УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Милош Самарџија

РАЗВОЈ МИКРОСЕРВИСНЕ АПЛИКАЦИЈЕ ЗА ANDROID КОРИШЋЕЊЕМ ОКРУЖЕЊА LUMEN

мастер рад

Београд, 2020.

Ментор:

др Милена Вујошевић Јаничић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Филип Марић, ванредни професор
Универзитет у Београду, Математички факултет

др Александар Картељ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Мојој породици

Садржај

Глава 1

Увод

Микросервисна архитектура представља један од популарнијих приступа развоју скалабилних дистрибуираних система. Главна филозофија на којој је овај приступ заснован је доменски оријентисано моделовање (енг. domain-driven design). Доменски оријентисано моделовање подстиче разбијање комплексних домена на што једноставније и независније поддомене, као и активно учешће доменских експерата (енг. domain experts) у развоју система.

Главна тематика којом се овај рад бави су основне идеје водиље доменски оријентисаног моделовања, на који начин их микросервисна архитектура инкорпорира, као и дизајн REST интерфејса за програмирање апликација (енг. RESTful API) који представља једну од техника за интеграцију поддомена система. У те сврхе је осмишљена и развијена апликација чији је циљ да илуструје примену микросервисне архитектуре у пракси, као и комуникацију са екстерним сервисима.

Апликација је названа MyRunningBuddy, а њена примарна намена је проналажење партнера за трчање на основу различитих параметара. Сам алгоритам за упаривање тркача уједно представља и најважнији поддомен (енг. core subdomain) целог система који имплементира пословну логику и на који је потребно обратити највише пажње током дизајна и имплементације. Део апликације је написан у програмском језику PHP коришћењем Lumen развојног оквира за развој микросервисних апликација и REST интерфејса за програмирање апликација, где је највећи део бизнис логике садржан. Други део апликације је написан у програмском језику Java, заједно са Android комплетом за развој софтвера (енг. Android SDK) и представља кориснички интерфејс, односно улазну тачку у систем, која није у фокусу овог рада, па

ће према томе пропорционално мање времена бити утрошено за њен развој. У пракси, корисничка апликација и њен кориснички интерфејс су нешто на шта би подједнако требало обратити пажњу током развоја производа. За потребе складиштења података од стране микросервиса користи се MySQL база података.

Глава 2

Доменски оријентисано моделовање

Доменски оријентисано моделовање је филозофија развоја чији циљ је управљање конструкцијом и одржавањем софтвера писаног за комплексне домене. Циљ се достиже употребом одређених образаца (енг. patterns), принципа и добрих пракси које, уколико се добро разумеју и примене, смањују шансу за прављење неких честих грешака које се јављају током развоја.

2.1 Чести проблеми развоја пословних апликација

Да бисмо разумели сврху постојања ове филозофије, потребно је да разумемо на какве изазове се најчешће наилази током развоја и одржавања софтвера. Један од популарнијих архитектуралних стилова коришћених у пословним апликацијама је тзв. велика лопта од блата (енг. Big Ball of Mud) коју карактерише лош дизајн кода са одсуством било какве организације. Такав софтвер је углавном доста спрегнут, и захтев за променом једне од функционалности може изазвати ланчану реакцију других измена које нису планиране, али су у том случају неопходне. Проблем је то што није лако уочљиво када пројекат скрене са правог пута, и од добре архитектуре дође до велике лопте од блата. Са временом се инкрементално квари, додавањем нових функционалности, и крпљењем постојећих, уз слабу бригу о одржању архитектуре, са изговором да ће некад касније бити издвојено време за

рефакторисање. Тако настаје технички дуг (енг. technical debt).

