



The Chandra

เสนอ

ดร. ปริญญา เอกปริญญา

จัดทำโดย

63010034	กฤษฎา	สารวิทย์
63010124	จักรภัทรณ์	ชื่นถาวร
63010187	ชนัญชิตา	ศรีทองดี
63010202	ชรินดา	สนธิดี
63010224	ชาญวณิชฐ์	นุชอยู่
63010229	ชินกฤต	ปิ่นคล้าย
63010872	วัชรภรณ์	ชาแท่น
63010979	สิรินดา	วังแพน
63010981	สิรินุช	เกตุคำ

รายงานนี้เป็นส่วนหนึ่งของวิชา 01076024 Software architecture and design

คณะวิศวกรรมศาสตร์ สาขาวิศวกรรมคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ที่มาและความสำคัญ

เนื่องจากปัจจุบันนี้เราจะสามารถเห็นได้ว่าในช่วงเวลาที่เราต้องการจะไปพักผ่อนหย่อนใจ ไปเที่ยว หรือทำกิจกรรมต่างสถานที่ที่จำเป็นจะต้องหาโรงแรมเพื่อเป็นที่พักอาศัยชั่วคราว และโรงแรมที่นั่นควรจะมาพร้อมกับระบบการจัดการที่ดีด้วย หากไร้ซึ่งระบบการจัดการที่ดีก็ทำให้เกิดปัญหาในการทำงานต่าง ๆ ภายในโรงแรม โดยที่ไม่มีความเป็นระเบียบเรียบร้อย ยุ่งเหยิง และยากในการติดตามข้อมูลข่าวสารต่าง ๆ จากทางโรงแรม หรือที่พักอาศัยนั้น ๆ ซึ่งอาจจะทำให้ส่งผลกระทบต่อความพึงพอใจของผู้ใช้บริการ และยิ่งเป็นการดี หากโรงแรมสามารถที่จะเก็บข้อมูลด้านพฤติกรรมของผู้ใช้บริการเพื่อนำมาวิเคราะห์ในการพัฒนาระบบการจัดการของโรงแรมให้ดียิ่งขึ้น

กลุ่มของพวกเราอยากที่จะพัฒนา Website application ที่เข้ามาช่วยจัดการงานต่าง ๆ ในโรงแรม (PMS : Property management System) ให้ง่ายขึ้นโดยของ Website เราจะประกอบไปด้วยระบบหลัก ๆ ดังนี้

1. ระบบการจัดการด้านการจองห้องพัก (Reservation management)
2. ระบบการจัดการสถานะของห้องพัก (Front-desk operations)
3. ระบบให้บริการแม่บ้าน (Housekeeping)
4. ระบบจัดการข้อมูลที่เกี่ยวข้องกับลูกค้า (Customer data management)

ข้อจำกัด

1. ส่วน Payment ทำการ mockup ไว้ แต่ไม่ได้ประสานงานกับหน่วยงานที่เกี่ยวข้องโดยตรง
2. ไม่สามารถใช้ Channel manager อื่นได้

สมมุติฐาน

1. สามารถจองห้องพักได้
2. เช็คสถานะห้องพักได้
3. มีการคำนวณราคาที่พักให้เหมาะสมในแต่ละช่วง
4. มีระบบแม่บ้านที่แสดงรายละเอียดการทำงานรายวันของแม่บ้านแต่ละคนตามหน้าที่
5. แม่บ้านสามารถเข้าดูรายละเอียดการทำงานของตนเองได้
6. มีการเก็บข้อมูลลูกค้าและรายละเอียดการชำระเงิน

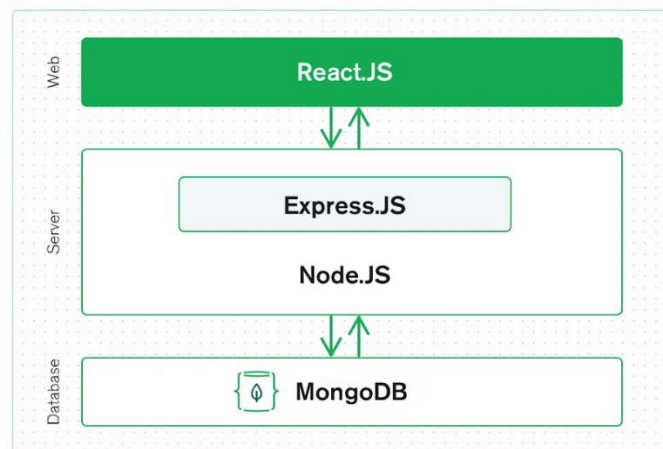
Boundaries

- สิ่งที่ทำ

1. ระบบห้องพัก สามารถเพิ่มห้องพักและข้อมูลของแต่ละห้องได้
 2. ระบบแสดงสถานะของห้องพัก
 3. ระบบสามารถค้นหาและจองห้องพักตามเงื่อนไขต่าง ๆ ได้
 - a. ระบบสามารถค้นหาห้องพักว่าง ตามวันที่และประเภทของห้อง เพื่อทำการจองได้
 - b. ระบบสามารถจองห้องพักได้ผ่านทางหน้าเว็บและผ่านระบบ Admin
 4. ระบบสามารถเพิ่มพนักงานได้ และจำแนกงานให้พนักงานแต่ละคนได้
 5. ระบบสามารถบันทึกผู้เข้าพัก ประวัติการเข้าพักและรายละเอียดการชำระเงินได้
- สิ่งที่ไม่ทำ
 1. การปรับเปลี่ยนราคาห้องพักตามเทศกาลได้
 2. แม่บ้านสามารถเข้าดูรายละเอียดของงานแต่ละวันในระบบได้

Software Architecture

เนื่องจากเราเลือกทำเป็นรูปแบบของ web application เราจึงได้เลือกใช้ Architecture เป็น **3-Tier (Client-Server) Architecture** พัฒนา software โดยใช้ MERN stack (MongoDB, Express.JS, React, Node)



1. Presentation tier (Front-end) : React.JS

Top tier ของ MERN stack คือ React ซึ่งเป็น JavaScript library ที่ใช้สำหรับการสร้าง dynamic client-side applications ใน HTML โดยที่ React จะช่วยสร้าง interfaces ที่ซับซ้อนผ่าน simple components, เชื่อมต่อกับข้อมูลบน back-end server และ render เป็น HTML.

ซึ่ง React's strong suit คือการจัดการ stateful, data-driven interfaces ด้วยโค้ดที่น้อยที่สุด แก่ไข่น้อยที่สุด และมีคุณสมบัติทั้งหมดที่คาดหวังจาก modern web framework ได้แก่ support for forms, error handling, events, lists และอื่น ๆ

