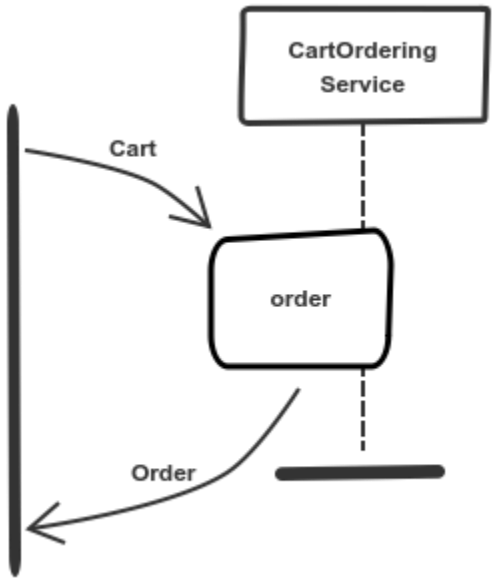
BUSUIOC Bianca , an IV , IS

TEMA 2 – Domain driven design – factory

DDD (Domain-driven design) este o abordare a dezvoltării software pentru nevoi complexe prin conectarea implementării la un model în evoluție. Domain-driven design nu este o tehnologie sau o metodă, ci un set de reguli și priorități. Pentru a realiza o dezvoltare rapidă a aplicației folosind domain-driven design, avem de-a face cu o serie de principii de design, design patterns și best practices.

Factory este un obiect care are singura responsabilitate de a crea alte obiecte.
Modelul DDD utilizează factory cu scopul de a încapsula logica creării de obiecte și agregate complexe. Factories nu sunt unice pentru DDD, dar joacă un rol important în cadrul uni proiect de proiectare.



Atunci când se creează o metodă factory în aggregate root, aceasta trebuie să fie una și nedivizată.Valorile obiectelor și entităților sunt create diferit. Deoarece valorile sunt nemodificabile, toate atributele trebuie transmise imediat după ce sunt create și se pot crea numai atribute specifice entităților.

Există trei tipuri de factory :

1. **Factory method :** este un model de design creator care oferă o interfață pentru crearea obiectelor în superclase, dar permite subclaselor să modifice tipul de obiecte care vor fi create.

Exemplu : Să ne imaginăm că avem o aplicație de management în care prima versiune se ocupă numai de transportul cu camioane,astfel cea mai mare parte a codului este într-o clasa Camioane. După un timp , ne dorim să ne mărim aria și vrem transport maritim , pe lângă cel cu camioane. Factory method sugerează înlocuirea creării directe a obiectului (folosind un nou operator) cu un apel la o metodă specială din " factory". Apelul constructorului trebuie mutat în interiorul metodei.Obiectele returnate de factory method sunt denumite produse.

Avantaje :

- nu returnează neapărat un obiect al clasei în care a fost aplelată. Adesea, acestea ar putea fi subclasele sale, selectate pe baza argumentelor date metodei
- poate returna un obiect deja creat, spre deosebire de un constructor, care creează întotdeauna o nouă instanță

2. **Abstract factory class :** folosită atunci când trebuie să determinăm de ce fel de obiect specific are nevoie clientul. Abstract factory class încapsulează atât logica deciziei, cât și logica creării.

Exemplu : Să ne imaginăm că avem un magazin de mobilă. În care codul aplicației conține familie de produse : canapele,scaune și fiecare produs este disponibil în mai multe variante: Ikea,VictorianStyle,ArtDeco. Avem nevoie de o modalitate de a crea obiecte individuale de mobilier astfel încât să se potrivească cu alte obiecte din aceeași familie. Dar nu dorim să modificăm codul existent de fiecare dată când adăugăm produse sau familii de produse. Astfel, Abstract factory sugerează să se forțeze variantele să urmeze interfețe comune : toate variantele pentru scaune trebuie să urmeze interfețele scaunelor și la fel pentru canapele . Apoi se va crea o interfață de bază care declară metodele pentru crearea tuturor produselor care fac parte dintr-o familie de produse. Metodele vor returna tipurile produselor abstracte reprezentate de interfețe : scaun,canapele. După se va implementa concret un factory, care returnează produse de un anumit tip : IkeaFactory se va rferi doar la obiectele IkeaScaun și IkeaCanapele. Codul trebuie să lucreze cu factory numai prin intermediul interfețelor.

3. **Builder class :** este un model de design care permite realizarea diferitelor tipuri și reprezentări ale unui obiect utilizând același proces de construcție. Permite construirea de obiecte compelxe pas cu pas.

De ce să folosim factory?

Pe măsură ce aplicațiile cresc în complexitate, factories devin mai frecvente.
Avantaje ale folosirii de factory :

- Standardizează instanțierea unui obiect – Factories oferă o modalitate standard de a instanția obiecte, astfel încât crearea de obiecte noi nu este pierdută în ambiguitate.
- Asigură încapsularea – Factory știe foarte multe despre structura internă și dependențele obiectului,dar va proteja informațiile de lumea exterioară prin

furnizarea de interfețe care să reflecte obiectivele cleintului și o vedere abstractă a obiectului creat.

- Obiectele trebuie să fie responsabile de creația lor - Puterea reală din spatele programării orientate pe obiecte este încapsularea logicii interne a obiectului. Un client ar trebui să poată utiliza un obiect fără a fi vreodată preocupat de implementarea internă a obiectului respectiv.Obiectele ar trebui să aibă o singură responsabilitate, astfel încât complexitatea unui obiect să fie distilată până când nu rămâne nimic care nu se referă la semnificația sa.

Factory și serviciile sunt similare în sensul că ambele iau un obiect și scot un obiect. Factories sunt un tip de serviciu. Factories întotdeauna scot un obiect nou, în timp ce serviciile nu fac neapărat acest lucru. Cu factories, facem o singură cerere și obținem un obiect afară, ieșirea fiind previzibilă. Un serviciu ar trebui să se ocupe de factory și factory ar trebui să se ocupe de logica creației.