Технички дуг је концепт у развоју софтвера настао као еквивалент новчаног дуга. Суштина концепта је да, занемаривањем дизајна кода, дуг постаје све већи, и све је мање вероватно да ће бити успешно измирен. У преводу, долази се до стадијума када је свака измена спецификације (која даље повлачи измене у коду) болна јер је софтвер постао изузетно спрегнут, и много компоненти зависи од других, иако у природи можда чак и не постоји таква веза између тих ентитета. Немогуће је направити изоловане измене, већ су оне раштркане широм система, и због тога се веома лако могу увести нови багови (енг. bugs) у систем. Јасно је да су измене спецификације у пословном софтверу неизбежне, па нам преостаје да се прилагодимо, или да пројекат полако али сигурно одведемо у пропаст. У тој ситуацији се, у зависности од тренутног стања кода, долази до закључка да је неопходно извршити опширно рефакторисање, или дизајнирање и писање софтвера испочетка. То све доводи до већих трошкова за развој, који су се могли избећи.

2.2 Идеја филозофије и основни концепти

Идеја водиља ове филозофије је разбијање великих и комплексних домена на једноставније поддомене. Ово има двојаки значај:

1. добијамо поддомене који су мање комплексни, и који имају јаснију одговорност

2. лакше се проналази и изолује најважнији поддомен у који је потребно уложити највећи део времена, јер је то оно што одређени софтвер чини другачијим од осталих решења, и представља разлог због којег је уопште и настао

Такође, филозофија инсистира на томе да се у пројекат активно укључе и доменски експерти чија би примарна улога била упознавање програмера и остатка тима са комплексним доменом како би се избегле недоумице и погрешна решења. Ово функционише тако што се током целог развојног циклуса организују формални и неформални састанци са експертима, где се кроз дискусију и различите активности (енг. knowledge crunching) врши анализа и разбијање домена на више мањих, и моделовање поддомена. Као пропратни ефекат ових састанака настају аналитички модел, и заједнички језик (енг.

ubiquitous language) чија је улога избегавање двосмислености у комуникацији између експерата и програмера. Током животног циклуса пројекта разумевање домена постаје боље, а заједно са разумевањем се развијају и расту заједнички језик и модели. Сва комуникација, модели и код су изражени у терминима заједничког језика.

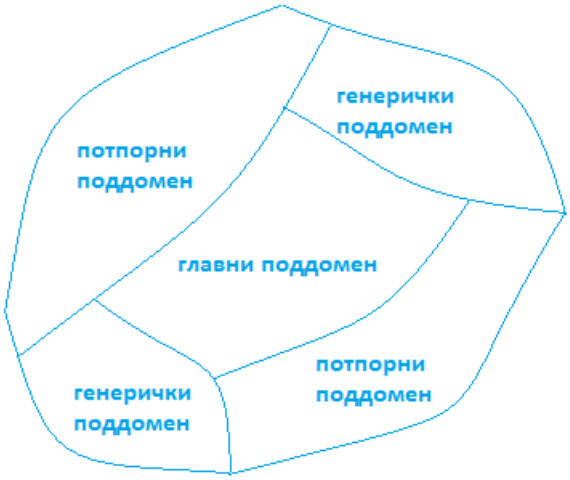
2.3 Аналитички модел

Аналитички модел је колекција артефаката који описују модел система, и намена му је да програмерима и пословним корисницима помогне да боље разумеју домен. Да би овај модел имао смисла, све време мора бити усклађен са имплементационим моделом, тј. измене у имплементацији се морају огледати у изменама у аналитичком моделу, и обрнуто. Ова усклађеност постиже се управо изражавањем оба модела у терминима заједничког језика.

Да би се ово остварило, имплементациони модел мора да буде ослобођен било каквих брига које су техничке природе, и да буде фокусиран искључиво на домен. Такође, битно је да аналитички модел буде једноставан за имплементацију, тј. да не буде превише апстрактан, или на превише високом нивоу. Пројекти који постану тешки за одржавање управо пате од недостатка фокуса на домен; технички проблеми бивају помешани са доменском логиком, и дизајн кода се везује за техничке појмове, уместо за домен. То доводи до повећања сложености, и отежаног разумевања пројекта од стране нетехничких лица попут доменских експерата. Комуникација се успорава услед велике количине времена утрошеног на објашњавање техничких концепата нетехничким лицима, врло лако долази до неразумевања и јављају се двосмислености, а потом настају и погрешна софтверска решења.

