LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

MODULĀRAS UN TESTĒJAMAS ARHITEKTŪRAS VEIDOŠANA TĪMEKĻA LIETOTNĒM

MAĢISTRA DARBS

Autors: **Andris Skudra**
Studenta apl. Nr.:as09632
Darba vadītājs: asoc. prof. Edgars Celms

RĪGA 2015

ANOTĀCIJA

Lietotnēs ar sarežģītu biznesa funkcionalitāti ir svarīgi domāt par testējamību un modularitāti arhitektūras līmenī. Darba mērķis ir aprakstīt principus, kā veidot šādu koda arhitektūru tīmekļa lietotnei, īpaši koncentrējoties uz testējamību.

Darba galvenās tēmas ir atkarību injicēšana, starpnozaru problēmu centralizēta risināšana, *Entity Framework* izmantošana kopā ar eksistējošu datubāzi, biznesa loģikas pārvaldīšana, servera un klienta puses uzturamu vienībtestu un integrācijas testu veidošana.

Lai realizētu pētījumu, autors eksistējošai uzņēmuma informācijas sistēmai veica arhitektūras uzlabošanu, kā rezultātā tā tika salīdzināta ar veco lietotni, secinot, ka ir uzlabojusies uzturamība, saprotamība un veiktspēja.

Atslēgvārdi: uzņēmuma lietotnes arhitektūra, vienībtesti, atkarību injicēšana, *Entity Framework*, ASP.NET MVC

ABSTRACT

It is important to consider testability and modularity at an architectural level in systems with complicated business logic. The objective of this paper “Modular and testable architecture for web applications” is to explore principles of creating architecture like this, with focus on testability.

Main topics of this paper are dependency injection, centralized cross cutting concern problem solving, using Entity Framework with existing database, business logic management and creation of maintainable unit and integration tests for server and client side code.

To perform this research, author made improvements to existing enterprise application architecture. As a result, new application was compared to old one, with conclusion, that maintainability, readability and performance have all increased.

Keywords: enterprise application architecture, unit testing, dependency injection, Entity Framework, ASP.NET MVC

AUTOREFERĀTS

Darba gaitā autors veica plašu informācijas avotu apkopošanu, grupēšanu un analīzi. Tika aprakstītas vairākas problēmas, kuras praktiskās izstrādes gaitā nācās risināt. Vairākām problēmām autors piedāvā savu risinājumu:

- Starpnozaru problēmu centralizēta risināšana ar ASP.NET MVC filtriem.
- *Entity Framework* izmantošana ar eksistējošu datubāzi – problēmas un risinājumi sakarā ar datubāzes shēmas nesakritību ar *Entity Framework* iespējām.
- Biznesa loģikas testēšana integrācijā ar datubāzi – saliktu testa objektu glabāšana JSON failos, ievietošana testa sesijas sākumā un atrite beigās.
- Klienta puses koda testēšana un integrācija ar *Angular* ietvaru.

Lai statistiski apskatītu izveidotās arhitektūras plusus, tika salīdzināts koda apjoms, koda metrikas (cikliskā sarežģītība, atkarību skaits utt.), ātrdarbība. Rezultāti apliecina, ka izveidotā arhitektūra ir uzturamāka, testējamāka, saprotamāka, ar potenciāli mazāk kļūdām. Sakārtojot kodu, ir izdevies uzlabot veiktspēju sākot ar 2 reizēm un vairāk.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

DI (*Dependency injection*) - projektējuma princips, kurš nosaka, ka objekti paši neveido instances klasēm, no kuriem tie ir atkarīgi, bet saņem tās no ārpuses.

IoC (*Invertion of Control*) - princips, kad ietvars kontrolē programmas izpildes gaitu.

SUT (*System Under Test)* - testējamās klases instance vienībtestā.

Klases atkarība (*dependency*) – objekts, kuru cits objekts izmanto savas funkcionalitātes realizēšanai.

Testu dubultnieks (*test double*) – objekts, ar kuru testos tiek aizstāta klases atkarība.

Automātiskā maketu veidošana (*auto-mocking*) – tehnika, kurā DI konteineris tiek izmantots, lai dinamiski aizstātu visas testējamās klases atkarības ar dubultniekiem.

AOP - aspektu orientētā programmēšana. Koncentrējas uz starpnozaru problēmu atdalīšanu no biznesa loģikas.

TDD (*Test driven development*) – izstrādes pieeja, kurā izstrāde apvienota ar vienībtestu veidošanu, fokusējoties uz klašu testēšanu izolācijā.

BDD (*Behaviour driven development*) – izstrādes pieeja, kurā izstrāde apvienota ar vienībtestu veidošanu, fokusējoties uz testu scenārijiem, kas atbilst lietošanas piemēriem.

MVC (*Model-View-Controller*) – arhitektūras princips tīmekļa lietotnēm.

ASP.NET MVC – *Microsoft* tehnoloģija *Model-View-Controller* arhitektūras principa realizācijai.

CQS (*Command Query Separation*) - princips, kurš funkcijas iedala divās grupās – tās vai nu atgriež datus, vai modificē sistēmas stāvokli.

CQRS (*Command Query Responsibility Segregation*) - princips, kurš datu apstrādes funkcijas un datu moduļus iedala divās grupās – datu lasīšanas un datu rakstīšanas.

CRUD (*Create Read Update Delete*) – apzīmē datu izveidošanas, lasīšanas, labošanas un dzēšanas operācijas attiecībā uz datubāzi.

ORM (*Object Relational Mapper*) – ietvars, kurš kartē objektus no/uz datubāzi.

ADO.NET - *Microsoft* datu piekļuves tehnoloģija

Starpnozaru problēmas (*cross cutting concern)* - funkcionalitāte, kas neattiecas uz biznesa loģiku, bet sistēmu kopumā, piemēram, transakciju kontrole, audits, kļūdu apstrāde.