2. Middle tier (Back-end) : Express.js และ Node.js

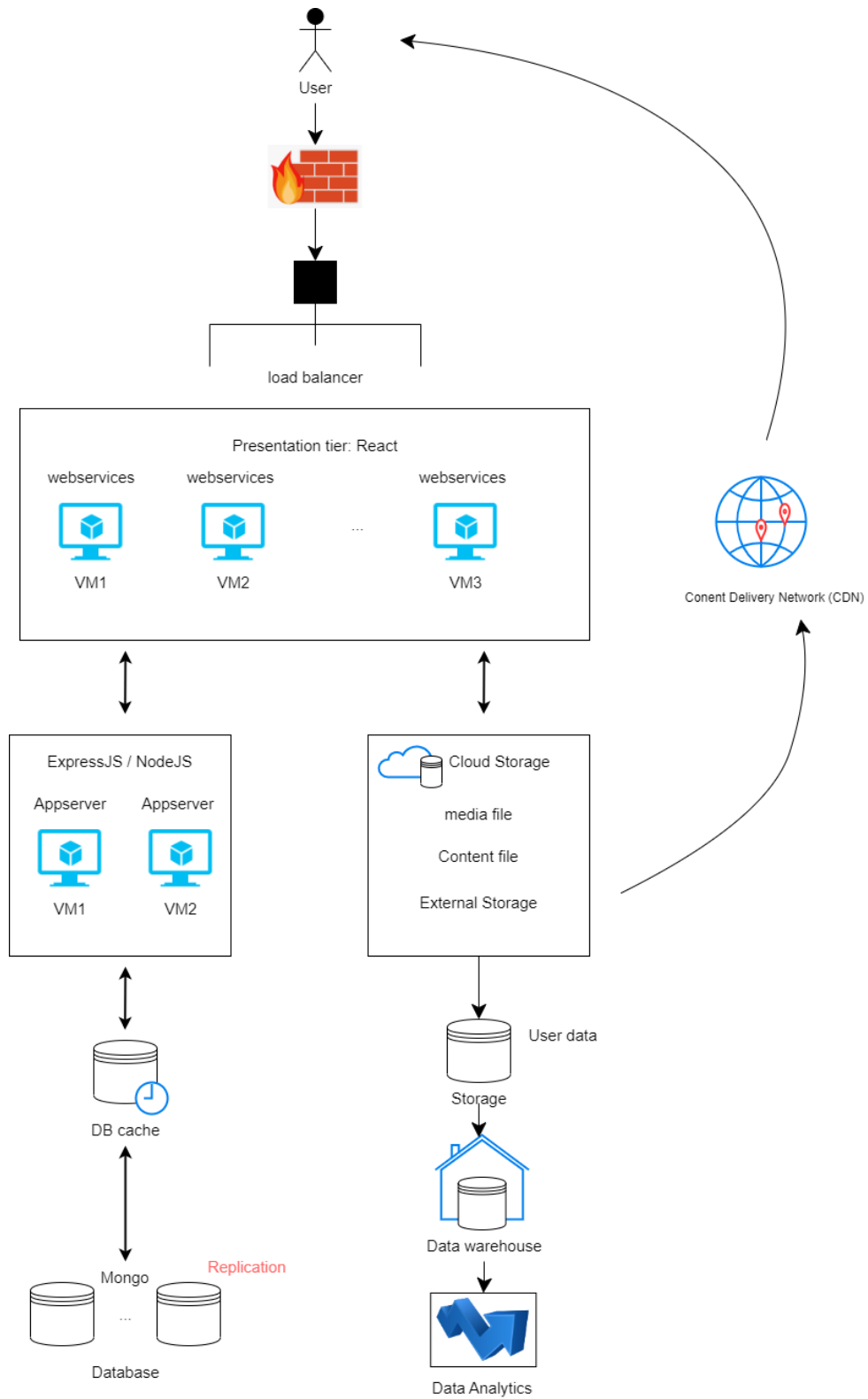
Level ถัดมา คือ Express.js server-side framework ซึ่งทำงานภายใน Node.js server ตัว Express.js จัดว่าเป็น "เฟรมเวิร์กเว็บที่รวดเร็ว, unopinionated และเรียบง่ายสำหรับ Node.js" โดย Express.js มีโมเดลที่มีประสิทธิภาพสำหรับการกำหนด URL routing (จับคู่ URL เข้ากับ server function) และจัดการ HTTP requests และ responses.

การทำ XML HTTP Requests (XHRs) หรือ GETs หรือ POSTs จาก React.js ในส่วน front end สามารถเชื่อมต่อกับฟังก์ชัน Express.js ที่เป็นตัวขับเคลื่อนแอปพลิเคชันได้ ในทางกลับกัน ฟังก์ชันจะใช้ MongoDB's Node.js drivers ไม่ว่าจะผ่านการ callbacks หรือ using promises เพื่อเข้าถึงและอัปเดตข้อมูลใน MongoDB database.

3. Data tier (Database) : MongoDB

แอปพลิเคชันสามารถเก็บข้อมูล (user profiles, content, comments, uploads, events, ฯลฯ) จะต้องการ database ที่ใช้งานได้ง่าย นั่นคือเหตุผลที่ทำให้มี MongoDB เข้ามา โดย JSON documents ที่สร้างขึ้นใน React.js front end สามารถส่งไปยัง Express.js server ซึ่งสามารถประมวลผลและ (assuming ว่า valid) เก็บไว้ใน MongoDB โดยตรง

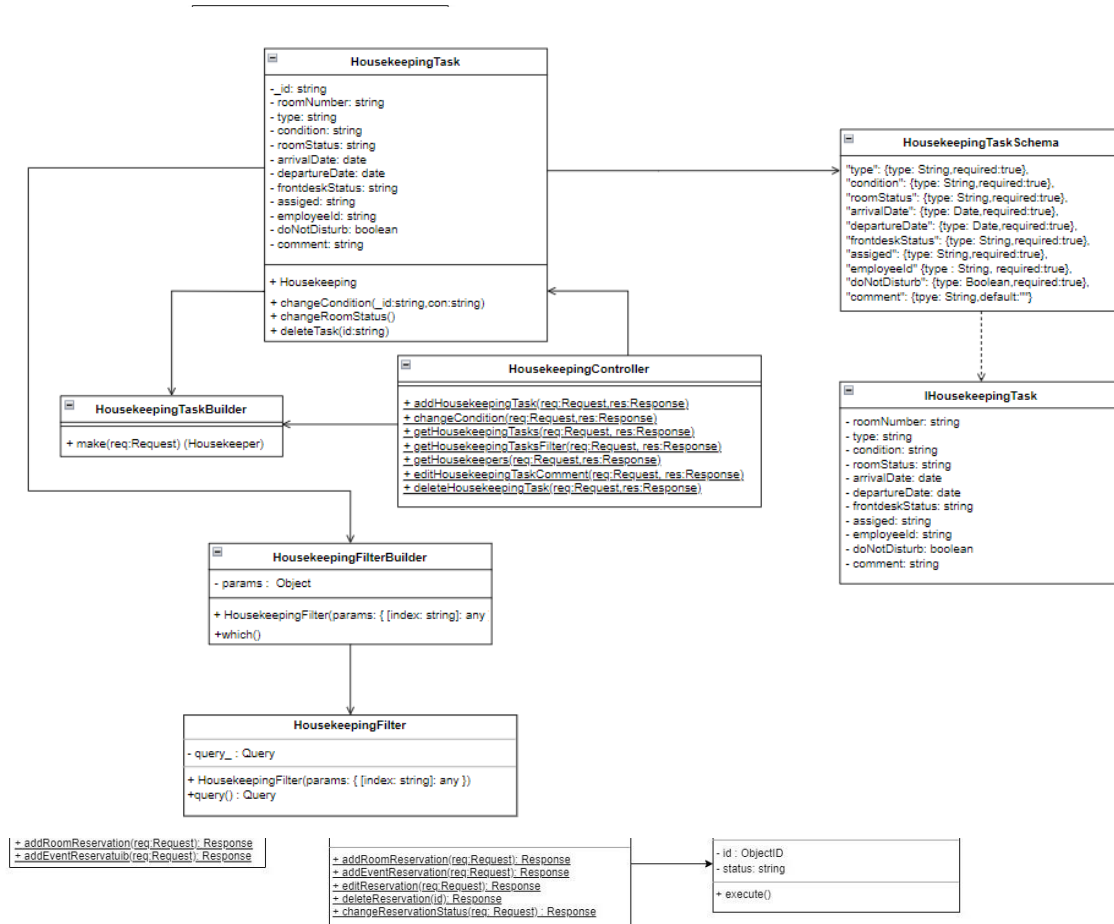
Software design



Software Architecture (3-Tier/Client-Server Architecture) (Use cases diagram)