2.4 Врсте поддомена

Разбијањем комплексног домена открива се више мањих и једноставнијих поддомена са мањим бројем одговорности. Сваки од поддомена се може сврстати у неку од категорија: главни, генерички (енг. generic) и потпорни (енг. supporting) поддомени. Уз сваки добијени поддомен се придружује његов одговарајући модел. Пример разбијања једног комплексног домена на више мањих може се видети на слици 2.1



Слика 2.1: Пример поделе комплексног домена на једноставније поддомене

Под генеричким поддоменом се подразумевају неке од најраспрострањенијих функционалности које и већина других система такође има. Један пример генеричког поддомена може бити подсистем за слање и примање порука. Ово наравно не мора увек бити тачно. Генерички поддомен једног система може бити главни поддомен унутар другог система. У нашем примеру са подсистемом за поруке, у већини система ће вероватно бити сврстан као генерички, али ће представљати главни поддомен у системима чији је највећи адут управо размена порука.

У потпорни поддомен спадају све остале ствари које представљају подршку за функционисање главног домена. Заједничка ствар за потпорне и генеричке домене је то што су у односу на главни поддомен мање битни, али су неопходни јер без њих систем не може бити комплетан.

Главни поддомен садржи највећи и најбитнији део пословне логике, и у зависности од његовог модела и одговарајуће имплементације зависи да ли ће софтвер бити популаран и добро прихваћен, или ће се придружити великом броју других осредњих софтверских решења. Спецификација главног поддомена ће бити најподложнија изменама, и дизајнирању његовог модела треба приступити пажљиво, како би био довољно флексибилан да подржи све накнадне измене.

Са друге стране имамо потпорне и генеричке поддомене чији дизајн може бити слободнији, па се чак могу користити и неке готове софтверске библиотеке (енг. third party libraries). С обзиром да њихов модел не мора бити превише

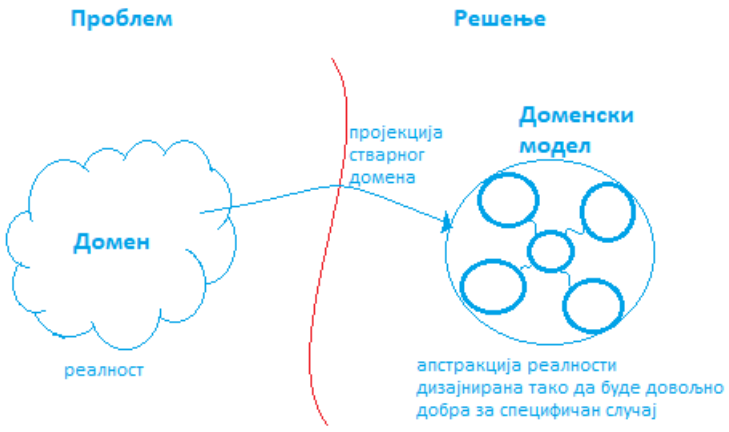
флексибилан, и највероватније ће трпети најмање измена, преживели бисмо и да се неки од њих временом претвори у велику лопту од блата. Кључно је то што је тај лош дизајн изолован од остатка система, и онемогућено му је даље ширење. Уместо претераног фокусирања на потпорне и генеричке поддомене, може се уложити додатни труд у развој главног поддомена. Програмери са мање искуства могу бити задужени за потпорне и генеричке поддомене, а они искуснији могу да се преусмере на главни поддомен.

2.5 Доменски модел

Доменски модел је централни појам доменски оријентисаног моделовања. На почетку се формира у виду аналитичког модела кроз заједничку сарадњу развојног тима и доменских експерата. Не осликава домен у потпуности онако како изгледа у реалности, већ представља поглед на домен из једног угла, односно његову апстракцију, која је довољно добра за наше случајеве употребе. Слика 2.2 приказује разлику између реалности и самог модела. Модел је описан заједничким језиком којим тим говори, као и дијаграмима које је тим креирао. Садржи само оно што је неопходно у контексту апликације која се креира, и мора да се развија паралелно са пословањем како би био користан и валидан. Модел је користан онолико колико је у стању да представи комплексну логику која решава пословне проблеме, а не колико добро осликава реалност.

