avoid "It works in my machine"
Release	CD	Container image	Deployments / Updates and Rollbacks
Run	Platform	Container instance	Speed, Management, Orchestration

Build

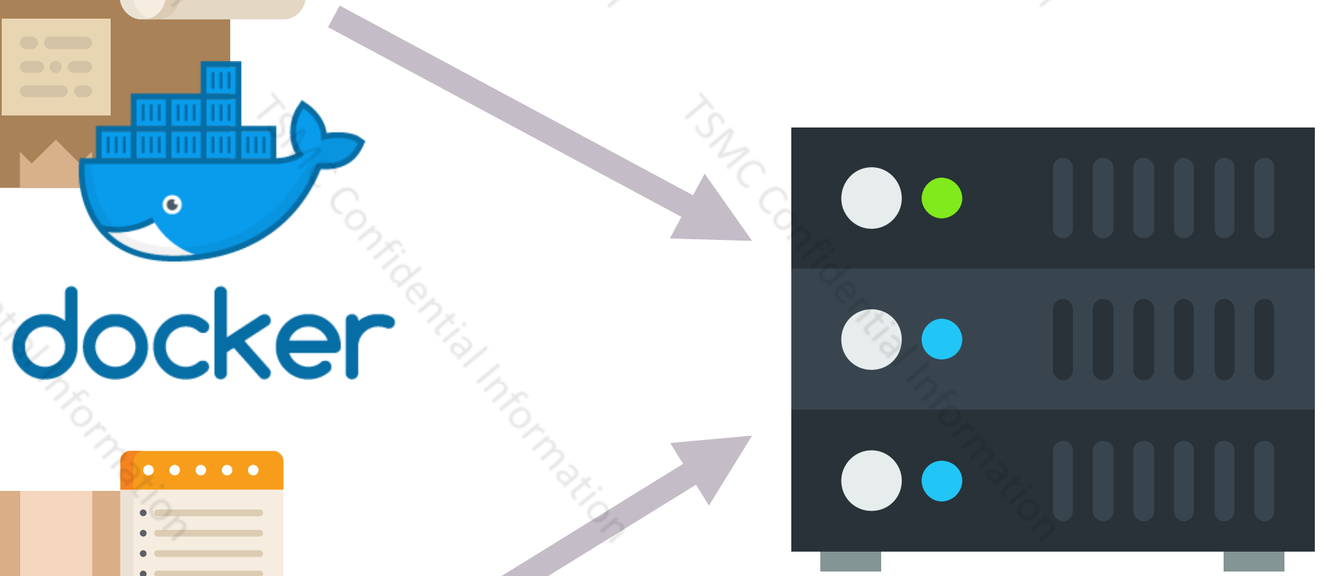


Release



Run

Task Definition
Release v1.0.0



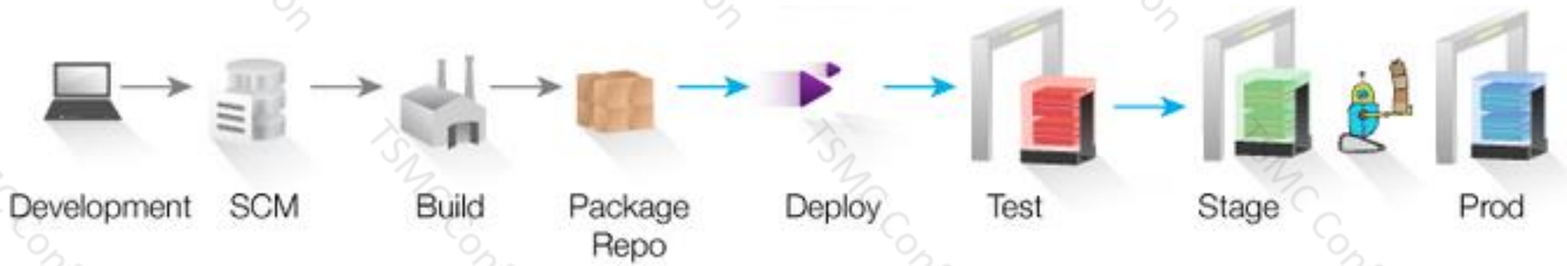
Task Definition
Release v1.0.1

10 - Dev/prod parity

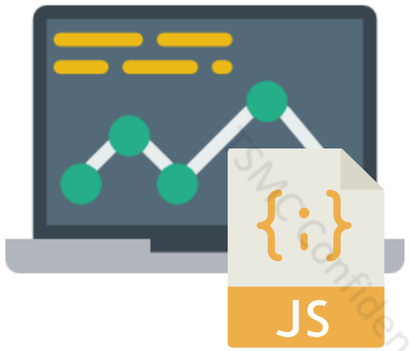
Keep development, staging, and production as similar as possible

10 - Dev/prod parity (What does it mean?)

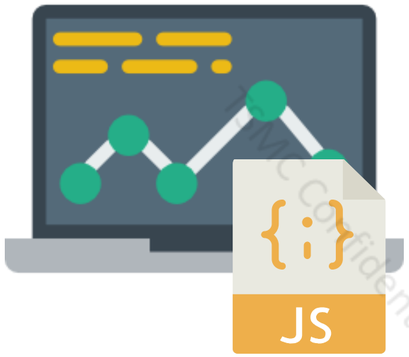
The twelve-factor app is designed for continuous deployment by keeping the gap between development and production small



Dev #1



Dev #2



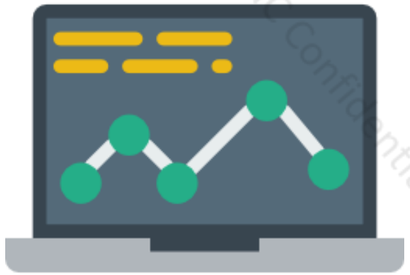
Staging / QA

Production

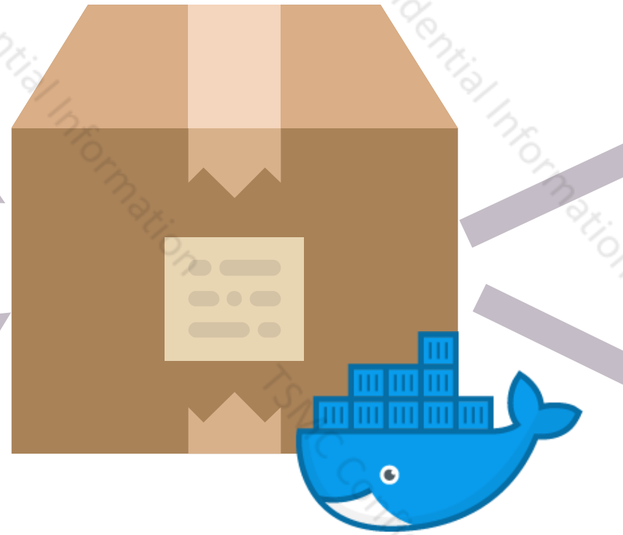
Local

Dev #1

Dev #2



Application



docker

Remote

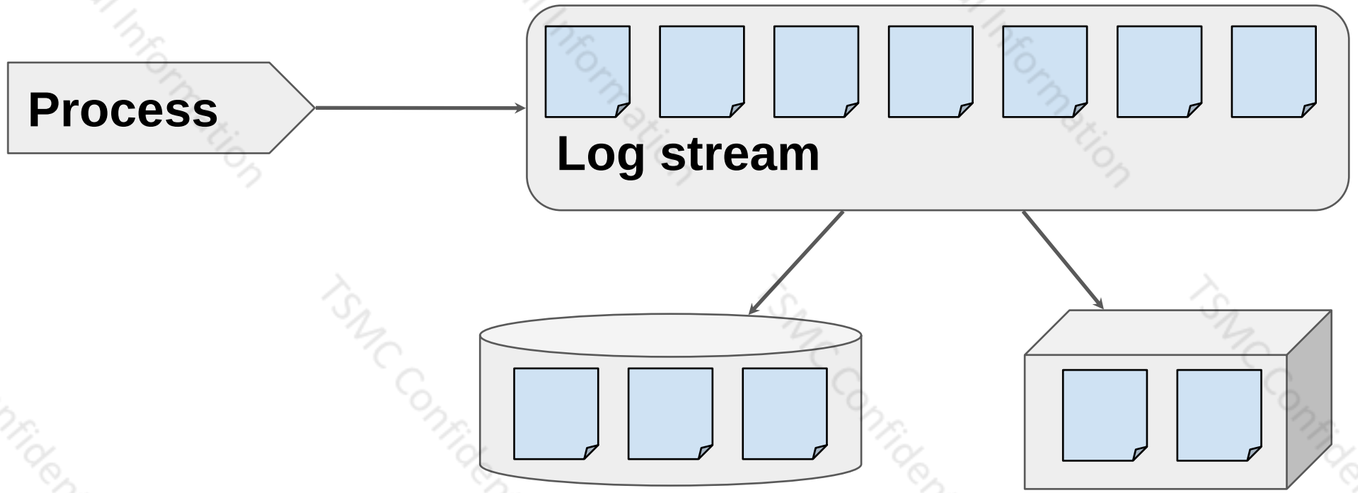
Staging / QA

Production

11 - Logs (What does it mean?)

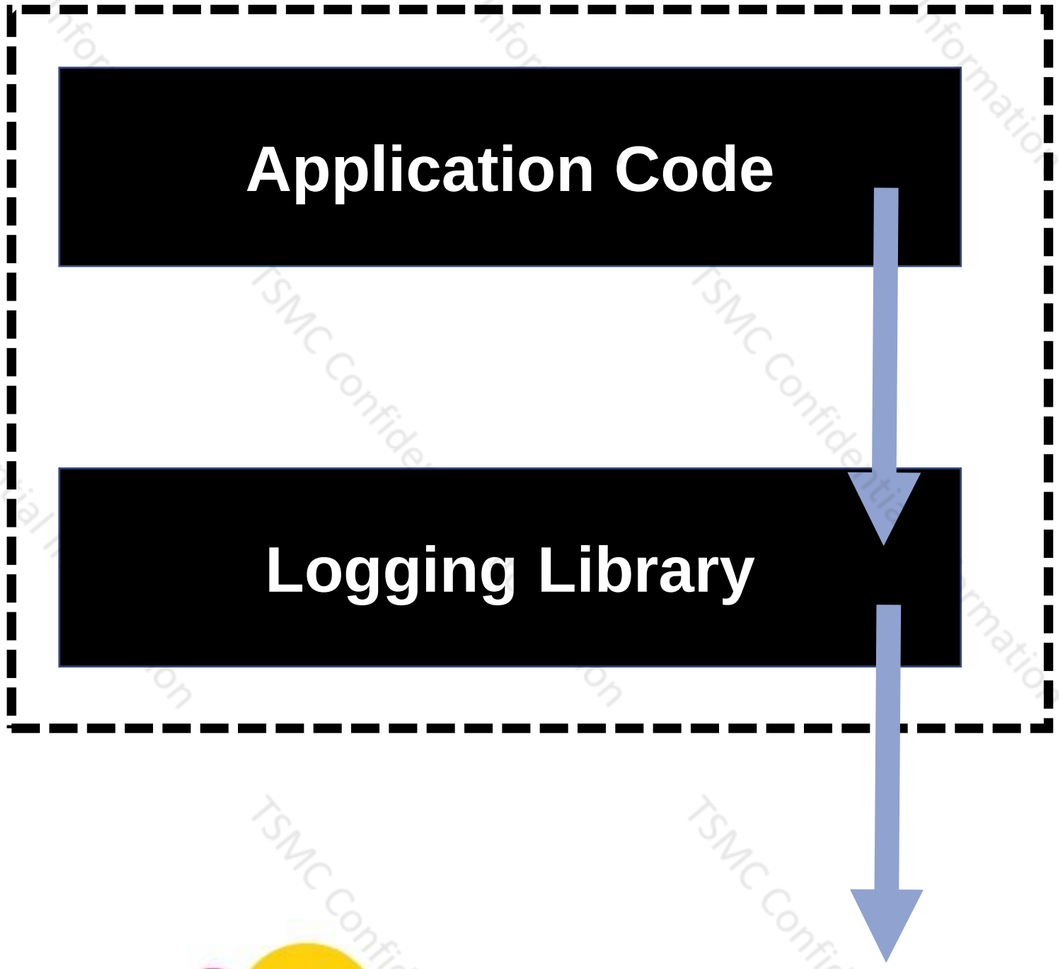
Treat logs as event streams

A twelve-factor app never concerns itself with routing or storage of its output stream



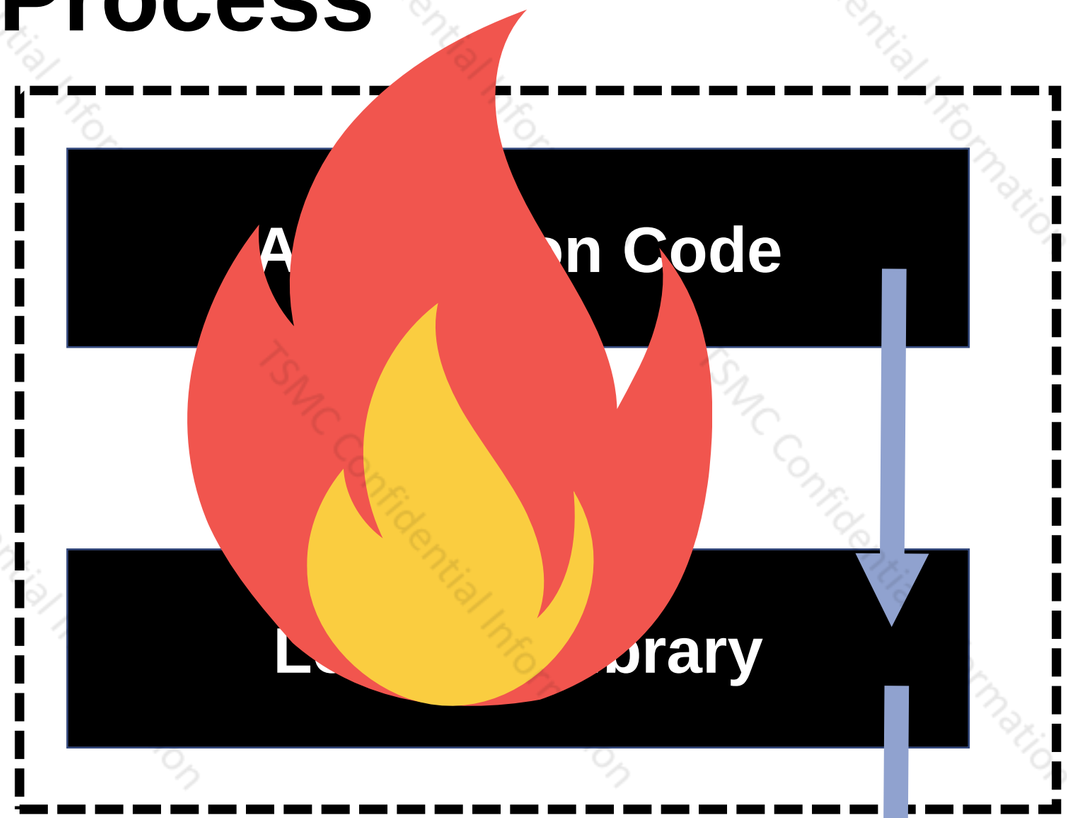
Treat logs as an event stream, and keep the logic for routing and processing logs separate from the application itself.

Process



elastic

Process



Some logs get lost if they haven't fully flushed

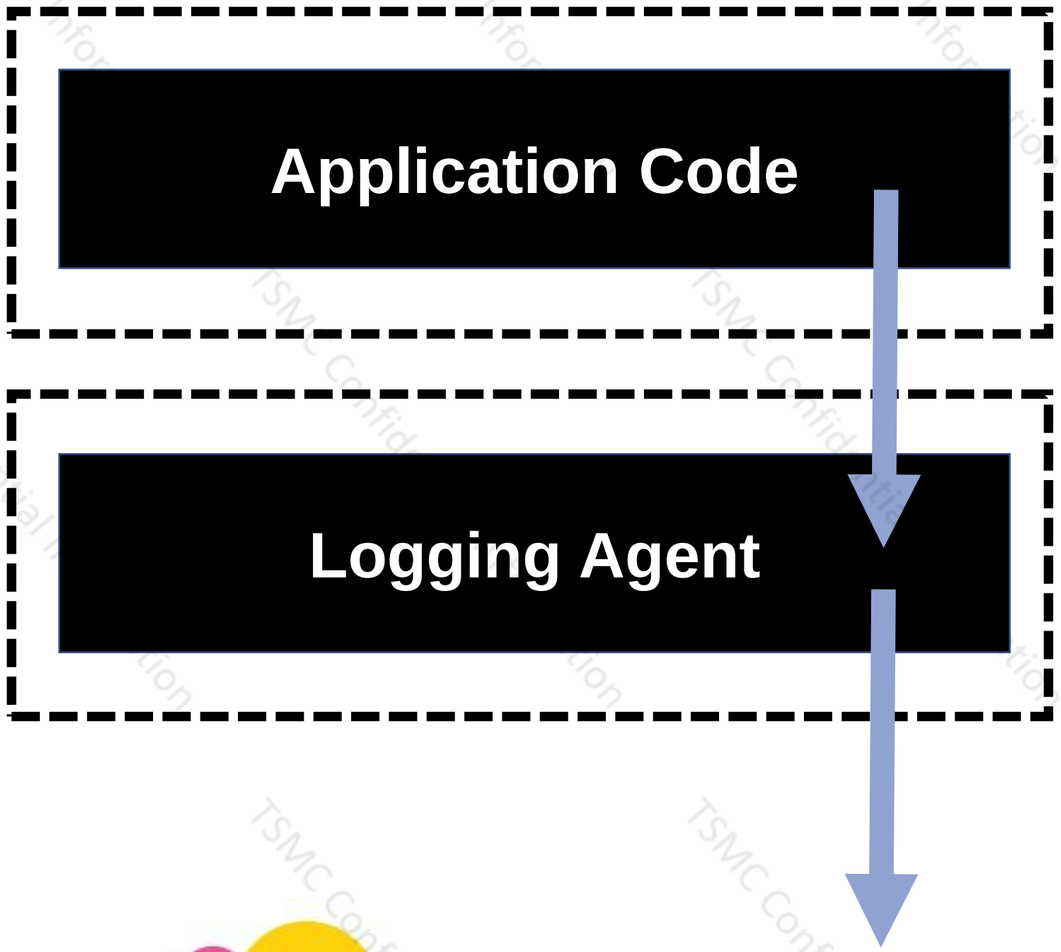
ANTIPATTERN

Process

Some logs get lost if they haven't fully flushed

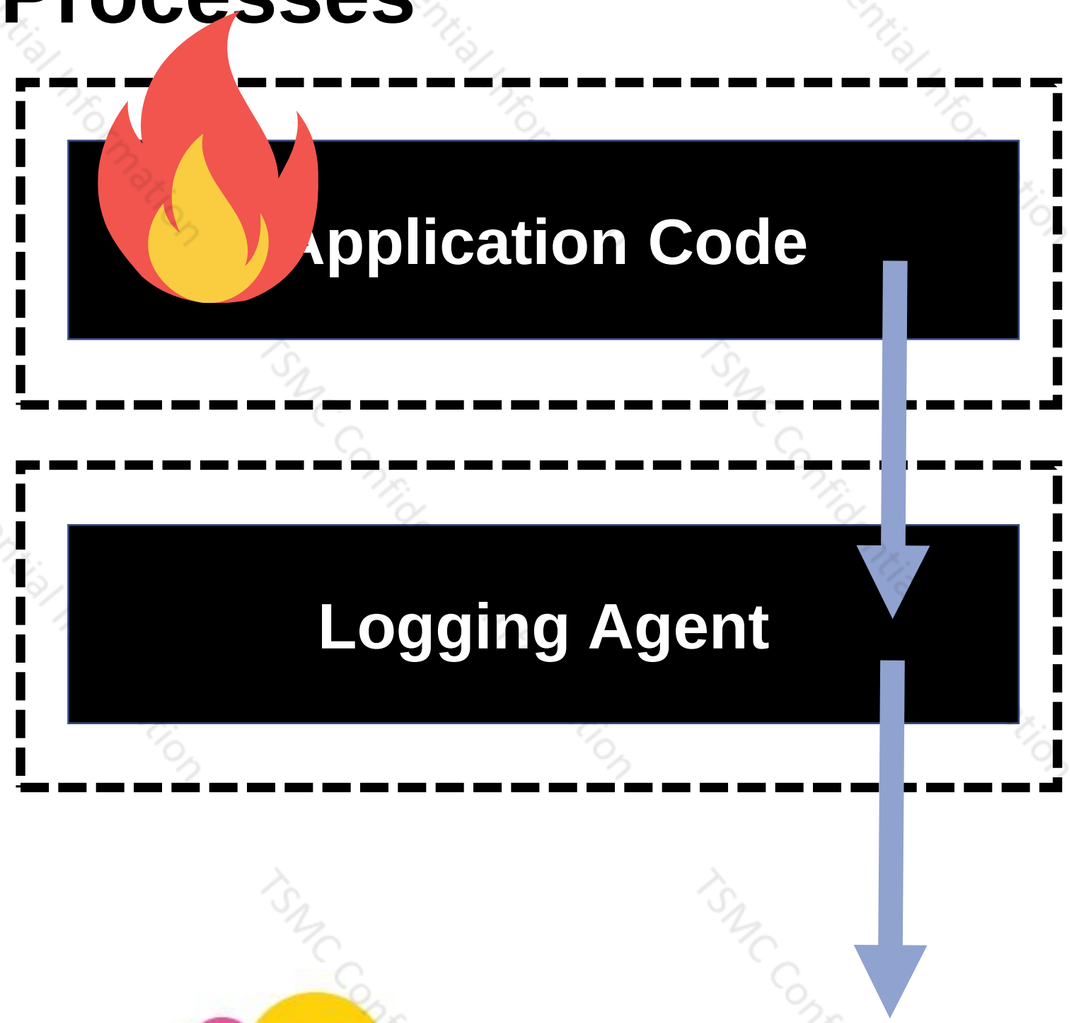
elastic

Processes



Logs go to an agent which handles exporting them off the host

Processes



Logs still reach agent, and still make it into ELK stack

elastic

Containerized code writes to stdout

```
var express = require('express');
var app = express();
var logger = require('morgan');

app.use(logger('tiny'));
```

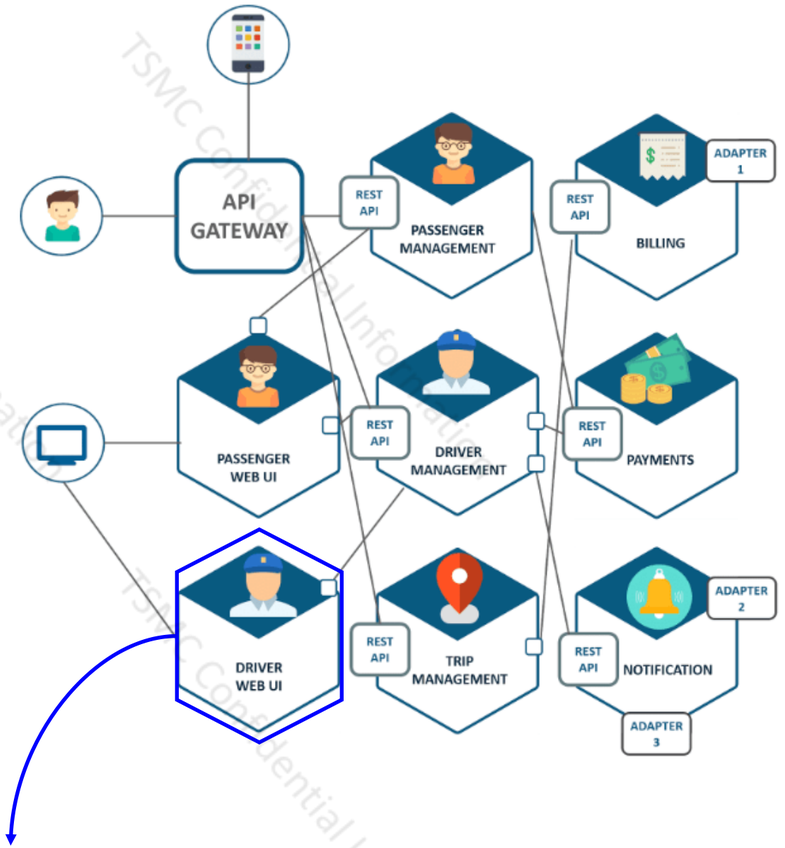
Docker connects container’s stdout to a log driver

```
docker run --log-driver awslogs myapp

docker run --log-driver fluentd myapp
```

1. Codebase
2. Dependencies
3. Config
4. Backing services
5. Build, release, run
6. Processes
7. Port binding
8. Concurrency
9. Disposability
10. Dev/prod parity
11. Logs
12. Admin processes

Design Principle

- Problem Solving Loop
- The 12 Factors to Cloud Success
- Design Patterns
  - Practice at Class (TSMC APC case study)
- 期末實作專題講解



Design Thinking inside the AP~!!

Principles + Problem = Pattern

Principles = SOLID + some general tips

Single Responsibility Principle

Open Closed Principle

Liskov Substitution Principle

Interface segregation principle

Dependency Inversion Principle

SRP (Single Responsibility Principle)

[SOLID]

A class or module should have one, and only one, reason to change.

(Clean Code, Robert C. Martin)

Why SRP?

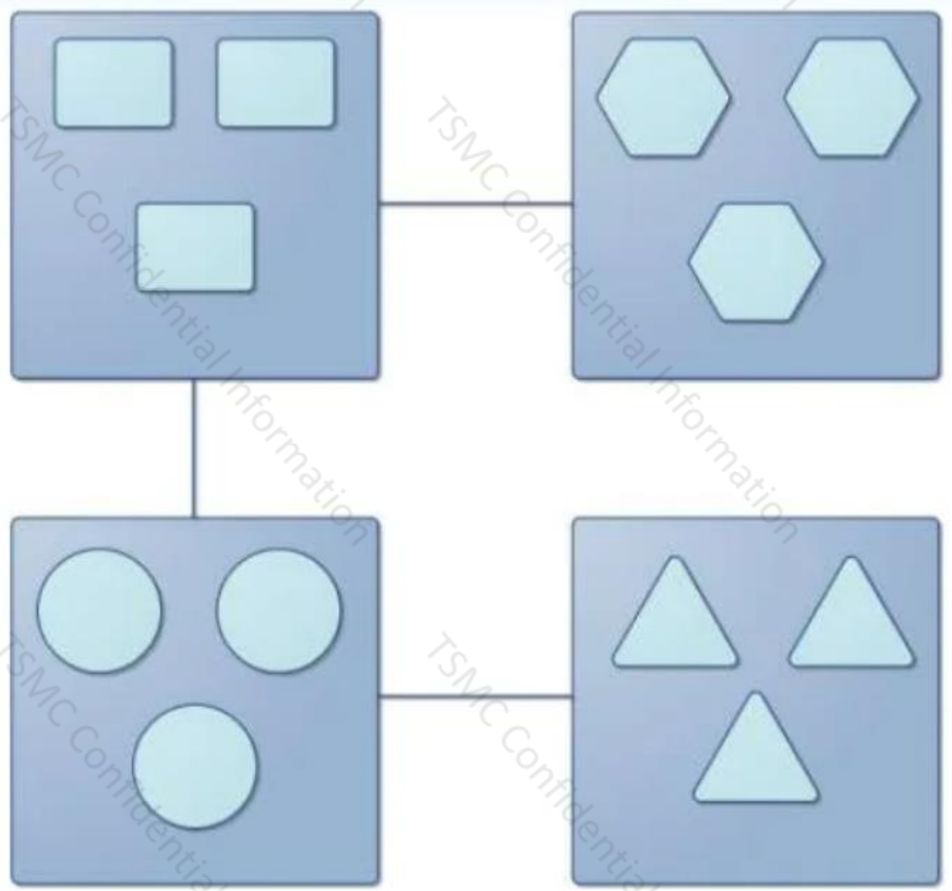
Organize the code

A place for everything and everything in its place

Why SRP?

