Sažetak teorije OOBL – 2.ciklus

"Standardna" arhitektura

- Podjela na:
  - Prezentacijski sloj
  - Aplikacijski servisi (*application layer*)
  - *Domain layer*
  - Tehnički servisi (*infrastructure layer*)

- Prezentacijski sloj
  - Prikazuje informacije korisnicima i interpretira njihove komande

- Aplikacijski sloj
  - Definira što bi softver trebao raditi i upravlja objektima u domeni prilikom obavljanja tog posla
  - Koordinira zadatke i delegira posao na kolaboracije objekata u sloju ispod

- *Domain layer*
  - Odgovoran za reprezentiranje osnovnih koncepata sustava, informacija o njegovom stanju i skupa njegovih pravila
  - Opisuje i kontrolira stanje objekata, iako su tehnički detalji njegovog pohranjivanja delegirani u niži sloj
  - **Ovaj sloj predstavlja dušu softvera !!!**

- *Infrastructure layer*
  - Pruža generičke tehničke mogućnosti drugim slojevima
    - Slanje poruka
    - Perzistencija objekata u baze podataka
    - Iscrtavanje UI elemenata

16

Prednosti i nedostaci poslovne logike u korisničkom sučelju!

Prednosti

- Visoka produktivnost (odmah se vide rezultati)
- I manje vješti developeri moru raditi na ovaj način
- Nedostaci u zahtjevima se mogu nadići brzom izgradnjom prototipa koji se prezentira korisnicima (dobije se brzi *feedback* ako nešto ne valja)
- Relacijske baze rade dobro i osiguravaju integraciju na nivou podataka
- 4GL alati rade dobro
- Kad se aplikacije predaju na korištenje, programeri za održavanje su u stanju brzo preurediti dijelove koje inicijalno ne razumiju jer su promjene lokalizirane u određenom dijelu UI sučelja

Nedostaci

- Teško se obavlja **integracija aplikacije**, osim kroz bazu podataka
- **Nema ponovne iskoristivosti** i **nema apstrakcija** poslovne domene. Poslovna pravila se moraju duplicirati u svakoj operaciji na koju se primjenjuju
- Rapidna izgradnja prototipova dostiže prirodni limit jer nedostatak apstrakcija limitira opcije za *refactoring*
- **Složenost vas pokopava vrlo brzo**, sistem raste samo kroz dodavanje novih jednostavnih aplikacija. Nema jednostavnog puta do bogatijeg ponašanja

Entity i Value Object

- Objekt koji je primarno definiran svojim identitetom -> **Entity**
- Što je **value object** ?
  - Objekt kod kojega nas zanimaju samo **vrijednosti atributa**
  - "Objekt koji predstavlja opisni aspekt domene bez konceptualnog identiteta"

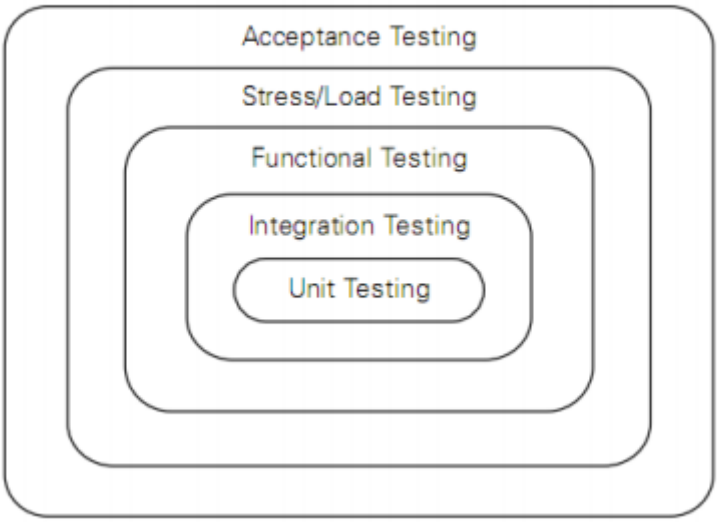
- ***Aggregate*** – *cluster* povezanih objekata koji tretiramo kao jedinku prilikom promjene podataka