Креирање доменског модела није једноставан посао, и представља итеративан процес. Велика је вероватноћа да први добијени модел неће бити задовољавајући, и то не треба да буде обесхрабрујуће. Тим би, заједно са доменским експертима, требало да настави са активностима, у циљу проналажења бољих модела. Не треба се задржати на само једном добром, већ треба пронаћи више задовољавајућих модела. Анализом доменских случајева употребе може се оценити корисност креираног модела, и потврдити да сви чланови тима разумеју домен.

Систем који се конструише се састоји од више целина, где су неке битније од других, па ће постојати и више модела различите сложености који ће се користити у различитим ограниченим контекстима (енг. bounded contexts). С обзиром на захтевност овог процеса, комплексни и богати модели се креирају само за битне поддомене система.



Слика 2.2: Пројекција реалног домена на једну његову апстракцију

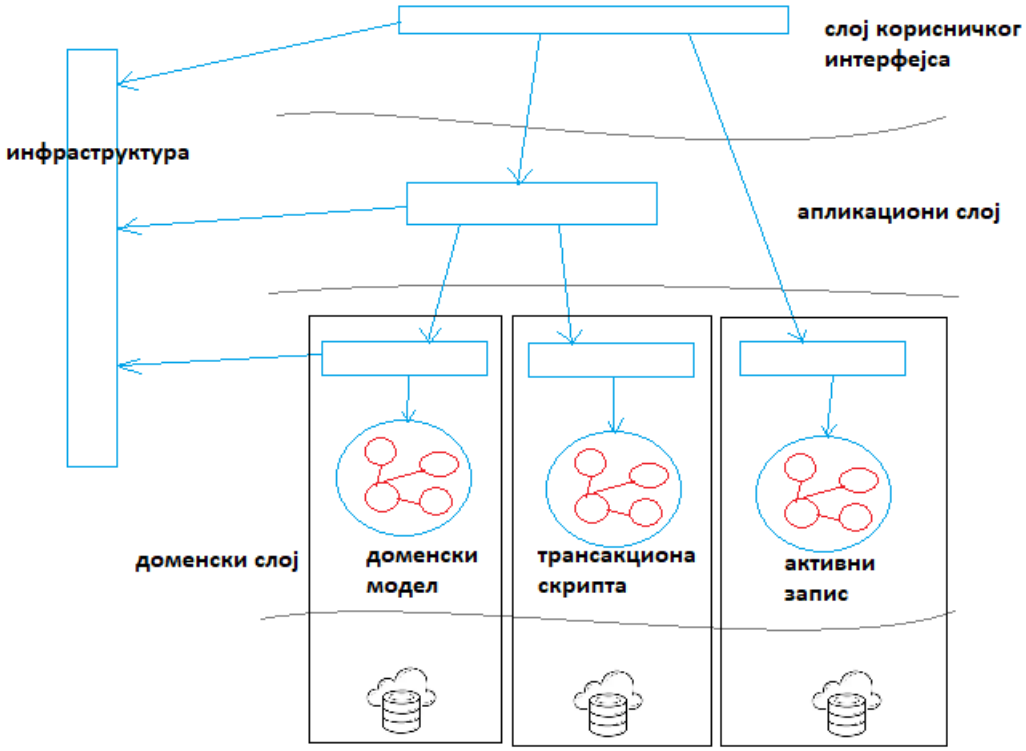
Имплементациони обрасци за доменски модел

Уколико посматрамо слојевите архитектуре софтвера, слој домена би представљао срж апликације. Он изолује сложеност доменског модела од сложености разних техничких делова софтвера. Његова улога је да осигура да се техничке ствари попут логике за управљање трансакцијама, складиштења података и приказивања не умешају у комплексност пословне логике. Мешањем инфраструктурне и пословне логике добили бисмо код који је тежи за разумевање, јер има више од једног задужења, и не можемо се фокусирати на само једну ствар. Слој домена обично представља само мали део целокупне апликације, што се може видети на слици 2.3.