Easier code changes (Refactoring)

- Less fragile code
- Low coupling
- High cohesion



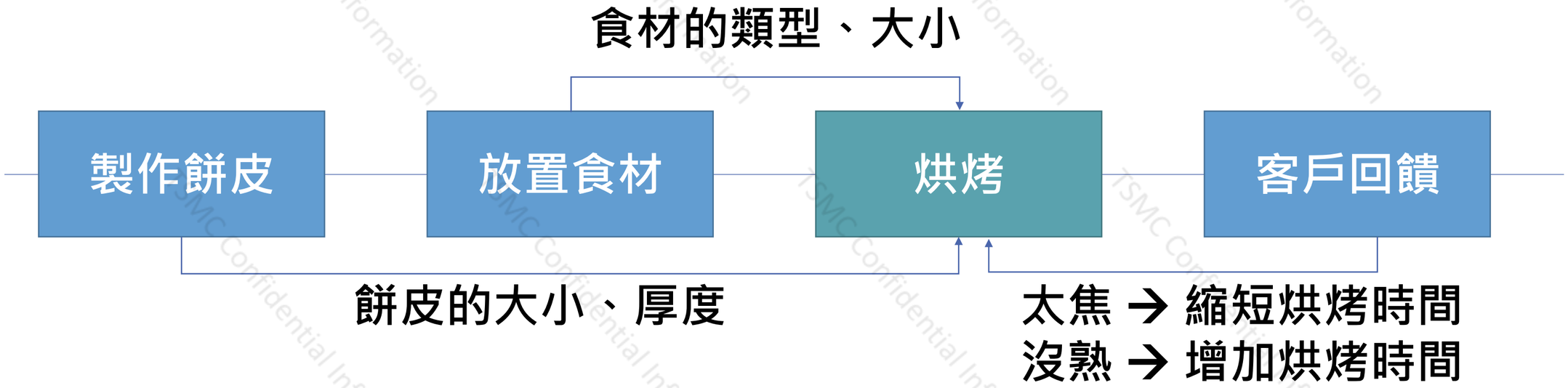
Why SRP?

- **Easier naming**
  - *The smaller and more focused class, it will be easier to name*

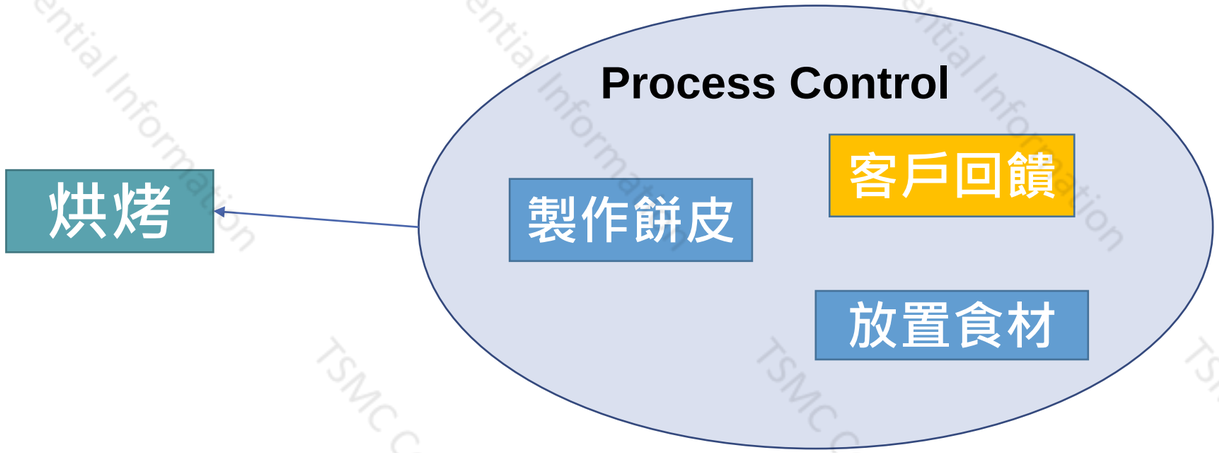
- **Maintainability**
- **Testability and easier debugging**

[回顧] Process Control Concept

- 在製作Pizza 時，有3道的製程步驟: 1. 製作餅皮, 2. 放置材料, 3. 烘烤
- 烘烤的時間，必須考慮到前2站的結果，並且透過客戶的回饋修正

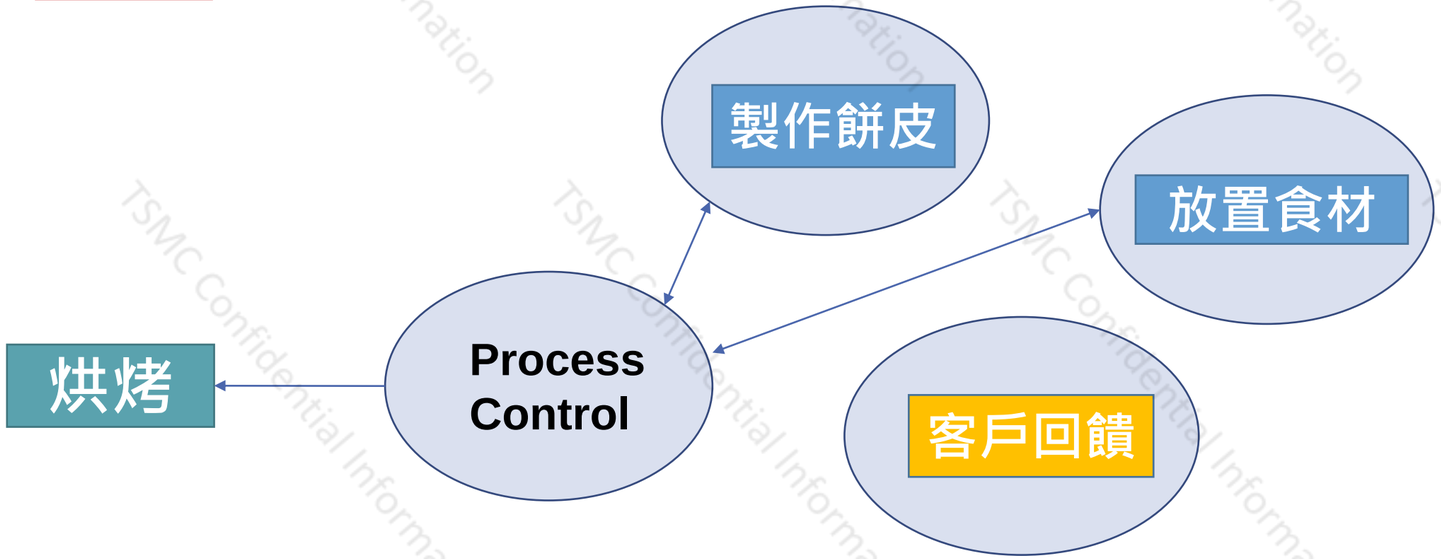


Bad:




```
function processControl(){
    if(客戶回饋){
        執行客戶回饋 detail;
    }else{
        製作餅皮因素 detail;
        fn=餅皮大小*a+厚度*b+…
        放置食材種類 detail;
        考量食材類型
        考量食材大小
        產生烘烤值 detail;
    }
}
```

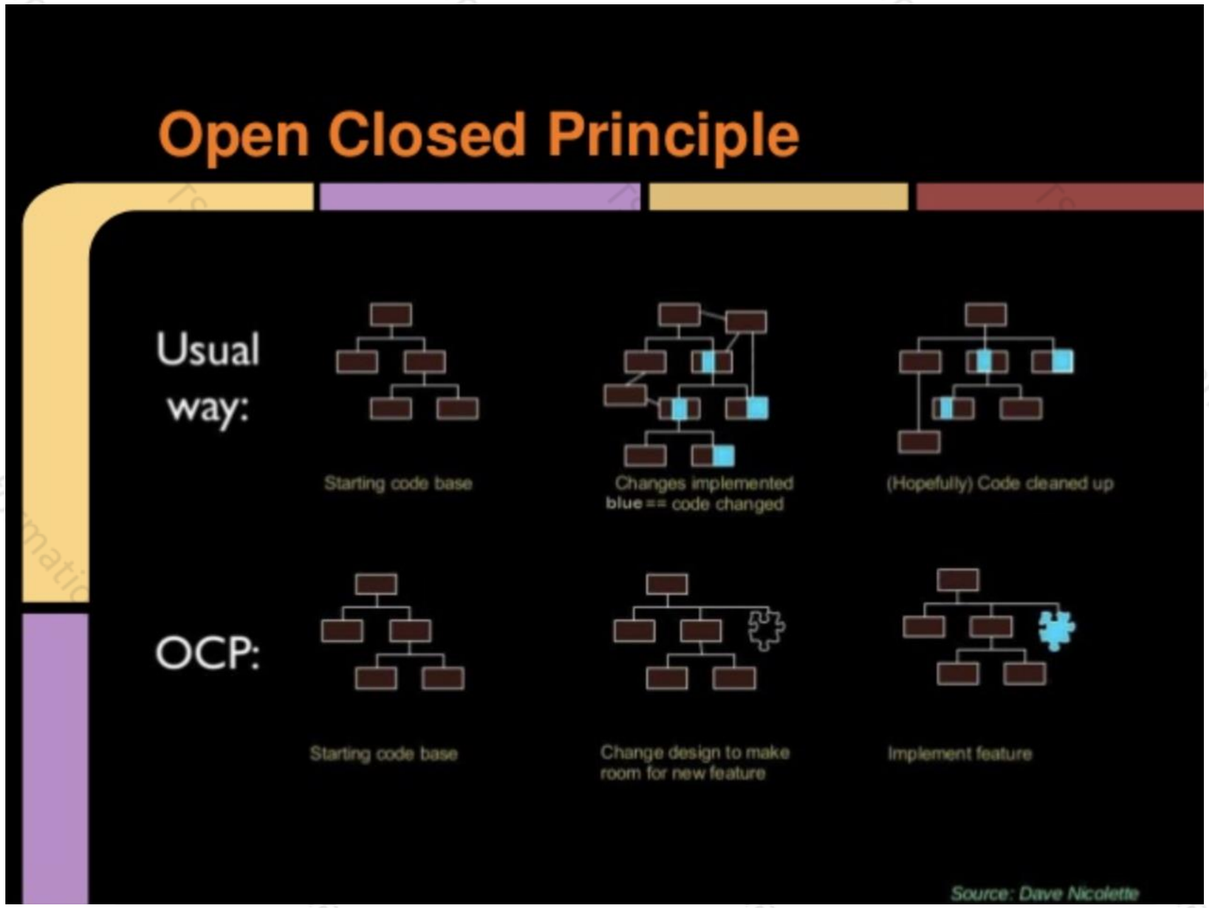
Good:




```
function processControl(){
    call interface 製作餅皮因素;
    call interface 放置食材種類;
    call interface 產生烘烤值;
}
function 製作餅皮(){
   return 餅皮大小*a + 厚度*b
};
function 放置食材(){
   考量食材類型, 大小
};

function 客戶回饋(){ …. };
```

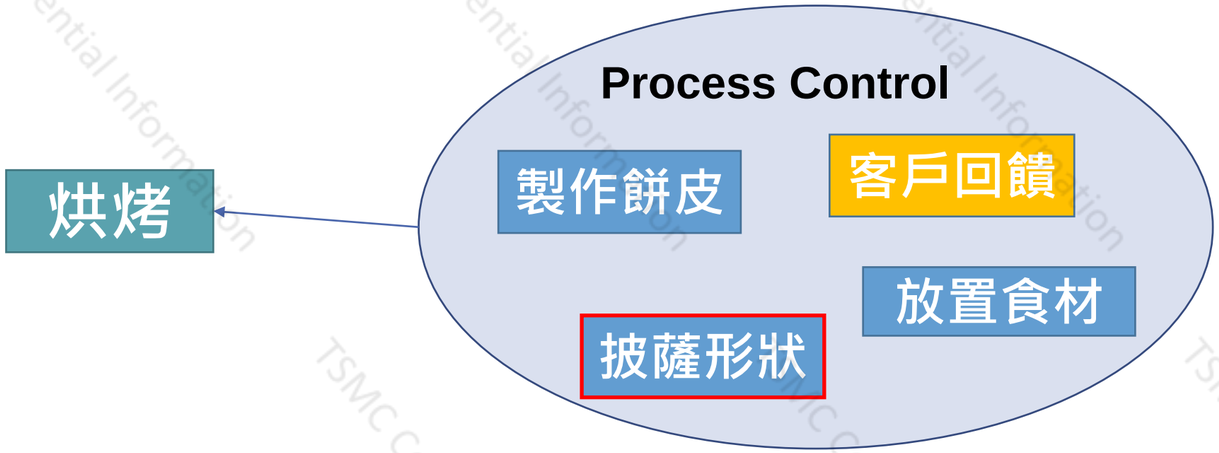
Keep It Simple, Stupid!

OCP (Open Closed Principle)

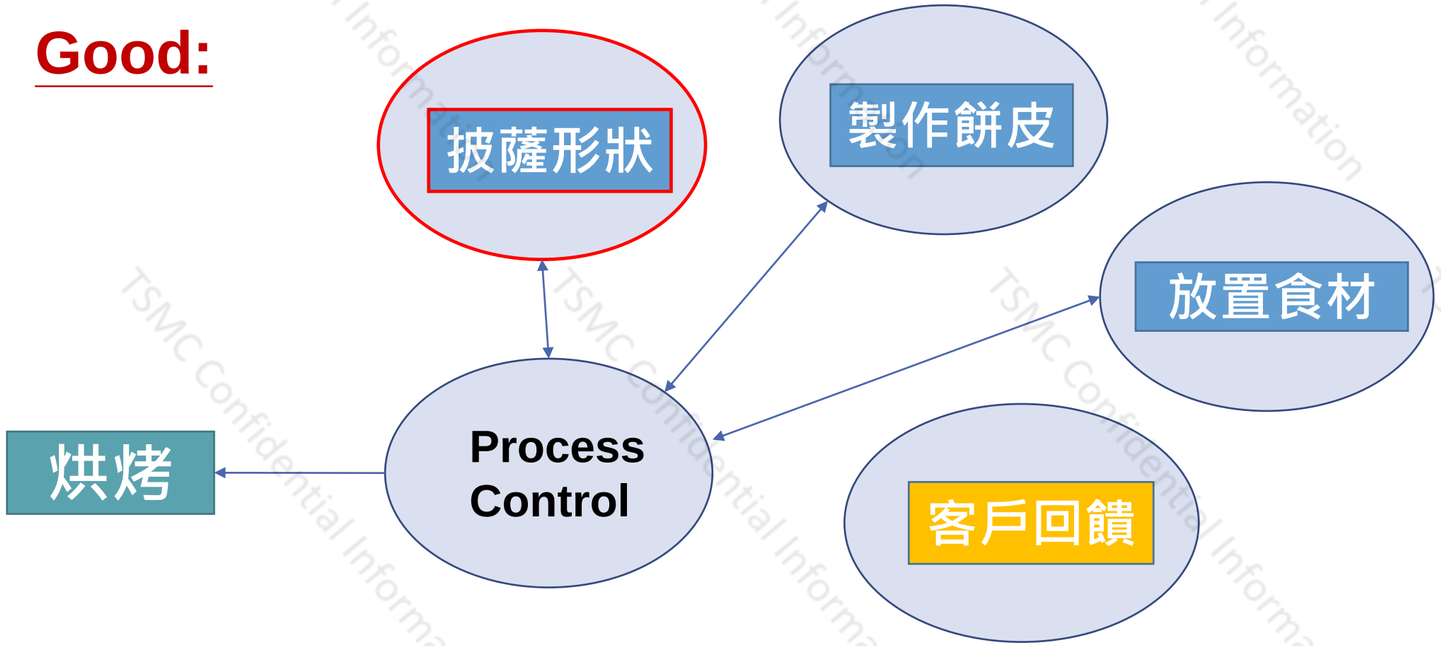


Software entities (classes, modules, functions, etc.) should be Open for extension, but Closed for modification.

Bad:



Good:



Modification

```
function processControl(){
    if(客戶回饋){
        執行客戶回饋 detail;
    }else{
        製作餅皮因素 detail;
        fn=餅皮大小*a+厚度*b+…
        放置食材種類 detail;
        考量食材類型
        考量食材大小
        選擇披薩形狀 detail;
        產生烘烤值 detail;
    }
}
```

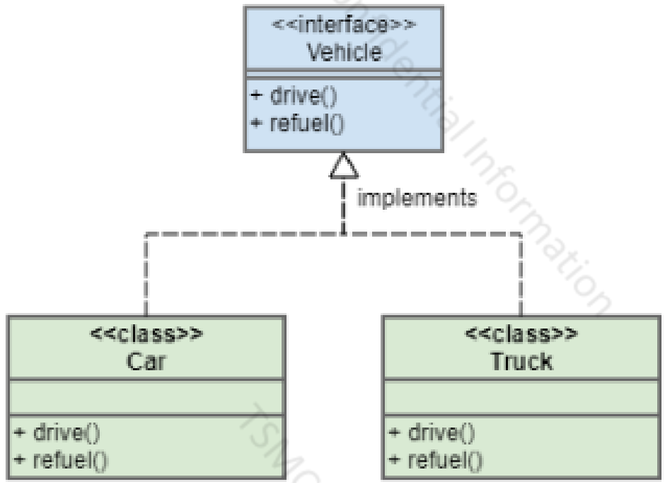
Extention

```
function processControl(){
    call interface 製作餅皮因素;
    call interface 放置食材種類;
    call interface 選擇披薩形狀;
    call interface 產生烘烤值;
}

function 製作餅皮(){ …. };
function 放置食材(){ …. };
function 披薩形狀(){ …. };

function 客戶回饋(){ …. };
```

Interface

- A classlike constructure used to specify the behavior of a class
- Only abstract methods, and Cannot be instantiated

Interface is used when you want to define a contract and you don't know anything about implementation.



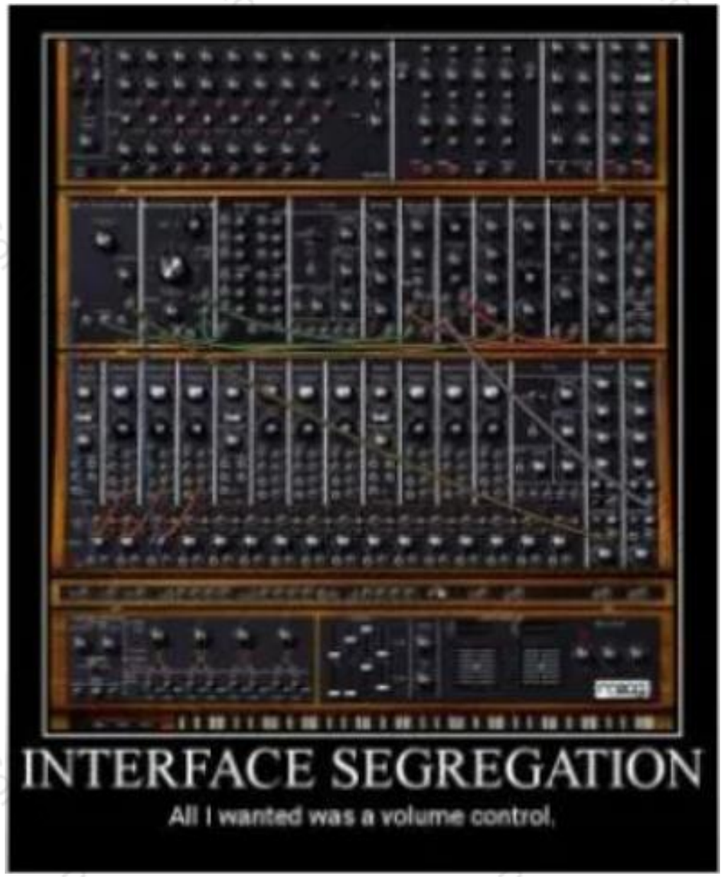

```
interface Vehicle{
    drive(): void;
    refuel(): void;
}

class Car implements Vehicle{
    public drive(): void{
        console.log('Drive Car');
    }

    public refuel(): void{
        console.log('Refuel Car');
    }
}

class Truck implements Vehicle{
    public drive(): void{
        console.log('Drive Truck');
    }

    public refuel(): void{
        console.log('Refuel Truck');
    }
}
```

```
var vehicle = new Car();
vehicle.drive();
vehicle.refuel();
```

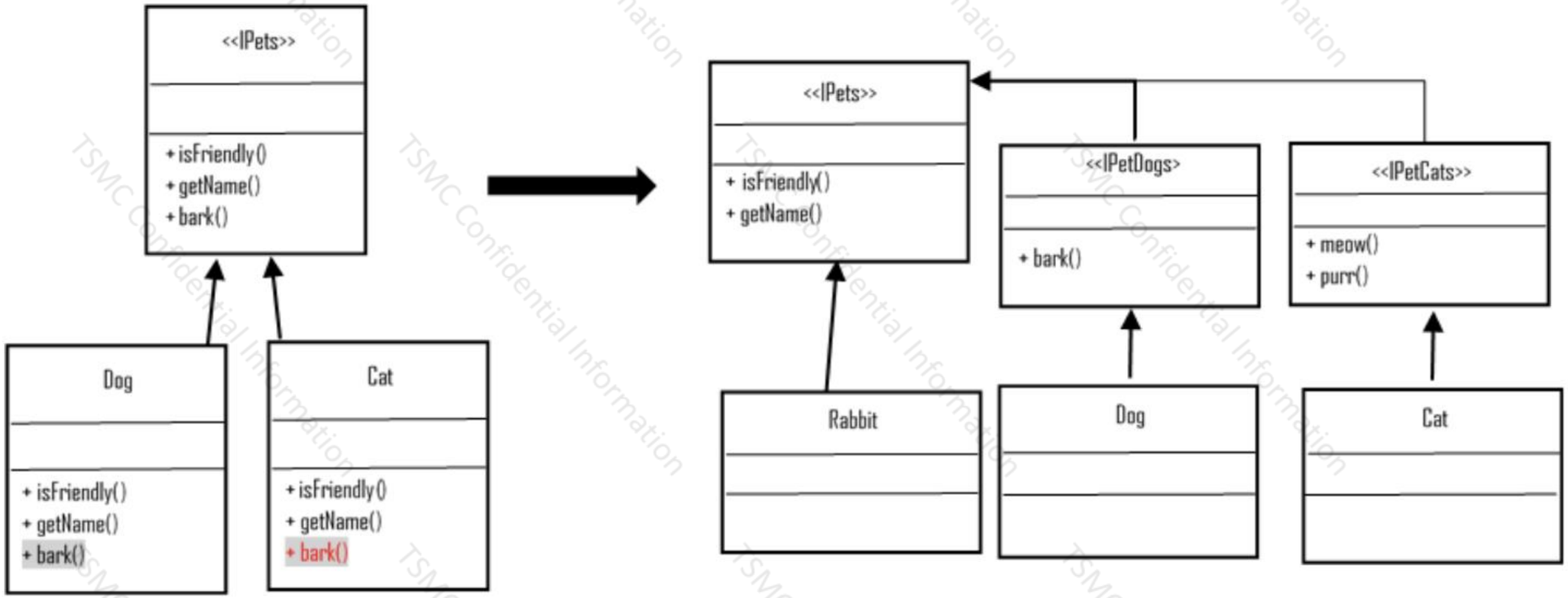
```
var vehicle = new Truck();
vehicle.drive();
vehicle.refuel();
```

Client 呼叫行為相同~!!

ISP (Interface segregation principle)

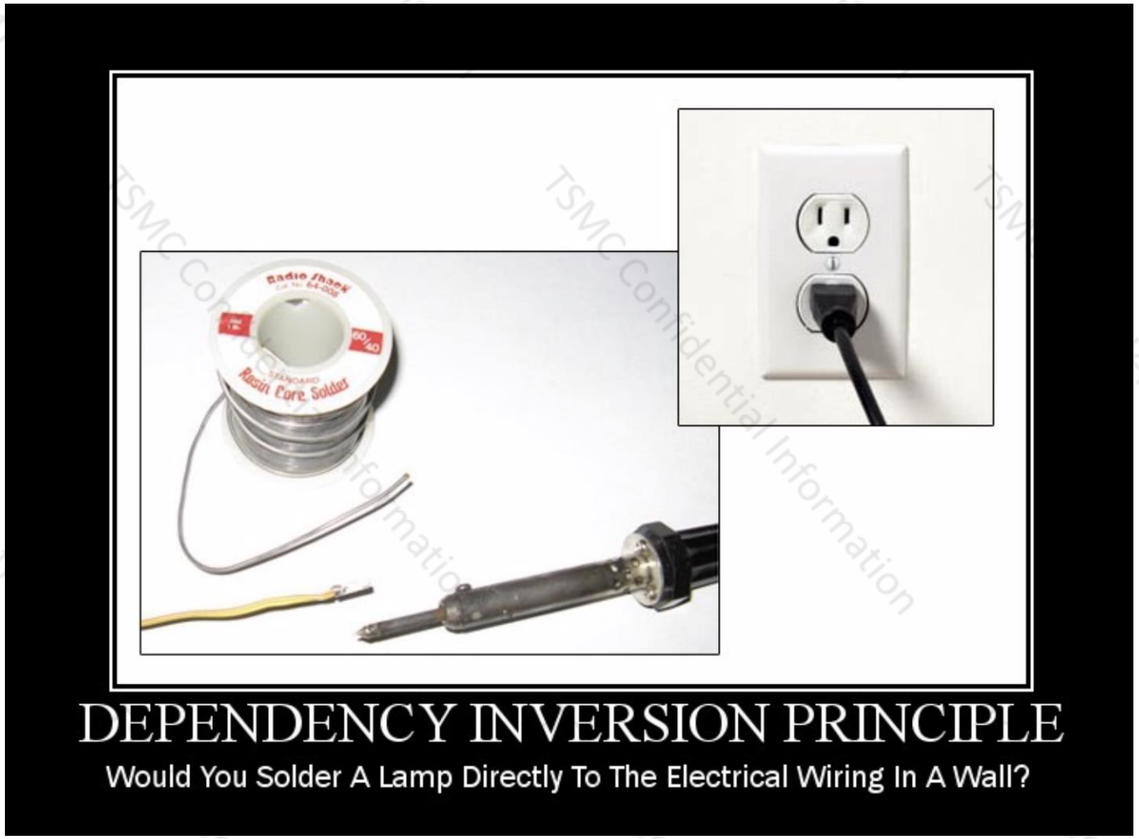


Many client-specific interfaces are better than one general-purpose interface.