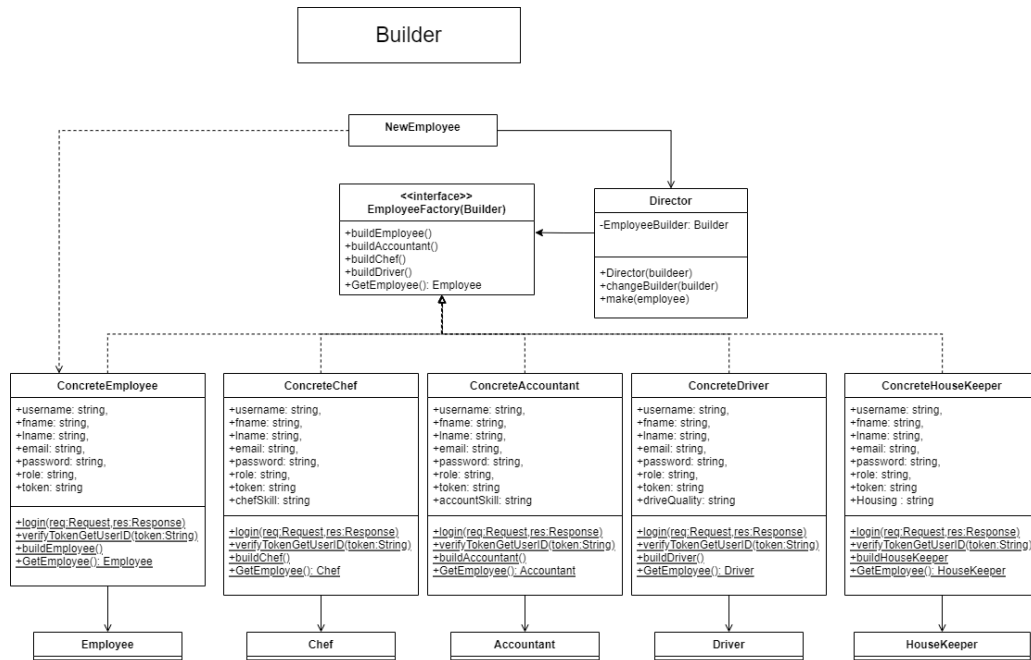
Bounded contexts

1. Reservation [By Customer]

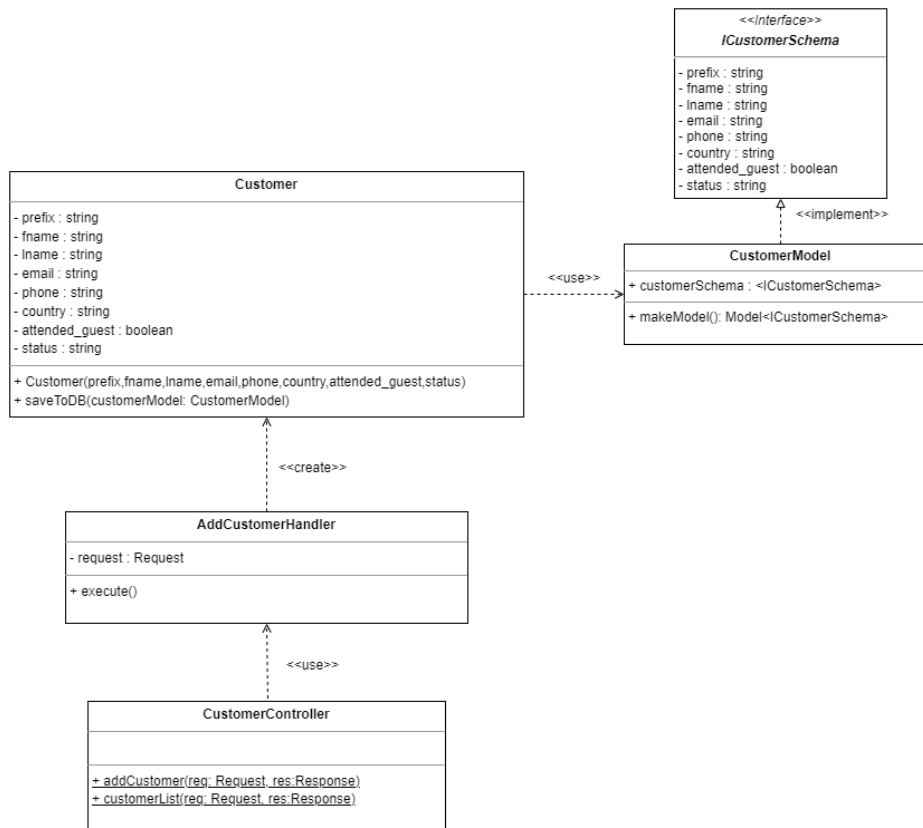


2. Housekeeping

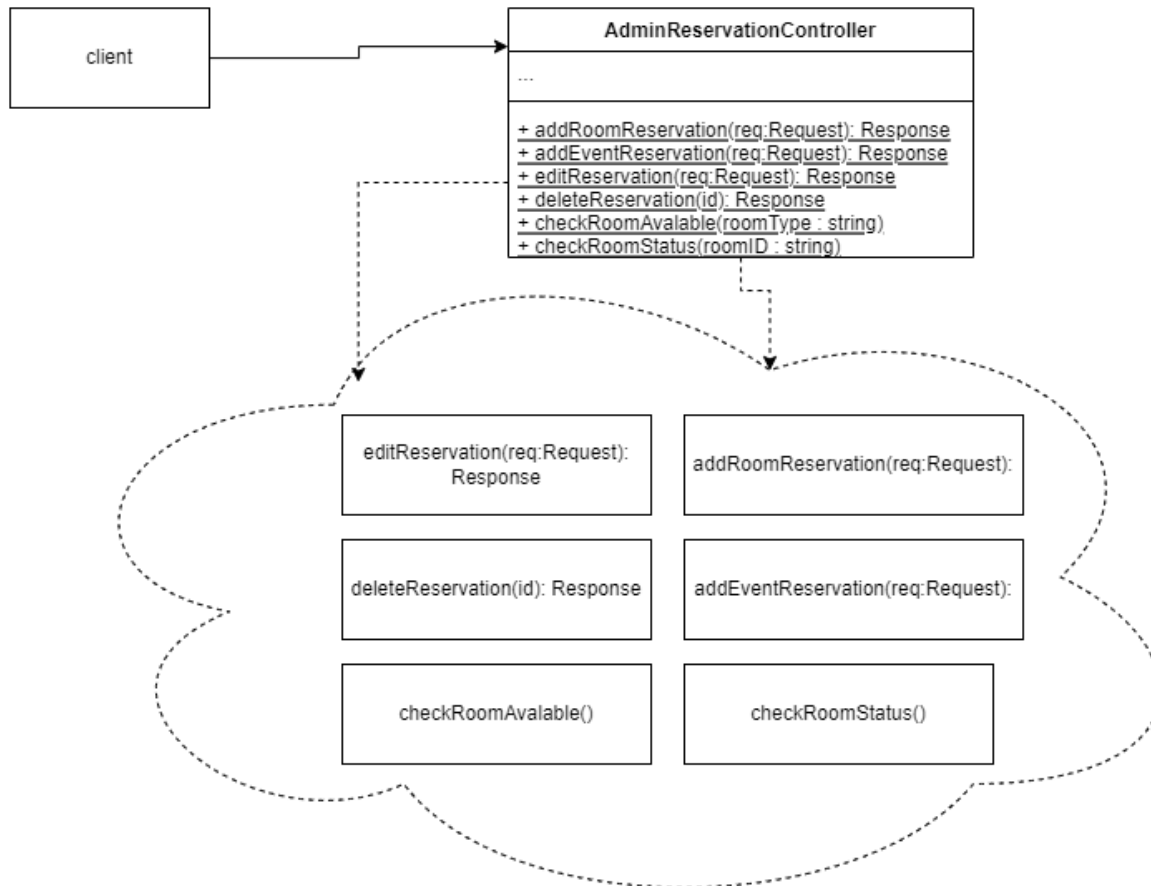
3. Employee



4. Customer



5. Reservation [By Admin]



Design pattern

Design Pattern Factory

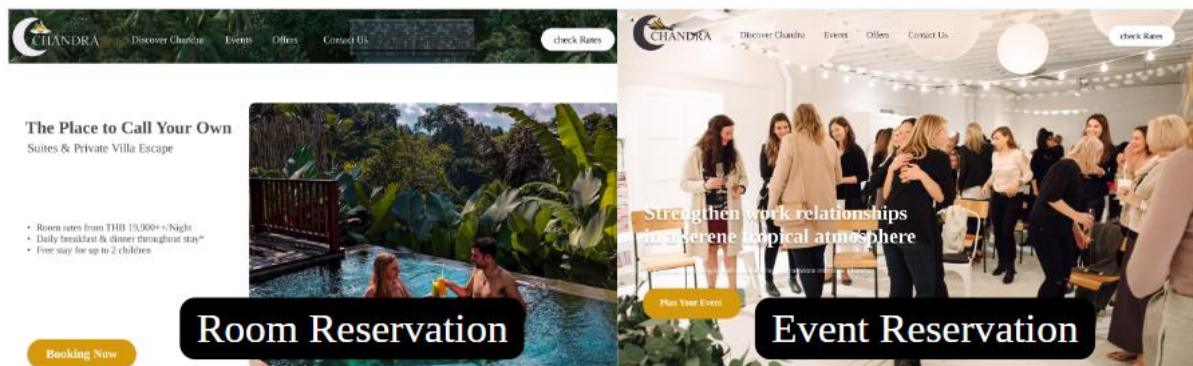
Factory คือการสร้างโรงงานมาผลิต object ที่มี attribute ต่างกัน เพื่อลดการบวมออกข้างถ้าหากเป็นการสร้างคลาสเพียงคลาสเดียวเราใช้ในส่วนของการ Reservation

- **ทำไมถึงใช้**

เนื่องจากเราได้สร้าง class RoomReservation สำหรับการจองห้องพัก และเพิ่ม class EventReservation ทำให้ code สับสน เนื่องจากจะมีข้อมูลที่ใช้ในการจองหลาย ๆ ส่วนที่เหมือนกันแต่ต่างกันที่ข้อมูลข้างในนั้นเช่น ชื่อลูกค้า, วันที่จอง, ราคาที่จอง, จำนวนแขก ฯลฯ โดยที่ทั้งสอง class หน้าหน้าเหมือนกันคือ เชฟเข้าไปในฐานข้อมูลสำหรับการจองห้องให้กับลูกค้า (SaveToDB())