Конструкција доменских модела је проблем који нема јединствено решење, али у пракси постоје одређена решења, односно имплементациони обрасци, који су општеприхваћени и углавном су се добро показали. Неки од њих више одговарају сложенијим поддоменима, док се други могу искористити код једноставнијих, за прављење простијих и сиромашнијих модела. Популарнији обрасци који се користе су доменски модел (енг. domain model), трансакциона скрипта (енг. transaction script) и активни запис (енг. active record).

Образац „доменски модел”

Овај образац се најбоље уклапа у комплексне домене са богатом пословном логиком, и подразумева креирање објектно-оријентисаног модела који садржи податке, пословне процесе и правила, и богату доменску логику. Премиса

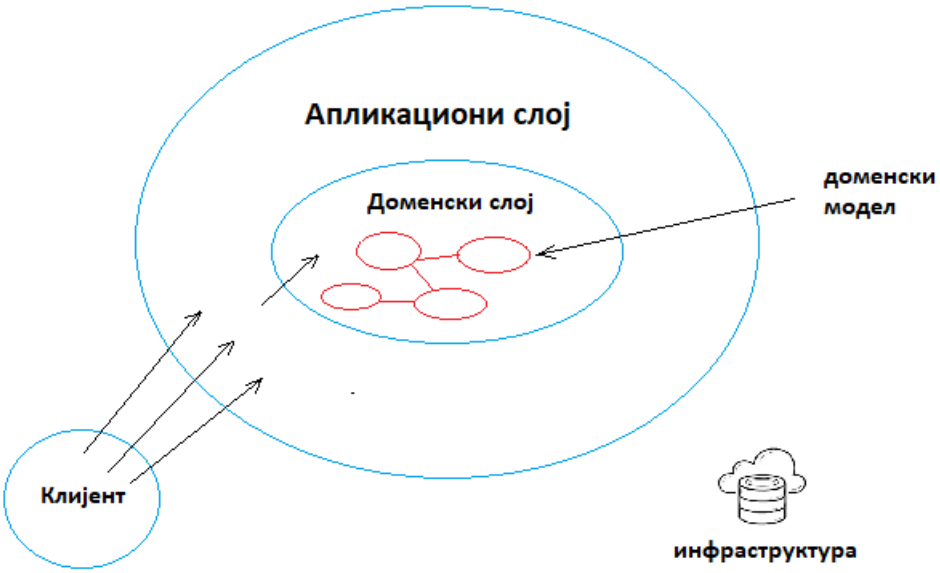


Слика 2.3: Слој домена као део сложеније архитектуре

обрасца је да не постоји база података, и да током еволуирања модела проблем складиштења података буде занемарен. Објекти у добијеном моделу су у потпуности ослобођени од инфраструктурних проблема, што нам омогућава да дизајн нашег модела буде фокусиран искључиво на домен. На слици 2.4 се може видети како је доменски слој раздвојен од техничких ствари попут складиштења података.

Образац „трансакциона скрипта”

Трансакциона скрипта је, за разлику од доменског модела, заснована на процедуралном стилу развоја, лакша је за разумевање и имплементацију. Идеја је да се за сваки пословни случај употребе креира по једна процедура, и да се све те процедуре групишу на једно место. Свака од процедура треба да садржи сву логику потребну да се обави одговарајући случај употребе, укључујући ствари попут пословних правила, логике за складиштење података, итд. Предност овог обрасца су једноставност и могућност да се лако



Слика 2.4: Раздвајање доменског слоја од остатка апликације

имплементирају нови случајеви употребе додавањем нове процедуре, без бојазни да ли ће бити утицаја на постојећу функционалност. Мане су, очигледно, потенцијално много понављања исте логике на различитим местима и лоша подела одговорности. Овакав образац је најприхватљивији за једноставније поддомене који садрже мало логике.