Windsor – atkarību injicēšanas ietvars

Testa armatūra (*test fixture*) – testu klase, kas apvieno vairākus vienībtestus.

EF (*Entity Framework*) – objektu-relāciju kartēšanas ietvars

DbContext – klase, caur kuru notiek datubāzes saskarne EF ietvarā.

LINQ – (*Language Integrated Query*) – vaicājumu valoda, kas cenšas izolēties no datu avota, sniedzot vienotu saskarni datu piekļuvei dažādos datu avotos

Code First – EF modeļa stratēģija, kur modelis tiek veidots kā C# klases

DOM (*Document Object Model)* – funkcijas HTML piekļuvei un apstrādei

JS (*Javascript)* – tīmekļa programmēšanas valoda.

jQuery - JS bibliotēka DOM manipulāciju atvieglošanai.

Mantotā lietotne (*legacy application)* - kods pārņemts no cita izstrādātāja un ir tehnoloģiski novecojis un ar laiku kļuvis slikti uzturams.

Angular – klienta puses ietvars, kurš koncentrējas uz DOM manipulāciju nodalīšanu no biznesa loģikas, atkarību pārvaldīšanu un testējamību.

DB – datubāze

DBVS – datubāzes vadības sistēma.

Slinkā datu atlase (*lazy load)* – projektējuma modelis, attiecībā uz ORM nozīmē, ka dati tiek atlasīti nevis uzreiz, bet tajā brīdī, kad tiem pirmoreiz mēģina piekļūt.

IEVADS

Arhitektūras mērķis ir pārvaldīt sarežģītību. Lielās, sarežģītās sistēmās vienībtesti kļūst īpaši svarīgi, jo sarežģītu biznesa funkcionalitāti testēt no lietotāja saskarnes ir lēni, sarežģīti un reizēm neiespējami, jo ir grūti izvērtēt rezultātu un noteikt, vai lietotne izdarījusi to, ko vajadzēja.

Tāpat svarīgi ir nodalīt starpnozaru problēmas (*cross cutting concern* – funkcionalitāte, kas neattiecas uz biznesa loģiku, bet sistēmu kopumā, piemēram, transakciju kontrole, audits, kļūdu apstrāde) no biznesa loģikas. Ja sistēma veic netriviālas darbības, aktuāli kļūst pārvaldīt biznesa loģikas sarežģītību.

Šajā darbā ar vārdu "arhitektūra" tiks apzīmēta programmatūras koda struktūra augstā līmenī - vairāk koncentrējoties uz koda organizāciju, slāņu/klašu savstarpējo sadarbību, mazāk uz tā tiešu rakstīšanu.

Ar vārdu "testējams" tiks apzīmēts kods, kuram var uzrakstīt vienībtestus vai zema līmeņa integrācijas testus. Lai tas būtu iespējams, kodam ir jābūt vāji saistītam un ar iespēju klasēm aizvietot atkarības (*dependency* – objekts, kuru cits objekts izmanto savas funkcionalitātes realizēšanai).

Pētāmā problēma – kādas metodes, tehnikas, principus un modeļus jāizmanto, lai izveidotu arhitektūru, kas ir modulāra, testējama un palīdz pārvaldīt biznesa loģikas sarežģītību.

Šie principi attiecināmi galvenokārt uz lietotnēm, ko raksturo šādas īpašības:

- Uzņēmuma informācijas sistēma - satur netriviālu, uzņēmumam specifisku biznesa loģiku.
- Sarežģīta datu atlase - daudz darba tiek veikts ar datiem.
- Datu (datubāzes) centriska – tiek izmantota relāciju datubāze un tai ir svarīga loma.

Tā kā praktiskajā darbā tika izmantota mantotā lietotne (*legacy application* - kods pārņemts no cita izstrādātāja un ir tehnoloģiski novecojis un ar laiku kļuvis grūti uzturams), daudz uzmanības tiks pievērsts arī šai īpašībai, galvenokārt tādās tēmās kā testēšana.

Neapšaubāmi, par šo tēmu ir pieejama informācija, tomēr lielākā daļa risinājumu par arhitektūru un testēšanu ir attiecināmi uz vienkāršām sistēmām vai arī autora gadījumā nebija pielietojami. Līdz ar to vairākām problēmām nācās meklēt savu risinājumu.

Darba mērķa sasniegšanai tiks veikti šādi uzdevumi:

- Tika veikta esošās informācijas, risinājumu sistematizēšana un analīze.
- Tika uzsākta izstrāde, izmantojot atrasto informāciju.

- Saskaroties ar problēmām tika veikta papildus izpēte un analīze. Ja risinājumu neizdevās atrast tīmekļa resursos, autors piedāvāja savu risinājumu.

Šajā darbā tika apskatītas šādas tēmas un risinātas šādas problēmas:

- Labas arhitektūras īpašības autora izpratnē, augsta līmeņa vadlīnijas un eksistējoši modeļi, kas risina dažādas problēmas – ciešu saistību starp slāņiem, sarežģītu biznesa domēna loģiku, ātrdarbības problēmas, kas rodas, izmantojot nepiemērotus modeļus datu lasīšanai un rakstīšanai.
- Vienībtestu teorija, modeļi, atkarību aizstāšana testos ar testu dubultniekiem.
- Atkarību injicēšana – teorija un praktiskas problēmas, kas saistītas ar *Windsor* (atkarību injicēšanas ietvars) izmantošanu.
- Starpnozaru problēmu risināšana, nodalot tos no pārējā koda.
- Datu piekļuves slānis – EF (*Entity Framework* – objektu-relāciju kartēšanas ietvars) modeļa veidošanas stratēģijas un integrācija ar eksistējošu datubāzi.
- Biznesa loģikas slānis – loģikas izolēšana no datubāzes un datu piekļuves slāņa.
- Biznesa loģikas testēšana – kā panākt testu konsistenci un atkārtojamību, testējot integrācijā ar datubāzi.
- Klienta puses arhitektūra – mantotas lietotnes testēšana un integrācija ar *Angular* ietvaru.

