1. Varför dessa mönster och hur de förbättrar lösningen

Factory Method

Varför?

- Vi vill kunna skapa olika typer av quiz (Multiple-Choice, True/False, Fill-in-the-blank, etc.) utan att koda om på flera ställen när nya quiztyper tillkommer.
- Vi slipper hårdkoda exakt *vilken* quizklass som ska instansieras överallt.

Förbättring:

- Ökad **extensibilitet**: Lätt att lägga till nya quiztyper.
- **Enhetlig skapandeprocess**: All quiz-instansiering sker via en enda "fabrik"-metod, vilket gör koden mer underhållbar.

Strategy

Varför?

- Olika sätt att räkna ut poäng eller bedömningar, t.ex. tidbaserat eller baserat på antal rätt.
- Vi kan dynamiskt byta algoritm beroende på situation (t.ex. svårighetsgrad eller kampanjregler).

Förbättring:

- **Flexibilitet**: Vi kan enkelt plugga in en ny poängberäkningsstrategi utan att ändra övrig kod.
- **Löskoppling**: Klienten bryr sig inte om *hur* poängen räknas, bara att ett "Strategy"-objekt utför beräkningen.

Observer

Varför?

- När en händelse (t.ex. en student klarar en kursmodul) inträffar, kan flera delar av systemet behöva reagera (skicka mail, tilldela badge, uppdatera leaderboard).
- Vi vill inte att huvudlogiken ska behöva veta exakt vilka andra komponenter som ska uppdateras.

Förbättring:

- **Löskoppling mellan avsändare och mottagare**: Koden som triggar händelsen (t.ex. “kurs slutförd”) behöver inte känna till alla beroenden.
- **Skalbarhet**: Det är enkelt att lägga till nya “Observers” (t.ex. SMS-notifiering) utan att röra grundlogiken.

---

2. Exempel – Use Case & User Story

2.1. Factory Method – Exempel

Use Case (kort version)

- **Namn**: Skapa en ny quiz (Multiple-Choice eller True/False).
- **Aktör**: Lärare (Teacher).
- **Händelseförlopp**:
    1. Läraren går in på “Skapa quiz”-sidan.
    2. Läraren väljer quiz-typ från en dropdown (t.ex. “Multiple-Choice”).
    3. Systemet anropar en fabriksmetod `QuizFactory.createQuiz(type)` som returnerar en instans av korrekt quizklass.
    4. Läraren fyller i frågor och svar och sparar.

User Story

> Som lärare vill jag kunna välja vilken typ av quiz jag skapar, så att jag kan anpassa frågorna efter kursens behov.

2.2. Strategy – Exempel

Use Case (kort version)

- **Namn**: Beräkna slutpoäng för en quiz.
- **Aktör**: System (bakgrundsprocess) när student slutför quiz.
- **Händelseförlopp**:
    1. En student skickar in sina svar.
    2. Systemet kontrollerar vilken poängstrategi som är vald för kursen (t.ex. “Tidbaserad”).
    3. Systemet anropar strategins metod `calculate_score(...)`.
    4. Systemet sparar resultatet i databasen.

User Story

> Som systemadministratör vill jag kunna konfigurera olika poängberäkningsstrategier för olika kurser, så att poängsättningen kan anpassas efter kursens behov.

2.3. Observer – Exempel

Use Case (kort version)

- **Namn**: Tilldela Badge när student slutför modul.
- **Aktör**: Student fullföljer en modul (händelsen).
- **Händelseförlopp**:
    1. En student blir “klar” med en modul (ex: quiz eller uppgift).
    2. Systemet “notifierar” alla observers (t.ex. BadgeSystem, EmailNotifier).
    3. BadgeSystem kollar ifall en ny badge kan utdelas.
    4. EmailNotifier skickar ut ett gratulationsmail.

User Story

> Som student vill jag automatiskt tilldelas badges och få notifikationer när jag slutför en modul, så att jag känner mig motiverad att fortsätta.

Sammanfattning

- **Factory Method** låter oss enkelt skapa nya typer av quiz utan att duplicera kod.
- **Strategy** gör poängberäkning flexibel och utbytbar beroende på kursens krav.
- **Observer** underlättar notifieringar och reaktioner på händelser (som att slutföra en modul eller quiz) utan att koppla ihop komponenter hårt.

Dessa mönster tillsammans skapar en mer skalbar och underhållbar arkitektur för en “Gamified E-Learning”-applikation.