Образац „активни запис”

Активни запис је популарни образац који подразумева да се сваки објекат модела пресликава у одговарајући ред неке табеле. Погодан је у случајевима када модел базе података одговара пословном моделу. Ова структура ће у себи садржати податке и понашање, као и додатне методе за складиштење, додавање нових инстанци и претраживање колекције објеката. С обзиром да свака од структура има методе за креирање, читање, ажурирање и брисање, могуће је коришћење алата и скриптова за аутоматско генерисање доменског модела.

2.6 Ограничени контексти и интегритет доменских модела

Већ смо споменули битност разбијања главног домена на неколико мањих. Као резултат тог процеса настају једноставнији поддомени са бољом поделом одговорности, и њихови одговарајући модели. У најбољем случају, пресликавање из поддомена у модел би требало да бијективно; међутим, у пракси се може десити да се неки модели протежу кроз више поддомена, или да један поддомен има више модела. Како би се остварила функционалност система, неопходно је да постоји нека интеракција између добијених модела. Кључно је да модели буду што независнији, и да ова интеракција буде добро дефинисана. У супротном, лако долази до преплитања концепата и логике из различитих поддомена. Да би се заштитио интегритет модела, дефинишу се јасне границе између различитих модела, а то се постиже везивањем модела за одређени контекст, тзв. ограничени контекст.

За разлику од поддомена, који је апстрактан појам, ограничени контексти представљају конкретну техничку имплементацију која ће форсирати одржање граница између модела апликације. Имплементирају се тако да поседују читав стек (енг. stack) функционалности једног поддомена, укључујући презентациони слој, доменску логику и инфраструктуру. Избор архитектуралног обрасца, па и самих технологија за имплементацију једног контекста се може разликовати од контекста до контекста. Сама филозофија не ограничава избор архитектуре, али је потребно да она буде у стању да изолује доменску логику, јер ће се презентација, инфраструктура и доменска логика мењати различитим интезитетом, и из различитих разлога.

Контексти се дефинишу на начин који би омогућио поделу послова између различитих тимова, и тако да унутар једног контекста нема двосмислених појмова који би увели забуну. Природно је да за један ограничени контекст буде задужен само један тим, унутар којег би се развијао заједнички језик. С обзиром да су ограничени контексти доста независни, и комуникацију са другим контекстима обављају преко јасно дефинисаних интерфејса, један тим би јако мало зависио од других тимова, и њихови рокови дистрибуције нових верзија софтвера не би зависили од другог тима и њихових проблема. Ово важи све док се не наруши компатибилност променом интерфејса преко којих се комуникација обавља.

Везе између различитих модела и тимова дефинишу на који начин ће се та комуникација одвијати. У наставку ће бити наведени неки од образаца веза међу контекстима.

Антикорупциони слој

Уколико се комуникација одвија између два ограничена контекста која се одржавају од стране два различита тима, велика је шанса да ће њихови модели бити међусобно некомпатибилни, и да појмови из њихових одговарајућих заједничких језика неће бити изражени на исти начин. Уколико се комуникацији приступи директно, један модел би морао да наруши своју структуру зарад компатибилности са другим моделом, али то се коси са филозофијом да модели треба да буду максимално независни. Најприхватљивије решење је креирање једног слоја индирекције, односно адаптера, који би требало да транслира интерфејс једног модела у интерфејс другог модела. Овај слој се назива антикорупциони слој (енг. anticoruption layer), и омогућава комуникацију између два некомпатибилна интерфејса, без потребе да се један модел прилагођава другом. Примена овог приступа није ограничена само на комуникацију са контекстима других тимова, већ и на комуникацију са екстерним контекстима на чији интерфејс не можемо да утичемо. На овај начин спречавамо да се лош дизајн других компоненти прелије на наш модел.