Lai realizētu pētījumu, praktiskajā daļā autors izmantoja lietotni, ko ikdienā uztur un attīsta. Lielākā daļa darbā aprakstīto problēmu radās izstrādes gaitā. Lai novērtētu padarīto darbu, izveidotā lietotne tika salīdzināta ar veco:

- Arhitektūras līmenī salīdzinātas koda metrikas – cikliskā sarežģītība, uzturamība.
- Salīdzināta veiktspēja svarīgākajām un prasīgākajām funkcijām.

1. ARHITEKTŪRA

Arhitektūras galvenais mērķis ir pārvaldīt sarežģītību. Šajā darbā ar vārdu "arhitektūra" tiks apzīmēta programmatūras koda struktūra augstā līmenī - vairāk koncentrējoties uz koda organizāciju un slāņu savstarpējo sadarbību.

Autoraprāt, laba arhitektūra atbilst šādām īpašībām:

- Modulāra – klašu atbildības ir skaidri saprotamas un nodalītas.
- Papildināma – viegli ieviest jaunu funkcionalitāti.
- Atkalizmantojama - sistēmas daļas/klases/funkcijas ir vienkārši izmantot atkal, iekļaujot tos citās sistēmās.
- Saliedēta (*highly cohesive*) - klases funkcijas ir fokusētas konkrēta mērķa sasniegšanai.
- Vāji saistīta (*loosely coupled*) – izmaiņas vienā klasē ietekmē pārējās pēc iespējas mazāk.
- Viegli saprotama - kods koncentrējas uz biznesa problēmas risināšanu un biznesa loģikas izpildi. Biznesa funkcionalitāte netiek jaukta kopā ar lietotnes līmeņa funkcionalitāti, piemēram, tādas lietas kā audits, transakciju kontrole, tiesību kontrole, datubāzes pieprasījumi ir nodalīti citos slāņos.
- Pragmatiska – arhitektūras sarežģītība atbilst sistēmas sarežģītībai. Ir svarīgi neaizrauties ar jaunāko modeļu vai tehnoloģiju izmantošanu tikai tāpēc, ka tā ir jaunā labākā prakse vai tamlīdzīgi. Tajā pat laikā, nedrīkst sarežģītu biznesa loģiku realizēt procedurālās programmēšanas stilā, visu uzrakstot vienā milzīgā funkcijā, kuru nav iespējams notestēt.
- Testējama - par testējamību jādomā jau arhitektūras veidošanas posmā. Galvenie jautājumi, kas jāatrisina arhitektūras veidošanas posmā – kā veidot modulāras klases, kuras pēc tam būs iespējams notestēt, kā veidot klašu atkarību pārvaldību, kā veidot datu piekļuves slāni.

1.1. SOLID principi

Meklējot informāciju par programmatūras arhitektūru, viena no pirmajām lietām, ar ko nākas saskarties, ir SOLID principi, ar ko 2000. gadu sākumā iepazīstināja programmēšanas eksperts Roberts C. Martins [1]. Neskatoties uz to, ka principi ir 15 gadus veci, tie savu

aktualitāti nav zaudējuši. Principi paši par sevi ir ļoti abstrakti, tomēr, tie kalpo kā pamats veidojot sarežģītu arhitektūru un rakstot modulāru, testējamu kodu.

Principi ir sekojoši:

- *Single Responsibility Principle* - katrai klasei ir viens un tikai viens iemesls mainīties. Principa pretējais piemērs ir milzīgas klases, kas veic daudzus nesaistītus uzdevumus (piemēram, veic ierakstus datubāzē, ģenerē HTML kodu, veic audita ierakstus, regulē transakcijas utt.).
- *Open / Closed Principle* - objektus (klases funkcionalitāti) jāvar papildināt, bet ne mainīt. Sarežģīta definīcija, bet pamatdoma ir tāda, ka jaunu funkcionalitāti sistēmā vajadzētu varēt ieviest ar jaunām klasēm, pēc iespējas mazāk modificējot eksistējošās.
- *Liskov Subsitution Principle* - klasēm, kas tiek mantotas no citas klases, jābūt aizstājamām ar bāzes klāsi. Doma ir tāda - padot mantojošo klasi bāzes klases vietā tā, lai saglabātu visu eksistējošo funkcionalitāti.
- *Interface Segregation Principle* - dot priekšroku vairākiem maziem, specifiskiem interfeisiem, nevis vienam masīvam interfeisam.
- *Dependency Inversion Principle* - klašu atkarībām jābūt abstraktiem objektiem, nevis konkrētām implementācijām. Praksē tas nozīmē, kas atkarības tiek padotas kā abstraktas klases vai interfesi. Šis princips cieši saistās ar testējamību, un sīkāk aprakstīts sadaļā par atkarību injicēšanu.

1.2. MVC arhitektūra

Pārejot no vispārīgiem, abstraktiem SOLID principiem uz mazliet konkrētākiem, nepieciešams aprakstīt arhitektūras modeļus, kas ieguvuši konkrētākus apveidus un ir nosaukti un atpazīstami ar konkrētu vārdu. Šie modeļi piedāvā noteikumus un principus, uz kuriem balstīties, izstrādājot arhitektūru. Būtiska lieta, ko svarīgi saprast - šie modeļi nepiedāvā gatavus risinājumus, bet tikai vadlīnijas. Tie risina specifisku problēmu, bet ir pietiekami brīvi definēti, lai tos būtu iespējams un pat vēlams izmantot kopā ar citiem modeļiem.

