Tarkvaraarhitektuur ja -disain

R.C. Martini “Clean Architecture”

Analüüs ja näidisrakendus

Anu Kuusmaa
Andres Käver

Sissejuhatus

Clean Architecture' ehk puhta arhitektuuri kontseptsiooni autoriks on Robert C. Martin. Mustri kirjeldus ilmus 2017. aastal samanimelise raamatuna [1], kuid lühemaid kokkuvõtteid on autor avaldanud varemgi näiteks oma blogis [2].

Martin sedastab, et arhitektuurilise lahenduse peamine eesmärk on toetada ja hõlbustada süsteemi kogu elutsüklit – väljatöötamist, muutmist ja muudatuste toodangusse saatmist. Tema teooria, kuidas sellist lahendust disainida, võib koondada järgmisteks universaalseteks põhimõteteks:

Klasside komponentideks jaotamise reeglid

1) Üheks komponendiks tuleb kokku koondada need klassid, mis enamasti muutuvad samadel põhjustel ja samal ajal.
2) Lähtuda tuleb SOLID põhimõtetest, mille sellisel kujul formuleerijaks on samuti Martin:
    a. Single Responsibility Principle ehk üheainsa vastutuse põhimõte. Igal tarkvarakomponendil peab olema ainult üks põhjus, miks teda muudetakse.
    b. Open-Closed Principle ehk avatud-suletud põhimõte. Tarkvarakomponent peab olema suletud muutustele, kuid avatud laiendamisele.
    c. Liskov Substitution Principle ehk Liskovi asendatavuse põhimõte. Et komponente oleks lihtsam üksteise vastu välja vahetada, peavad nad täitma lepingut/liidest.
    d. Interface Segregation Principle ehk liideste eraldatuse printsiip. On seotud esimese põhimõttega, ning sedastab, et ühte liidesesse ei ole mõtet kuhjata funktsionaalsusi, millest kõiki ei tahaks täita ükski liidese täitja. Need on mõtekam jagada mitme liidese vahel.
    e. Dependency Inversion Principle ehk pööratud sõltuvuste printsiip. Kõik sõltuvused peaksid olema suunatud abstraktsioonide/liideste, mitte konkreetsete klasside poole. (Vt ka andmebaasikihi sõltuvuse näidet allpool.)

Komponentideks jaotamine ehk nendevaheliste piiride (boundaries) tõmbamine on hea arhitektuurilise lahenduse alustala.

Süsteemitaseme reeglid

1) Süsteemi keskmeks on kriitilised ärireeglid, need, millega ettevõte teenib raha, need, mis on tema äri südameks. Kõik sõltuvused süsteemis peavad olema suunatud kriitiliste ärireeglite suunas. Kriitilistest ärireeglitest n-ö järgmine tase on kasutusjuhtumite tase, kus otsustatakse, millal ja milliseid kriitilisi ärireegleid kasutatakse.
2) Komponendid võib liigitada nende kauguse järgi sisenditest ja väljunditest ehk konkreetse rakenduse spetsiifikast. Sõltuvused peavad olema suunatud sisenditele ja väljunditele lähemal asuvate komponentide poolt kaugemal asuvate poole. See toetab sisendite ja väljundite lihtsamat väljavahetamist. Mõnest teisest

arhitektuurimustrist erinev on siin see, et sisendite-väljundite, ehk Martini sõnakasutuses detailide alla kuulub ka andmebaas.

3) Sõltuvused peavad olema suunatud volatiilsemate, tihedamini muutuvate komponentide poolt stabiilsemate poole. See on kooskõlas eelmise punktiga.