ดังนั้น Factory Design Pattern ที่มีคลาสแม่เป็น interface จะรับผิดชอบในการเซฟข้อมูลการจองลงฐานข้อมูล จึงสร้างฟังก์ชัน SaveToDB() ไว้แล้วมีคลาสลูกไปรับฟังก์ชันนั้นมาดูแลจะเป็นการเหมาะกับการที่เรามีคลาสสองคลาสที่มีตัวแปรเก็บค่าข้างในคล้ายกันอย่างมากต่างกันแค่ข้อมูลข้างในตัวแปรนั้น โดย Factory Design Pattern นี้จะให้ผลลัพธ์คล้ายกับการ

new object ก็จริงแต่จะช่วยหากในอนาคตเรามีการจองเพิ่มขึ้นอีก เช่น จองร้านอาหารของทางโรงแรม, จอง lobby ของโรงแรม ก็จะทำให้การเพิ่มการจองลงฐานข้อมูลทำได้ง่ายขึ้น



โรงแรมมีรูปแบบการจอง 2 รูปแบบ

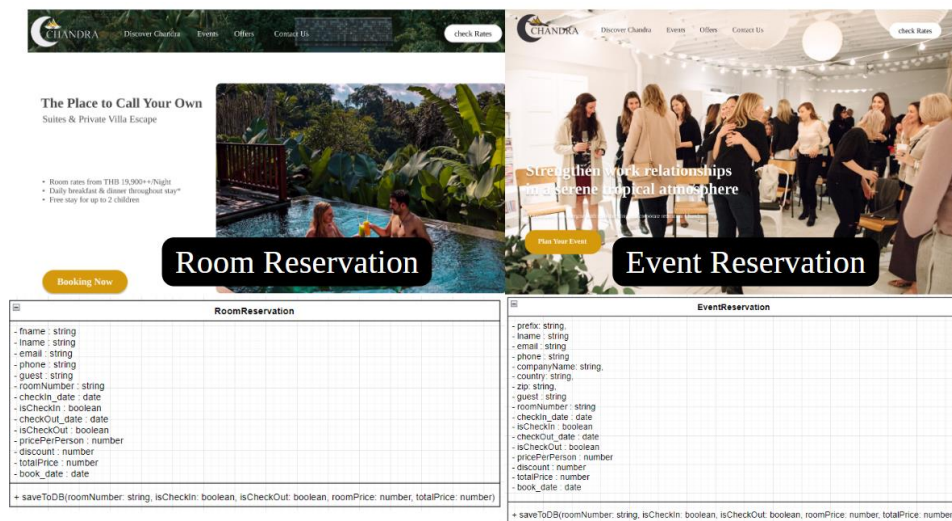
1. Room Reservation จองห้องพัก
2. Event Reservation จองจัดงานอีเวนต์

ข้อมูลที่ต้องใช้ในการจองทั้งสองรูปแบบ ได้แก่

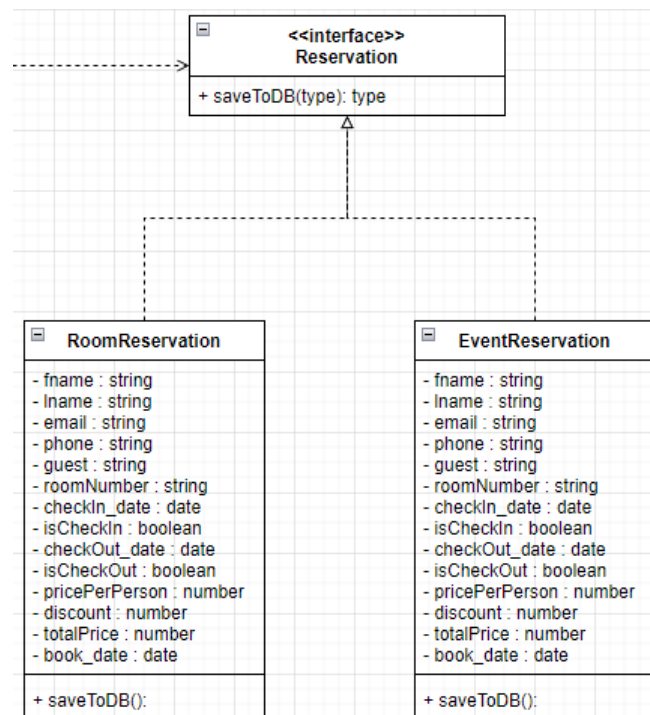
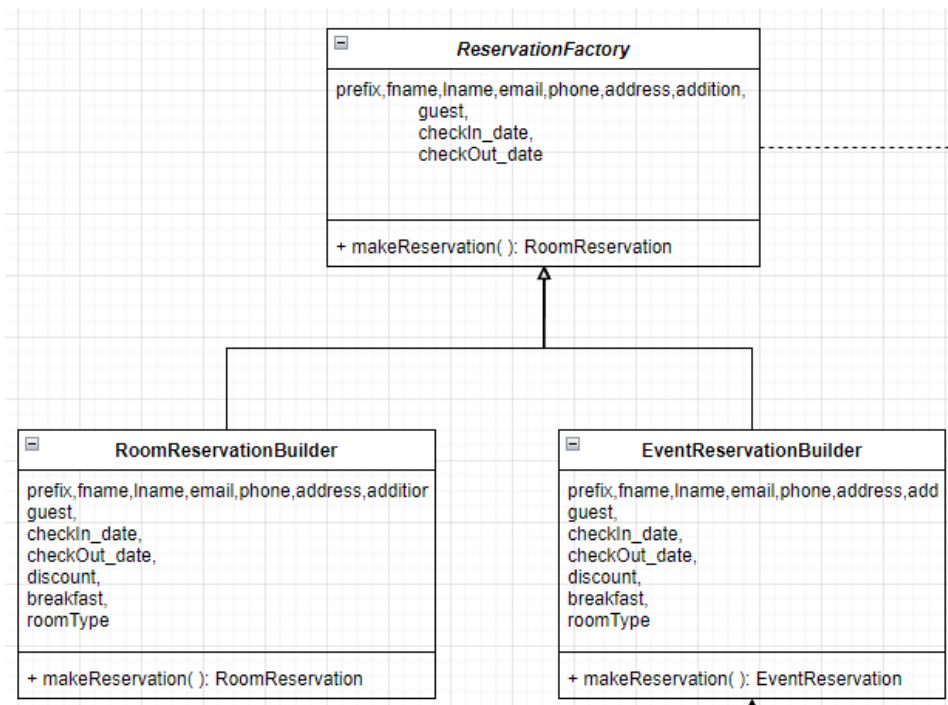
1. Firstname Lastname
2. email
3. Phone number
4. CheckIn-CheckOut date
5. price
6. discount
7. TotalPrice
8. Book_data