MVC ir standarta arhitektūras modelis tīmekļa lietotnēm. To veido trīs komponentu veidi [2]:

- Modelis (*model*) – datu objekts, satur biznesa loģiku un datu piekļuves loģiku.
- Skats (*views*) – attēlo datus. Tīmekļa lietotnes gadījumā tas būtu HTML kods, kurā ievieto modeļa datus.

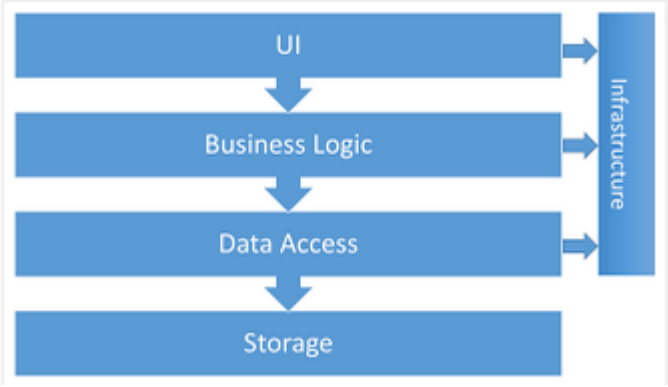
- Kontrolieris (*controller*) – apstrādā lietotāja darbības, izsauc modeļus, lai iegūtu datus un veiktu citas darbības, izvēlas skatu un atgriež to lietotājam.

Šī ir orģinālā definīcija, kura laika gaitā ir novecojusi. Apskatot modeļu definīciju, redzama problēma – biznesa loģika ir cieši saistīta ar datu piekļuves loģiku. Netriviālā tīmekļa lietotnē, kas nodarbojas ar ko vairāk nekā vienkāršu CRUD (*Create Read Update Delete* – apzīmē datu izveidošanas, lasīšanas, labošanas un dzēšanas operācijas attiecībā uz datubāzi), biznesa loģika ir pārāk apjomīga un sarežģīta, lai to visu aprakstītu pāris modeļos, turklāt vēl to visu apvienojot ar datu piekļuvi. Tieši tāpēc, attiecinot MVC uz lielākām sistēmām, tas tiek izmantots, lai realizētu tikai vienu slāni – prezentācijas [3]. Līdz ar to no modeļiem tiek izņemta visa loģika un vēl jo vairāk datu piekļuves funkcionalitāte – šīs rūpes tiek pārceltas uz citiem slāņiem. Līdz ar to veidojas nākamais arhitektūras līmenis – slāņveida arhitektūra, kas novērš tradicionālas MVC galveno trūkumu – biznesa un datu piekļuves loģikas ciešo saistību.

1.3. Tradicionāla slāņveida arhitektūra

Tipiskā slāņveida arhitektūrā funkcionalitāte tiek grupēta un nodalīta dažādos slāņos (.NET vidē parasti tas ir projekts, kas apzīmē slāni). Nodalījums nav strikti definēts, tomēr biežāk izmantotie slāņi ir [4] [5] [6]:

- Prezentācijas slānis / UI (*user interface*).
- Biznesa loģikas slānis / BLL (*business logic layer*) / domēna slānis.
- Datu piekļuves slānis / DAL (*data access layer*).
- Infrastruktūras slānis – vispārēja, kopīga funkcionalitāte, piemēram, audits. Tajā pat laikā reizēm šajā slānī iekļauj arī datu piekļuves slāni.



Attēls 1.1 - Tradicionālā slāņveida arhitektūra [4]

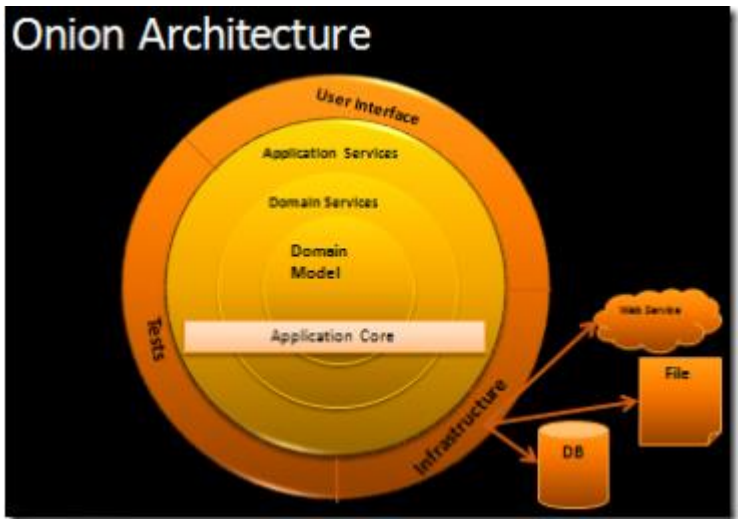
Šajā arhitektūras modelī katrs slānis ir atkarīgs no slāņiem zem tā. Katrs slānis tiešā veidā drīkst izsaukt tikai vienu slāni zem tā, piemēram, lai prezentācijas slānis iegūtu datus, tas izsauc biznesa loģikas slāni, kurš tālāk izsauc datu piekļuves slāni.

Galvenā problēma - atkarību plūsma šajā modelī veido nevajadzīgas atkarības. Prezentācijas slānim nebūtu jābūt atkarīgam no slāņa, kurš atbild par datu piekļuvi. Biznesa loģikai nevajadzētu būt saistītai ar to, kā dati tiek saglabāti. Tā ir šī modeļa galvenā problēma - ciešā saistība starp slāņiem. Tie ir fiziski nodalīti koda līmenī, bet joprojām cieši saistīti. Tas rezultējās ar to, ka praktiski visi slāņi beigās ir atkarīgi no datu piekļuves. Datubāze tiek nostādīta šī modeļa centrā, un viss pārējais tiek pielāgots tās vajadzībām. Šāda, datubāzes centriska arhitektūra, padara to grūti testējamu, jo biznesa loģikas funkciju testi rezultēsies ar datubāzes izsaukumiem, kas krietni palēnina un apgrūtina šo procesu.

