E-poe arendamisel soovitan kasutada järgmiseid disainmustreid, mis sobivad hästi Java back-end'i ja TypeScript front-end'i arhitektuuriga:

**Back-end disainmustrid**

1. **Repository Pattern** (juba sisaldub Spring Data JPA-s)
   * Eraldab andmepöörduskihi äriloogikast
   * Lihtsustab andmebaasi operatsioonide testimist
   * Võimaldab kergemini vahetada andmebaasi tehnoloogiaid
2. **Service Layer Pattern**
   * Kapseldab äriloogika eraldi kihti
   * Võimaldab transaktsioonide haldust ühes kohas
   * Tagab, et kontrollerid tegelevad ainult päringu töötlemise, mitte äriloogikaga
3. **DTO (Data Transfer Object) Pattern**
   * Eraldab andmebaasi mudelid API vastustest
   * Võimaldab täpselt kontrollida, millist infot API kaudu jagatakse
   * Aitab vältida andmete lekkimist ja üleliigset infot vastustes
4. **Builder Pattern**
   * Kasulik keerukate objektide, nagu tellimused, loomiseks
   * Eriti hästi toimib Lombok @Builder annotatsiooniga
5. **Strategy Pattern**
   * Kasulik erinevate makseviiside, saadetiste jälgimise või hindade arvutamise loogika puhul
   * Võimaldab lisada uusi strateegiaid koodi muutmata

**Front-end disainmustrid**

1. **Container/Presentational Component Pattern**
   * Eraldab andmehalduse (Container) ja UI-ga (Presentational) seotud komponendid
   * Container komponendid kasutavad Redux'i või konteksti, Presentational komponendid saavad andmed props'idena
   * Lihtsustab UI komponentide taaskasutamist ja testimist
2. **Flux/Redux Pattern** (Redux Toolkit'iga)
   * Ühesuunaline andmevoog (one-way data flow)
   * Keskne olekuhaldus (state management)
   * Ennustatav olekuhaldus läbi puhaste funktsioonide (reducers)
3. **Custom Hook Pattern**
   * Eraldab logika taaskasutatavate hookide sisse
   * Näiteks useCart, useAuth, useProducts jne.
   * Vähendab komponendi koodi keerukust
4. **HOC (Higher-Order Component) Pattern** või **Render Props Pattern**
   * Kasulik ühise funktsionaalsuse (nt autentimise kontroll) jagamiseks komponendite vahel
   * React uuenduste järel sageli asendatud hook'idega, kuid endiselt kasulikud teatud juhtudel

**Arhitektuuriline lähenemine**

Soovitan kasutada **Clean Architecture** põhimõtteid, mis kombineerivad need mustrid:

1. **Domain-Driven Design (DDD)**
   * Struktureerib koodi äridomeeni mudelite ümber
   * Defineerib selged piirid erinevate alamdomeenide vahel
   * Sobib hästi e-poe funktsionaalsuse (tooted, tellimused, kasutajad) loogiliseks eraldamiseks
2. **Hexagonal Architecture / Ports and Adapters**
   * Eraldab rakenduse tuuma (äriloogika) välisest infrastruktuurist (andmebaas, API)
   * Võimaldab süsteemi osi iseseisvalt testida ja arendada
   * Muudab süsteemi paindlikumaks muudatuste suhtes
3. **Microservices Pattern** (juhul kui rakendus on ulatuslikum)
   * Eraldab funktsionaalsuse (näiteks tootehaldus, tellimused, maksed) eraldi teenusteks
   * Võimaldab skaleerida üksikuid komponente vastavalt vajadusele
   * Spring Boot on suurepärane valik mikroteenuste loomiseks

Alustades väiksema rakendusega, soovitan alguses keskenduda **Repository**, **Service Layer**, **DTO** ja **Container/Presentational Component** mustritele, mis annavad hea struktuuri, ning seejärel vastavalt vajadusele lisada teisi mustreid, kui rakendus kasvab.

Kas soovid mõne konkreetse mustri või arhitektuurilise lähenemise kohta täpsemat infot?