- Definiraju ga:
  - Korijen *(root)*– jedan, točno određeni *entity* objekt koji je dio agregata
    - To je jedini dio agregata na koji vanjski objekti smiju imati referencu !
  - Granica – definira što se nalazi unutar agregata

- ***Factory*** – programski element koji je odgovoran za kreiranje drugih objekata
  - Enkapsulira potrebno znanje za kreiranje objekta na jednom mjestu

- ***Repository*** – konceptualni okvir koji enkapsulira rješenje za dohvaćanje objekata
- Repository – djeluje kao kolekcija, ali sa poboljšanim mogućnostima postavljanja "upita"
  - Dodajemo i brišemo objekte prikladnog tipa, a mašinerija repozitorija se brine o interakciji s bazom podataka
  - Na jednom mjestu definiramo kohezivni skup odgovornosti za dohvaćanje objekata

Vrste testiranja



Unit testing

- Već smo ih vidjeli u primjerima
  - Namijenjeni testiranju funkcionalnosti malih ("jediničnih") dijelova izvornog kôda

Integration testing

- Testiramo kako komponente integriraju
  - Može biti intergacija na razini klase, modula, komponente ...

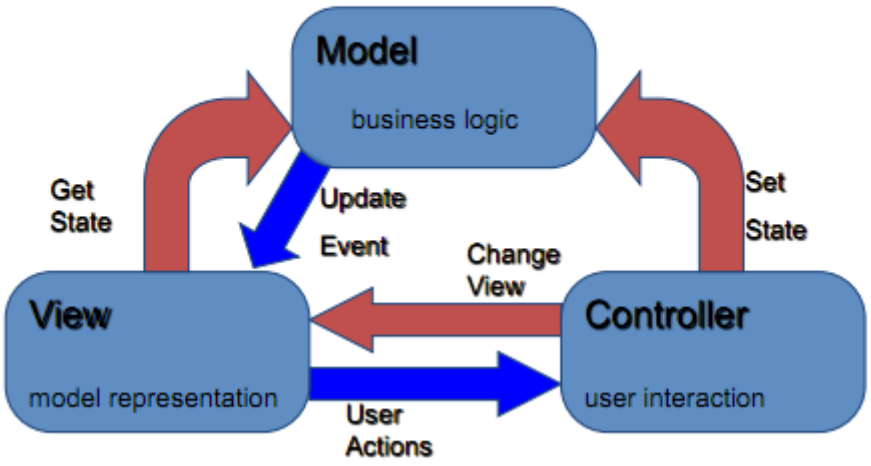
Functional testing

- Funkcionalno testiranje podrazumijeva ispitivanje kôda na granici njegovog javnog API-ja
- Efektivno, testiramo *use case*-ove aplikacije

Model-View-Controller (MVC)

- Model sadrži podatke koji predstavljaju problem (model domene kod DDD)
  - Naravno, nisu samo podaci već i **ponašanje**
    - Sadrži i pravila koja omogućavaju pristup do podataka i njihovo mijenjanje
    - Podaci su privatni, a postoje *accessor* (getter and setter) metode
  - ***I/O free!!!***
- Controller "odgovara" na korisnikove akcije "govoreći" Modelu kako da se promijeni
  - Prevodi interakcije s View-om u akcije koje se obavljaju u Modelu (a koje mogu i mijenjati stanje modela)
  - Ovisno o interakciji s korisnikom i rezultatima akcija u modelu, kontroler odabire odgovarajući view
- View prikazuje trenutno stanje Modela korisniku
  - Ne prikazuje što bi se **trebalo** događati u modelu!
    - Npr. ako se zatraži snimanje fajla, View ne bi trebao sam reći da je fajl snimljen, već bi trebao prenijeti poruku koja je "došla" iz Modela