1.4. Onion arhitektūra

Onion arhitektūras pamatā ir doma par vāju saistību starp slāņiem, kā rezultātā tiek novērsta galvenā problēma tradicionālajā arhitektūrā – cieša atkarība no datubāzes un datu piekļuves. Šīs arhitektūras izmantošana ļauj koncentrēties uz svarīgo – biznesa funkcionalitātes ieviešanu, nevis dažādiem ārējiem (ietvari, datubāze, serveri) faktoriem. Literatūrā atrodami citi stipri līdzīgi modeļi - sešstūra (*hexagonal*) arhitektūra [7], porti un adapteri (*ports and adapters*) [7], "tīrā" arhitektūra (*clean architecture*) [8]. Praktiski tīmekļa resursos visi šie nosaukumi tiek izmantoti gandrīz vai kā sinonīmi. Tomēr *Onion* ir biežāk sastopamais – tas radās 2008. gadā, kad tā autors Džefrijs Palermo (*Jeffrey Palermo*) izveidoja tīmekļa dienasgrāmatas ierakstu, piešķīra šim modelim savu nosaukumu un aprakstīja tā principus savā skatījumā [5]. Šī arhitektūra ir piemērota lielām uzņēmumu lietotnēm, kas satur sarežģītu biznesa loģiku.

Galvenais noteikums - slānis var būt atkarīgs tikai no slāņiem, kas atrodas tuvāk kodolam, bet ne otrādi. Pašā centrā atrodas svarīgākais slānis - domēna modelis, kurš satur visu biznesa loģiku. Šī ir galvenā atšķirība no tradicionālas slāņveida arhitektūras, kuras centrā atrodas datubāze. *Onion* arhitektūrā datubāze ir ārējs faktors – tās izsaukumi ir nodalīti infrastruktūras slānī, kurā atrodas arī tādas lietas kā ārējie tīmekļa servisi un failu sistēma (būtībā, jebkurš ārējs informācijas datu nesējs), kā arī ietvari un starpnozaru problēmu risinājumi.



Attēls 1.2 - Onion arhitektūra [5]

Modelis nedefinē ļoti konkrētus slāņus un to nosaukumus. Tomēr, biežāk izmantotie ir:

- Kodols – sastāv no biznesa loģikas un ārējo komponentu interfeisu definīcijām (realizācija ir ārējā slānī):
    - Domēna modelis (*domain model*) - centrālais slānis, satur biznesa objektu klases (entītijas) un biznesa loģiku.
    - Domēna servisi (*domain services*) – interfeisu definīcijas datu piekļuves (CRUD) funkcijām.
    - Lietotnes servisi (*application services*) - interfeisu definīcijas starpnozaru funkcionalitātei – audits, transakcijas.
- Ārējie slāņi
    - Kodolā definēto interfeisu (domēna un lietotnes servisi) realizācija.
    - Infrastruktūras slānis - viens no ārējiem slāņiem, satur kodu, kas veic komunikāciju ar datubāzi un starpnozaru funkcionalitāti (audits, drošība).
    - Lietotāja interfeiss (prezentācijas slānis) – izsauc un darbina kodolu.
    - Vienībtesti, integrācijas testi – testi tiek uztverti kā vēl viens ārējais klients, tāpat kā lietotāja interfeiss, kas darbina kodu sev specifiskā veidā, šajā gadījumā testu nolūkos.

Problēmas, ko risina *Onion* modelis:

- Mazina moduļu sapārotību un uzlabo modularitāti, galvenokārt slāņu starpā, bet tikpat labi arī klašu līmenī.
- Uzlabo testējamību.

- Noturība pret ārējo faktoru (ietvaru, datubāzes) izmaiņām. Ietvariem ir tendence novecot, kad to vietā nāk jauni, modernāki, ar vairāk un labākām iespējām. Ietvaru iznešana infrastruktūras slānī dod iespēju nepieciešamības gadījumā tos salīdzinoši viegli aizstāt, jo visa sadarbība ar tiem notiek netiešā veidā - caur interfeisu.

1.5. Command Query Responsibility Segregation (CQRS)

CQRS ir princips, kurš datu apstrādes funkcijas un datu moduļus iedala divās grupās – datu lasīšanas un datu rakstīšanas. Lielā daļā tīmekļa resursu CQRS definīcija tiek nevajadzīgi sarežģīta, aprakstot to kombinācijā ar *event sourcing* (aprakstīts zemāk), tomēr būtībā CQRS ir mazliet vienkāršāks. Galvenā ideja ir radusies no CQS modeļa (*Command Query Separation* - modelis, kurš funkcijas iedala divās grupās – tās vai nu atgriež datus, vai modificē stāvokli). Komanda (*command*) ir jebkura metode, kas modificē datus (bet neatgriež), savukārt vaicājums (*query*) - jebkura metode, kas atgriež datus (bet nemodificē) [9].

Savukārt, CQRS šo principu realizē attiecībā uz biznesa loģikas un datu piekļuves funkcijām. Šo principu var ieviest vairākos līmeņos.

Pirmais līmenis ir – sadalīt nevis funkcijas (kā CQS), bet gan klases - komandās un vaicājumos. Līdz ar to, vienas klases vietā (piemēram, *UserService*) tiek radītas 2 klases (*UserReadService* un *UserWriteService*). Klasēm jāspēj strādāt neatkarīgi vienai no otras. Ja komandai nepieciešams datu objekts, tad tas tiek padots kā parametrs, komanda nedrīkst tieši izsaukt vaicājuma metodi.