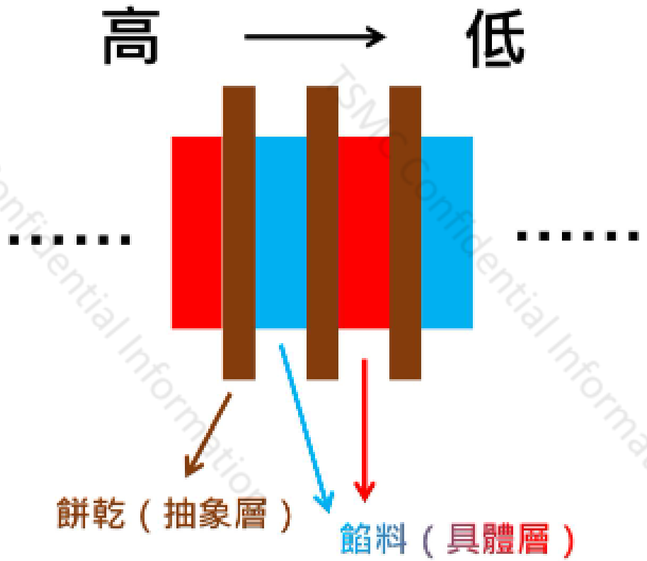
No any client should be forced to depend on methods it does not use.

DIP (Dependency Inversion Principle)



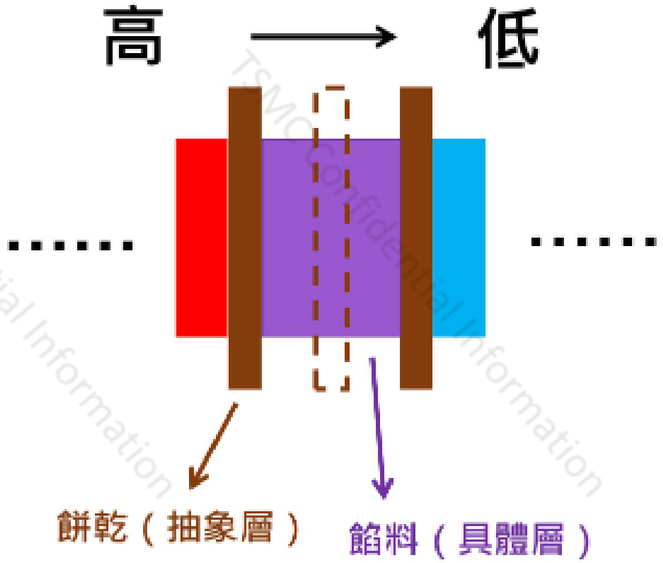
- **Abstractions should not depend on details.**
- **Don’t depend on anything concrete**
  - **Work with interfaces**

通常一個系統**變化最頻繁的地方就是商業邏輯**，可能**因客戶的需求變動時常有變化**，所以我們可以將商業邏輯視為是比較高層次的（意為**商業邏輯都會寫在比較高層次模組的位置**）。
換句話說，通常**變化不大的計算邏輯或功能型程式**，我們都會**實作在比較低層次模組的位置**



∞ Sandwich cookie

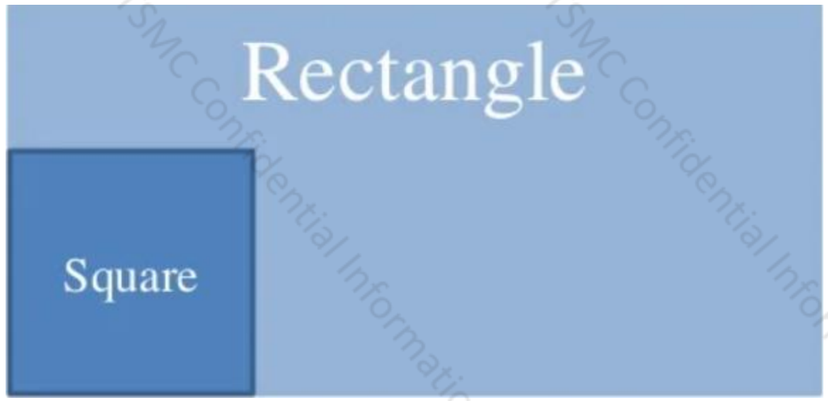
因為有餅乾（抽象層）的關係，兩種餡料（具體層）並不會實際碰到彼此，換句話說，這兩餡料可以各自加上一些佐料（實作各自發展），但**倘若兩餡料需要做到保持各自風味的前提下做結合，就需要餅乾來幫忙**



∞ Sandwich cookie

倘若沒有餅乾的保護，兩種餡料就會混合在一起（程式有相依性，即為**程式耦合**）

LSP (Liskov Substitution Principle)



Replace objects with instances of their subtypes without altering the correctness of that program.

SRP (Single Responsibility Principle)

Keep It Simple, Stupid!

Work with Interface~!

Principles + Problem = Pattern

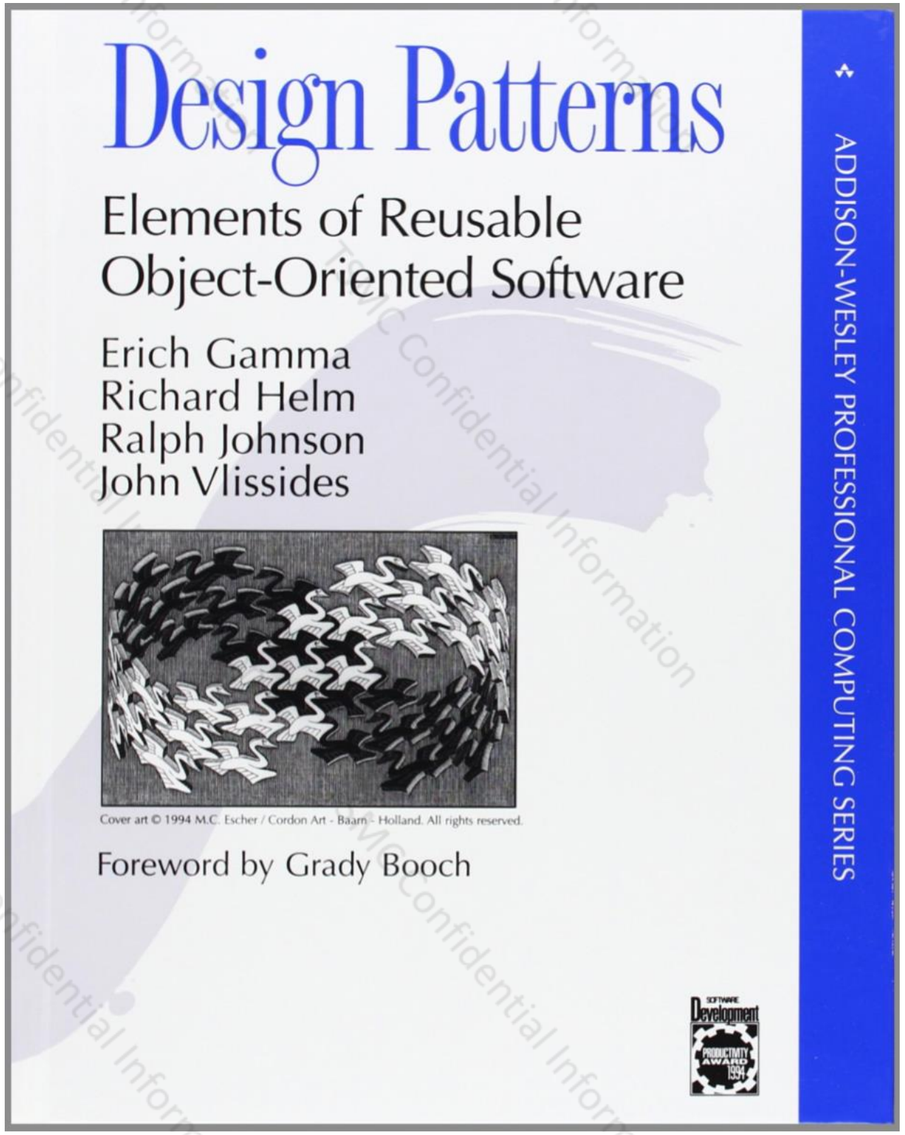
Brief History

- **By 1995, the Design Patterns book by the Gang of Four was published (Gamma et al. 1995).**

  Many of the patterns are based on existing idioms in programming languages.

- **New patterns have been created over the years:**

Kind	GoF	Wikipedia
Creational	5	10
Structural	7	12
Behavioral	11	15
Concurrency	0	16

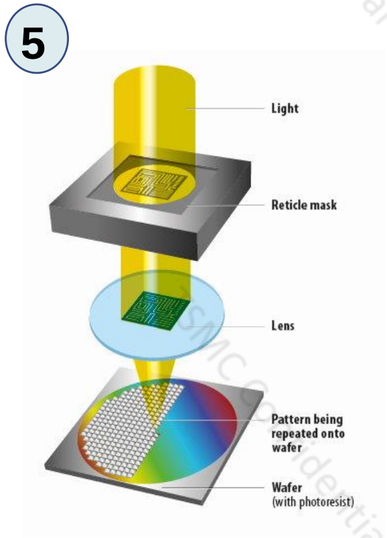


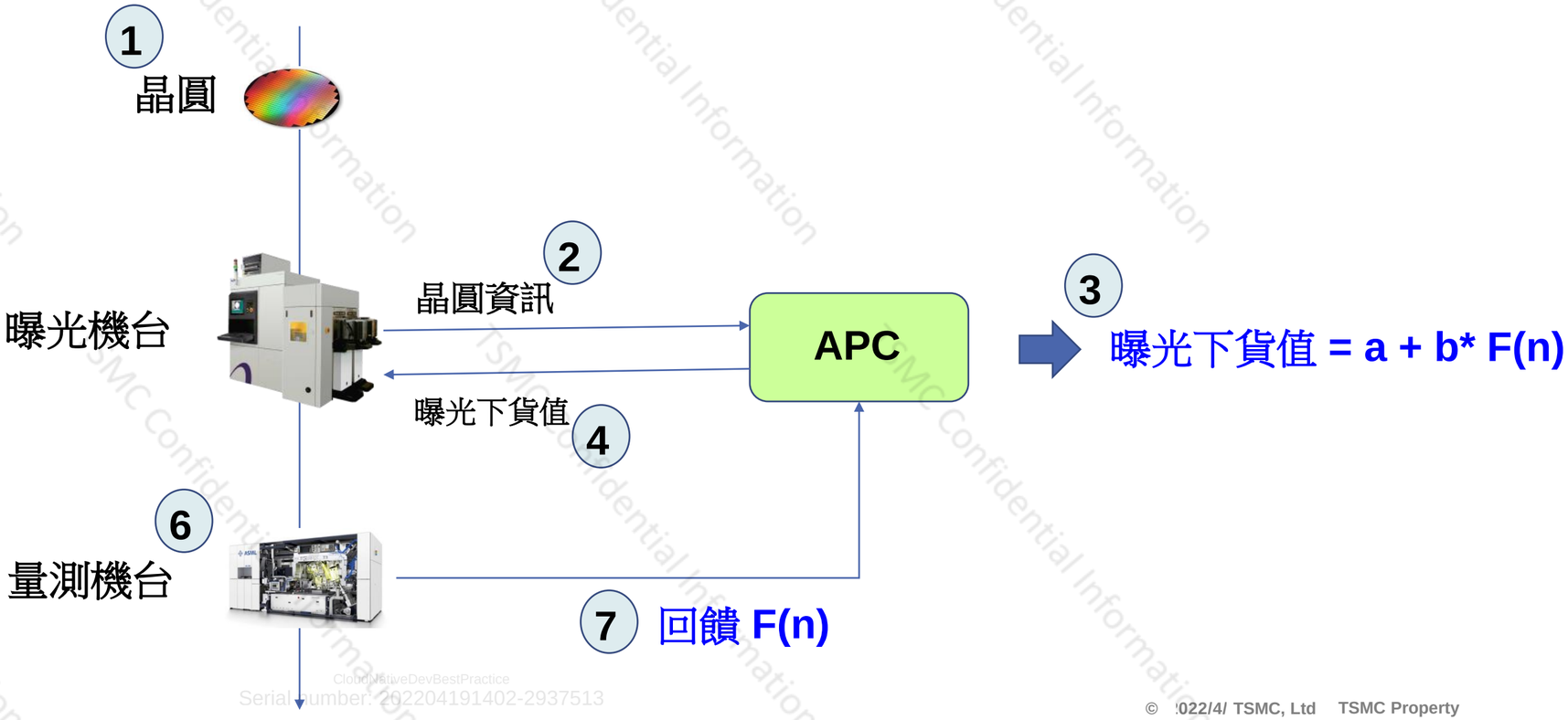
Gang of Four (GoF) patterns

- **Creational Patterns** *(abstracting the object-instantiation process)*
  - **Factory Method**
  - **Abstract Factory**
  - **Singleton**
  - **Builder**
  - **Prototype**
- **Structural Patterns** *(how objects/classes can be combined)*
  - **Adapter**
  - **Bridge**
  - **Composite**
  - **Decorator**
  - **Facade**
  - **Flyweight**
  - **Proxy**
- **Behavioral Patterns** *(communication between objects)*
  - **Command**
  - **Interpreter**
  - **Iterator**
  - **Mediator**
  - **Observer**
  - **State**
  - **Strategy**
  - **Chain of Responsibility**
  - **Visitor**
  - **Template Method**

Advanced Process Control

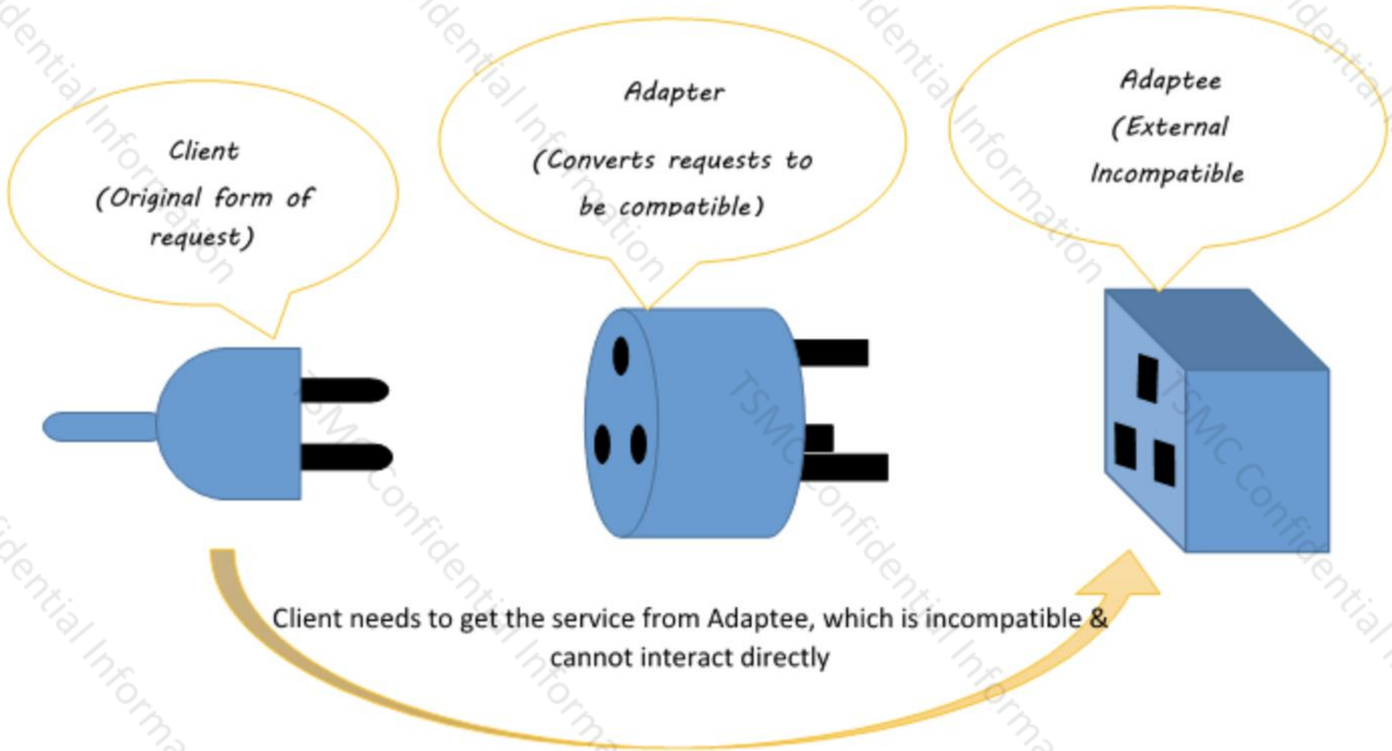
- 台積工廠內的機台運作以晶圓(Wafer)為最小單位, 在曝光機台(Process Tool)下貨前, 曝光機台會提供晶圓資訊(Wafer Info), 向 APC 系統詢問, 製程控制機制的調整值(Fn), 透過公式(a+b*Fn)以產生曝光下貨值(Sub Recipe), 提供給曝光機台下貨曝光用。
- 曝光後會進入量測機台, 產生量測結果, 以回饋給製程控制調整值(Fn), 使下次曝光更準確。





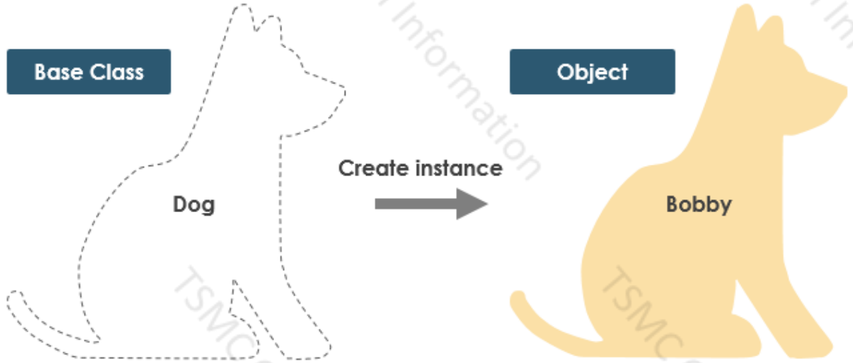
Pattern: Adapter

- *an object that fits another object into a given interface*



轉接頭~!!

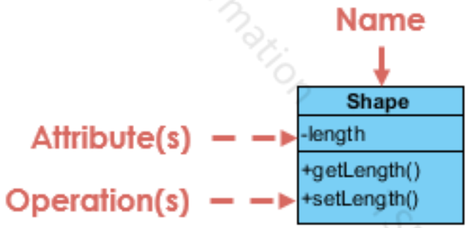
About UML



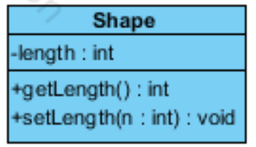
Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

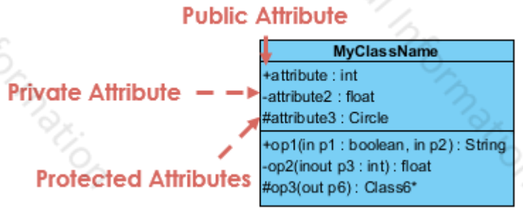
UML Class Notation



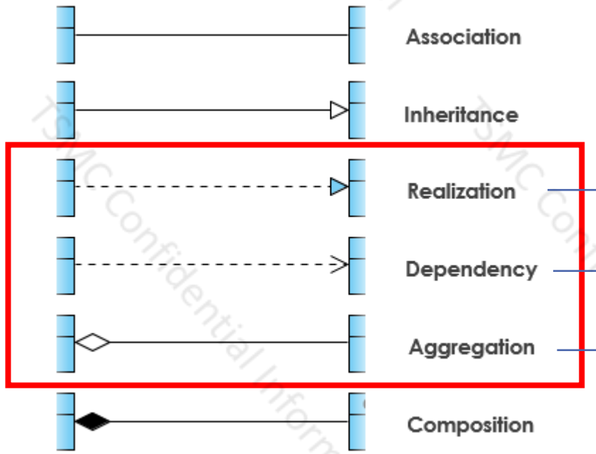
Class without signature



Class with signature



Class Visibility



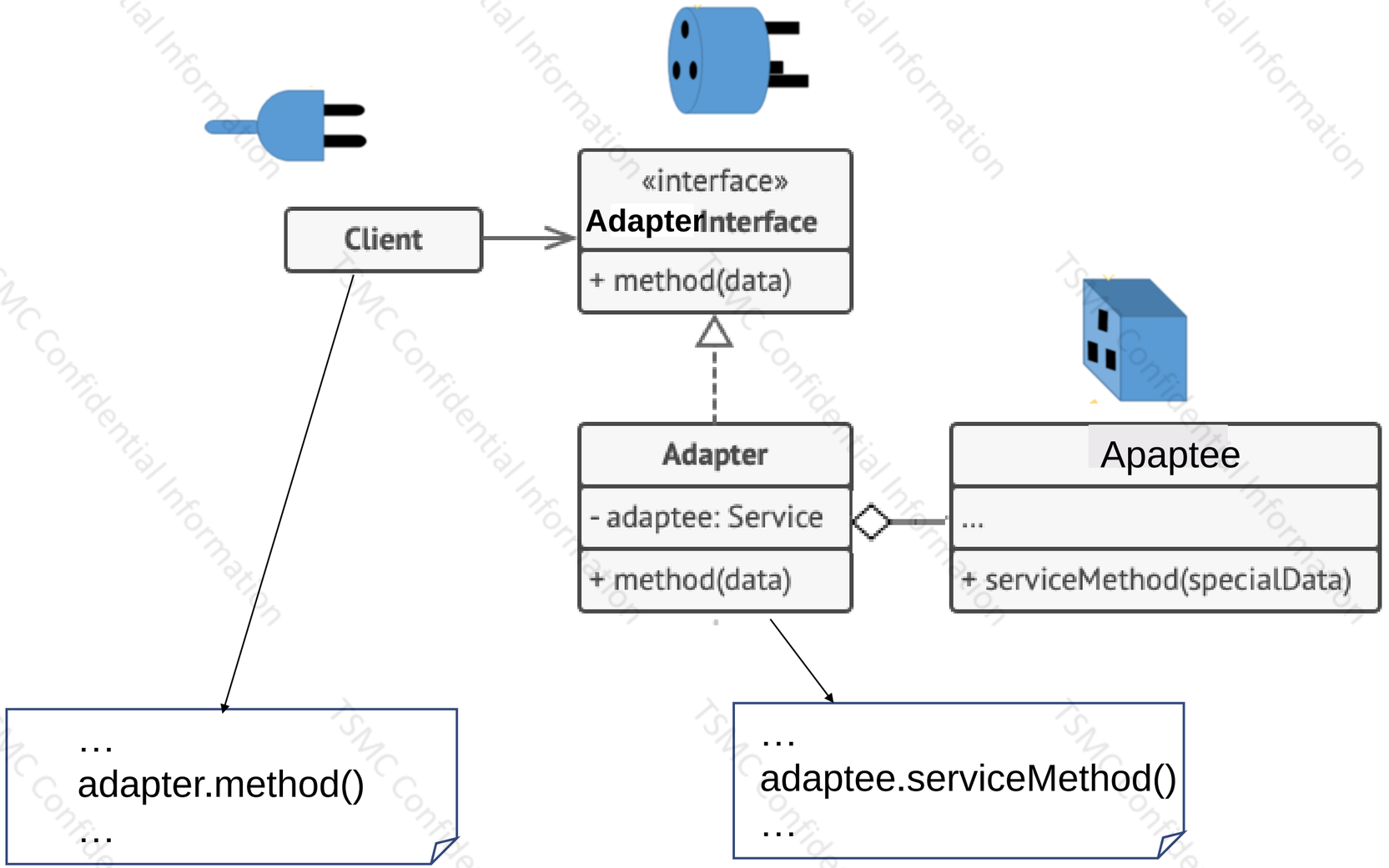
Class implements Interface

Use the object

Has the object

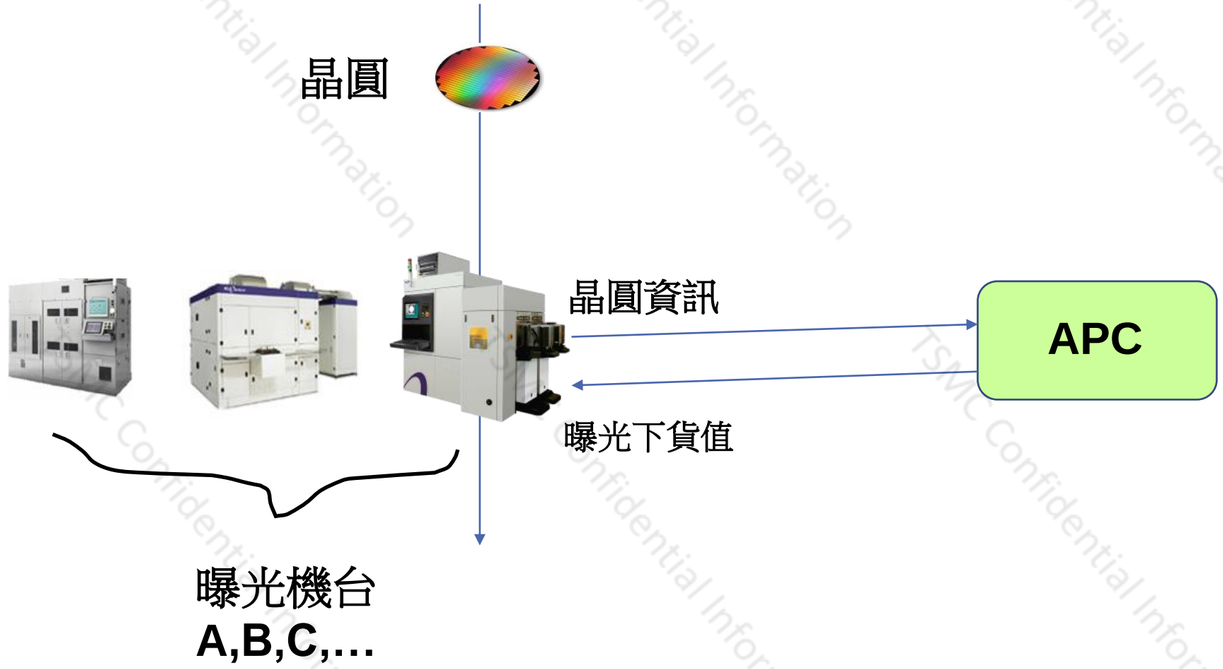
Relationships between classes

Sample code 請參照 /Example/Adaptor/*

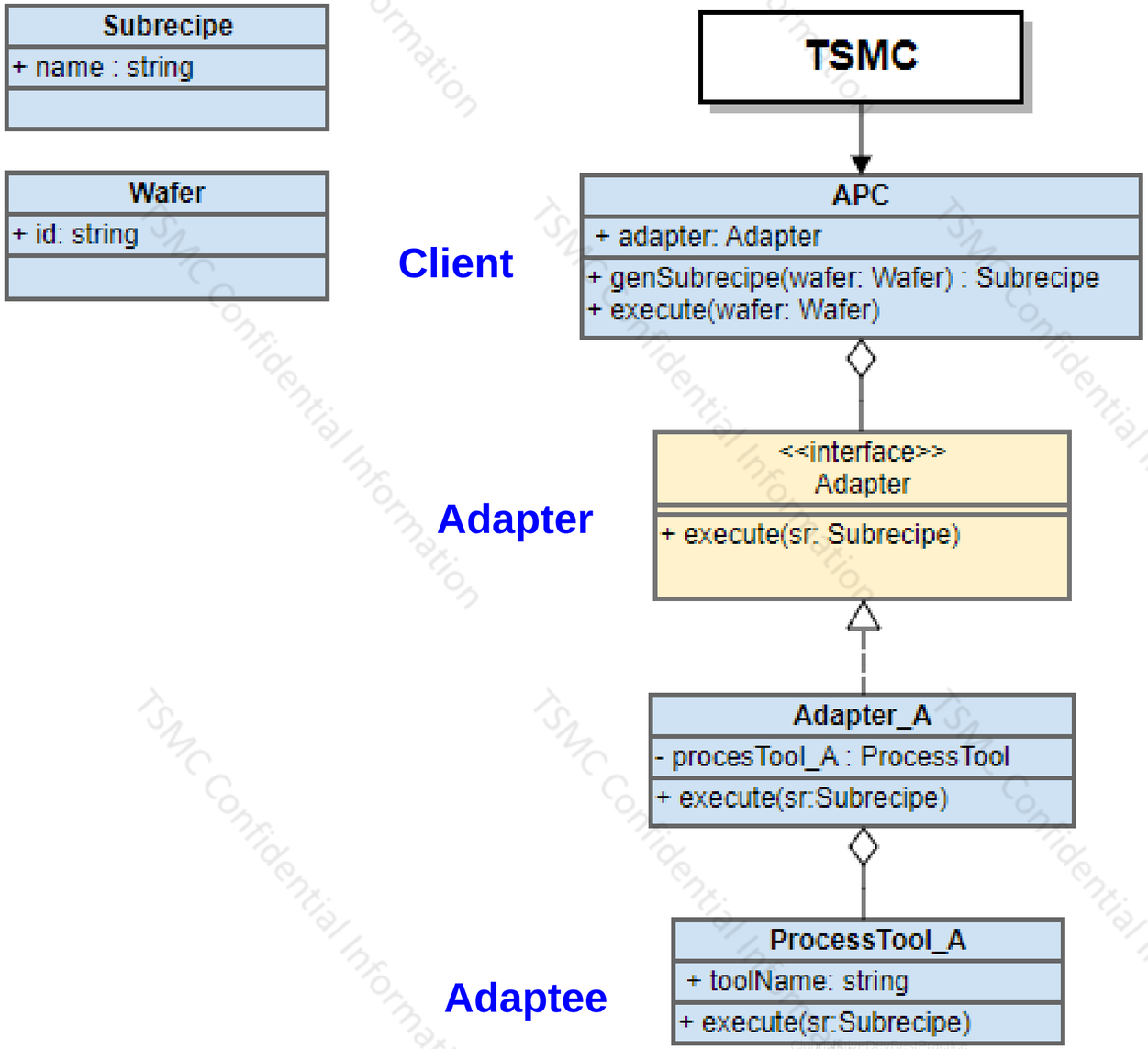


Adapter Pattern 課堂練習(1/2)