- **ใช้อย่างไร**

จากการวิเคราะห์เราจะได้มาว่าการจองทั้งสองแบบนี้เป็นตัวกำหนดลักษณะข้อมูลข้างในของการจอง เราจึงแยกการเซฟข้อมูลลง Database ตามประเภทการจอง เราจะได้โรงงานผลิตการจองออกมา 2 โรงงาน



ซึ่งแต่ละโรงงานจะมีหน้าที่คอยดูแลข้อมูลในการจองของใครของมันเอง ซึ่งตรงกับหลัก SRP(Single-Responsibility Principle) แต่เนื่องจากมีปัญหาที่ถ้าเวลามีอะไรเข้ามาใหม่ต้องไปนั่งไล่แก้โค้ดเก่า เลยทำให้เรารวมร่างโรงงานทั้งคู่ใน class interface ที่ชื่อ Reservation (Factory) ໄວ້เลย






โดยเราทำให้มี class interface ชื่อ Reservation มารับผิดชอบเรื่องการเซฟข้อมูลลง Database แทนและมี subclass 2 subclass คือ Room Reservation และ Event Reservation

ใช้แล้วมีผลดียังไง (ช่วยอะไร)

1. ช่วยให้ตอนสร้าง object ข้อมูลการจอง ที่รับหลาย ๆ parameters ไม่ให้ซ้ำซ้อนกัน ระหว่างประเภทจองห้อง หรือจองจัดงานอีเว้นท์ และทำให้ object นั้นพร้อมสำหรับการใช้งานเลย โดยไม่ต้องระบุข้อมูลที่แท้จริงของ object นั้น
2. เปิดโอกาสให้คลาสลูกเป็นคนตัดสินใจในการสร้าง object ที่เหมาะสมแทน(ใส่ข้อมูลรายละเอียดการจองต่าง ๆ เอง)

Design Pattern Builder

			
name	Susan	Susan	Suksan
lastname	ploenjit	prompong	prompong
role	Accountant	Chef	Driver

ทำไมถึงใช้

เรามีปัญหาในการสร้าง Class employee ที่แต่ละ employee จะมีข้อมูลภายในแตกต่างกันจึงต้องการ employee ที่เหมือนกันออกมาจากแป้นพิมพ์เดียวกัน โดยการที่เราจะสร้าง employee ได้ 1 คน ต้องมี*

1. ชื่อ นามสกุล
2. อีเมล
3. ตำแหน่ง
4. โทเคน

แต่ถ้าเราอยากได้ employee แบบใหม่ที่มีบอกว่า*

1. พนักงานแต่ละคนขับรถได้ไหม
2. พนักงานมีความสามารถในการทำบัญชีไหม
3. ทำอาหารเก่งไหม

โดยแต่ละคนอาจจะไม่ต้องใส่ก็ได้เนื่องจาก บางครั้งการทำบัญชีได้ก็ไม่ได้ใช้กับเชฟทำอาหาร แต่ใช้แค่กับพนักงานบัญชี โดยถ้าหากต้องเขียนเป็นคลาสแต่ละคลาส จะได้ combination ของ class ทั้งหมดที่ต้องสร้างคือ จำนวน combination = $3C0 + 3C1 + 3C2 + 3C3 = 1+3+3+1 = 8$ คลาสที่ต้องสร้างต่างกัน

3C0 ไม่เลือกเลยทุกoption มีแค่ 1 กรณี

1. คลาสที่มีแต่ mandatory attribute

3C1 เลือกแค่ตัวoptionตัวใดตัวนึงจาก 3 ตัว มี 3 กรณี คือ

1. ขับรถได้ไหม
2. ทำบัญชีได้ไหม
3. ทำอาหารเก่งไหม

3C2 เลือกตัว option มา 2 ตัวจาก 3 ตัว มี 3 กรณี

1. ขับรถได้ไหม, ทำบัญชีได้ไหม
2. ทำบัญชีได้ไหม, ทำอาหารเก่งไหม
3. ทำอาหารเก่งไหม, ขับรถได้ไหม

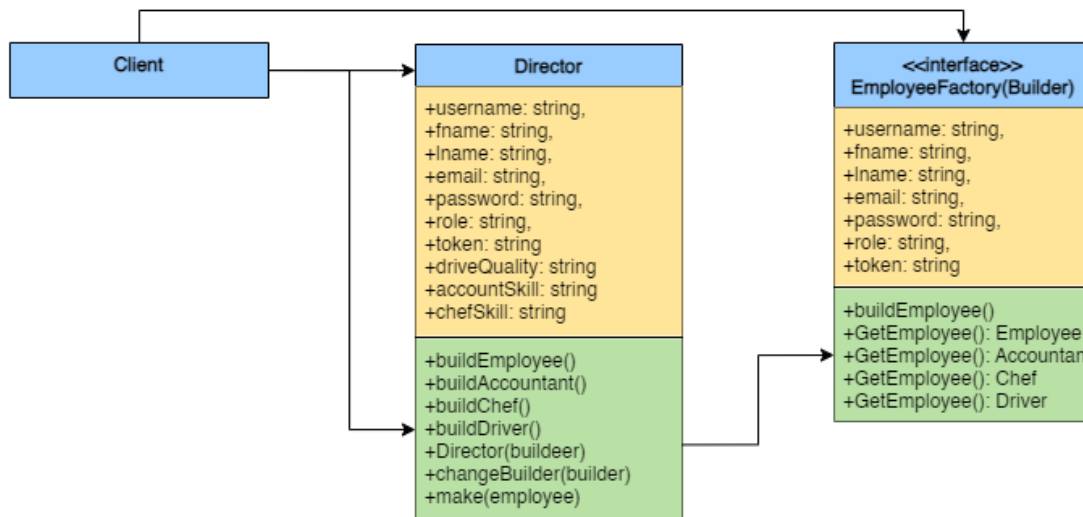
3C3 เลือกทุก option มีกรณีเดียว

1. ขับรถได้ไหม, ทำบัญชีได้ไหม, ทำอาหารเก่งไหม

ซึ่งถือว่าเยอะมาก ยกตัวอย่างเช่นประมาณนี้

```
public Employee (...Mandatory..Attribure...){ ...}  
public Employee (...Mandatory..Attribure..., String accountSkill){ ...}  
public Employee (...Mandatory..Attribure..., String canDrive){ ...}  
public Employee (...Mandatory..Attribure..., String chefSkill){ ...}  
public Employee (...Mandatory..Attribure..., String canDrive, String accountSkill){ ...}  
public Employee (...Mandatory..Attribure..., String accountSkill, String chefSkill){ ...}  
public Employee (...Mandatory..Attribure..., String chefSkill, String canDrive){ ...}  
public Employee (...Mandatory..Attribure..., String chefSkill, String accountSkill, String canDrive){ ...  
}
```

ซึ่งจะเห็นได้ว่านี่คือปัญหาของเรา ดังนั้นเราจึงนำ Builder Design มาใช้



Ref: <https://medium.com/coding-becomes-easy/creational-design-pattern-builder-pattern-1aef7cd052f8>