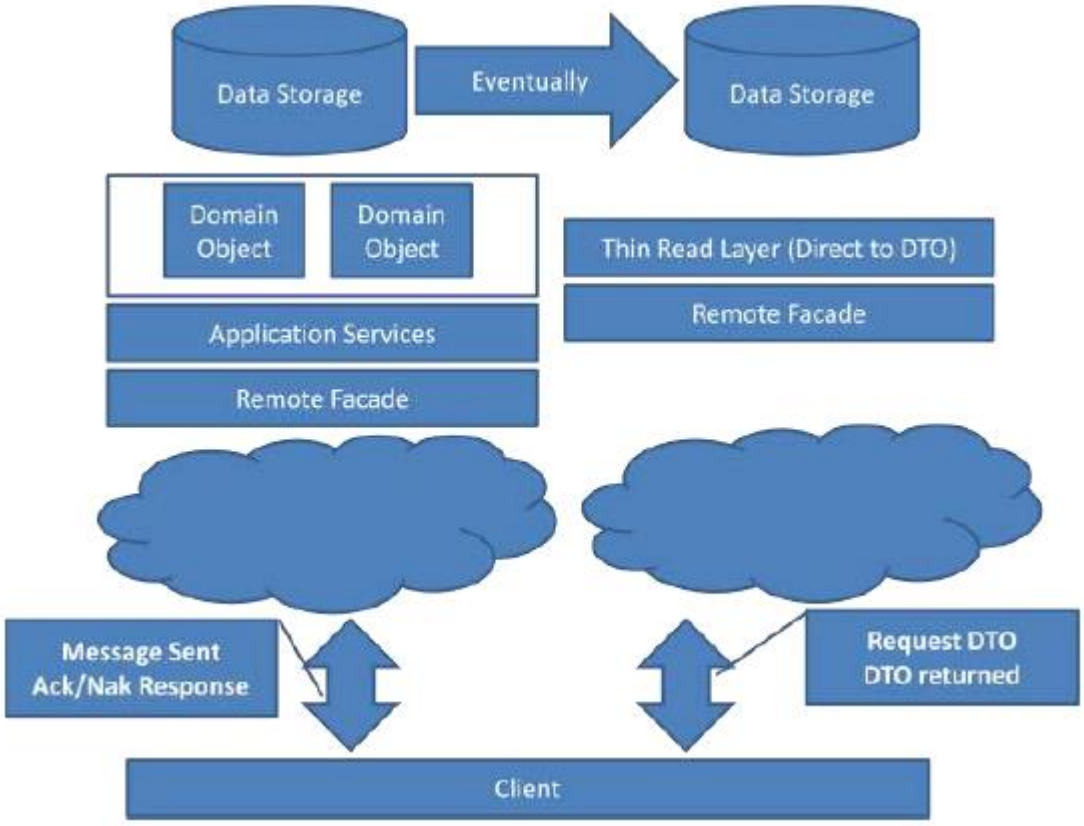
Nākamais līmenis ir domēna modeļa sadalīšana – izmantot dažādas klašu grupas datu atlasei un datu labošanai. Piemēram, datu labošanai izmantot entītiju klases un agregātus, savukārt datu lasīšanai piekļūt datubāzei tieši, aiztaupot datu kartēšanu (*mapping*) uz entītijām. Šādam sadalījumam ir vairāki iemesli un ieguvumi [9].

Pirmkārt, domēna modelim kļūstot arvien sarežģītākam, starp lasīšanas un rakstīšanas funkcijām kļūst arvien mazāk saistītas, tām ir mazāk kopīga koda, iezīmju un struktūras. Lai mazinātu sarežģītību, kļūst izdevīgi šīs metodes atdalīt no koda saprašanas un lasīšanas viedokļa.

Otrkārt, veiktspēja un mērogojamība. Parasti, lietotnē ir vairāk datu lasīšanas nekā labošanas pieprasījumu. Līdz ar to, vienā brīdī var kļūst izdevīgi datu attēlošanai pāriet uz tīru ADO.NET (*Microsoft* datu piekļuves tehnoloģija) risinājumu, nevis izmantot kādu ORM (*Object Relational Mapper* – ietvars, kurš kartē objektus no/uz datubāzi) ietvaru.

Treškārt, ORM veidotajās entītiju klasēs bieži vien ir dažādas validācijas un cita papildus funkcionalitāte, kas vienkāršai datu attēlošanai nav vajadzīga.

CQRS ieviešana koda līmenī ļauj veikt komandu un vaicājumu sadalīšanu jau augstākos arhitektūras līmeņos – datubāzē. Šajā brīdī CQRS attiecas uz arhitektūru vairs ne tikai koda līmenī, bet jau datubāzes, līdz ar to ir ārpus šī darba sfēras, tomēr tiks īsi aprakstīts lai sniegtu pilnu pārskatu par CQRS. Šajā līmenī mērķis ir izmantot funkcionalitātei atbilstošāko datu modeli, jo viens modelis nekad nebūs optimāls visām darbībām. Piemēram, datu agregātu labošana un atlase ir krietni ātrāka dokumentu datubāzēs, jo saistīti glabājas denormalizētā veidā, vienā vietā. Savukārt, atskaišu gatavošanai šāds modelis būs būtiski lēnāks, un tur vajadzētu izmantot kādu datubāzes vadības sistēmu, kas labāk piemērota analītiskai datu apstrādei un atlasei.



Attēls 1.3 - CQRS arhitektūra ar dažādiem datu avotiem [9]

Pateicoties datu atlases un rakstīšanas sadalīšanai dažādos līmeņos, CQRS kalpo par pamatu, veidojot augsti sadalītas sistēmas (*distributed applications*), kurām svarīga augsta veiktspēja. Tādos gadījumos CQRS parasti tiek izmantots kombinācijā ar *event sourcing*.

