# Domain Driven Design

## Uvod u Domain Driven Design

**Domain Driven Design (DDD)** predstavlja pristup u razvoju softvera koji se fokusira na razvoj **modela domena nekog problema.** Pritom se teži razumevanju procesa i pravila samog domena.

DDD pruža principe i šablone za rešavanje složenih problema. Tako se dobija **jasan, čist, testabilan kod** koji predstavlja neki domen. Međutim, glavni cilj nije pisanje koda, već **rešavanje problema**. To zahteva puno rada sa klijentima kako bi se shvatile njihove potrebe i razumeo domen problema. Neophodan je neprestani rad tima za razvoj softvera sa **domen ekspertima**-ljudima koji su stručnjaci za određeni domen.

Jedan od glavnih principa prilikom rada sa DDD-em je: **“Divide and conquer“** – “Zavadi pa vladaj“. Ovaj princip je takođe poznat i kao: “**Separation of Concerns**“ – razdvajanje odgovornosti. Odnosno, potrebno je podeliti domen na manje delova i fokusirati se na deo po deo. Tako se dobijaju takozvani subdomeni (**subdomain**). Svaki od njih je potrebno **modelovati** i rešiti problem sa kojim se on susreće.

Benefiti DDD-a su:

* fleksibilan softver (lako dodavanje novih funkcionalnosti)
* jasna putanja prilikom rešavanja kompleksnog problema
* čist kod
* kod lak za održavanje
* biznis logika je na jednom mestu
* korisni design pattern-i

Iako Domain Driven Design pruža navedene benefite, treba se jako pažljivo koristiti. Nije uvek najbolje rešenje koristiti DDD. Kao što i sam **Eric Evans**, začetnik Domain Driven Desgn-a kaže, DDD se treba koristiti samo kod kompleksnih problema gde se i mogu uvideti pravi benefiti DDD-a. U suprotnom, rad sa DDD-em može postati **overengineering**-razvoj proizvoda koji je bespotrebno ukomplikovan.

S obzirom da je DDD jedan jako kompleksan pristup, postoji nešto što se zove **DDD Mind Map**. On prikazuje koncepte DDD-a, njihovu interakciju i položaj. Mind Map-u je moguće videti na sledećoj slici:

A diagram of a company

Description automatically generated

## Bounded context

Bounded context predstavlja centralni pattern Domain Driven Design-a. Prilikom rešavanja kompleksnih problema, model podataka može postati jako veliki. Zbog toga DDD deli veliki model na više različitih delova, a svaki deo se zove **Bounded context**. To je zapravo granica (boundary) u okviru domena gde se konkretni domenski model primenjuje. Bounded context sprečava širenje nekih koncepta modela tamo gde im nije mesto (u druge bounded context-e).

Najpravilnije korišćenje DDD-a bi značilo da svaki Bounded context ima različite podatke, kod, kao i tim koji radi na razvoju modela jednog Bounded contexta. Međutim, u praksi se jako retko dešava da postoji baš ovoliki nivo odvajanja.

**Bounded context i subdomain** su dosta slični, ali opet dosta različiti koncepti. Subdomain predstavlja **prostor problema**, odnosno kako je odlučeno da se podeli poslovna logika, ili neka domenska aktivnost. Sa druge strane bounded context je **prostor rešenja**, odnosno kako je softver i razvoj tog softvera organizovan za rešavanje problema. Uglavnom se ova dve prostora poklapaju, ali ne mora nužno da znači uvek. Eric Evans daje jedan konkretan primer. Neka se posmatra soba čiji je pod potrebno prektiti tepihom. Soba je prostor problema, dakle, predstavlja subdomain, a tepih je prostor rešenja. Problem je moguće rešiti na više načina. Prvi način je prekriti sobu sa tepihom identičnog oblika kao i pod sobe i onda se subdomain i Bounded context odnose na istu stvar. Drugi način bi bio prekriti sobu sa više različitih tepiha čija površina ne pokriva čitavu površinu sobe. U ovom slučaju se subdomain i Bounded context ne poklapaju. [1]

Uglavnom postoje više različitih Bounded contexta. **Context map** demonstrira kako su povezani Bounded context-i međusobno kroz komunikaciju između timova koji rade na posebnim Bounded context-ima.

Pravilo je da svaki Bounded context ima svoju bazu podataka. Objašnjenje za to daje Eric Evans u svojoj knjizi Domain Driven Design:

“Ako ste u kompaniji koja deli bazu podataka i tu bazu podataka ažurira na hiljade procesa, veoma je teško kreirati model po DDD-u i pisati softver koji radi nešto konkretno sa ovim modelom“. [3]

Kada pričamo o mikroservisnoj arhitekturi takođe postoji jedno (nepisano) pravilo da svaki servis ima svoju bazu podataka. Kao što će biti kasnije objašnjeno, ovde se uviđa i paralela između Bounded context-a i mikroservisa. **Uglavnom** je deljenje odrađeno na sledeći način, **ali** **nije pravilo**: **1 Bounded context - 1 mikroservis - 1 baza**.

**Shared kernel** (zajednička osnova) predstavlja još jedan jako bitan koncept kod DDD-a. Odnosi ne na način interakcije između različitih bounded context-a. Sastoji se od skupa domenskog modela koji je zajednički za više različitih bounded context-a. Na ovaj način se taj zajednički domenski model deli između više različitih bounded context-a. Zove se “kernel“ zato što predstavlja osnovne/kernel elemente između context-a.

## Ubiquitous language

Ubiquitous language je ključni pojam u DDD-u. On predstavlja **zajednički jezik** između tehničkih i netehničkih osoba koje rade na rešavanju nekog problema putem DDD-a. Dakle, koriste ga svi učesnici razvojnog procesa nekog proizvoda, domain eksperti, developeri i ostale zainteresovane strane za razvoj proizvoda. Pomaže prilikom razumevanja domena problema.

Pomoću ubiquitous language-a se dobija konzistentna terminologija. Termini koji se koriste u kodu se poklapaju sa terminima koje koriste domen eksperti tokom diskusije o domenu problema. Klase, imena, metode i ostali bitni termini dolaze od domenskih eksperta i ubiquitous language-a.

Domenski eksperti igraju ključnu ulogu u razvojnom procesu kao što je već pomenuto. Ubiquitous language im pomaže prilikom komunikacije da developeri razumeju sam problem. Na kraju se dobija bolji softver.

Neophodno je pomenuti da ubiquitous language nije statičan i konačan. Razvija se tokom vremena kako celokupni tim uči i razume problem. Međusobna komunikacija članova tima i konstantne povratne informacije treba da budu prisutne kako bi se menjao i poboljšavao ubiquitous language.

Različiti bounded context-i u sistemu mogu da imaju svoj poseban ubiquitous language. Takođe, neki termini iz jednog bounded context-a mogu da imaju totalno neko drugo značenje u drugom bounded context-u. Baš zbog toga je bounded context i jako bitan kod DDD-a, zato što on omogućava postojanje i međusobno funkcionisanje različitih modela.

Tehnički aspekti prilikom modelovanja bounded contex-a su isti kao i pattern-i prilikom programiranja problema.

## Literatura

* <https://app.pluralsight.com/library/courses/fundamentals-domain-driven-design/table-of-contents> [1]
* [https://martinfowler.com/bliki/BoundedContext.html [2](https://martinfowler.com/bliki/BoundedContext.html%20%5b2)]
* Domain Driven Design Eric Evans [3]