Lahendamaks olukordi, kus intuitiivselt kulgeb sõltuvus teistpidi kui Martini kontseptsiooni järgi (näiteks kasutab äriloogikakiht andmebaasi ja võiks sellest sõltuda), kasutab Martin liideseid. Äriloogika jaoks vajalikud andmekihi kasutamise meetodid on deklareeritud liideses, mida saavad täita erinevad andmete haldamise implementatsioonid (vt joonist, kus punane joon esitab klasside jaotust komponentide vahel). Sellise lähenemise korral saab näiteks SQL-andmebaasi välja vahetada veebiteenusest andmete küsimise või CSV-failist lugemise vastu, ilma et äriloogika kihti tuleks muuta.

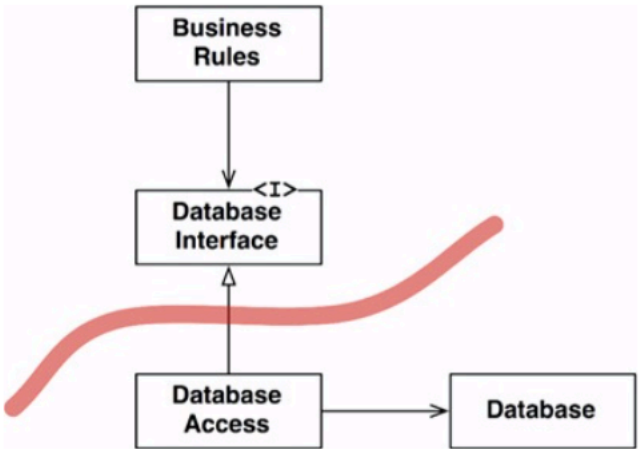
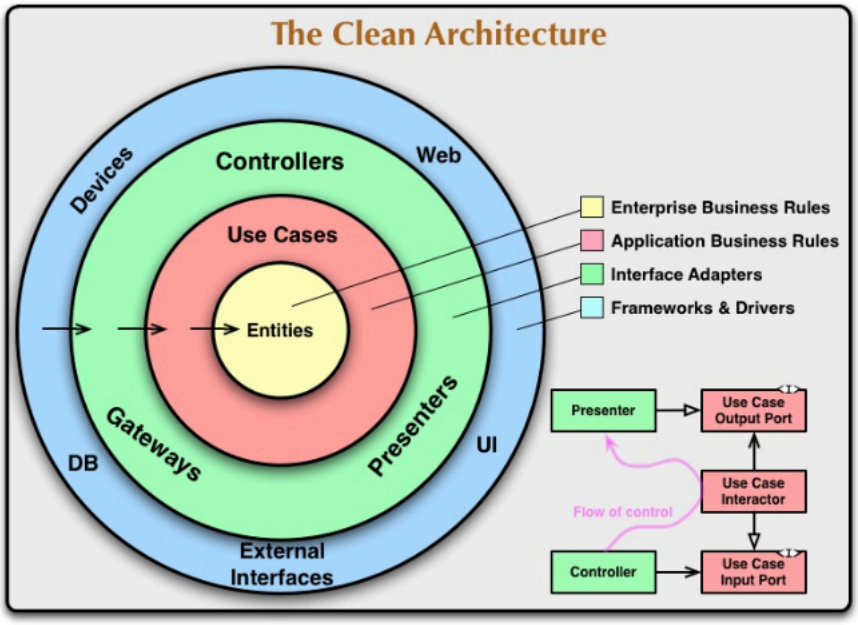


Figure 17.2 The boundary line

Paljude teiste mustrite eeskujul näeb Martin väljavahetatava detailina ka kasutajaliidest, mis peab sõltuma äriloogikast, mitte vastupidi. Kasutajaliides peab olema lihtsalt väljavahetatav, üht ja sama äriloogikat peab saama muutusteta kasutada nii konsoolirakendusest, teenusest kui veebirakendusest.

Parim arhitektuur on Martini sõnul nii-öelda karjuv arhitektuur, millele peale vaadates on selgelt võimalik öelda, millise sisuga äri ta esindab, nii nagu esiku, köögi ja kolme toaga hoonearhitektuur ütleb selgelt, et tegu on koduga. Milliseid raamistikke ja sisendeid-väljundeid on kasutatud, on siin teisejärguline detail, mille suhtes tehakse otsus arhitektuuri ellurakendamisel – nii nagu ühepereelamu puhul on näiteks katusematerjal. Oluline on mõistlik komponentideks jaotus ja komponentide paiknemine üksteise suhtes ehk nendevahelised sõltuvused.