Event sourcing šajā salikumā kalpo kā diezgan neparasts datu glabāšanas mehānisms. Pielietojot šo principu, sistēmas galvenais datu avots ir nevis aktuālais stāvoklis, bet gan visas

darbības, kas notikušas līdz šim. Piemēram, ja lietotājs objektam X samaina atribūtu Y un nospiež saglabāt, tad datubāzē notikumu tabulā tiek pievienots fakts, ka lietotājs veicis šādu darbību, bet nekur netiek pārrakstīts objekta X atribūts kā tāds. Līdz ar to, sistēmas aktuālais stāvoklis tiek izsecināts no visiem faktiem. Protams, veiktspējas nolūkos tiek glabāti vairāki momentuzņēmumi dažādos laika brīžos, tai skaitā tuvu aktuālajam, bet tie nav primārais datu avots. No šiem momentuzņēmumiem tad arī tiek veikta datu lasīšana [9].

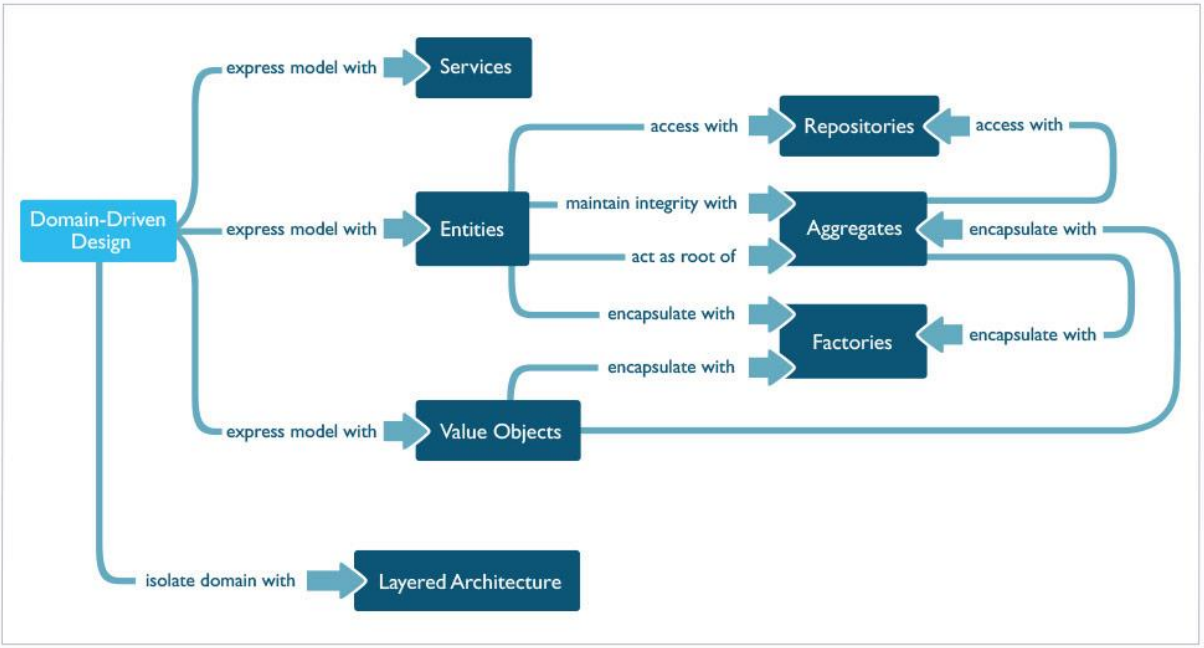
Event sourcing tiek izmantots sistēmās, kur nepieciešama augsta pārliecība par datu pareizību (bankas, finanses, apdrošināšana), kā arī augsts auditēšanas līmenis. Šajā modelī dati ir praktiski nezūdoši - pat dzēšanas operācija ir notikums, ka objekts ir dzēsts, nevis fiziska izdzēšana no datu glabātuves. Līdz ar to ir iespēja sistēmu atgriezt vai apskatīt jebkurā stāvoklī, kādā tā ir bijusi. *Event sourcing* tiek izmantots kombinācijā ar CQRS, jo tas dod iespēju efektīvi izmantot dažādas datu bāzes lasīšanai un rakstīšanai.

1.6. Domain Driven Design (DDD)

Domain Driven Design (DDD) pamatā ir bagātīgs domēna slānis, kur atrodas visa biznesa loģika. Šis termins tika radīts 2003. gadā, kad Ēriks Evanss uzrakstīja grāmatu ar šādu nosaukumu [10]. Datubāze šajā modelī ir sekundāra, primārais ir biznesa problēma, kas jārisina. Šis modelis tiek pielietots sistēmām ar sarežģītu domēnu un netriviālu biznesa loģiku. Parastai CRUD lietotnei šāds piegājiens radītu tikai lieku sarežģītību. Šis modelis parasti tiek implementēts ar vairākslāņu arhitektūru, kuras centrā atrodas domēna modelis.

Lai pārvaldītu biznesa loģikas sarežģītību, DDD norobežojas no problēmām, kas skar datu saglabāšanu datubāzē, tā vietā koncentrējoties uz biznesa objektiem (entītijām) un likumiem, novietojot tos lietotnes arhitektūras kodolā.

Svarīgs aspekts ir vienotas valodas (angliski bieži tiek lietots termins *ubiquitious language*) izveidošana izstrādātājiem un biznesa cilvēkiem, tādējādi mazinot nesaprašanos starp šīm grupām un ļaujot tām sarunāties "biznesa valodā".