โดยทำการสร้าง EmployeeBuilder class มาข้างใน Employee class และจุดประสงค์คือการสร้าง object employee แบบที่ไม่ต้องประกาศ constructor ของ employee ข้างนอกเลย ซึ่งเราทำให้เป็น public ดังนั้นเวลาเราจะสร้าง employee ใหม่ เราเพียงแค่สร้างผ่าน builder และส่ง parameter ที่เราต้องการลงไป

```

public class Employee {

    private String name;
    private String driveQuality;
    private String accountSkill;
    private String chefSkill;

    // No setters defined , only getters are defined to retain immutability
    public String getName() {
        return name;
    }

    public String getdriveQuality() {
        return driveQuality;
    }

    public String getAge() {
        return accountSkill;
    }

    public String getchefSkill() {
        return chefSkill;
    }

    private Employee(EmployeeBuilder builder) {
        this.name = builder.name;
        this.driveQuality = builder.driveQuality;
        this.accountSkill = builder.accountSkill;
        this.chefSkill = builder.chefSkill;
    }

    @Override
    public String toString() {
        return "Employee{" +
            "name='" + name + '\'' +
            ", driveQuality='" + driveQuality + '\'' +
            ", chefSkill='" + chefSkill +
            ", accountSkill='" + accountSkill + '\'' +
            '}';
    }

    public static class EmployeeBuilder {
        private String name;
        private String driveQuality;
        private String chefSkill;
        private String accountSkill;

        public EmployeeBuilder name(String name) {
            this.name = name;
            return this;
        }

        public EmployeeBuilder driveQuality(String driveQuality) {
            this.driveQuality = driveQuality;
            return this;
        }

        public EmployeeBuilder chefSkill(String chefSkill) {
            this.chefSkill = chefSkill;
            return this;
        }

        public EmployeeBuilder accountSkill(String accountSkill) {
            this.accountSkill = accountSkill;
            return this;
        }

        public Employee build() {
            Employee employee = new Employee(this);
            return employee;
        }
    }
}

```



```

public class Employee {

    private String name;
    private String driveQuality;
    private String accountSkill;
    private String chefSkill;

    // No setters defined , only getters are defined to retain immutability
    public String getName() {
        return name;
    }

    public String getdriveQuality() {
        return driveQuality;
    }

    public String getAge() {
        return accountSkill;
    }

    public String getchefSkill() {
        return chefSkill;
    }

    private Employee(EmployeeBuilder builder) {
        this.name = builder.name;
        this.driveQuality = builder.driveQuality;
        this.accountSkill = builder.accountSkill;
        this.chefSkill = builder.chefSkill;
    }

    @Override
    public String toString() {
        return "Employee{" +
            "name='" + name + '\'' +
            ", driveQuality='" + driveQuality + '\'' +
            ", chefSkill='" + chefSkill + '\'' +
            ", accountSkill='" + accountSkill + '\'' +
            '}';
    }

    public static class EmployeeBuilder {
        private String name;
        private String driveQuality;
        private String chefSkill;
        private String accountSkill;

        public EmployeeBuilder name(String name) {
            this.name = name;
            return this;
        }

        public EmployeeBuilder driveQuality(String driveQuality) {
            this.driveQuality = driveQuality;
            return this;
        }

        public EmployeeBuilder chefSkill(String chefSkill) {
            this.chefSkill = chefSkill;
            return this;
        }

        public EmployeeBuilder accountSkill(String accountSkill) {
            this.accountSkill = accountSkill;
            return this;
        }

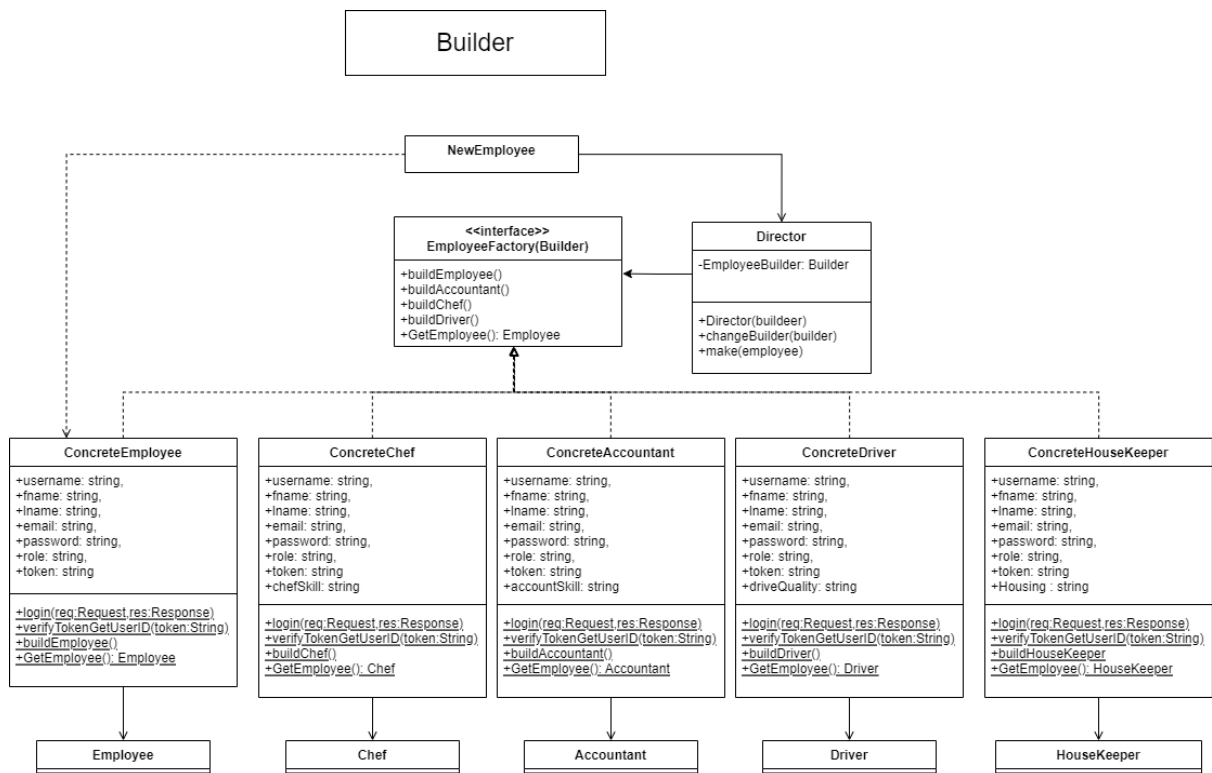
        public Employee build() {
            Employee employee = new Employee(this);
            return employee;
        }
    }
}

```


จะได้ผลลัพธ์ประมาณนี้

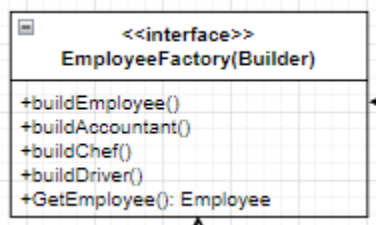
```
"C:\Program Files\Java\jdk-19\bin\java.exe" "-javaagent:D:\IntelliJ IDEA Community Edition 2022.2.3\lib\idea_rt.jar=5000:D:\IntelliJ IDEA Community Edition 2022.2.3\bin" -Dfile.encoding=UTF-8
Employee{name='Susan ploenjit', driveQuality='null', chefSkill=null, accountSkill='Spectacular'}
Employee{name='Susan prompong', driveQuality='null', chefSkill=Michalin5star, accountSkill='Null'}
Employee{name='Suksan', driveQuality='presidentDriverQuality', chefSkill=Null, accountSkill='Null'}
```