Дељено језгро

Понекад се може десити да два контекста имају доста заједничких концепата и логике, и покушаји да се они раздвоје могу бити контрапродуктивни, и увести непотребну сложеност. Уместо раздвајања, пракса је да такви контексти деле поједине делове, односно да имају дељени модел преко којег ће се одвијати комуникација. Овај дељени модел се зове дељено језгро (енг. shared kernel). Мана овог приступа је дељени код који уводи зависност, а који је настао као компромис зарад олакшане интеграције. Ово уводи ризик да један контекст може да утиче на интегритет другог, и због тога је потребно обратити додатну пажњу.

Узводна/низводна веза

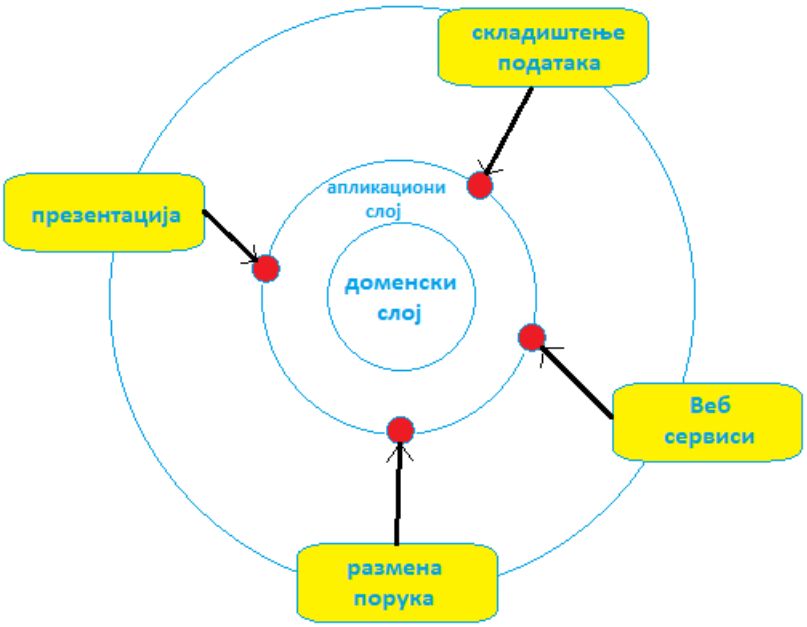
Веза између два контекста такође може бити дефинисана на основу смера тока података, где ће један контекст бити узводно, а други низводно. У том случају узводни контекст има велики утицај на низводни. Дистрибуције нове верзије узводно ће највероватније имати утицаја на низводне контексте, посебно уколико дође до промене интерфејса.

2.7 Слојевита архитектура

У претходној секцији је речено да архитектура мора да подржи раздвајање доменске логике од остатка апликације, односно раздвајање одговорности. Да би се ово постигло, могу се креирати различити слојеви апликације, где би сваки слој био задужен за неку од одговорности. На слици 2.5 се може видети слојевита архитектура са доменским слојем у самом језгру. Доменски слој је у потпуности ослобођен било каквих зависности, и фокусиран је искључиво на доменску логику. Апликациони слој који окружује доменску логику има улогу да имплементира пословне случајеве употребе, и то тако што ће оркестрирати доменским моделима и делегирати захтеве доменском слоју. Једина зависност коју апликациони слој треба да има је зависност од доменског слоја.

Јасно је да ће ова два слоја имати потребу за складиштењем података, и за разним другим инфраструктурним услугама. Да би се ово остварило без увођења додатне зависности, примењује се принцип познатији као инверзија зависности (енг. dependency inversion). Инверзија зависности у овом случају подразумева да, уместо да се апликациони слој прилагођава специфичностима сваке инфраструктурне услуге, свака од услуга имплементира унапред дефинисан интерфејс од стране апликационог слоја. Пропратни ефекат независности апликационог и доменског слоја од остатка апликације је њихово олакшано тестирање, тј. тестирање пословне логике у изолацији, без инфраструктурних зависности.

2.8 Интеграција ограничених контекста



Слика 2.5: Слојевита архитектура апликације са инверзијом зависности

Глава 3

Закључак