- 台積廠內 APC 只有一套製程控制機制, 但一個廠有上百個機台, 不同廠牌, 不同機型, 在機台操作使用的語法細節可能不同, 產生的曝光下貨值需針對不同機型轉換成各自的曝光工作。
- 如何維持APC系統一致，開發轉接頭機制以協助處理機型的差異，當新增機台或機台版本改變時, APC 系統可不需改變, 僅新增or修改轉接頭機制即可。
- [課堂練習] 請以 Adapter Pattern 撰寫簡單之針對不同曝光機型，執行各自的曝光工作之程式。



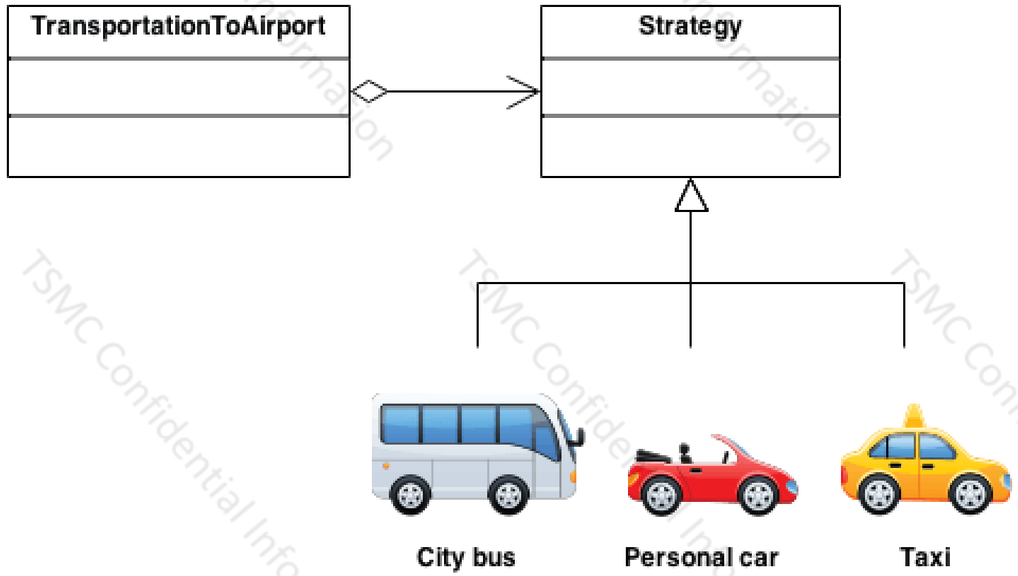
Adapter Pattern 課堂練習(2/2)

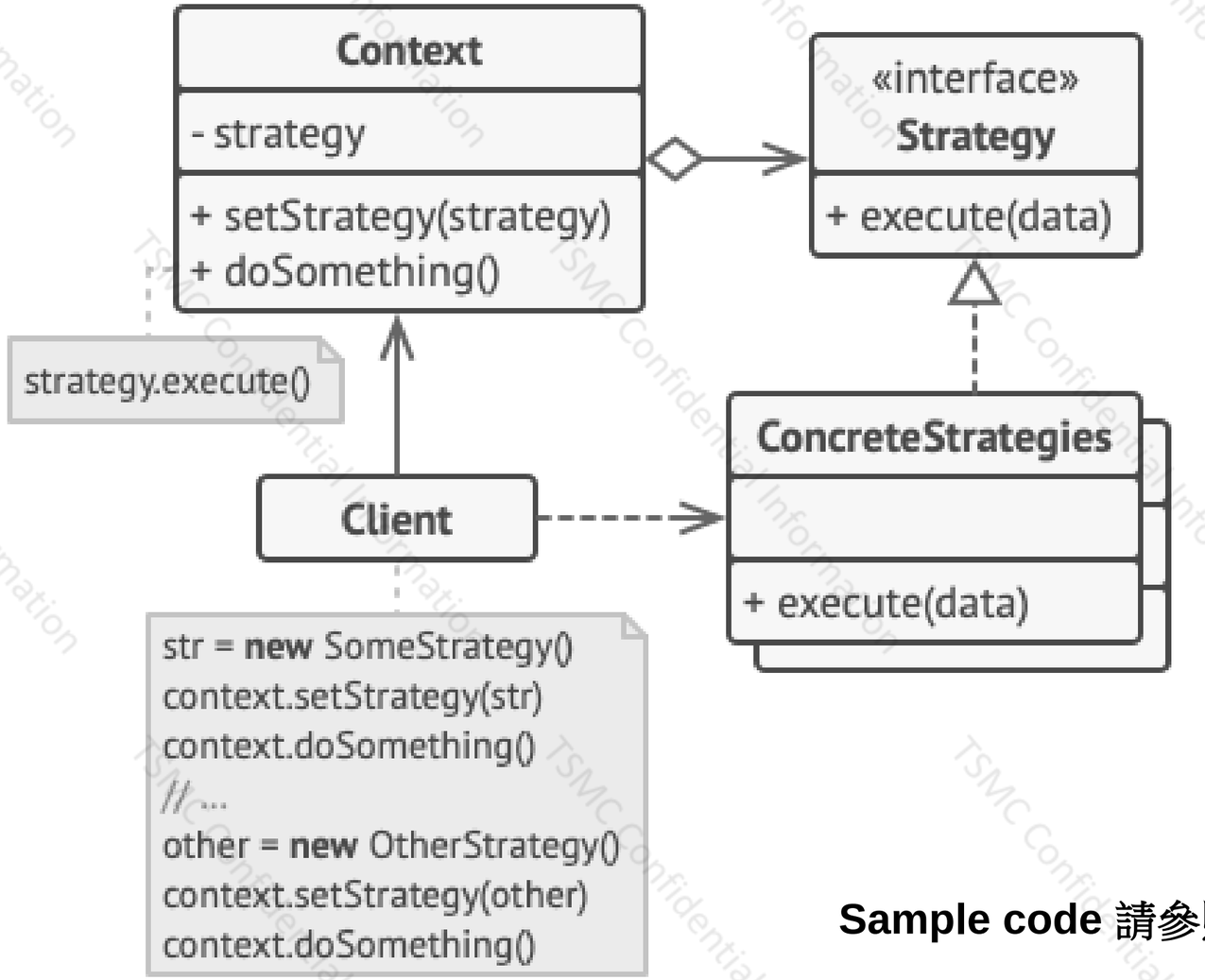


Sample code 請參照 /Practice/Adaptor/*
請新增 ProcessTool B,C 之 Adapter
並完成 APC之呼叫程式

Pattern: Strategy

- ***objects that hold alternate algorithms to solve a problem***

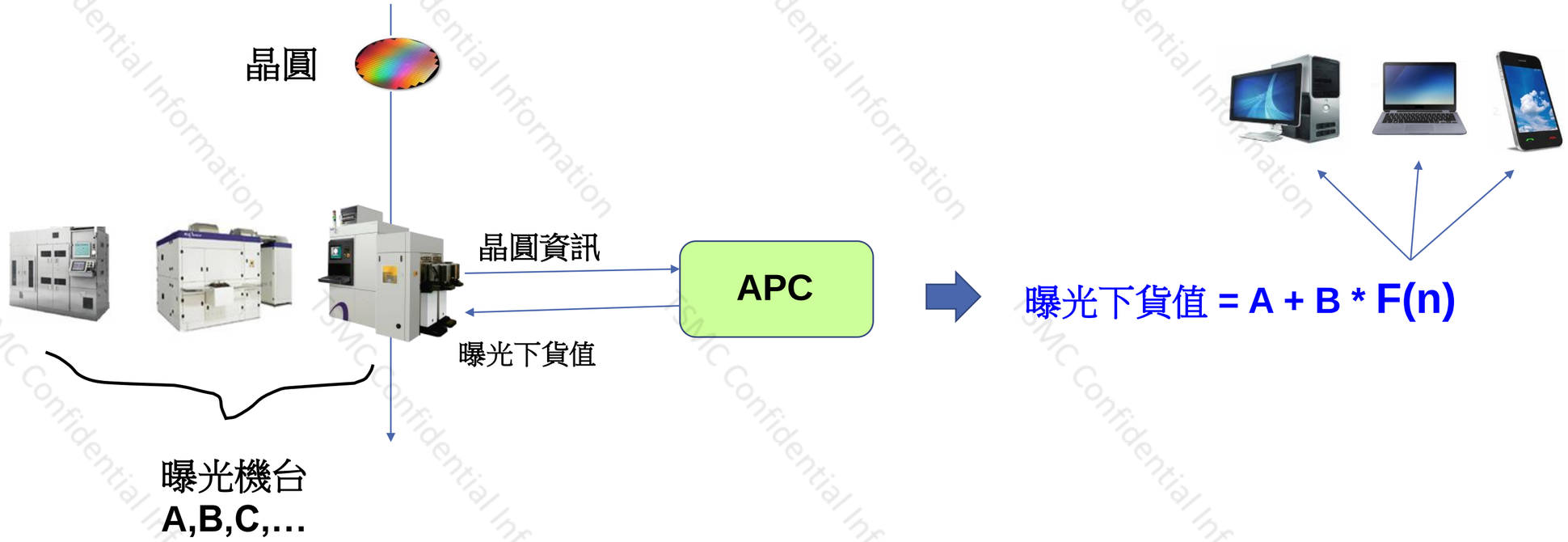




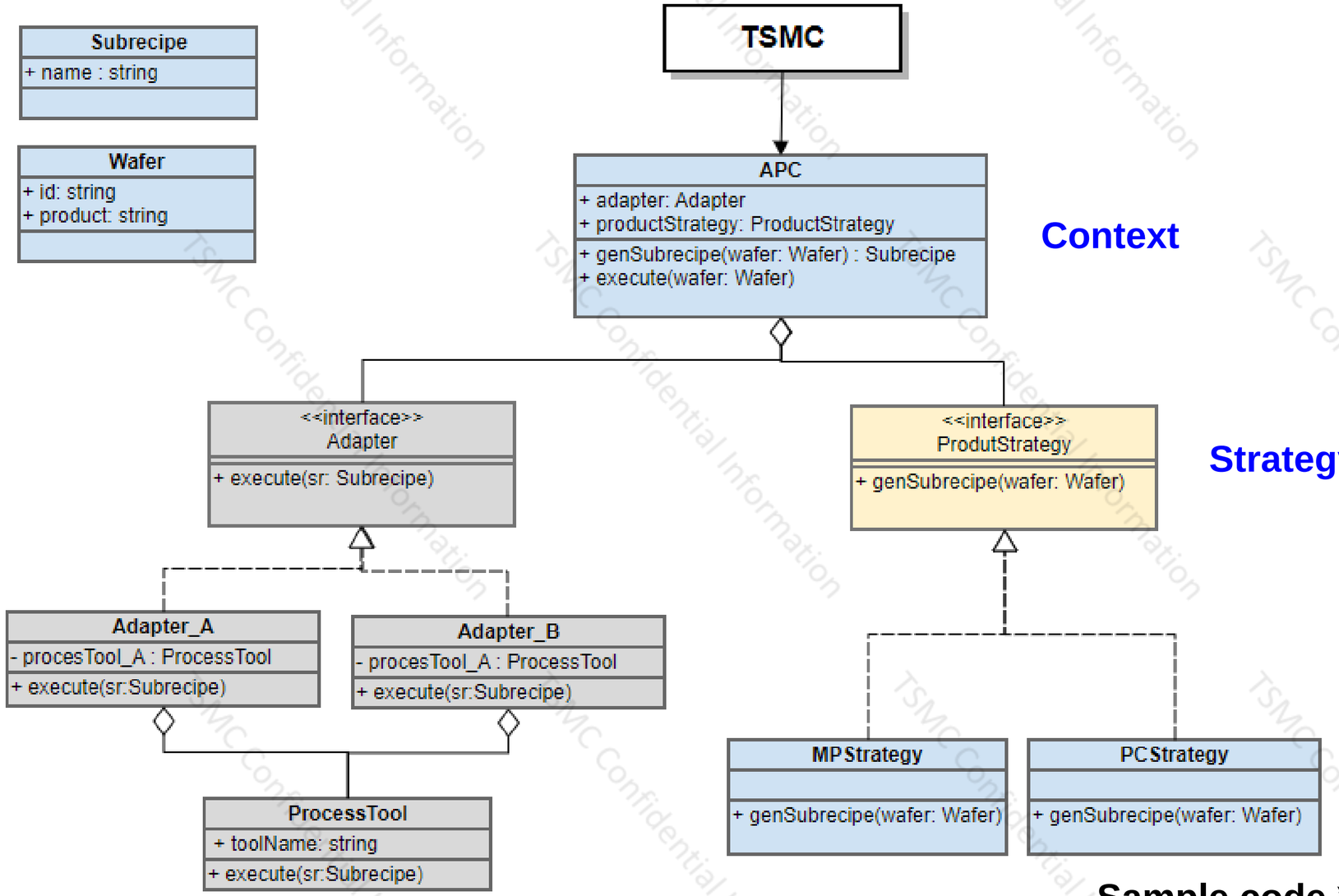
Sample code 請參照 /Example/Strategy/*

Strategy Pattern 課堂練習(1/2)

- 台積的機台，為追求產能最大利用化，同樣的機台必須可曝光不同產品。
- 假設APC產生曝光下貨值的公式為 A + B * F(n)，A and B 是固定值，F(n) 會隨產品種類不同而變動，假設現在有 手機(Mobile Phone)/電腦(Personal Computer)/筆電(Note Book) 等不同產品需要曝光。
- [課堂練習] 請延續上題，以 Strategy Pattern 撰寫簡單之產生曝光下貨值之程式。



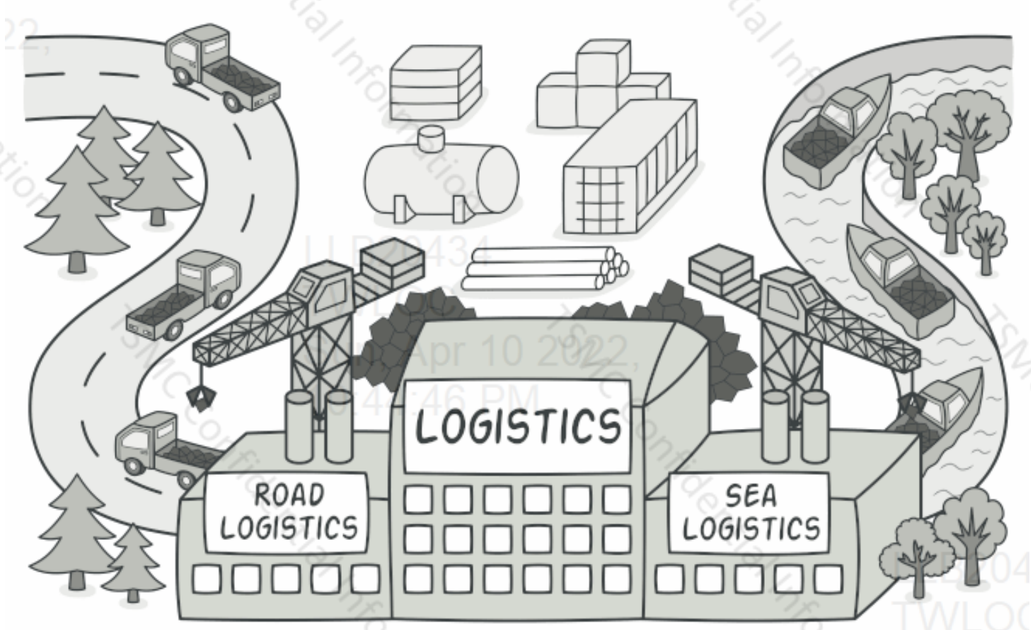
Strategy Pattern 課堂練習(2/2)

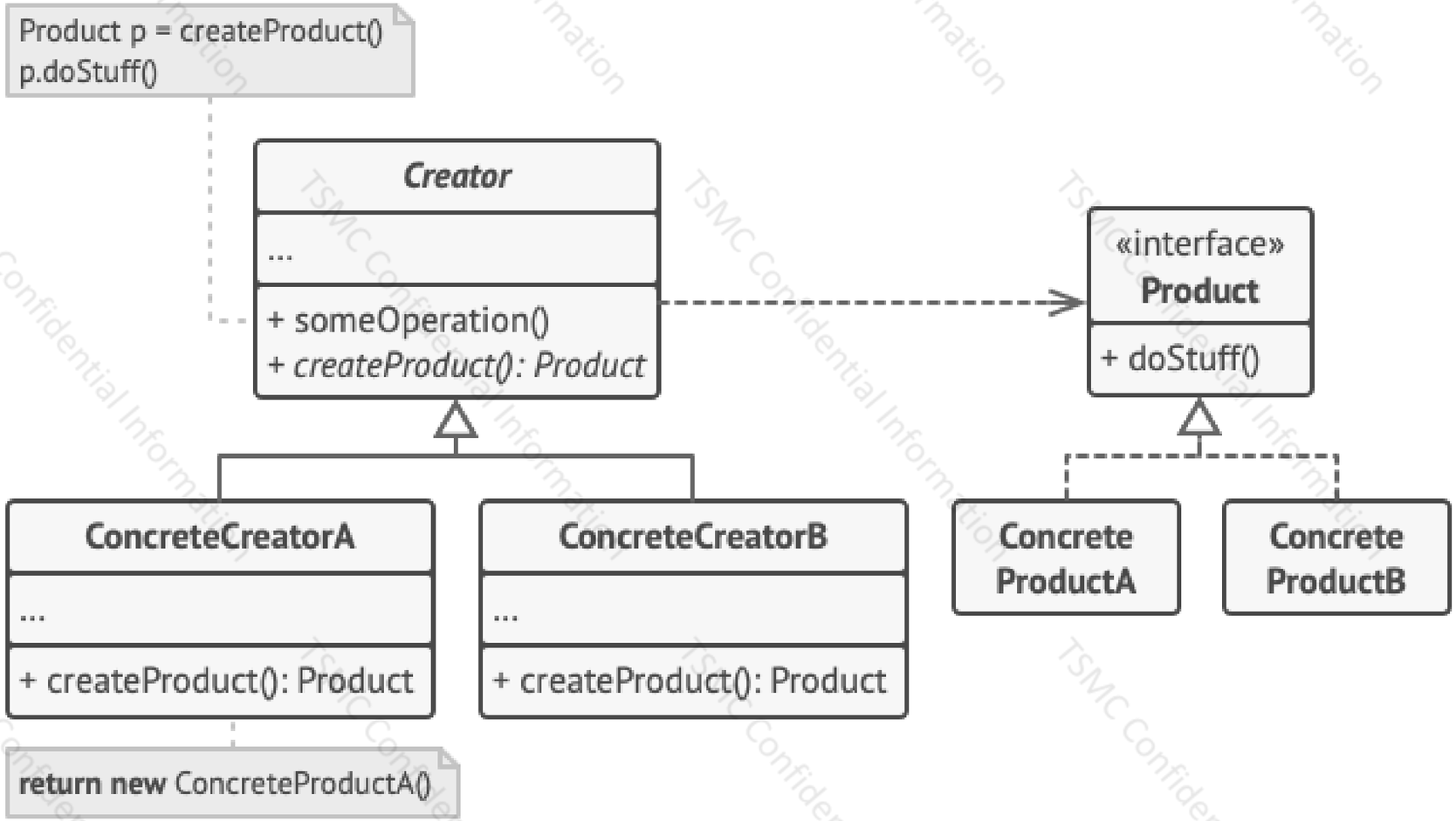


Sample code 請參照 /Practice/Strategy/*
請新增 NB Strategy 之程式
並完成 APC之呼叫程式

Pattern: Factory

- ***create objects without exposing the creation logic to the client***

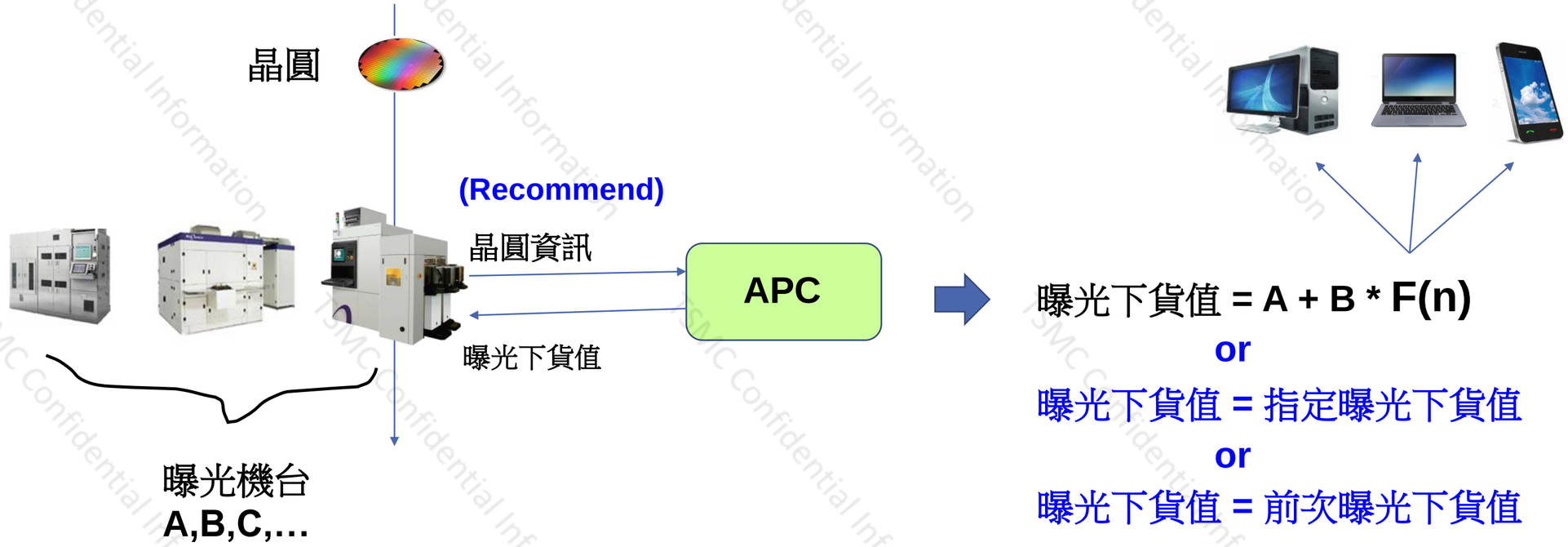




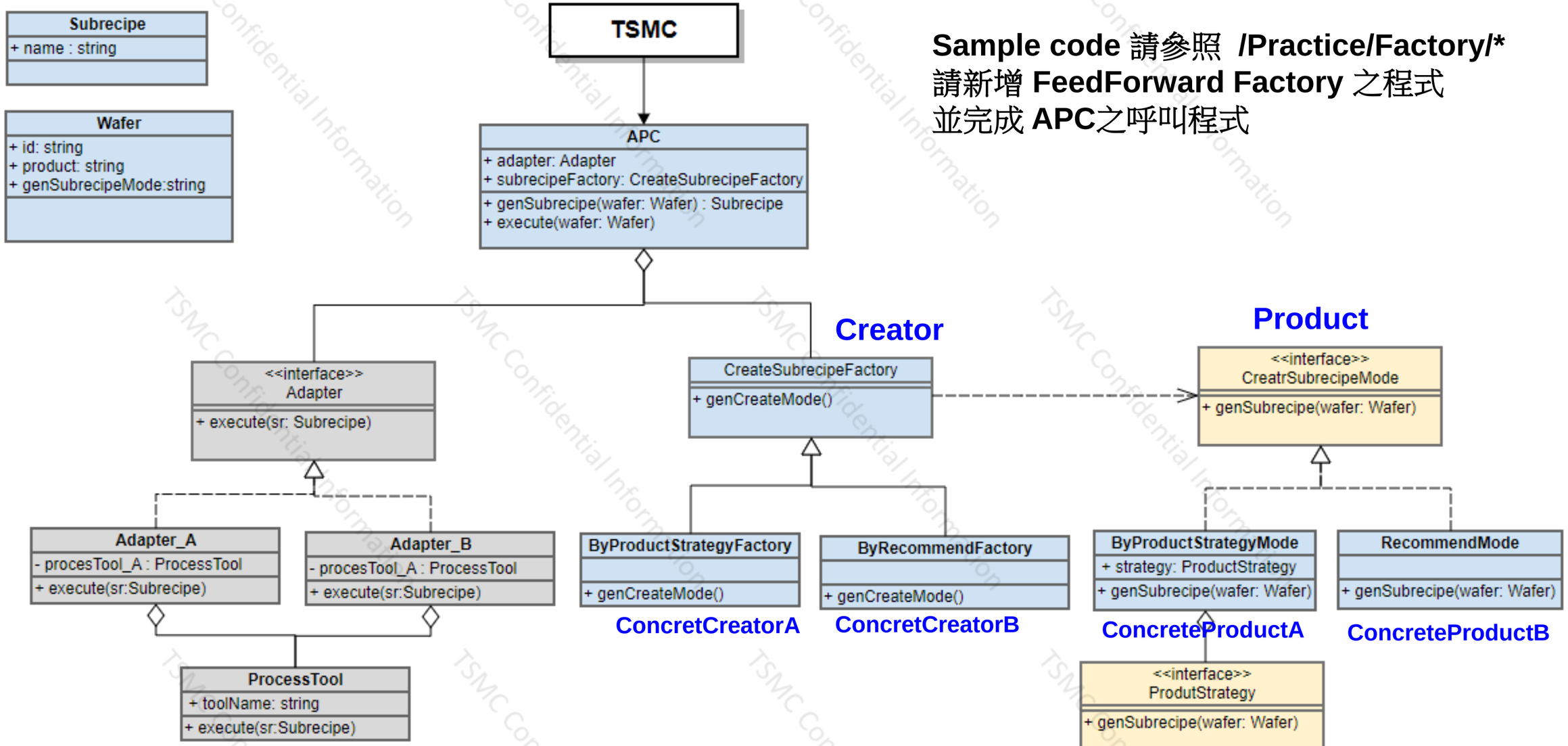
Sample code 請參照 /Example/Factory/*

Factory Pattern 課堂練習

- 產生曝光下貨值的邏輯，除了既有APC產生曝光下貨值的公式模式(A + B * F(n)) 之外，製程工程師可能因為該批Lot的特定問題，無法用一般的下貨模式產生曝光下貨值，而必須透過手動指定曝光下貨值(Recommend)之指定曝光值模式(Recommend)，或是直接照前次下貨值的參照前次模式(Feedforward)。
- [課堂練習] 請修改原程式, 在晶圓資訊中新增指定下貨值之資訊，並以Factory Pattern 撰寫指定曝光下貨值下貨值之程式。