ใช้อย่างไร

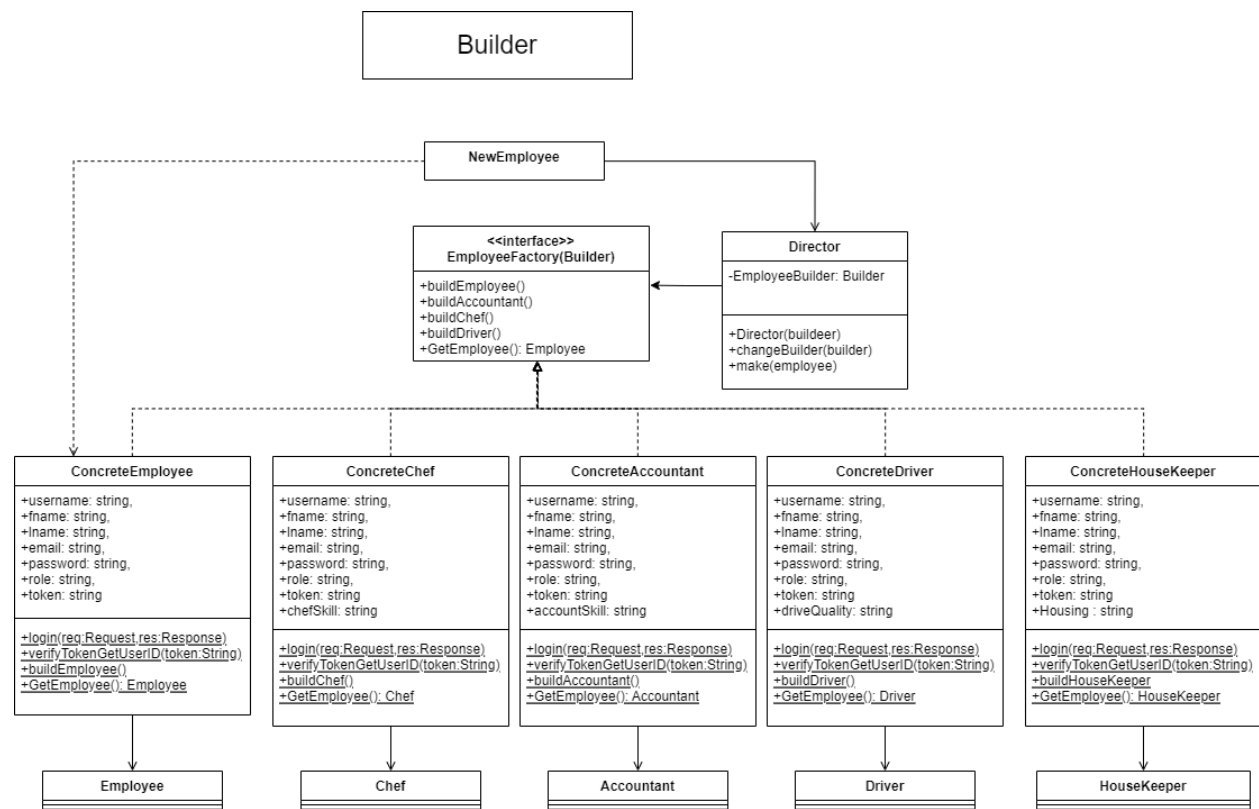


ภาพรวมที่นำมาประยุกต์

สิ่งแรกที่เราสนใจคือ Product (ในที่นี้คือ Employee, Chef, Accountant, Driver)



สร้าง class EmployeeFactory(Builder) โดยพวก options ต่าง ๆ เราสามารถเพิ่มเข้าไปโดยใช้ชื่อ buildEmployee() และสุดท้ายเพิ่มฟังก์ชันสร้าง product จริง ๆ (GetEmployee():Employee)



คลาสที่สร้าง product จะต้องมืคลาสที่บอกรายละเอียดในการสร้าง product ต่าง ๆ อยู่ เราใช้ชื่อว่า Concrete.. ซึ่งถ้าเรามี product หลาย ๆ แบบก็จะมี ConcreteBuilder หลาย ๆ ตัว (ในที่นี้ ConcreteEmployee, ConcreteChef, ConcreteAccountant, ConcreteDriver)

ใช้แล้วมีผลดียังไง (ช่วยอะไร)

- ช่วยลดการบวมในแถบ parameters ที่ส่งไปบางทีก็ไม่จำเป็นต้องส่งไปได้สำหรับบางกรณี
- ลดความซ้ำซ้อนในขั้นตอนการสร้าง employee class เนื่องจากขั้นตอนในการสร้างที่ตายตัว เช่น ConcreteAccountant Class ก็มีแค่ parameters พวกนี้ ดังนั้นแทนที่เราจะเขียนโค้ดเดิมซ้ำๆ เราก็นำโค้ดพวกนั้นไปไว้ใน director เพื่อลดการเขียนโค้ดซ้ำได้ แลมนโค้ดนั้นสามารถผลิต product ที่มีขั้นตอนการสร้างแบบเดียวกัน และถ้าหากจะเพิ่มก็แค่ทำการเพิ่ม builder style อีกแบบเข้าไป

Housekeeping ใช้ Design Pattern Builder

ทำไมถึงใช้

เนื่องจาก construction ของ class Filter มี parameter เยอะและบางตัวไม่จำเป็นต้องระบุจึงใช้ Design Pattern Builder มาแก้ไขปัญหามาเพื่อให้สามารถ set เพียง parameter ที่สนใจได้

ใช้อย่างไร

ใช้เพื่อ build Filter object เพื่อที่จะทำไป query housekeeping task ตามเงื่อนไขที่ต้องการ

ใช้แล้วมีผลดียังไง (ช่วยอะไร)

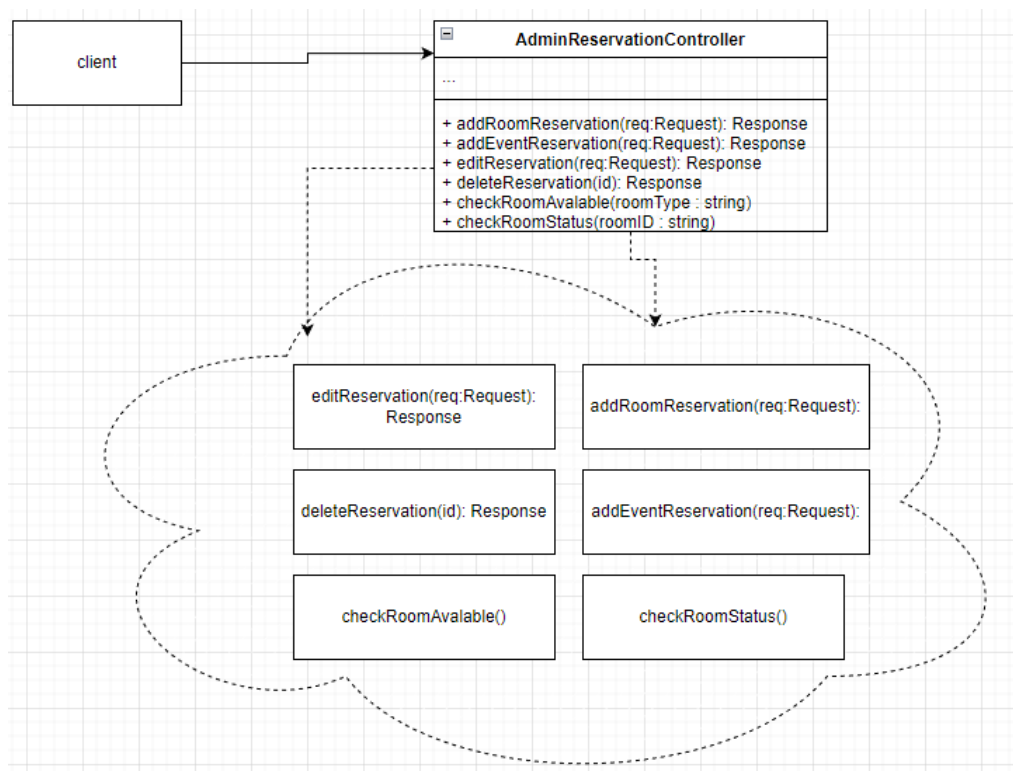
ทำให้ช่วยลดความยุ่งยากในการสร้าง Filter object เพราะสามารถกำหนด parameter ได้อย่างอิสระและเพิ่ม readability ของโค้ด