Puhta arhitektuuri ülevaatlik skeem:



Metoodika

Meie näidisrakenduseks, mille puhul kasutame puhta arhitektuuri põhimõtteid, on kontaktide andmebaas. Selle kriitilisteks ärireegliteks, meie äri nii-öelda tuumaks on kontaktide, isikute ja kontaktitüüpide kuvamine, lisamine, muutmine ja kustutamine. Rakendus on kirjutatud C#-s.

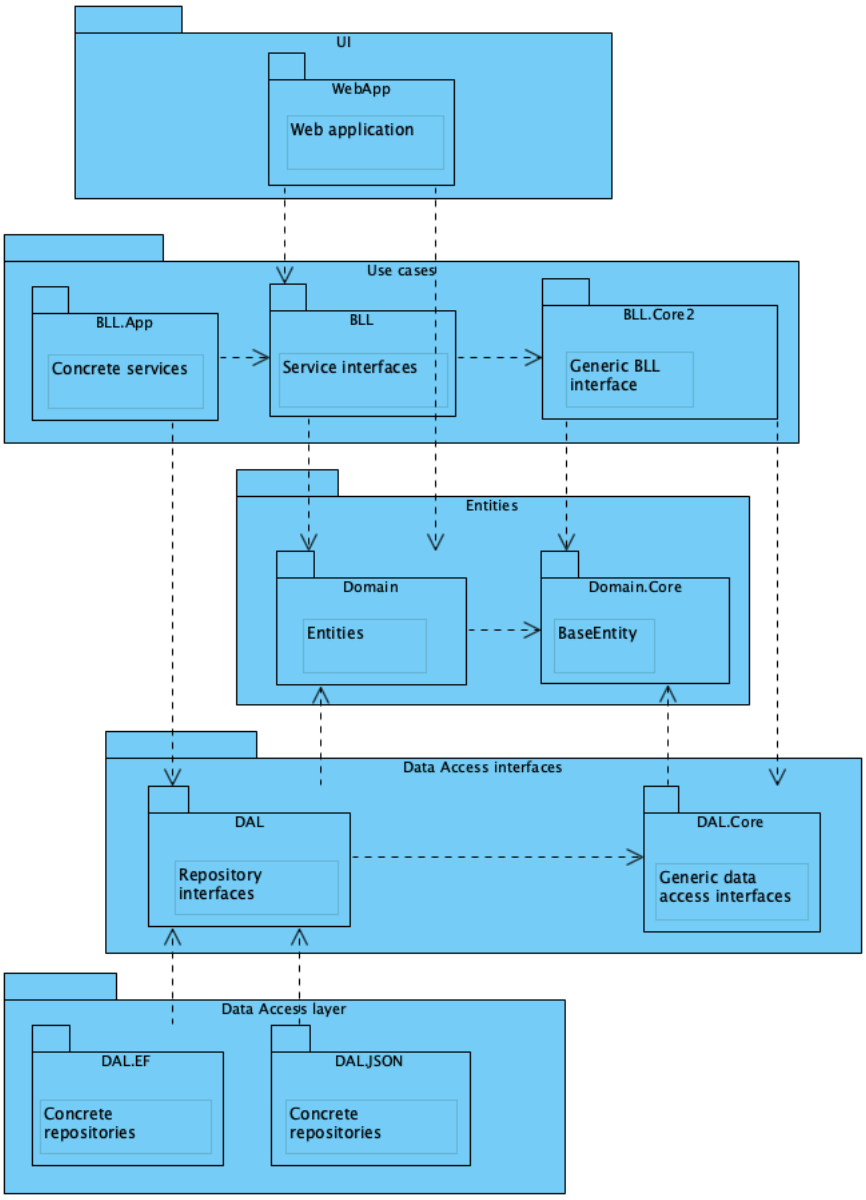
Meie rakenduse domeeniobjektideks on inimene (Person), kontakt (Contact) ja kontaktitüüp (ContactType). Ühel inimesel võib olla mitu kontakti, millest igaüks vastab ühele kontaktitüübile (nt telefoninumber, Skype'i konto, email). Puhta arhitektuuri mudeli järgi moodustavad eelmainitud kolm objekti meie äriloogika tuuma. Nende ümber, neist sõltuvuses kasutusjuhtude ehk teenuskiht. Veebikontroller ja andmeligipääsukiht kasutavad omakorda teenuskihti. Andmekihi sõltuvus teenuskihist on viidud ellu eelpool mainitud põhimõtete järgi ehk andmekiht täidab liidest, mille meetodeid omakorda kutsub välja teenuskiht. Erinevalt Martini mudelist ei kuulu andmekihi liidesed siiski füüsiliselt teenuskihiga samasse komponenti, vaid on eraldi komponendis, mida kasutavad nii teenus- kui andmekiht (vt ka valdkonnamudeli joonis). Omavaheliste sõltuvuste seisukohalt on tulemuses vahe väike, kuid rakendus on nii modulaarsem, selle ülesehitus lihtsamini mõistetav.