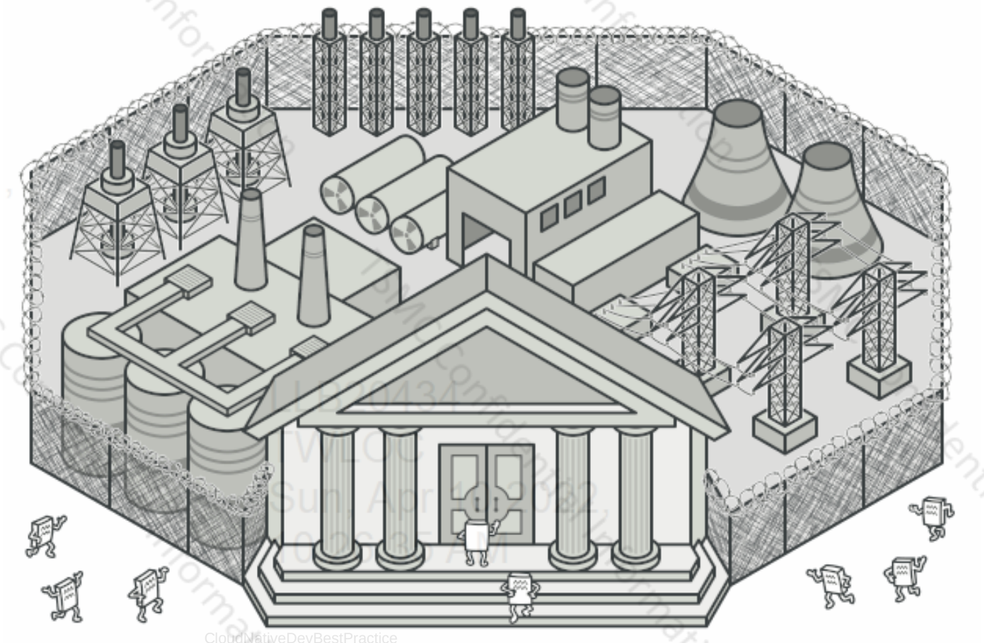


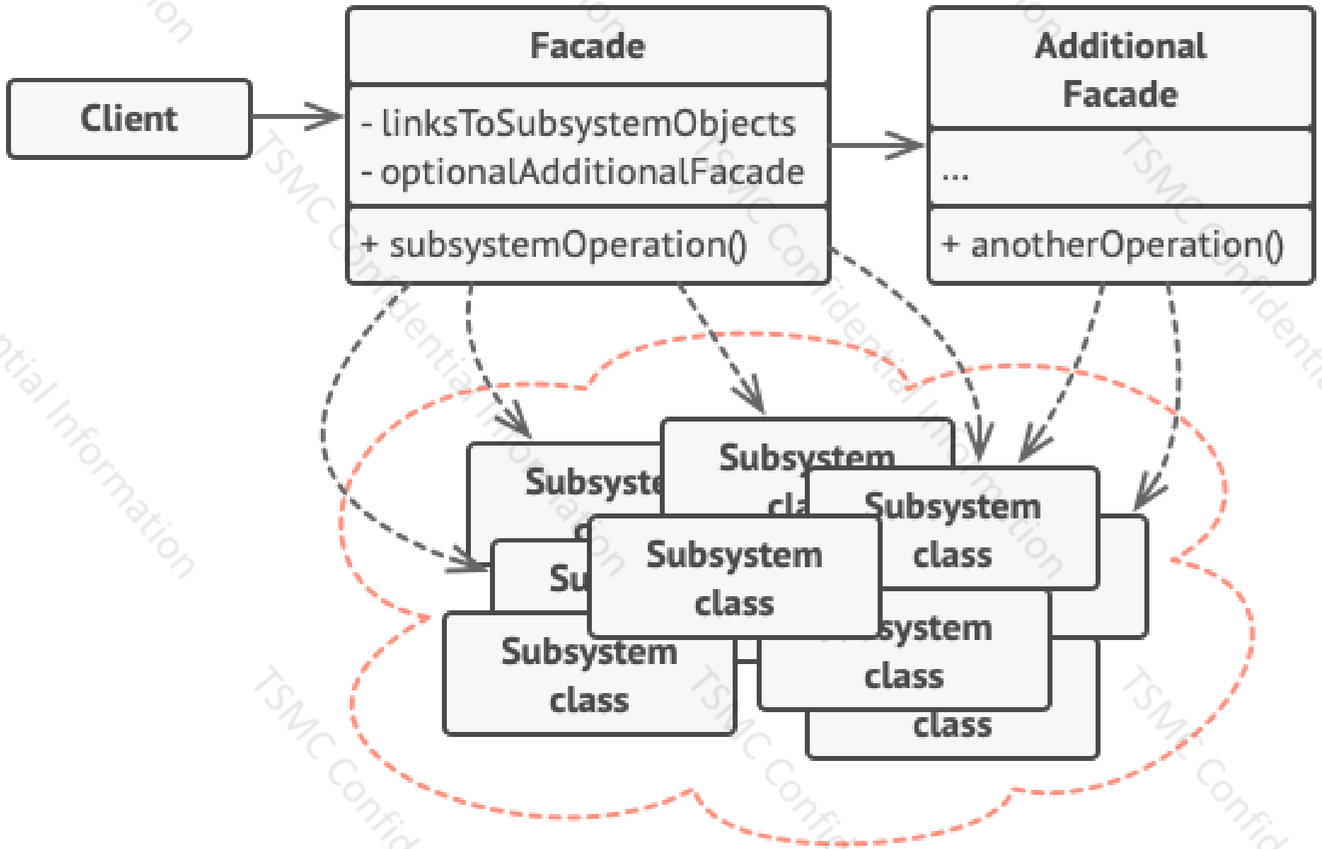
Sample code 請參照 /Practice/Factory/*
請新增 FeedForward Factory 之程式
並完成 APC之呼叫程式



Pattern: Facade

- Object *that* provides a simplified interface to a library, a framework, or any other complex set of classes.

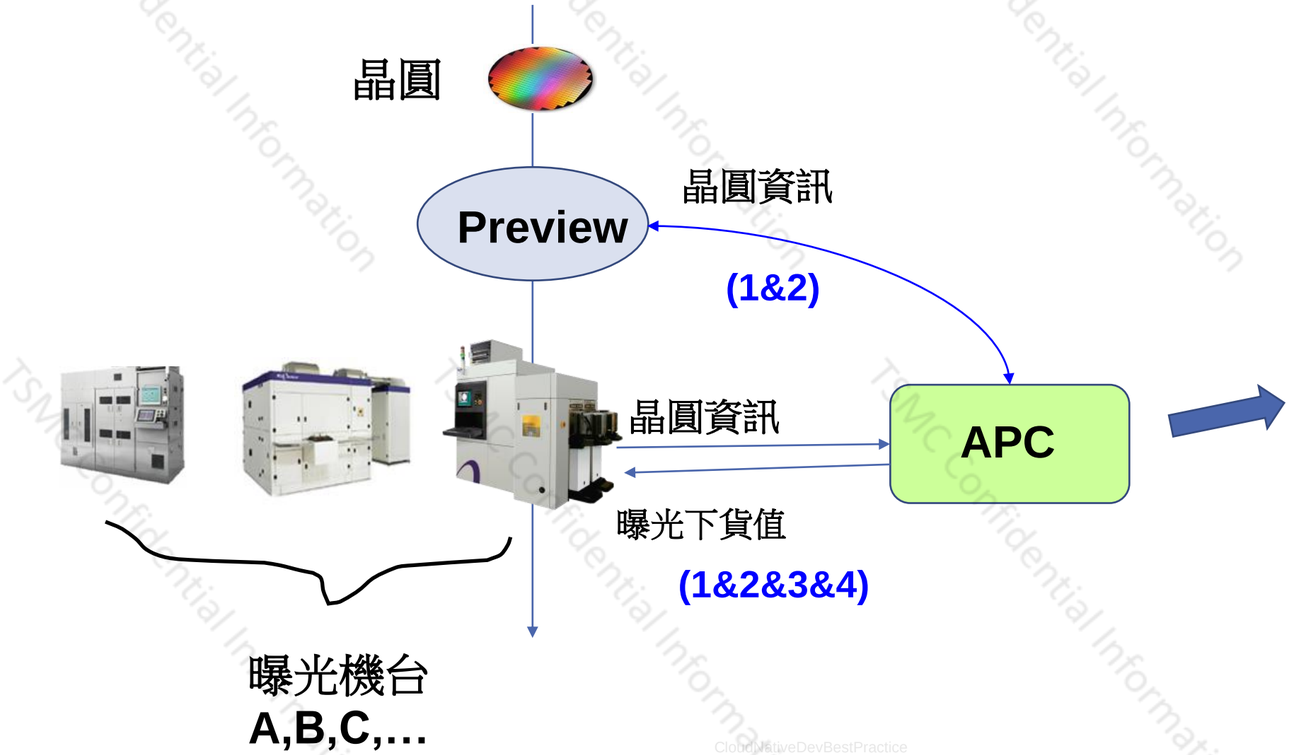




Sample code 請參照 /Example/Facade/*

Facade Pattern 課堂練習

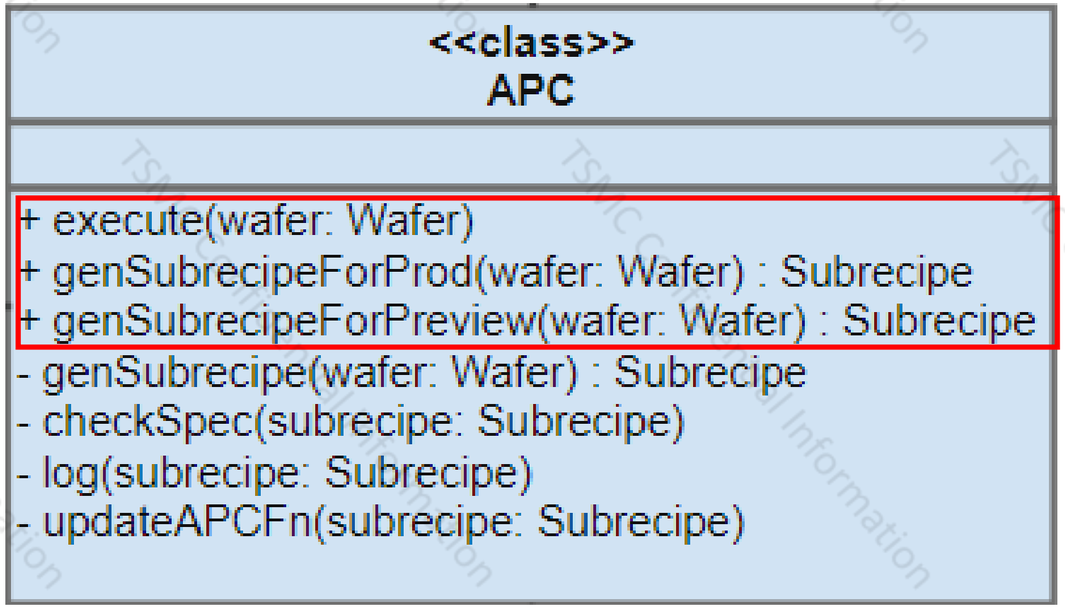
- APC 系統除了產生曝光下貨值(genSubrecipe)，還會作檢查下貨值是否超出規格(checkSpec)，留下下貨記錄(log)，更新製程控制調整值(updateAPCFn)等動作。
- APC 產生曝光下貨值過程可能會失敗，因此工廠為了避免晶圓因為曝光下貨值無法產生而卡貨在機台，會在下貨前提前詢問APC 該批貨是否可產生曝光下貨值，此動作稱為 Preview，但 Preview 行為，並未真正下貨，因此只會執行產生曝光下貨值(genSubrecipe) ，檢查下貨值是否超出規格(checkSpec)兩個動作。
- [課堂練習] 請以 Facade Pattern，將 APC 行為包裝出 façade interface 。



1. 產生曝光下貨值

SR = a + b*f(n)
or
SR = 指定SR
or
SR = 前次SR

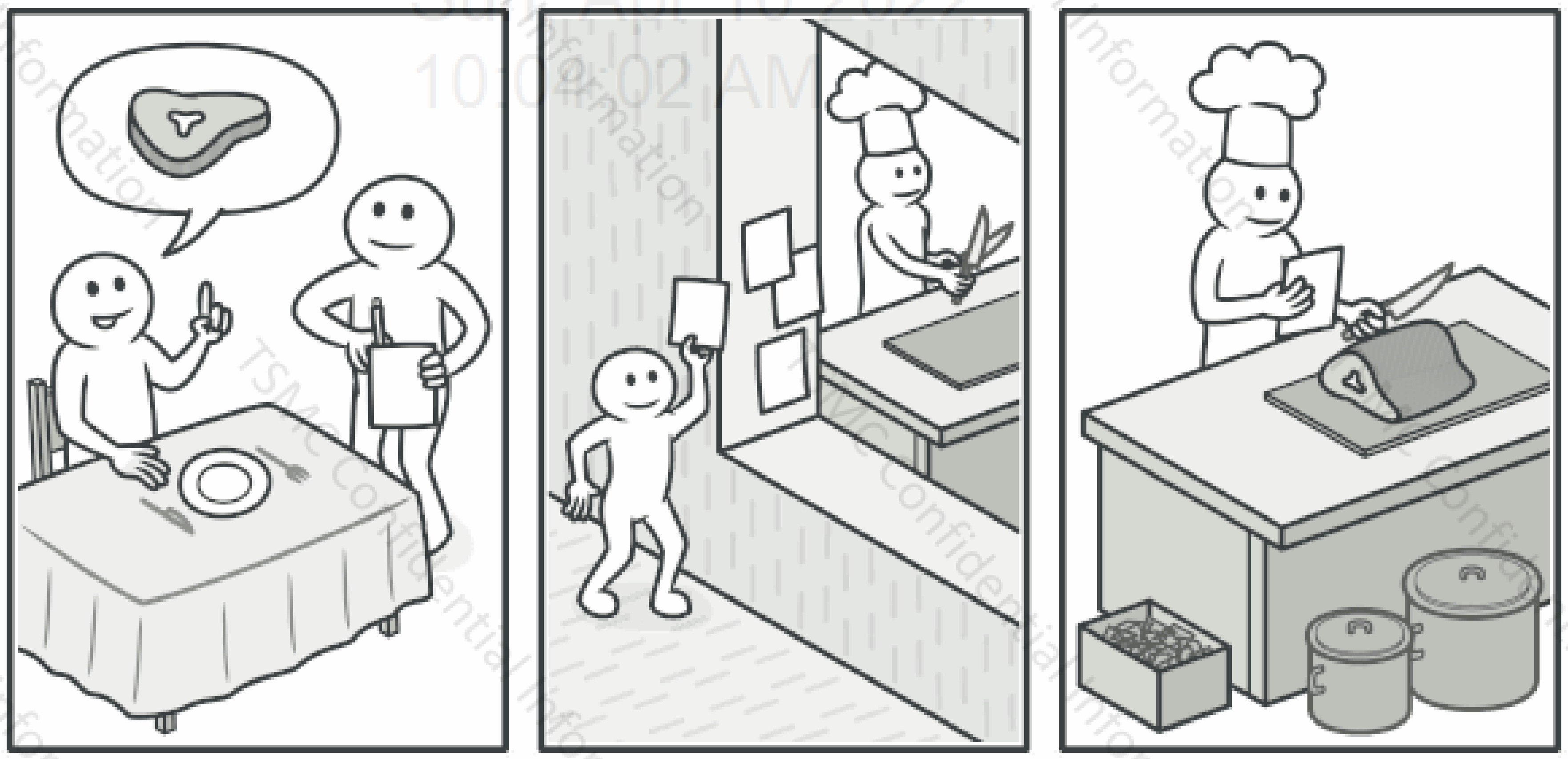
2. 檢查曝光下貨值是否超出規格
3. 留下下貨記錄
4. 更新製程控制調整值



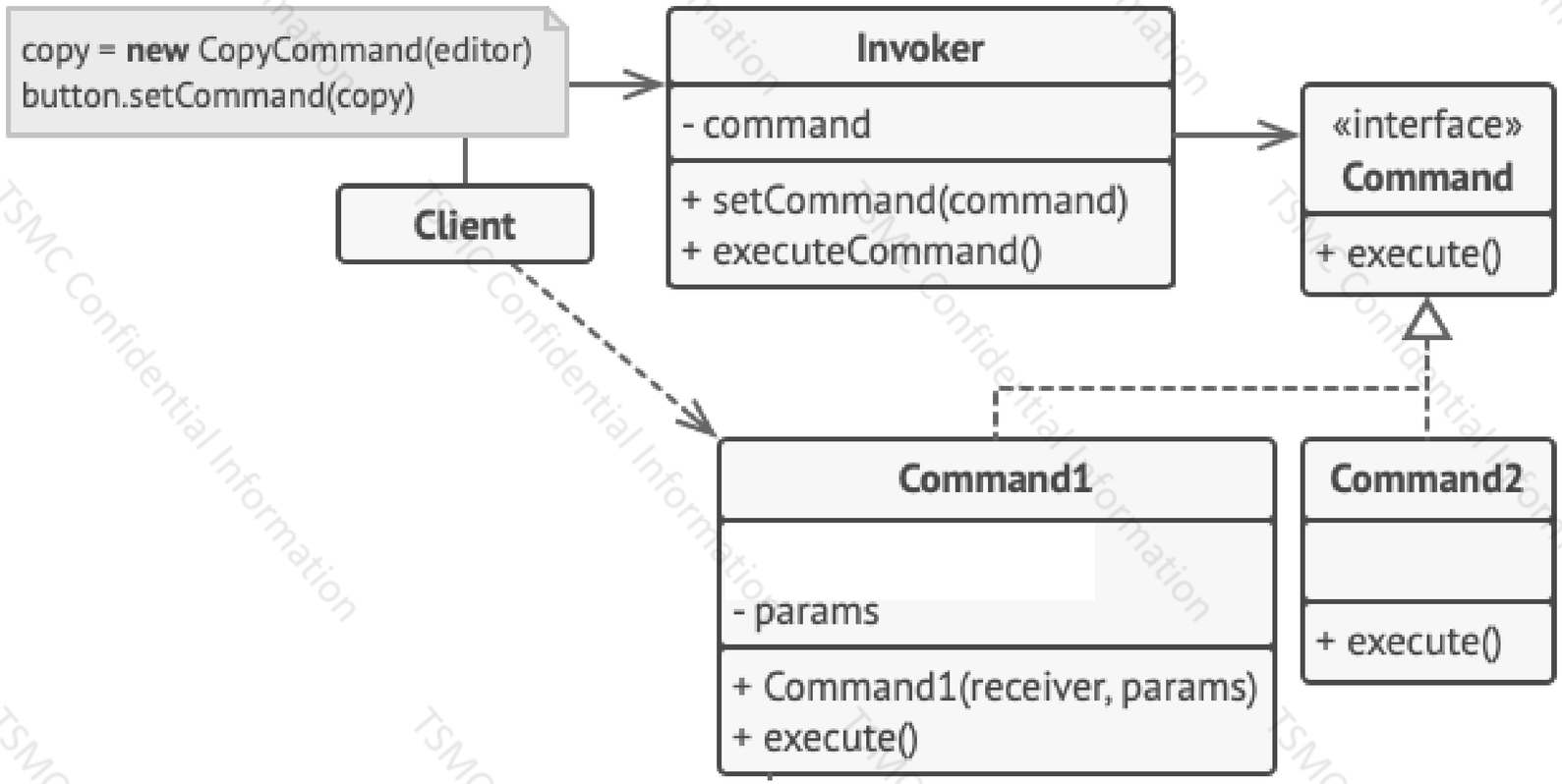
Sample code 請參照 /Practice/Facade/*
請新增 APC function *3
並完成 APC之呼叫程式

Pattern: Command

- ***object is used to encapsulate all information needed to perform an action or trigger an event at a later time***



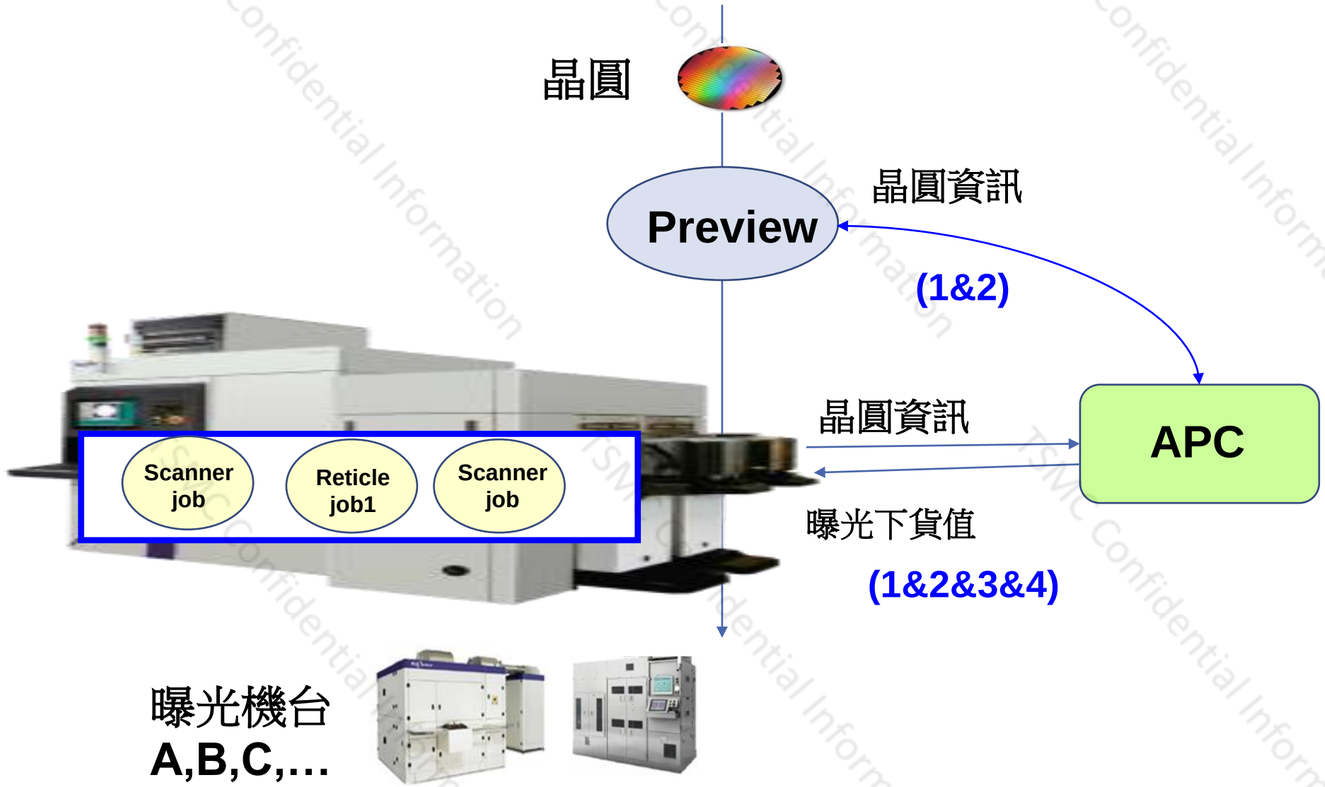
Making an order in a restaurant.