Design Pattern Facade

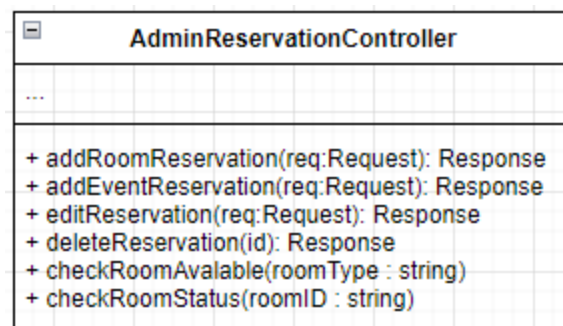
ทำไมถึงใช้

เนื่องจากเวลาที่ลูกค้ามาถึงหน้า counter ต้องการที่จะจองห้องพัก ซึ่งปกติแล้วถ้าหากลูกค้าจองผ่านเว็บไซต์ลูกค้าจะต้องระบุว่ามีแขกกี่ท่าน, ตั้งการจองว่าจะเข้าพักวันไหนถึงวันไหน, อยากแก้ไขการจองไหมหรืออยากยกเลิกการจองหรือเปล่า, นั่งเลือกราคาห้อง, เลือกประเภทห้องเอง เป็นต้น ซึ่งจะเห็นได้ว่ามีความวุ่นวายมาก ถ้าหากลูกค้ามาที่หน้า counter แล้วต้องกดแต่ละขั้นตอนด้วยตัวเอง ทั้งๆที่มีพนักงานอยู่ข้างหน้าทุกๆ ดังนั้นเพียงแค่พนักงานทำการบอกรายละเอียดข้อมูลห้อง, ราคา, สอบถามว่าเข้าพักถึงวันไหน และให้ลูกค้าบอกกับเรามาตามคำถามข้างต้น ให้พนักงานไปทำงานต่อเอง เราจึงนำ Facade เข้ามาช่วย เมื่อลูกค้าจะมาจองห้อง เขาแค่เรียก AdminReservationController ของ Facade ก็พอ ส่วนที่เหลือให้เป็นหน้าที่ของพนักงานเราจัดการต่อเอง

ใช้อย่างไร



Facade pattern ของ Admin Reservation



เราได้ทำการสร้าง AdminReservationController interface ที่จะใช้ฟังก์ชันต่างๆ โดยข้างในนั้นจะมี method 6 method อยู่ในนั้น

+ addRoomReservation(req:Request): Response

+ addEventReservation(req:Request): Response

+ editReservation(req:Request): Response

- + deleteReservation(id): Response
- + checkRoomAvalable(roomType : string)
- + checkRoomStatus(roomID : string)

ใช้แล้วมีผลดียังไง (ช่วยอะไร)

ข้อดี คือ โค้ดเราไม่ไปผูกติดกับ subsystem ที่มีความวุ่นวาย

ข้อเสีย คือ Facade จะเป็น God object เข้าถึงได้ทุกอย่าง รู้ทุกอย่าง มี coupling สูง

Quality attribute scenarios

Availability

เมื่อ User ต้องการที่จะเรียกใช้ฟังก์ชัน ไต ๆ จากระบบ ผลการตอบรับที่ควรจะได้ คือระบบควรจะตอบสนองการเรียกใช้ฟังก์ชันที่เรียกโดย user ได้ทันที ซึ่งจะวัดผลจากระยะเวลาที่ระบบใช้ในการตอบสนอง

Portion of Scenario	Possible values
Source	User
Stimulus	The user calls a function
Artifact	The system
Environment	The system under normal load
Response	The system should respond to the function called by the user executed.
Response Measure	Response time

Integrability

ในอุดมคติ การควบรวมระบบ เราจะใช้กับการที่เราทำ domain ต่าง ๆ มาเป็น independently ให้การทำงานเป็นแบบเดี่ยว แต่เราจะทำการให้แต่ละ domain สามารถเข้าถึงอีก domain ได้โดยการใช้ integrability

Developer ได้ทำการเพิ่ม customer booking ในหน้าเว็บไซต์ จากการทำในโค้ดซึ่งจะได้ผลการตอบรับจากระบบคือ form ที่มีไว้ทำการจองสามารถมีส่วนเกี่ยวข้องกับระบบ Hotel management อันเก่าของเราได้

Portion of Scenario	Possible values
Source of stimulus	Developer
Stimulus	Add the customer booking form in user interface
Artifact	Code
Environment	Development
Response	New booking form can be relate on the hotel management system
Response Measure	Response time

Modifiability

Developer ต้องการที่จะเปลี่ยนหน้าตาของ UI ซึ่งปกติต้องเข้าไปยุ่งเกี่ยวกับโค้ดมหาศาล แต่เราได้ทำการวางโครงสร้างโค้ดมาเป็นอย่างดีทำให้การแก้โค้ดของเราไม่กระทบต่อโค้ดทุกส่วน

Portion of Scenario	Possible values
Source of stimulus	Developer
Stimulus	Want to change user interface
Artifact	Code
Environment	At design time
Response	Modification is made with no side effect
Response Measure	Effort, number of affected artifacts

User ต้องการที่จะปรับเปลี่ยนข้อมูลการจอง (อยากเลื่อนวันจอง)จากในเว็บ ในเว็บจะตอบสนองต่อการเปลี่ยนแปลงการจองนี้ในฐานข้อมูล โดยข้อมูลอื่น ๆ ในฐานข้อมูลจะไม่ได้รับผลกระทบใด ๆ

Portion of Scenario	Possible values
Source of stimulus	User
Stimulus	Modify data for reservations
Artifact	System (Website)
Environment	The system under normal load
Response	Modification without affecting other functionality.
Response Measure	period to change without affecting

Performance

Admin(พนักงาน) ต้องการที่จะตรวจสอบการจองภายใต้การทำงานที่ปกติของระบบ และขณะที่ server กำลังเกิด overload ผู้ใช้งานเยอะมาก โดยระบบก็ไม่มีควมล่าช้าแต่อย่างใด และสามารถส่งข้อมูลที่พนักงานต้องการกลับมาได้

Portion of Scenario	Possible values
Source of stimulus	User(admin)
Stimulus	Want to check booking information
Artifact	System
Environment	normal mode, overload mode
Response	the information can be displayed in entirely
Response Measure	Latency

Usability

การใช้งานง่าย ลูกค้าต้องการที่จะจองห้องพัก เพียงแค่ลูกค้าใส่ข้อมูลต่าง ๆ ไป

กด submit จากนั้นระบบก็จะประมวลผลรับทราบการจองนั้นและทำสำเร็จในไม่ช้า และจะมีการส่งอีเมลล์ไปแจ้งเตือนลูกค้า

Portion of Scenario	Possible values
Source of stimulus	End user
Stimulus	Want to booking room
Artifact	System
Environment	Runtime, System configuration time
Response	The system informs that the reservation is successful
Response Measure	It takes less than 30 minutes

Portion of Scenario	Possible values
Source of stimulus	End user
Stimulus	Want to cancel booking room
Artifact	System
Environment	At runtime
Response	The system informs us that the reservation is canceled successfully.
Response Measure	Cancellation operation take less than 2 seconds

Testability

Dev ต้องการที่จะทำ unit test เพื่อตรวจสอบโค้ดที่เขียนมามีโค้ดส่วนไหนใช้งานไม่ได้บ้าง ซึ่งได้ผลลัพธ์ออกมาว่า การทำงานของระบบเป็นปกติ และมี Path coverage 85% ถือว่ามีการเขียนโค้ดที่สุญเปล่า้น้อยมาก และใช้เวลาไม่นาน

Portion of Scenario	Possible values
Source of stimulus	Developer
Stimulus	Performs unit test
Artifact	Component of a system
Environment	Completion of the component
Response	Perform a test sequence
Response Measure	Path coverage of 85% is achieved within 30 minutes

Reference

Software Architecture, <https://www.mongodb.com/mern-stack>

Design pattern, <https://refactoring.guru/design-patterns>