Demonstreerimaks andmekihi väljavahetatavust, on meil kaks erinevat konkreetset andmebaasikihti, mis mõlemad etteantud liidest täitavad. Üks on traditsiooniline, Entity Frameworki abil mälus hoitavat relatsioonilist andmebaasi kasutav lahendus, teine küsib ja kirjutab andmeid JSON-failidest.

Kasutajaliideseid on meil implementeeritud üks, ASP.NET Core 2.2 veebiliides.

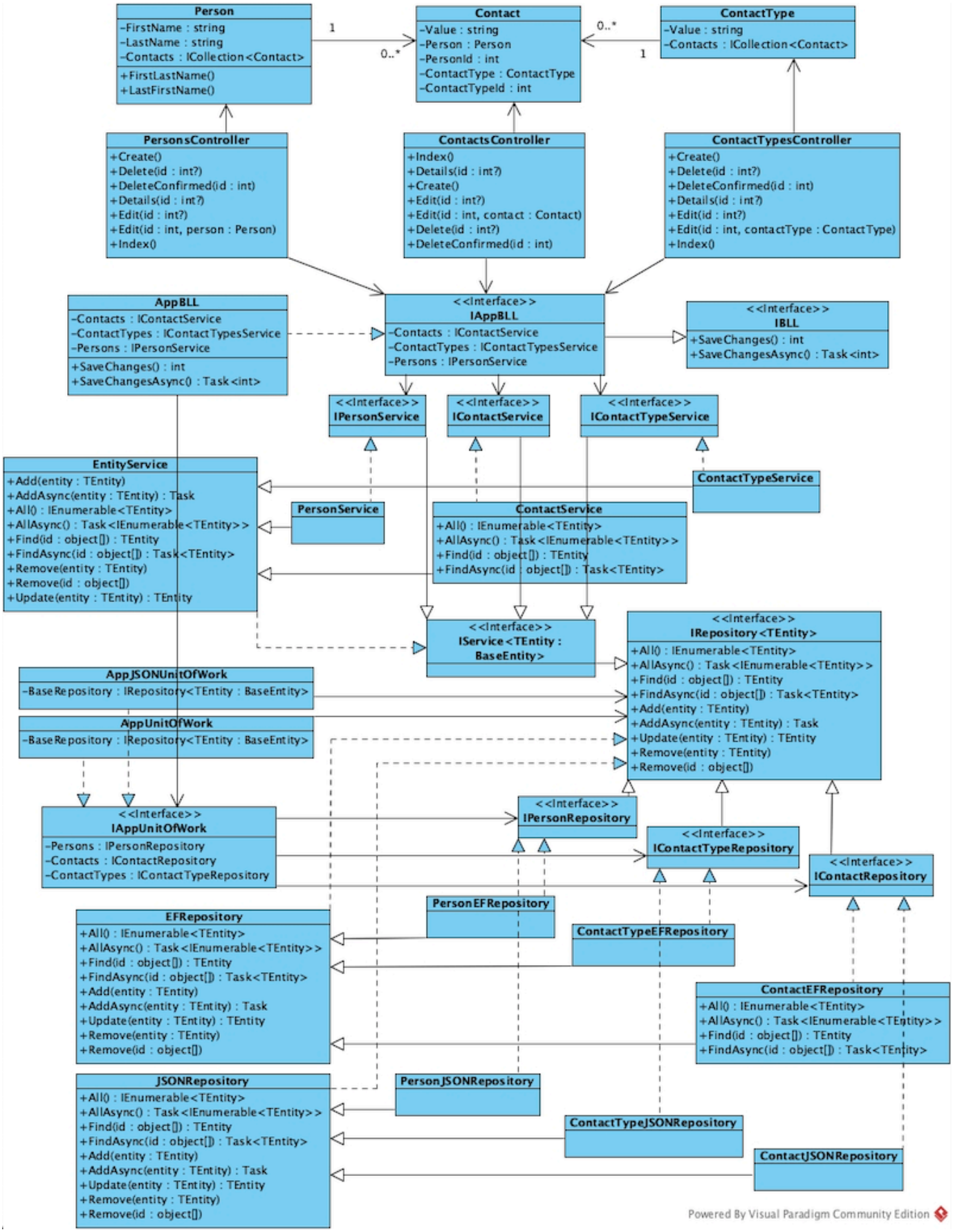
Mudel

1. Valdkonnamudel

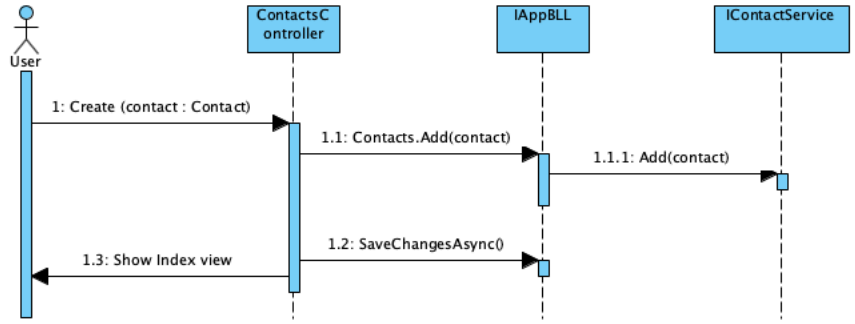


Väiksemate pakettidena on välja toodud meie rakenduse iseseisvad komponendid. Suuremate pakettidena on tähistatud Martini poolt eristatud süsteemikihid. Andmete ligipääsu kihis on näha kaks erinevat implementatsiooni – Entity Frameworki ja JSON-faile kasutavad lahendused.