Sample code 請參照 /Example/Command/*

Command Pattern 課堂練習

- 機台一天必須處理大量晶圓之曝光下貨，且除曝光動作(ScannerJob)外，可能會有各種系統會對機台下不同操作命令, ex. 光罩操作(ReticleJob)的 退光罩(Check-Out)，進光罩(Check-In)，機台必須獨立管理所有動作，並照順序確實完成，且要能因應隨時有緊急貨必須插隊之處理。
- [課堂練習] 請延續上題，以 Command Pattern，撰寫簡單之產生曝光下貨值之程式，需至少包含2個scanner job以及2個 reticle job。



1. 產生曝光下貨值

SR = a + b*f(n)

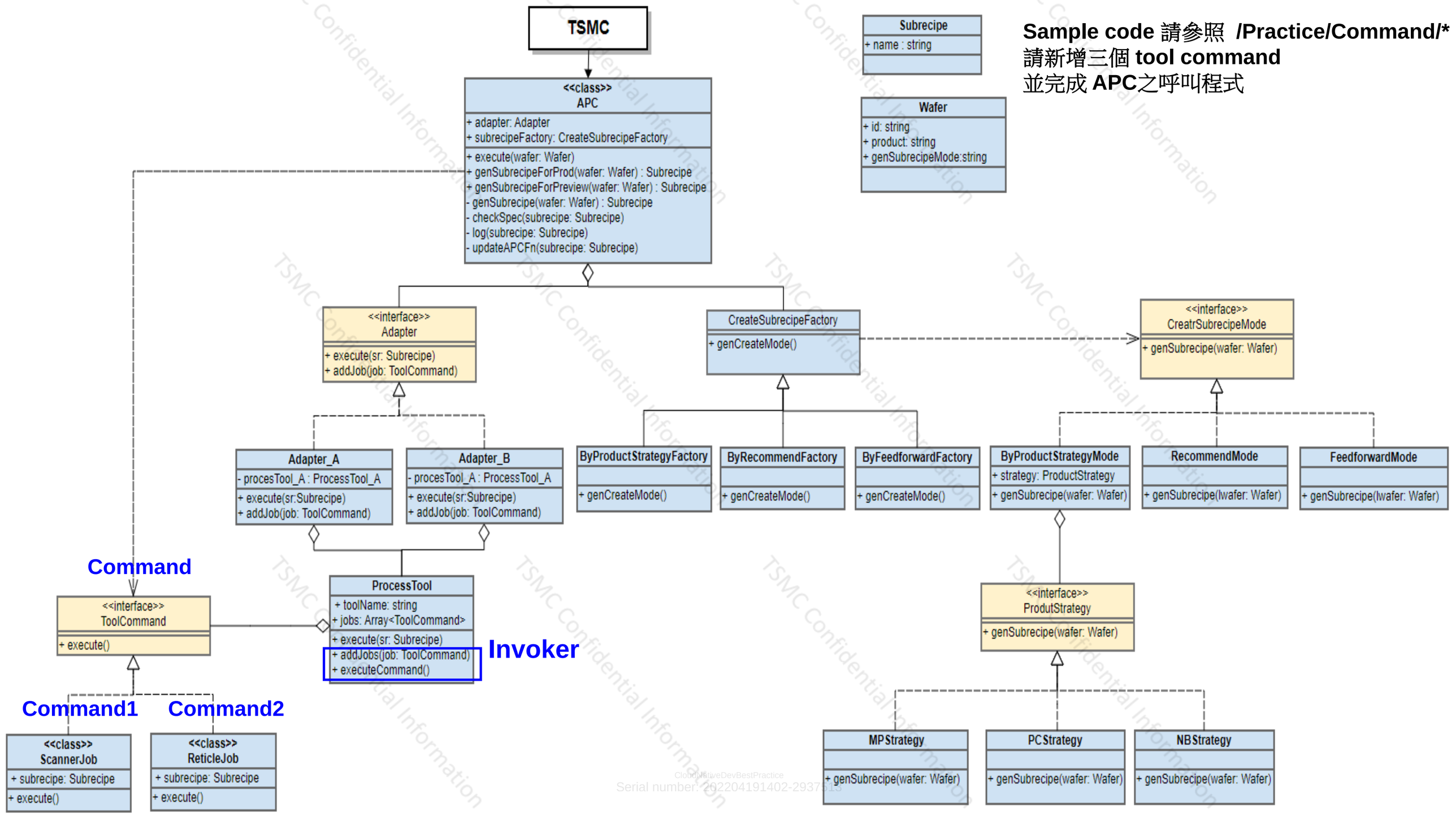
or

SR = 指定SR

or

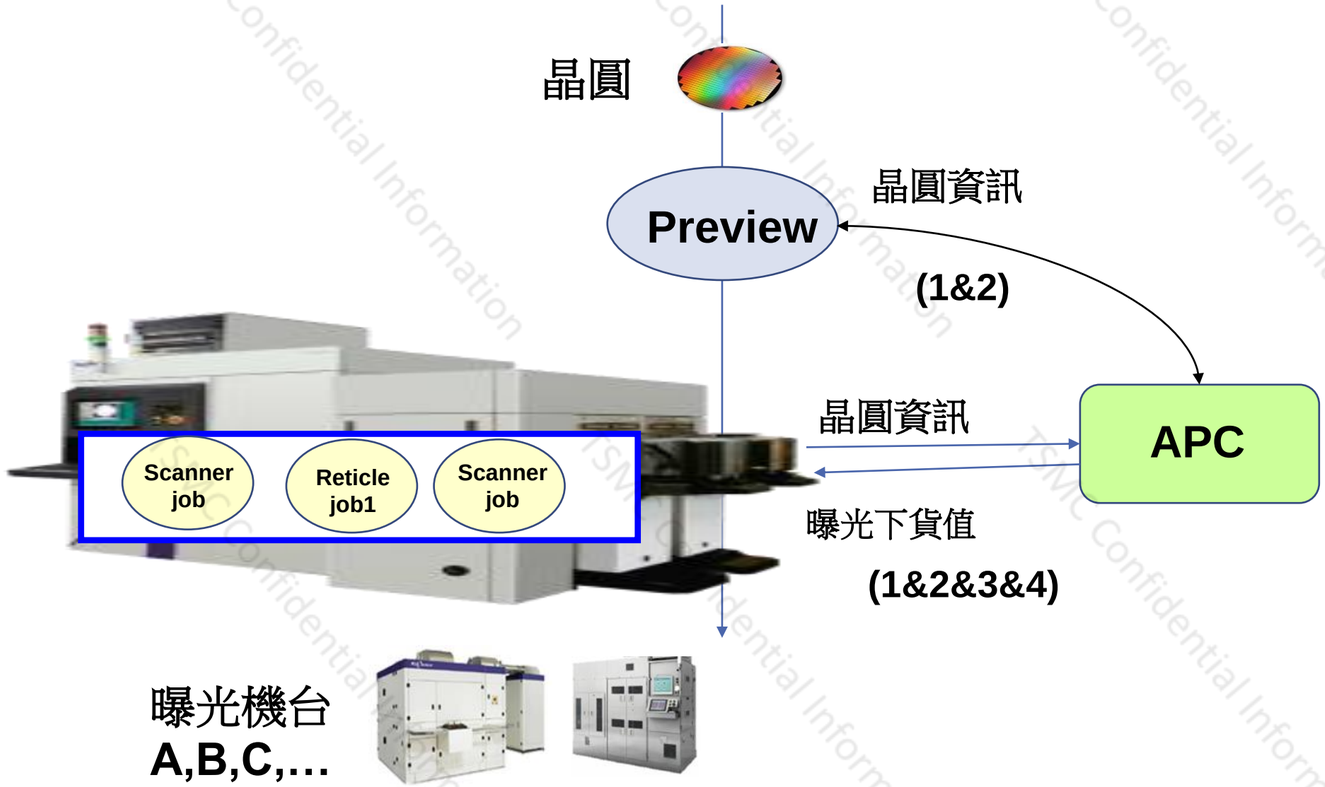
SR = 前次SR

2. 檢查曝光下貨值是否超出規格
3. 留下下貨記錄
4. 更新製程控制調整值

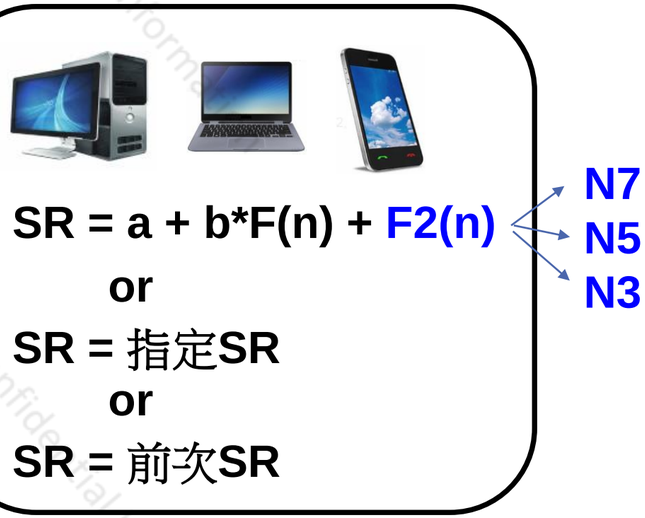


課後作業

- 台積每隔幾年會進入一個新的技術(Technology)世代 (N7→N5→N3→…), 不同世代對產品的製程控制機制調整值F2(n)都會不同, 假設公式為 SR = a + b*F(n) + c*F2(n)
- [課堂練習] 請延續課堂練習，新增 TechStrategy (StrategyPattern) 以完成APC呼叫。



1. 產生曝光下貨值

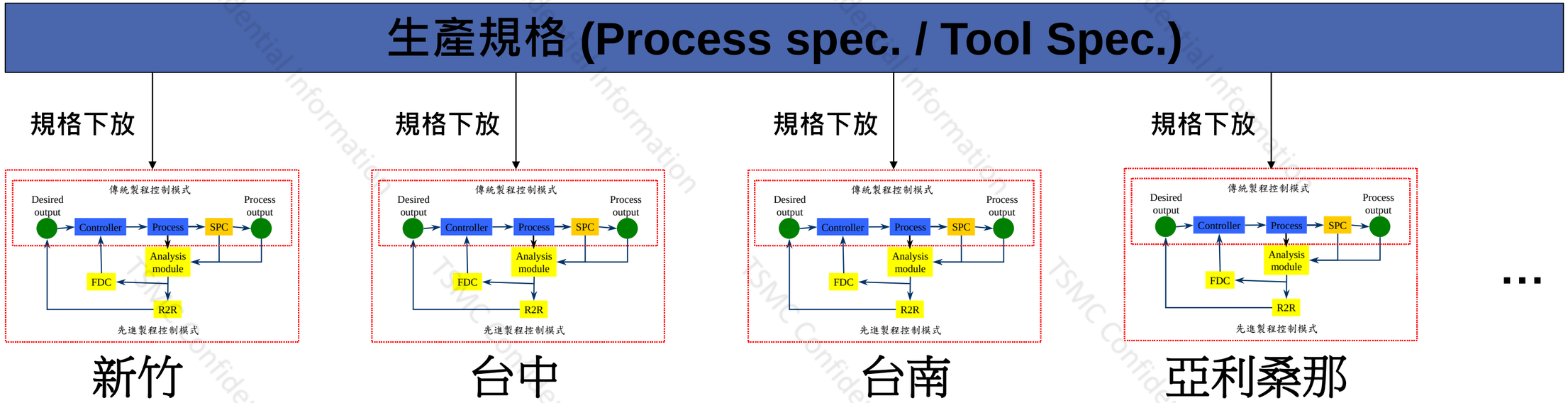


2. 檢查曝光下貨值是否超出規格
3. 留下下貨記錄
4. 更新製程控制調整值

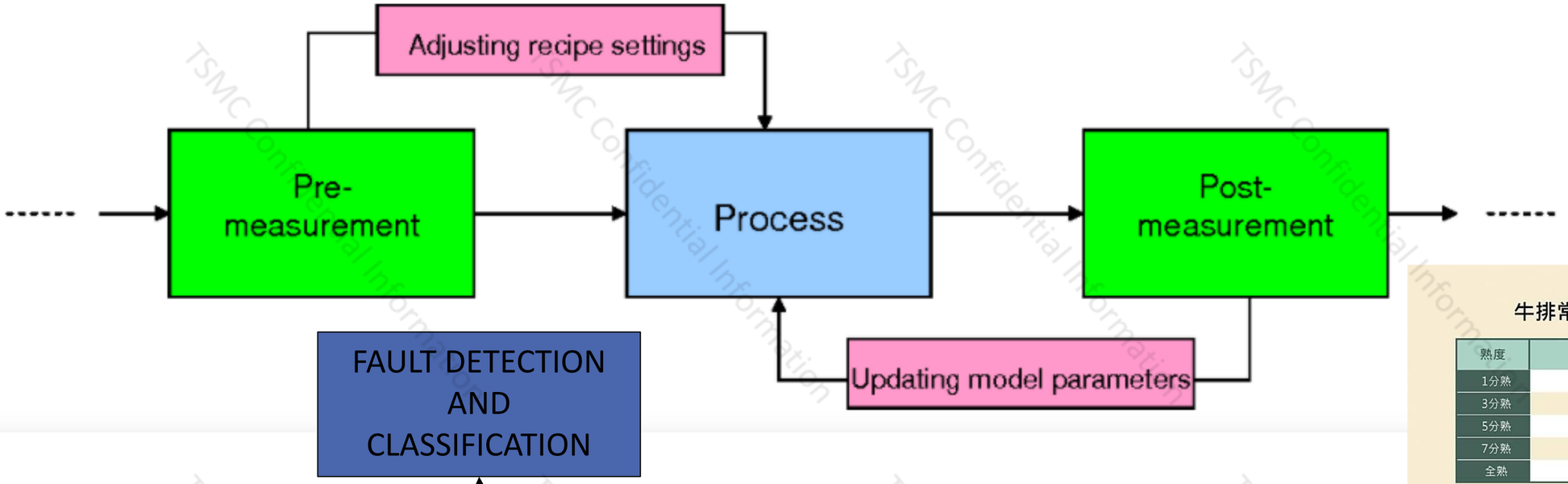
期末實作 – Smart Manufacturing

課堂實作 – Smart Manufacturing

- 為了達到不同廠區生產出來的產品規格一致, 產品規格由中央訂定, 規格下放到各個獨立生產的廠區
- 廠區的生產穩定由APC (Advanced Process Control) 控管 (不同產品的Spec. 在各個不同獨立廠區都相同)
- 機台是否正常由FDC (Fault Detection and Classification) 管控 (機台 sensor 每秒報出機台溫度, 溫度的上下限也由中央控管)



課堂實作 – Smart Manufacturing



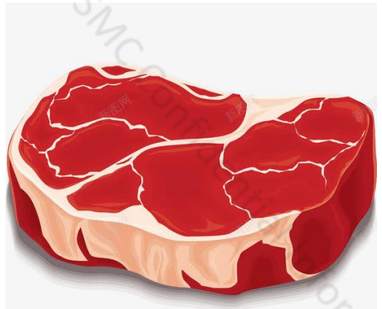
食力 foodNEXT

牛排常見的熟度判斷基準

熟度	切面顏色	中心溫度
1分熟	90%血紅色	45℃
3分熟	60%血紅色	50℃
5分熟	30%血紅色	55℃
7分熟	粉紅色	65℃
全熟	灰色	70℃

內容來源＝《專業肉舖的牛肉料理教本》，幸福文化出版

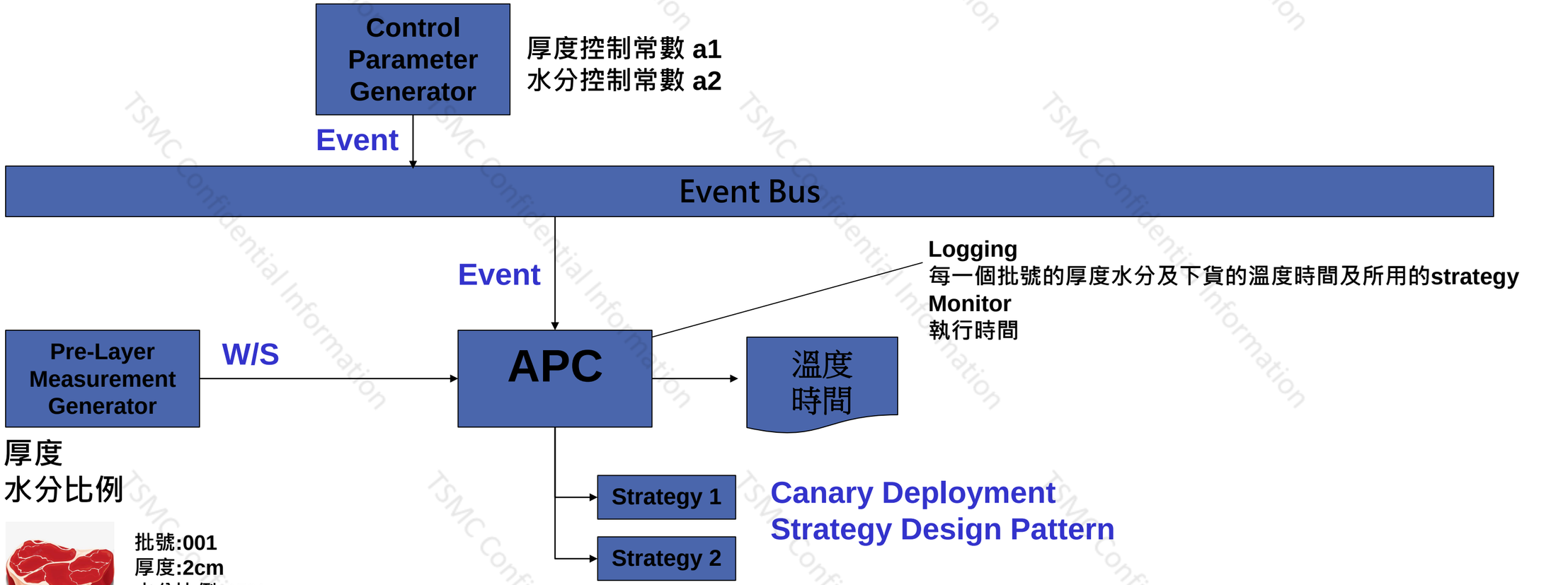
水分比例
厚度



溫度
時間

核心溫度

課堂實作 – Smart Manufacturing Reference Architecture



批號:001
厚度:2cm
水分比例:65%

批號:002
厚度:2.2cm
水分比例:67%

APC 控制Strategy
Strategy 1 : 以水份控溫度,時間固定 (20min), 溫度=厚度(cm) * 厚度控制常數-a1 (52.1) + 模型常數 (2)
Strategy 2 : 以厚度控制時間, 溫度固定 (100度C), 時間 = 水分(%) * 水分控制常數- a2(1.66) + 模型常數(3)
70%由Strategy 1控制
30% 由Strategy 2 控制

課堂實作 – Smart Manufacturing

Programming and Runtime

- Programming language不限, 但prefer Java/Javascript/Go/Python, 儘量不要使用C/C++(相關的套件不容易找)
- Deploy on GKE (Google Kubernetes Engine).

期末專題Evaluation Criteria

- Application meet requirement – 20%
- Application Architecture – 15%
- Code Contribution – 15%
- Testing Coverage – 15%
- Code Healthy Metric – 10%
- Deployment (Canary, High Availability) – 15%
- Application Monitor – 10%

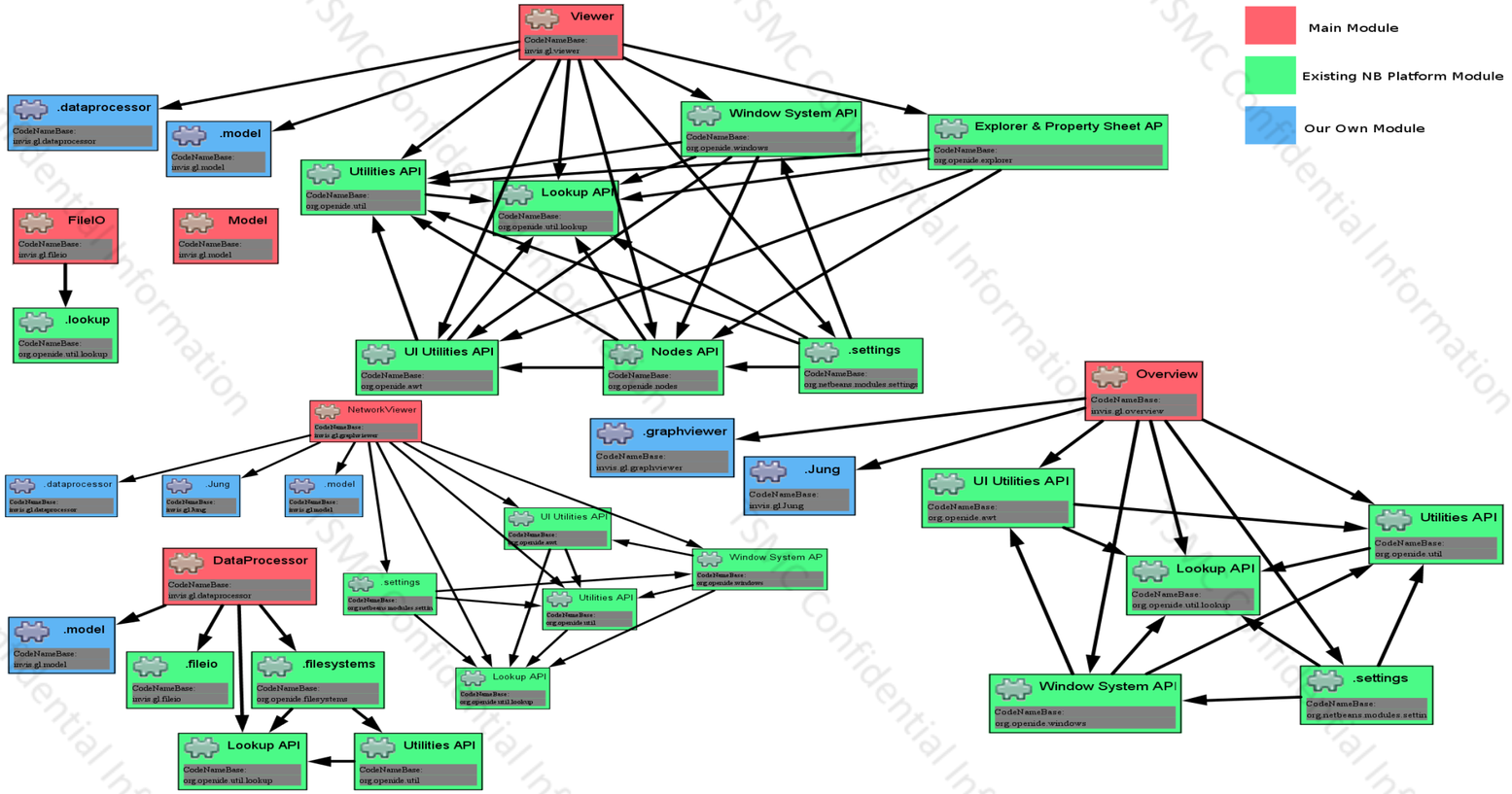
Backup Material

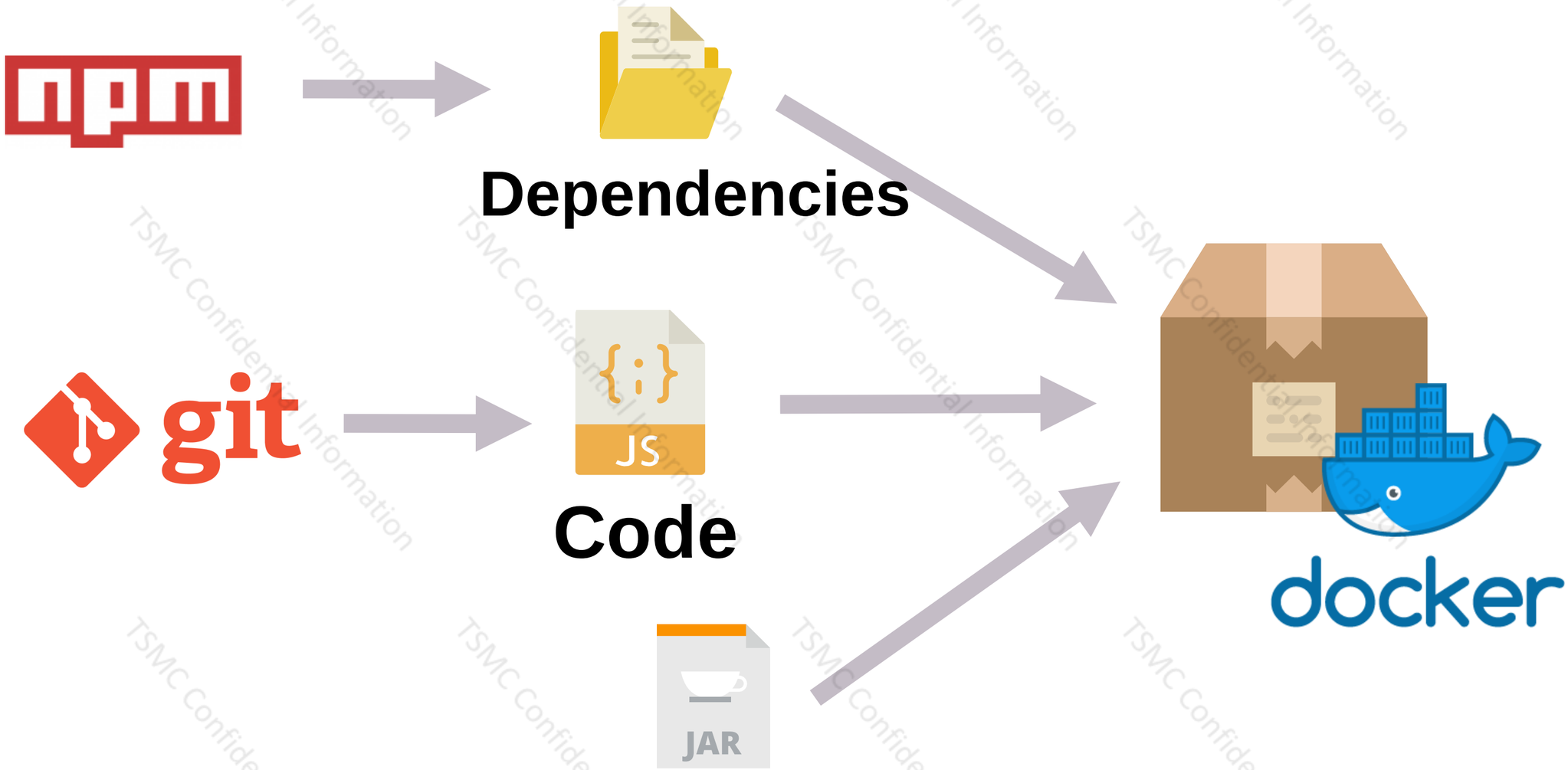
2 - Dependencies (What does it mean?)

- Use a package manager to avoid dependency hell.
- Don't commit dependencies in the codebase repository.