MVC arhitektura



Model-View-Presenter (MVP)

- Dopušta da View prima input od korisnika!
  - Klasika za moderna GUI okruženja
  - .NET kontrole, npr., već imaju ugrađene *handlere* za razno razne *keybord & mouse events*
- ALI!!!! – View "prima" input i **samo delegira Controlleru** podatke dalje na obradu!
- Controller je "in charge" i govori View-u što da radi

- Probleme koji nastaju prilikom spremanja objekata u relacijske baze možemo ovako preciznije "klasificirati":
  - Problemi granularnosti
  - Problemi s podtipovima
  - Problemi s identitetom
  - Problemi asocijacija
  - Problemi s navigacijom po grafovima objekata
  - Troškovi

Problem *granularnosti*

- *Granularity* – relativna veličina objekata s kojima se radi

Problem *podtipova*

- Što ćemo s nasljeđivanjem?
  - Izvedene i bazne klase definiraju drugačije podatke i funkcionalnost
  - Kako ćemo tu hijerarhiju pospremiti u relacijske tablice?

Problem *identiteta*

- Kako provjeriti da li su dva objekta "jednaka" ?
  - Možemo provjeravati identitet objekta [==]
  - Možemo provjeravati jednakost vrijednosti [.equals()]
- SQL – ako su primarni ključevi isti, onda se radi o istim podacima
  - Ali, dva ili više objekata mogu predstavljati jedan te isti redak iz baze!!!???

Problem s *asocijacijama*

- Asocijacije predstavljaju odnos među entitetima
  - OO – realizacija preko referenci na objekte
  - SQL – preko *stranih ključeva* (*foreign key*)
- Reference na objekte su direkcionalne, strani ključevi nisu!

"Navigacija" među objektima

- Objektima se različito pristupa u OO i SQL paradigmi

```
obj.getDetails().getFirstField()
```

Suma sumarum – TO KOŠTA!

- Nadvladavanje OO-RDBMS *impedance mismatch*-a zahtijeva značajan trud
- Procjene – do 30 % kôda se troši na povezivanje OO i SQL-a

Vrste ORM-a

- Možemo navesti 4 varijante
  1. Čista relacijska:
     - Cijela aplikacija se dizajnira oko relacijskog modela, nema OO-a (a bazaši su presretni ;-)
     - Portabilnost?
  2. *Light object mapping*
     - Ručno kodiranje SQL/ODBC
     - Popularno, ali ...
  3. *Medium object mapping*
     - SQL generiran tijekom *build time*-a pomoću alata ili tijekom runtime-a korištenjem nekog framework-a
  4. Potpuno objektno orijentirani
     - Ima sve mogućnost modeliranja svega: kompozicija, nasljeđivanje, polimorfizam i prezistenciju

Prednosti ORM-a

- Produktivnost
- Jeftinije/lakše održavanje
- Performanse
  - O ovome ima različitih mišljenja ☺
- Nezavisnost o DBMS tehnologiji

Mapiranje hijerarhije nasljeđivanja

- "najvidljiviji" dio *impedance-mismatch* između OO i SQL paradigmi
- Tri moguća pristupa:
  - *Table per concrete class*—"izbacujemo" polimorfizam i nasljeđivanje iz relacijskog modela
  - *Table per class hierarchy*—omogućujemo polimorfizam denormalizacijom relacijskog modela uz korištenje *diskriminatorskog stupca* koji sadrži podatke o tipu
  - *Table per subclass*—reprezentira odnos "is a"(nasljeđivanje) kao "has a" (strani ključ) odnos

Table per concrete class

- Najjednostavnije – jedna tablica za svaku (ne-asptraktnu!) klasu

Table per class hierarchy

- Sve klase iz hijerarhije u jednu tablicu
  - Znači, svojstvo svake izvedene klase kao poseban stupac
  - Definira se *diskriminator* – određuje kojeg je točno tipa objekt u tom retku

Table per subclass

- Nasljeđivanje "modeliramo" stranim ključevima