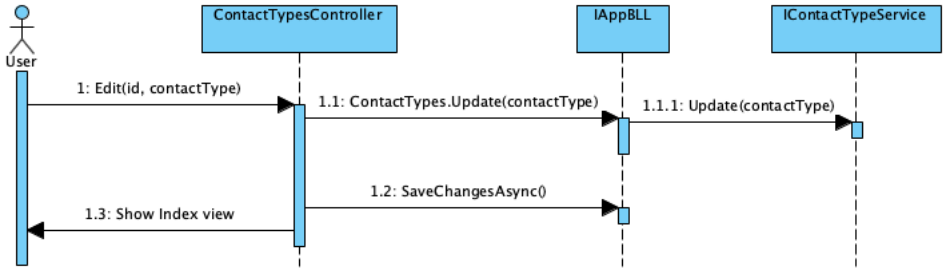
2. Klassidiagramm



3. Interaktsioonidiagramm: uue kontakti lisamine



4. Interaktsioonidiagramm: kontaktitüübi redigeerimine



Mudeli analüüs

Tegu on väga lihtsalt laiendatava ja skaleeritava mudeliga. Nagu näitas meie eksperiment, oli Martini mõistes detailide ehk uue andmekihi implementatsiooni lisamine väga lihtne, taandudes JSON-failidga seonduvatele tehnilistele nüanssidele. Muudatused senises koodis piirdusid uute sõltuvuste injekteerimise ning JSONi serialiseerimise atribuutide lisamisega domeeniobjektidele.

Nagu näha valdkonnadiagrammilt, on kõik sõltuvused suunatud õiges suunas, äriloogikakihi poole. Äriloogikasiseselt on sõltuvused omakorda suunatud domeenimudeli poole, ning kasutusjuhtude kiht sõltub andmekihi liidestest. Konkreetne andmetele ligipääsu kiht ning kasutajaliidesekiht ehk detailid sõltuvad teistest kihtidest, kuid neist endist ei sõltu midagi.

Nagu näha klassidiagrammilt, on kõik sõltuvused lahendatud sõltuvustega liidestest. Kui välja arvata domeeniobjektid, ei liigu ükski sõltuvus ühegi konkreetse klassi suunas.

Puhta arhitektuuri kontsentrilist mudelit võiks võrrelda nt kolmekihilise arhitektuuriga. Kui vaadata Martini püstitatud eesmärki, tarkvarasüsteemi kogu elutsükli toetamist, siis on puhtal arhitektuuril siin eeliseid. Sõltumatus andmekihi realisatsioonist võib näida nüansina, kuid kui tegu on suure süsteemiga, võib selle väljavahetamine tähendada nädalate- ja kuudepikkust veaohtlikku tööd. Samuti, kui näiteks äriline eesmärk on arendatavat tarkvaratoodet müüa, nii et seda saaks integreerida suurematesse süsteemidesse, siis mida rohkem komponente selle juures on lihtsasti väljavahetatavad, seda parem. Martin toob ka näite, kus ühe tema tarkvaralahenduse ostnud klient suutis vähem kui päevaga vahetada välja failipõhise andmekihi MySQL andmebaasi vastu.

Kuna rakenduse lähtepunktiks oli meie meeskonna liikme poolt reaalselt toodangus kasutatav kood, võiks nullist alustades teha ka lihtsama näidisrakenduse. Me ei näinud aga põhjust hakata koodi sihipäraselt lihtsustama, kuna ta täidab oma eesmärgi puhta arhitektuuri kõigi positiivsete külgede näitlikustamisel.

Kokkuvõte

Käesoleva töö eesmärk oli tutvustada ja ellu rakendada Robert C. Martini puhta arhitektuuri kontseptsiooni. Esitasime tema teooria kokkuvõtliku ülevaate. Kirjutasime selles toodud põhimõtteid järgiva lihtsa funktsionaalsusega rakenduse koos testidega ning esitasime rakendust kirjeldavad mudelid UML-skeemidena.

Demonstreerimaks Martini põhimõtet, et andmete ligipääsu kiht peab olema lihtsasti väljavahetatav „detail“, implementeerisime kaks andmete ligipääsu lahendust ja näitasime, et nende väljavahetamine piirdus minimaalsete muudatustega senises koodis. Jõudsime järeldusele, et puhta arhitektuuri mudel toetab Martini püstitatud eesmärke muuta tarkvarasüsteem lihtsasti hallatavaks ja laiendatavaks.

Näidisrakenduse kood: https://github.com/akaver/CleanArchitectureDemo.git

Kasutatud allikad

1. Clean Architecture: A Craftsman’s Guide to Software Structure and Design. 2017. Martin, R.C. Prentice Hall.
2. http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html