2 - Dependencies

Explicitly declare and isolate dependencies





Dependencies

Code

Binaries

Application Bundle

Dependency Declaration: Node.js

`package.json`

```
{
  "dependencies": {
    "async": "2.1.4",
    "express": "4.16.2",
    "express-bearer-token": "2.1.0",
    "body-parser": "1.18.2",
    "jwt-simple": "0.5.1",
    "lodash": "4.17.4",
    "morgan": "1.7.0",
    "request": "2.81.0"
  }
}
```

`npm install`

Dependency Declaration: Python

`requirements.txt`

```
django==1.6
bpython==0.12
django-braces==0.2.1
django-model-utils==1.1.0
logutils==0.3.3
South==0.7.6
requests==1.2.0
stripe==1.9.1
dj-database-url==0.2.1
django-oauth2-provider==0.2.4
djangorestframework==2.3.1
```

`pip install`

Dependency Declaration: Ruby

Gemfile

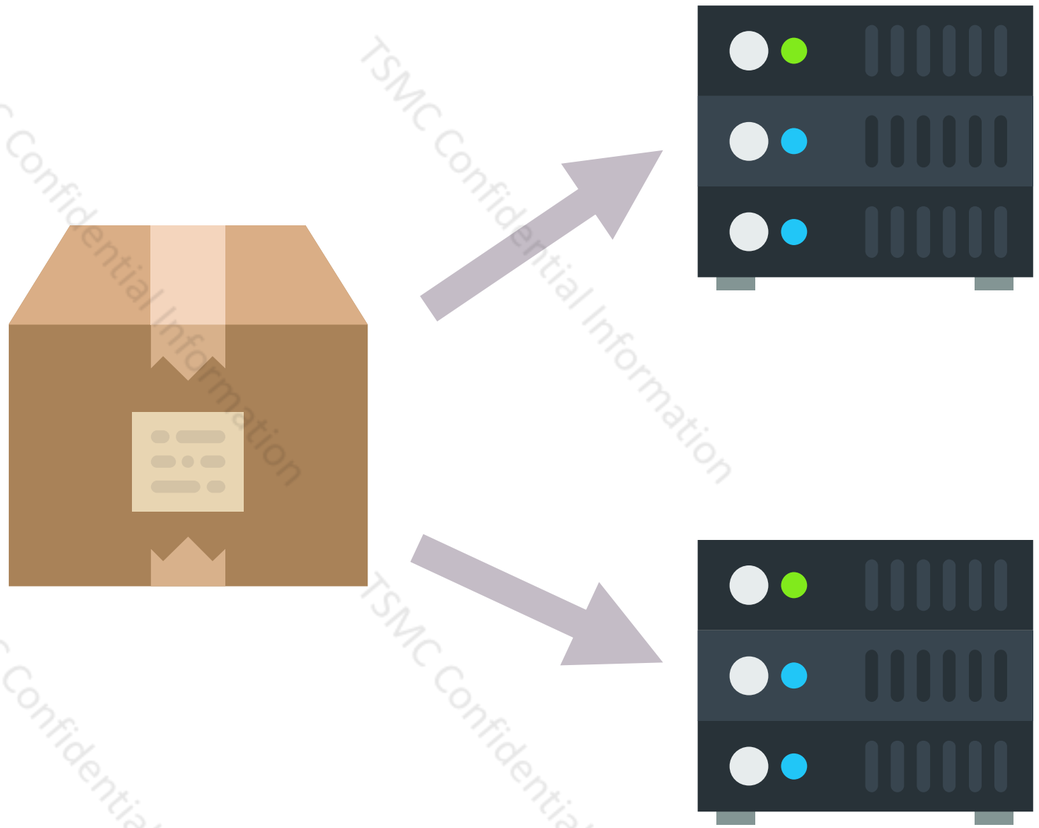
```
source 'https://rubygems.org'

gem 'nokogiri'
gem 'rails', '3.0.0.beta3'
gem 'rack',  '>=1.0'
gem 'thin',  '~>1.1'
```

```
bundle instal
```

Dependency Isolation

Never depend on the host to have your dependency.

Application deployments should carry all their dependencies with them.



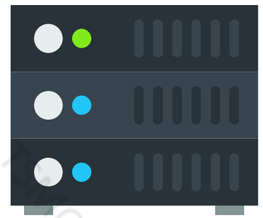
Dependency Declaration & Isolation: Docker

Dockerfile

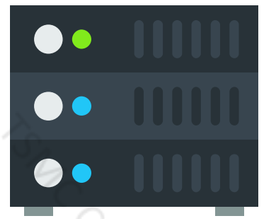
```
FROM mhart/alpine-node:8

RUN apk add --no-cache make gcc g++ python

WORKDIR /srv
ADD . .
RUN npm install

EXPOSE 3000
CMD ["node", "index.js"]
```

docker build

```
docker run
```



Development



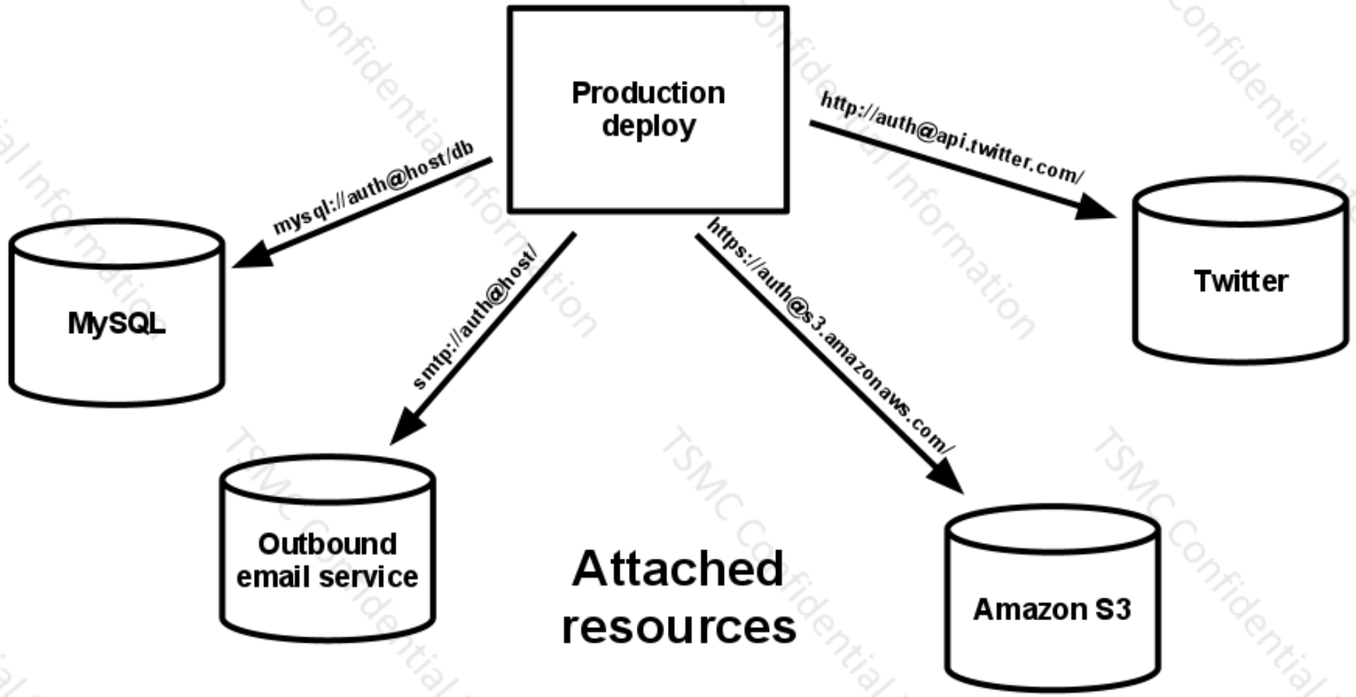
Production

Let's take a detour to factor 5

We will return to factors 3 and 4 later

4 - Backing Services (What does it mean?)

Treat backing services as attached resources

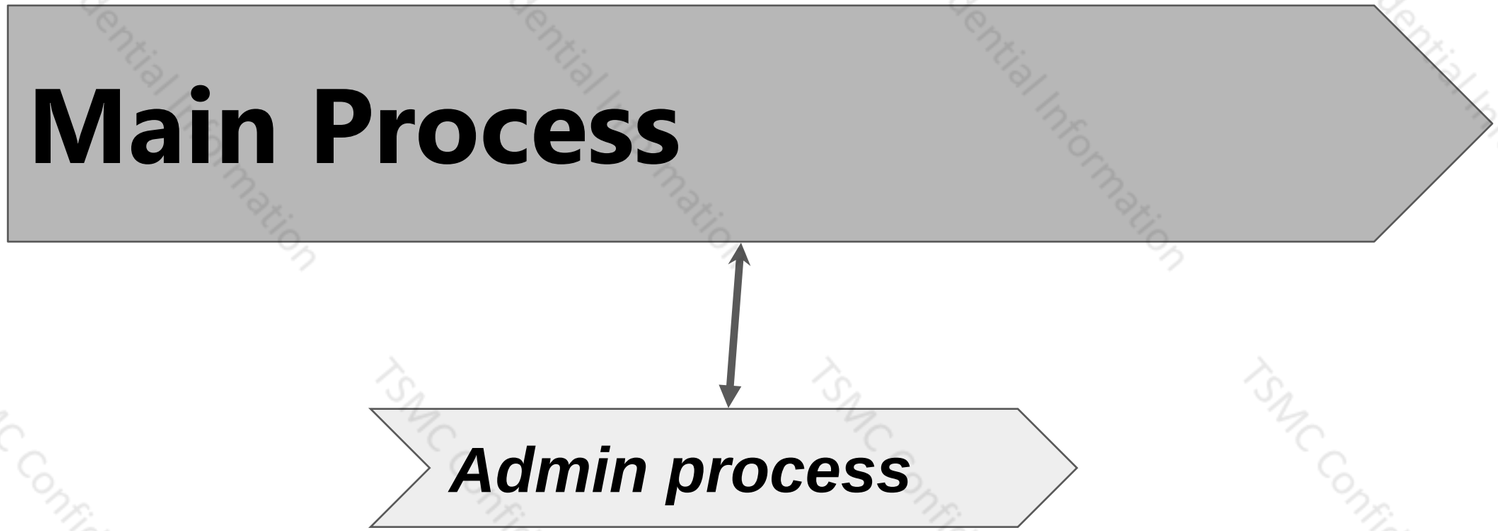


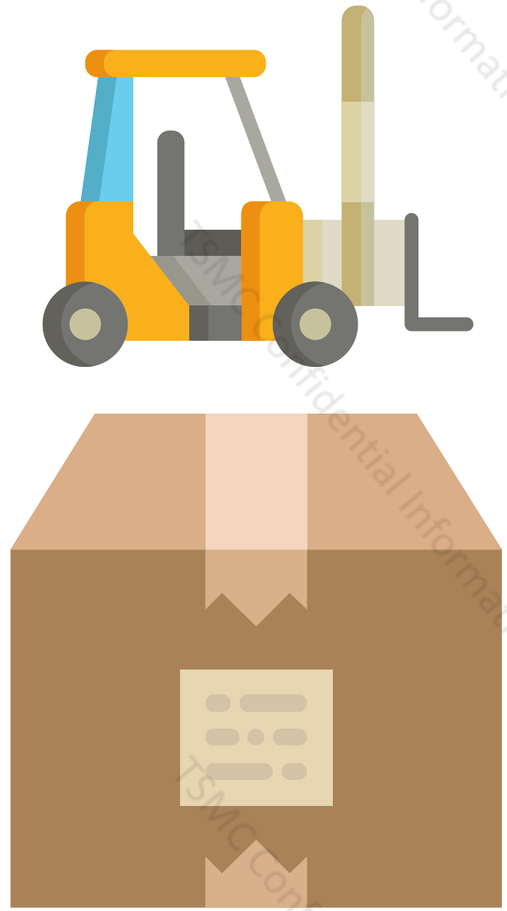
Let's take a detour to factors 11 and 12

We will return to factor 6 later

12 - Admin processes (What does it mean?)

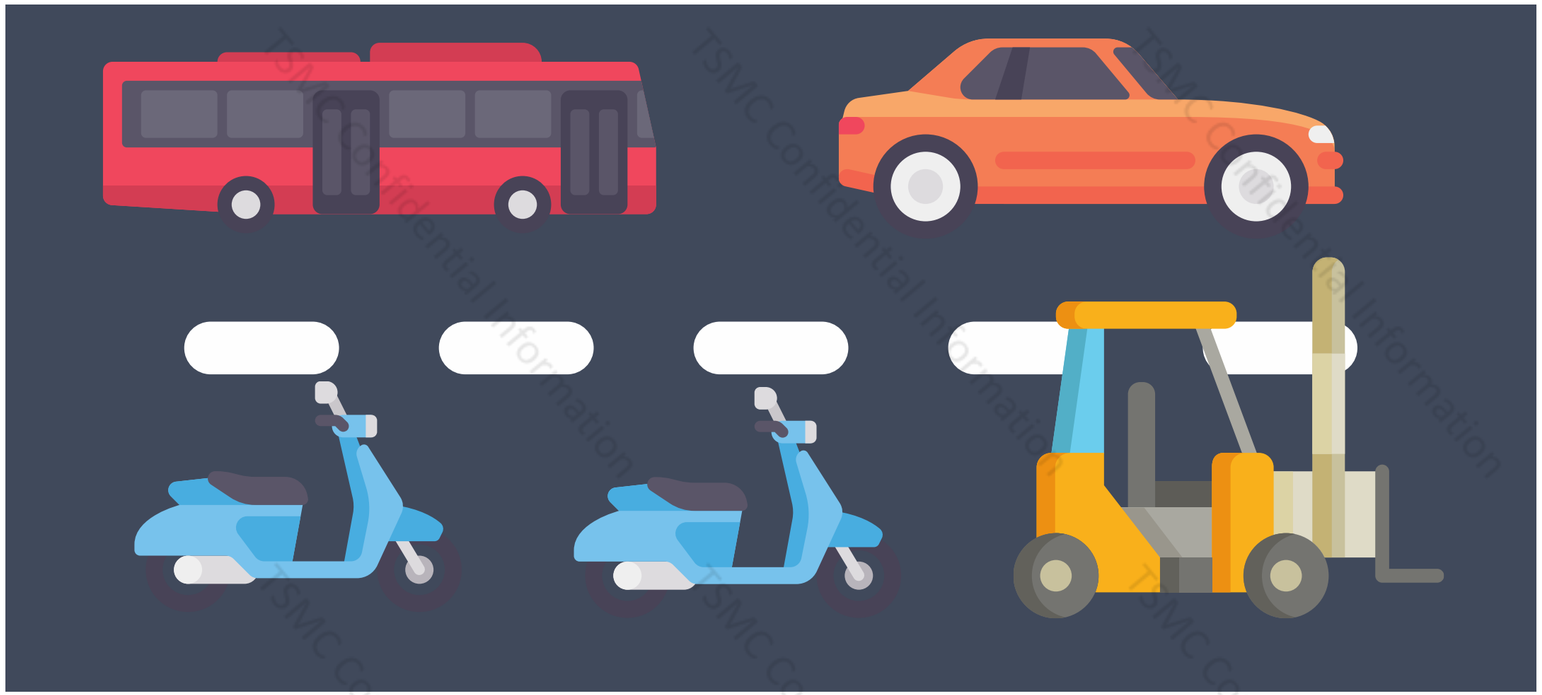
Run admin/management tasks as one-off processes





Admin / management processes are inevitable:

- **Migrate database**
- **Repair some broken data**
- **Once a week move database records older than X to cold storage**
- **Every day email a report to this person**



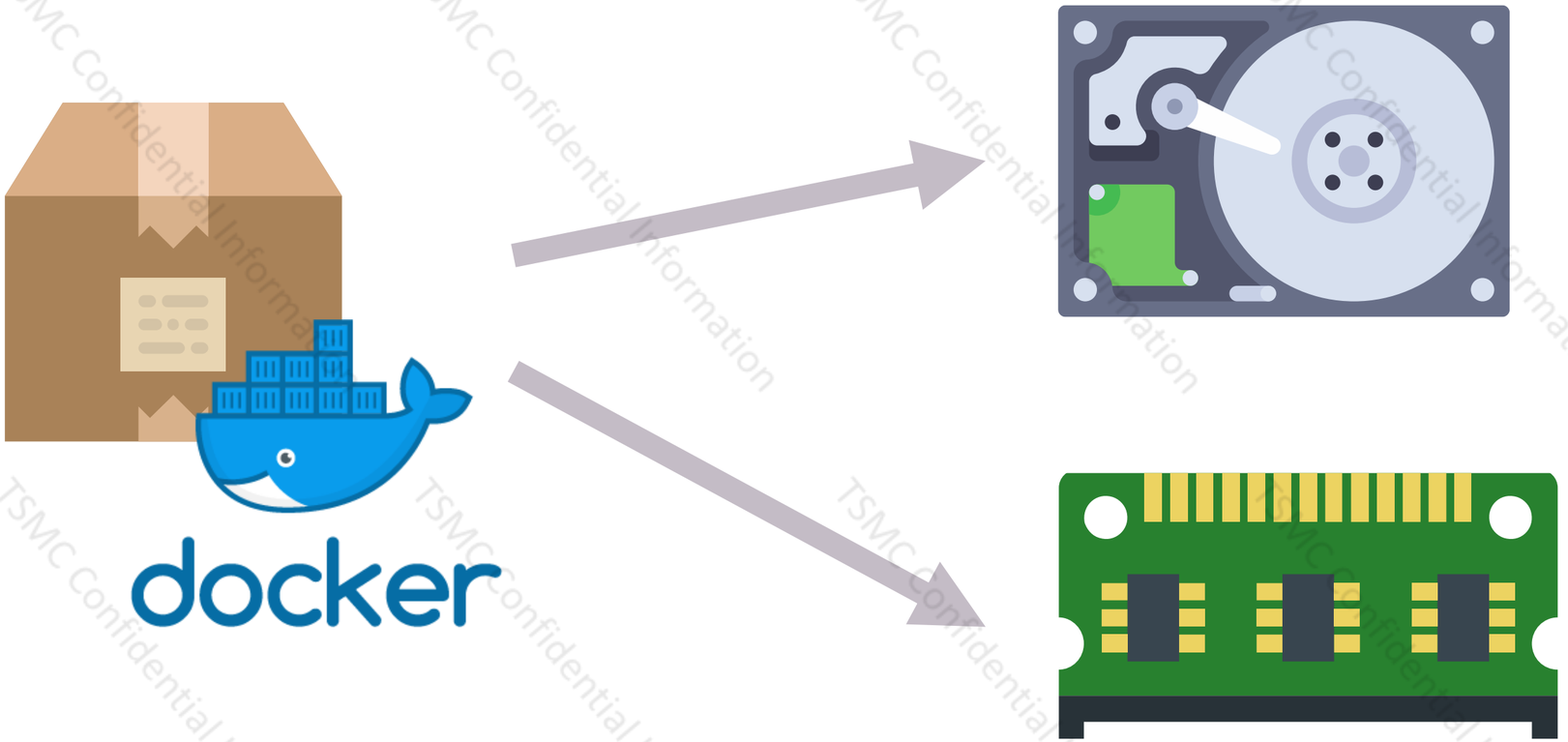
Run admin processes just like other processes.

6 - Processes (What does it mean?)

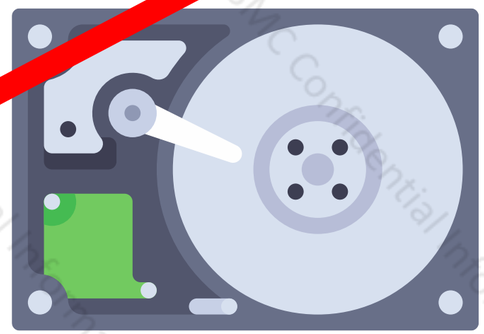
Execute the app as one or more stateless processes

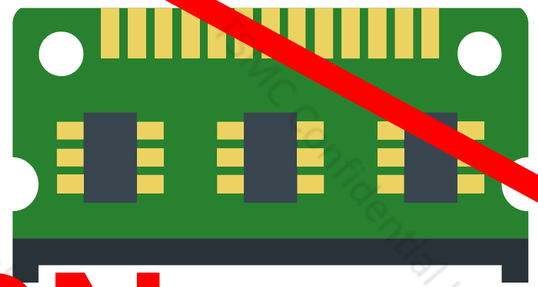
Twelve-factor processes are stateless and share-nothing

Stateful container stores state in local disk or local memory. Workload ends up tied to a specific host that has state data.

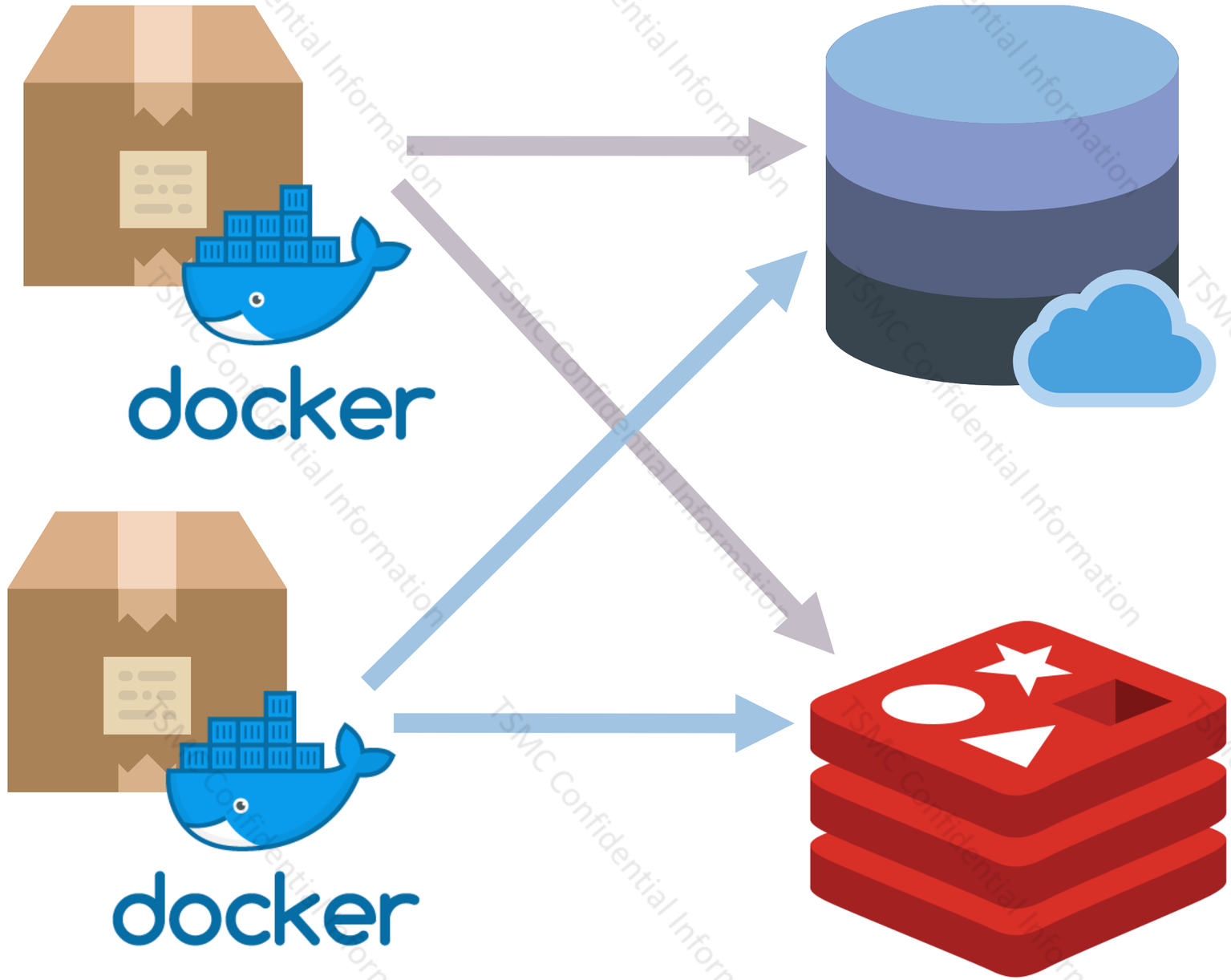


Stateful container stores state in local disk or local memory.
Workload ends up tied to a specific host that has state data.





ANTIPATTERN

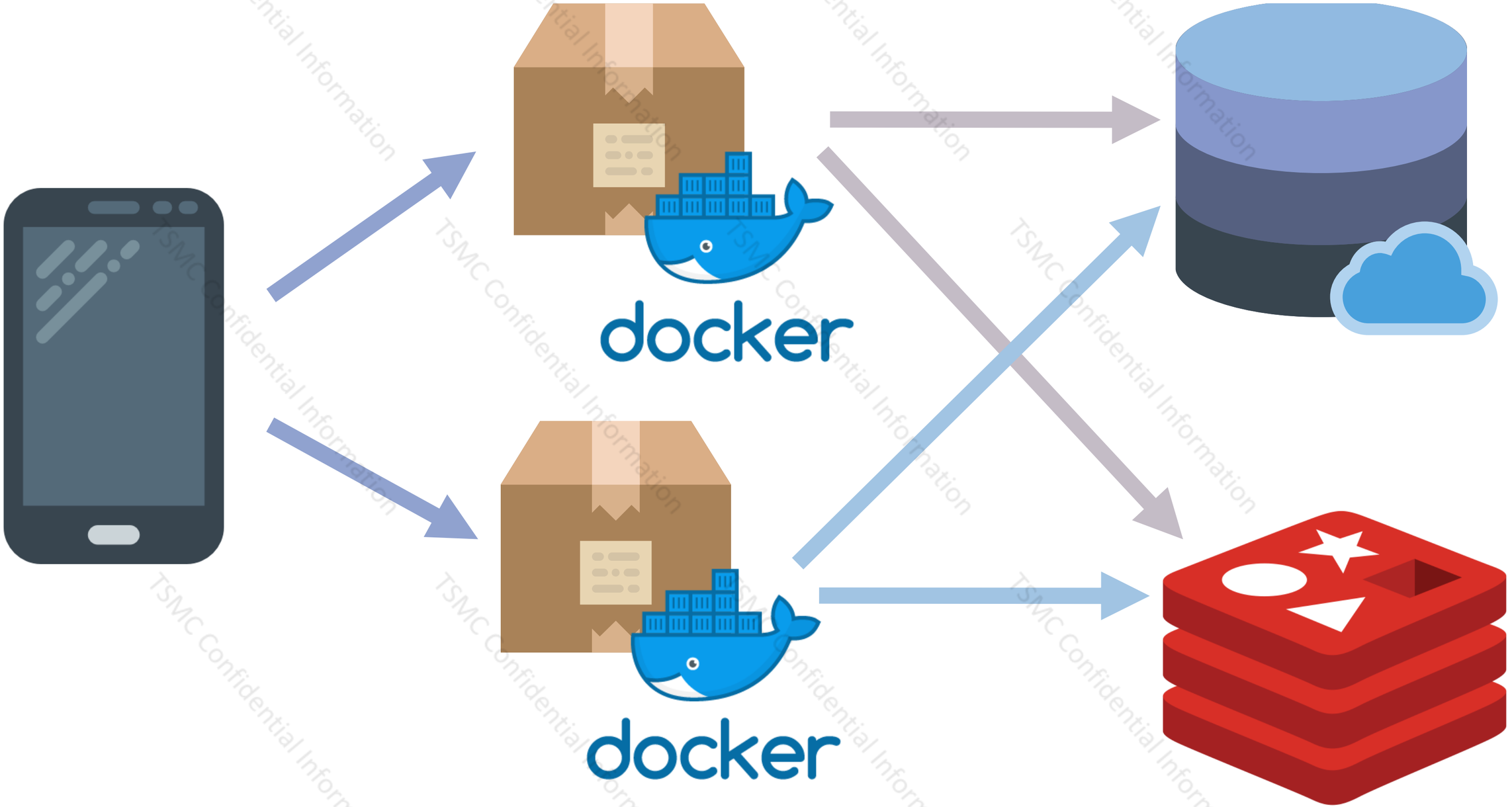


Database

Durable store of truth.
MySQL, PostgreSQL, MongoDB, DynamoDB

Cache

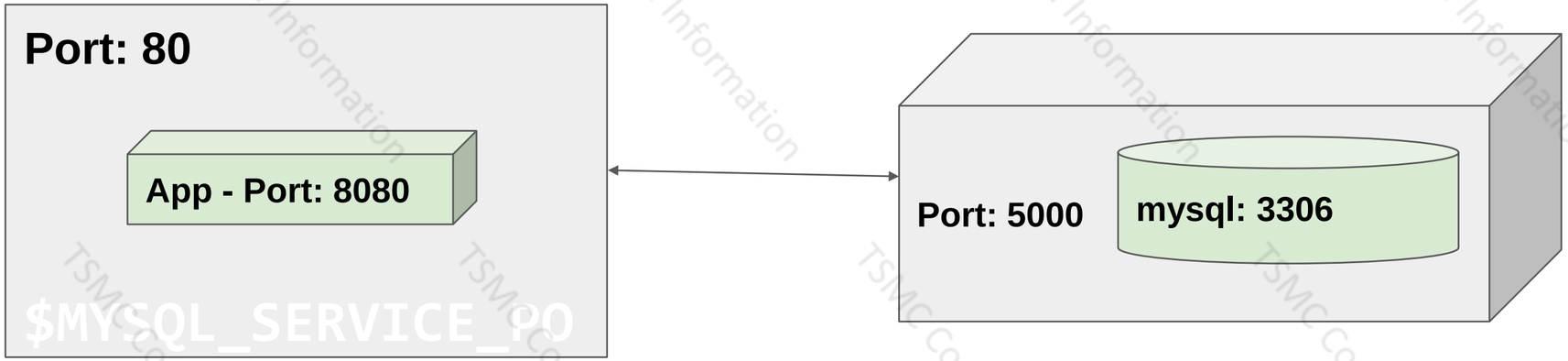
Fast, temporary state store.
redis, memcached

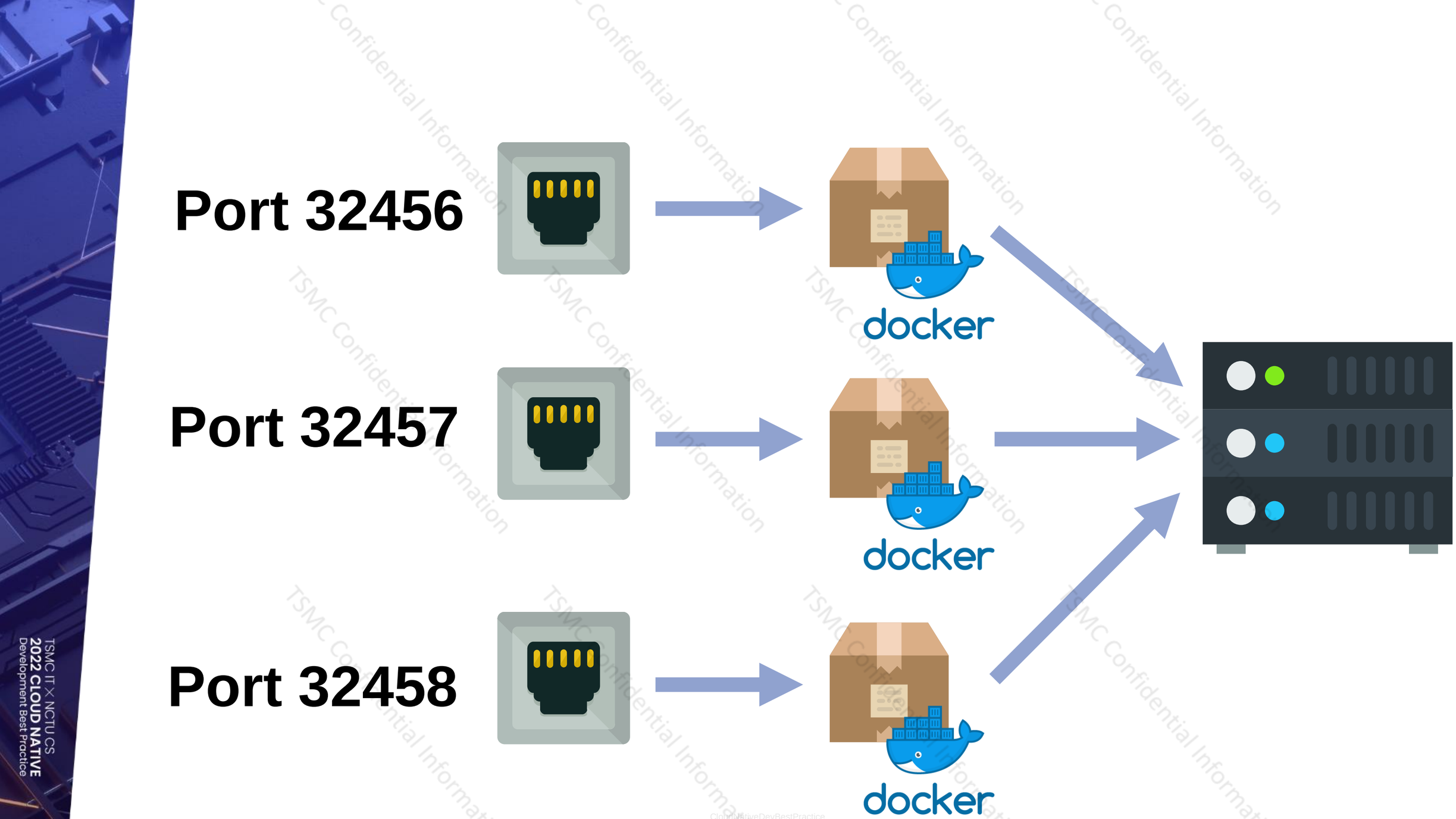


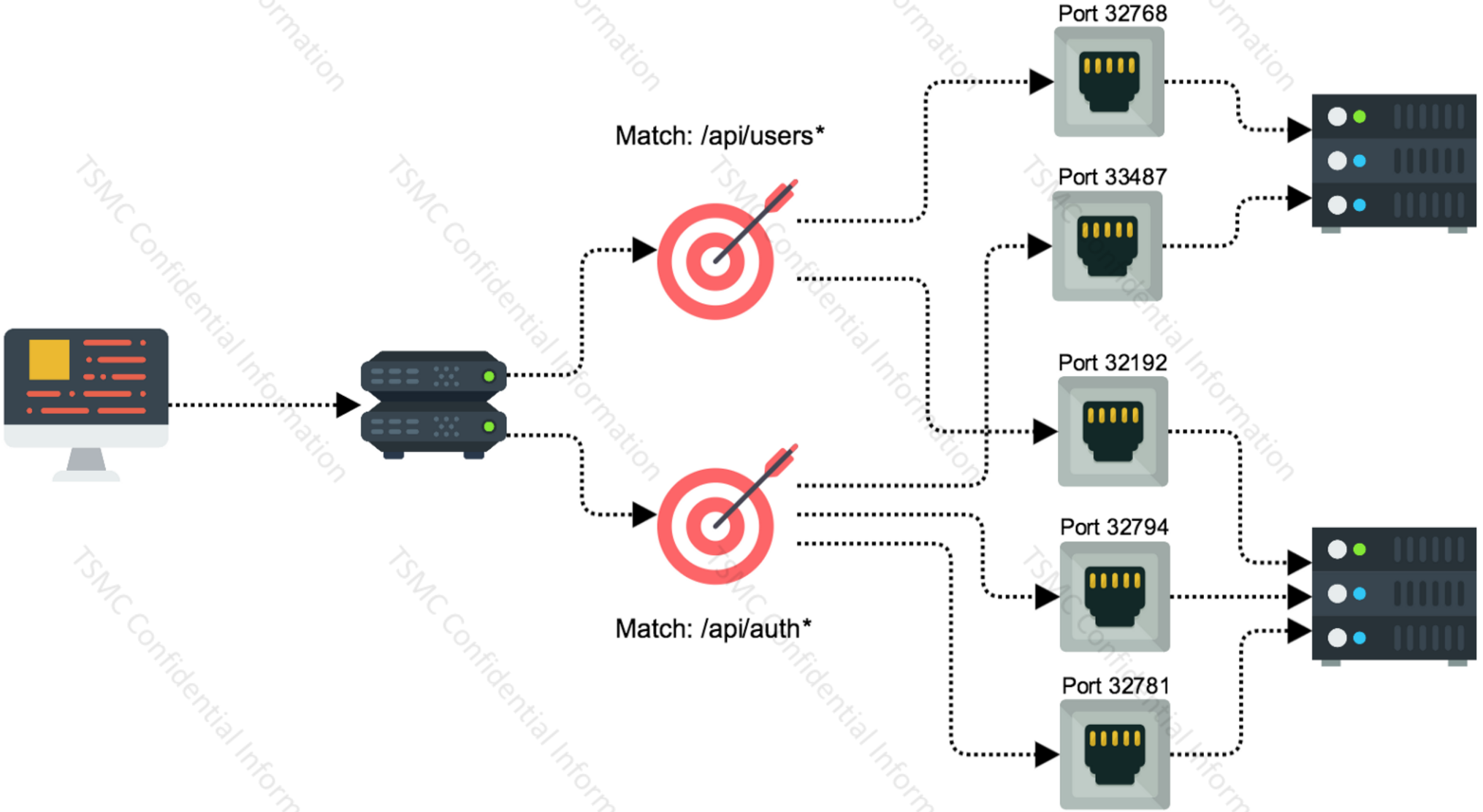
7 - Port Binding (What does it mean?)

Export services via port binding

The twelve-factor app is completely self-contained

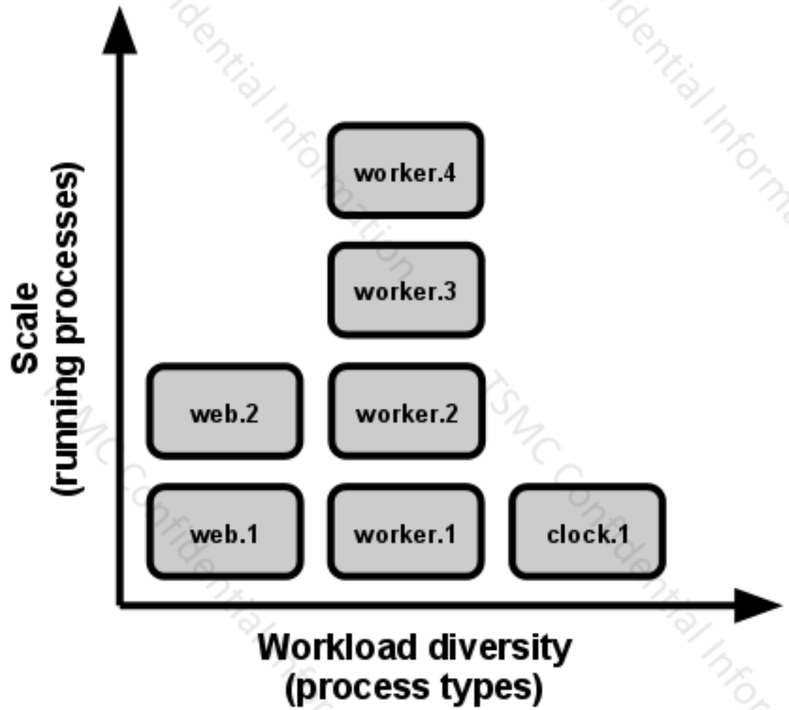






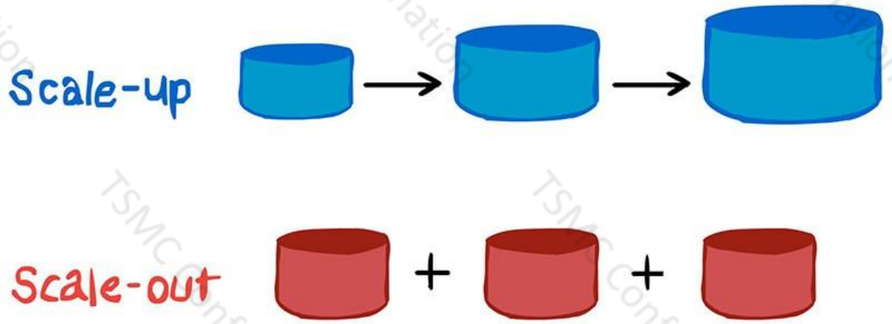
8 - Concurrency

Scale out via the process model



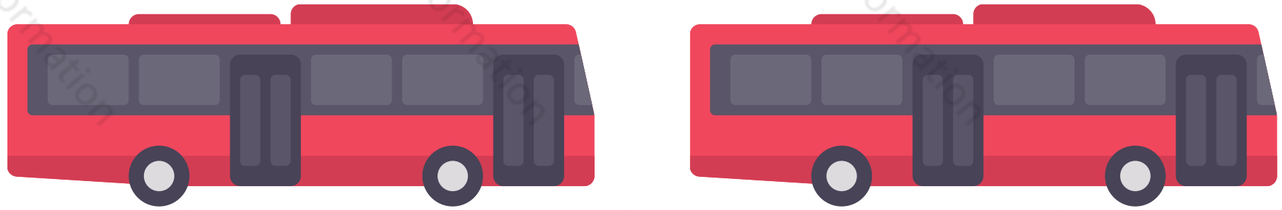
8 - Concurrency (What does it mean?)

- You can scale up and out
- Scale processes types
- Workload diversity
- It "*advocates*" for Microservices



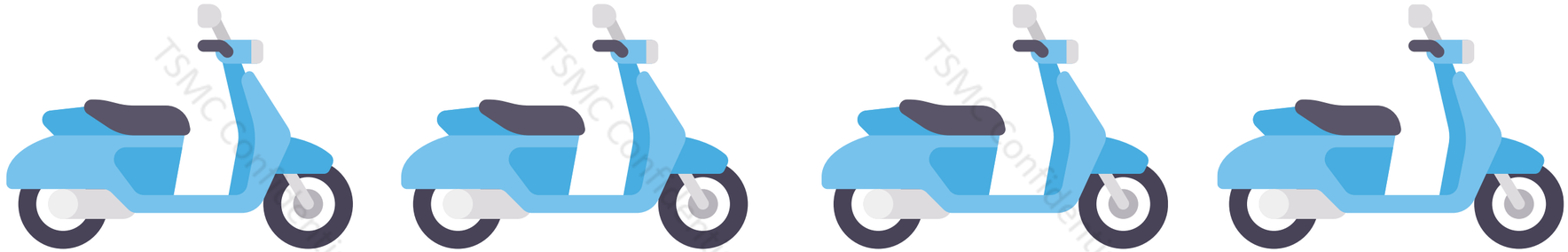
API **Web** **Worker**

Web



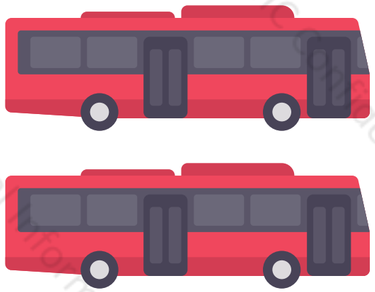
API

Worker



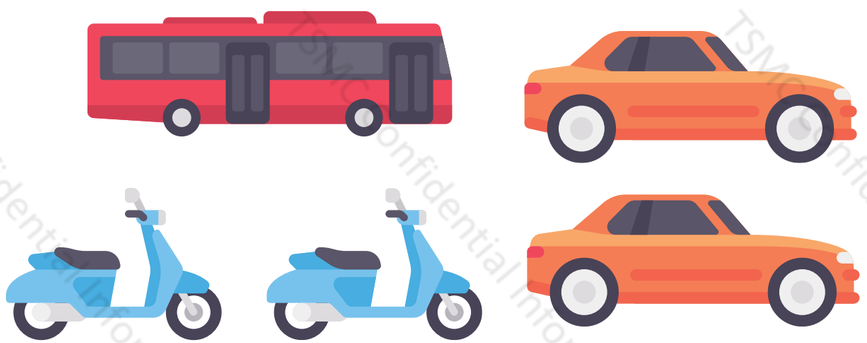
Hosts

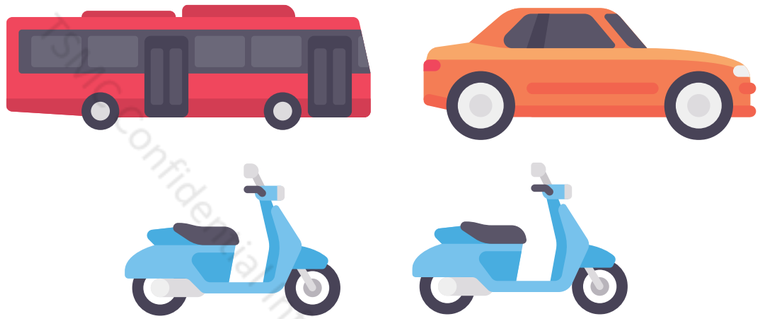
Processes

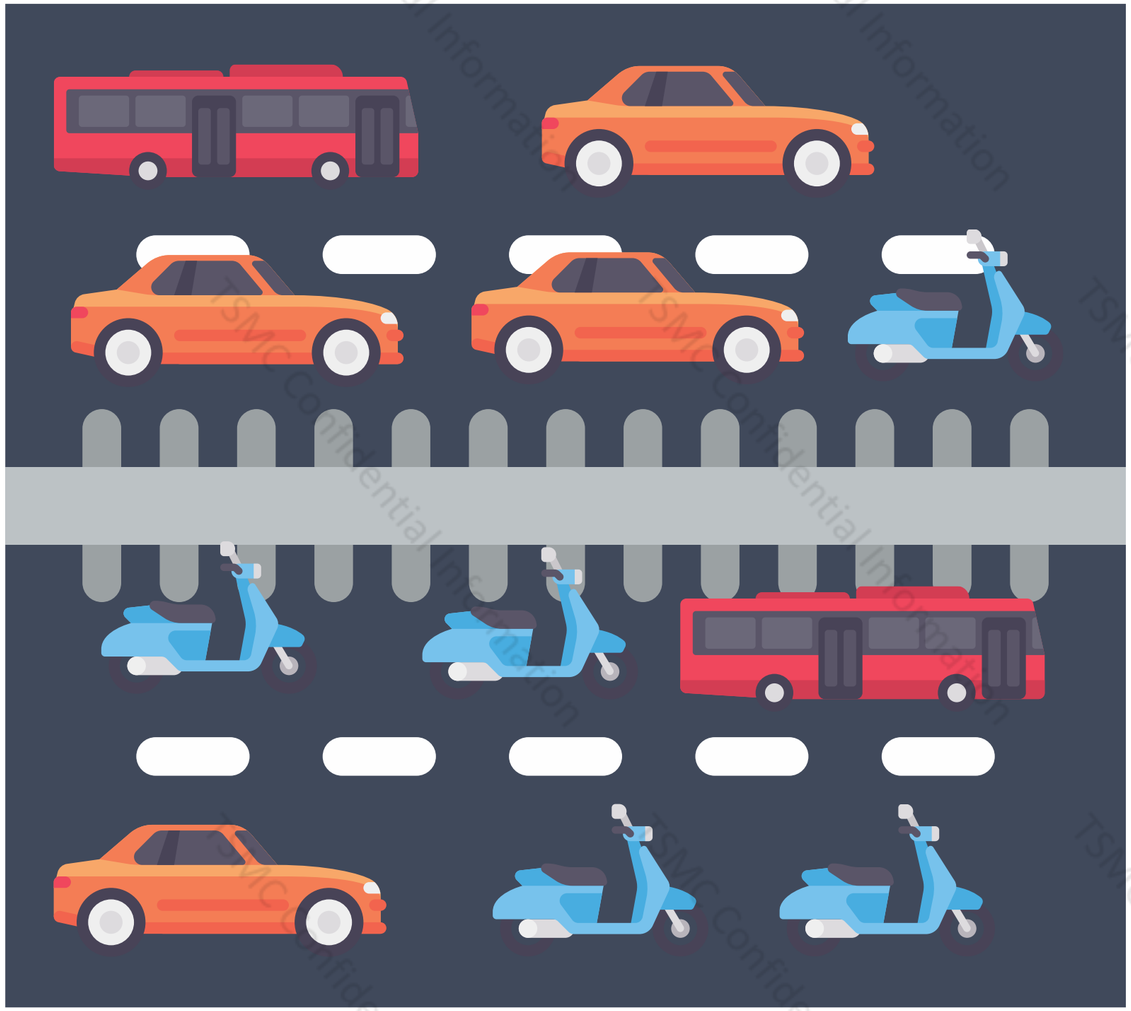


Hosts

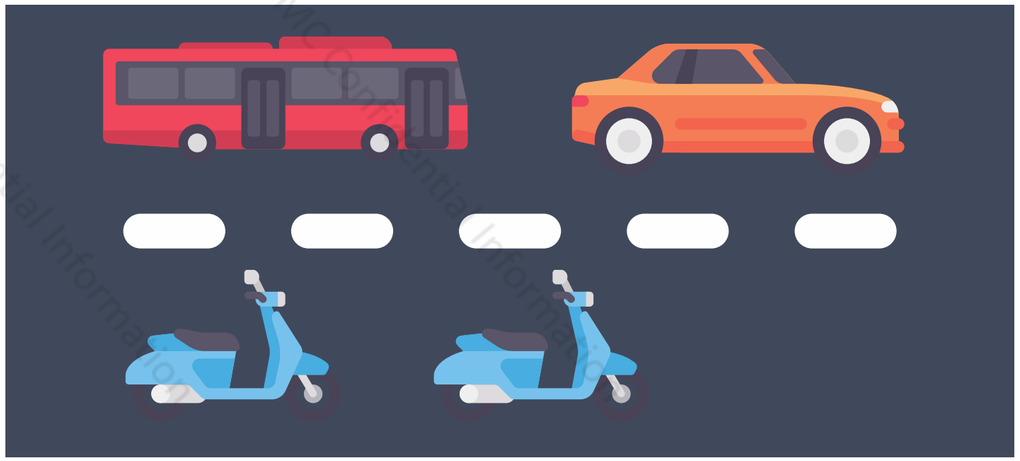
Processes







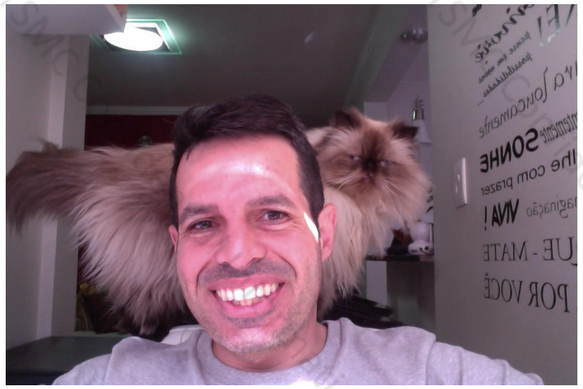
Large Host = More Concurrent Processes



Small Host = Fewer Concurrent Processes

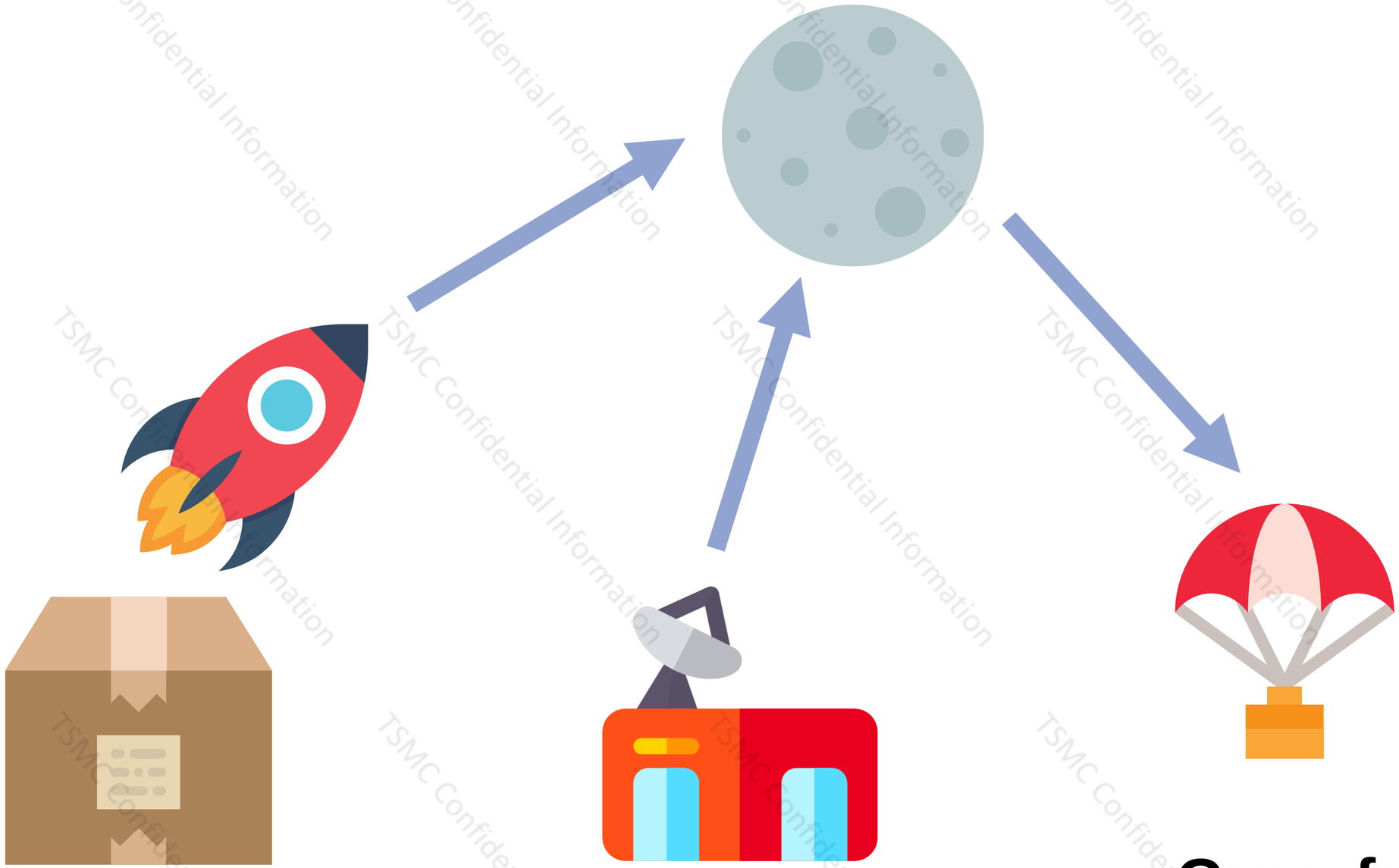
9 - Disposability (What does it mean?)

Maximize robustness with fast startup and graceful shutdown



Pets Vs Cattle

- **Processes can be started or stopped at a moment’s notice**
- **Processes should minimize startup time**
- **Processes shut down gracefully when they receive a SIGTERM**
- **Processes should also be robust against sudden death**



Fast Launch

Responsive

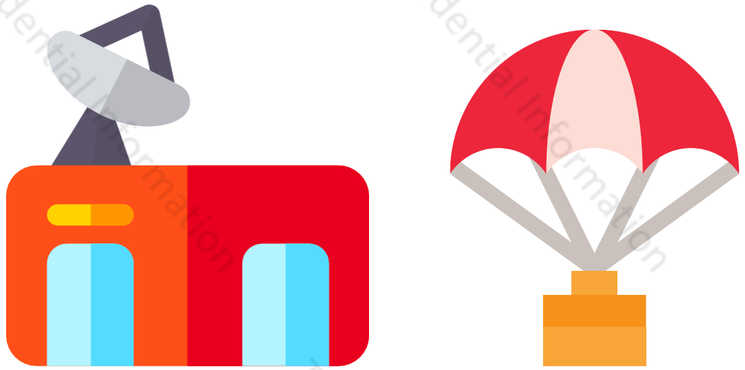
Graceful Shutdown

Fast Launch

Minimize the startup time of processes:

- **Scale up faster in response to spikes**
- **Ability to move processes to another host as needed**
- **Replace crashed processes faster**

Responsive, Graceful Shutdown



Should respond to SIGTERM by shutting down gracefully

```
var server = app.listen(3000);

console.log('Message service started');

process.on('SIGTERM', function() {
  console.log('Shutting down message service');
  server.close();
});
```