Attēls 1.4 - DDD pamatelementi [11]

DDD var atšķirt dažāda veida objektus, kas piedalās problēmas risināšanā un veido domēna modeli [10 lpp. 60-106]:

- Entītijas – objekti ar identitāti. Entītija var mainīt atribūtus, bet tās identitāte saglabājas.
- Vērtību objekti – objekti bez identitātes. Atribūtu vērtības nav maināmas.
- Agregāti – veidojas, grupējot entītijas un vērtību objektus grafā, tādā veidā attēlojot vecāka-bērna grafveida struktūru.
- Domēna servisi – satur biznesa loģiku, kas neiekļaujas viena agregāta ietvaros un neiederas pie kādas konkrētas entītijas. Parasti tiek implementēti kā bezstāvokļa (*stateless*) klases.
- Infrastruktūras servisi – realizē lietotnes ārējās saskarnes, piemēram, failu sistēmu, datubāzi, tīmekļa resursu izsaukumus.
- Lietotnes servisi (*application services*) – paši nesatur biznesa loģiku, bet pārvalda izsaukumus starp loģiku, ko satur domēna servisi un agregāti. Atkarīgi no infrastruktūras servisiem.
- Repozitoriji – realizē saskarni starp domēna objektiem un datu glabātuvi, satur datu atlases un modificēšanas funkcionalitāti.

- Rūpnīca (*factory*) – satur objektu izveidošanas loģiku, ja tā kļūst sarežģīta un satur pārāk daudz implementācijas detaļu. Tipisks piemērs ir objektu izveide datubāzē, kuru identitātes (*OBJETCID)* kolonnas iegūšanai nepieciešama īpaša apstrāde.

DDD varētu izskatīties līdzīgs *Onion* arhitektūrai, tomēr šie modeļi risina atšķirīgas problēmas. DDD koncentrējas uz sarežģītas biznesa loģikas realizēšanu salīdzinoši vienkāršā, objektorientētā veidā, ignorējot datubāzi. Savukārt *Onion* modelis risina problēmas, kas rodas no slāņu ciešas saistības.

1.7. Secinājumi

Šajā nodaļā aprakstītie arhitektūras modeļi nav salīdzināmi, jo risina dažādas problēmas un viens otru neizslēdz. Būtībā tie viens otru ļoti labi papildina, tieši tāpēc nereti tie tiek apvienoti dažādās kombinācijās (DDD + *Onion,* DDD + CQRS + *Event sourcing*), jo lielās sistēmās bieži vien ir vairākas no minētajām problēmām.

2. ATKARĪBU INJICĒŠANA

Atkarību injicēšana (*dependency injection* (DI)) ir projektējuma princips, kurš nosaka, ka objekti paši neveido instances klasēm, no kuriem tie ir atkarīgi, bet saņem tās no ārpuses. Šis ir viens no galvenajiem modernas arhitektūras pamatprincipiem, bez kura izmantošanas praktiski nav iespējami ne vienībtesti, ne arī modulāras, vāji saistītas klases.

Bieži kā DI sinonīms tiek izmantots arī termins *Inversion of Control* (IoC), bet tas patiesībā ir plašāks termins un iekļauj sevī DI kā vienu no veidiem. IoC sākumā nozīmēja principu, kad ietvars kontrolē programmas izpildes gaitu. Ar laiku šo terminu sāka lietot, lai apzīmētu apgrieztu kontroli pār atkarībām. Tad tika ieviests DI termins, lai novērstu nesaprašanos [12 lpp. 41] [13].

Padodot atkarības, parasti tiek izmantotas nevis konkrētas klases, bet interfeisi. Tas dod iespēju testos īstās realizācijas vietā izmantot testu dubultnieku (kurš implementē to pašu interfeisu), kuru uzvedību var kontrolēt, piemērojot to testā veicamajam scenārijam.

Šīs metodes izmantošanai ir vairāki pozitīvi iznākumi:

- Koda modularitāte - klases atkarības ir skaidri definētas un pārskatāmas, kas rezultējas ar koda uzturamību, pārskatāmību, atkalizmantojamību.
- Iespēja rakstīt vienībtestus - DI ir praktiski vienīgais veids, kā iegūt kontroli pār klases atkarībām, kas ļauj testēt klasi izolācijā.

2.1. Injicēšanas veidi

Ir trīs metodes, kā klasei padot atkarības.

- Konstruktora injekcija - atkarības tiek padotas kā parametri, izsaucot konstruktoru. Tās tiek saglabātas kā klases mainīgie, lai vēlākos funkciju izsaukumos tos varētu izmantot klases iekšienē. Šī ir biežāk izmantotā metode. Galvenais pluss - ļoti uzskatāmi tiek parādītas klases atkarības, viss ir redzams konstruktora signatūrā

```
private IDependency _dependency;
public ExampleClass(IDependency dependency)
{
      this._dependency = dependency;
}
```

- Atribūtu injekcija (*property injection*) - atkarības tiek uzstādītas, izmantojot klases atribūtus (*get/set*). Izmanto gadījumos, kad klase pati izveido noklusēto atkarības instanci,

bet ir vajadzīga iespēja mainīt šo instanci, piemēram, testēšanas nolūkos [12 lpp. 104]. Noderīgs veids, ja sistēmā nav vēlama DI izmantošana, tomēr testos ir vajadzība aizstāt atkarības.

```
public class ExampleClass
{
      public IDependency Dependency { get; set; }
}
```

- Metožu injekcija - atkarības tiek padotas kā funkcijas parametri. Izmanto gadījumos, kad atkarības instance var mainīties katrā izsaukumā [12 lpp. 112].

```
public class ExampleClass
{
      public void FunctionWithDependency(IDependency Dependency) { ... };
}
```

2.2. DI ietvari

Atkarību veidošanu un padošanu, protams, var veikt manuāli. Tomēr, objektu un atkarību grafam kļūstot arvien lielākam, šis process kļūst pārāk laikietilpīgs un prasa pārāk daudz koda rakstīšanas, ko varētu uzticēt darīt kādam citam. Vēl viens iemesls ir konstruktoru signatūras biežas izmaiņas (piemēram, ja klase ir aktīvā izstrādes stadijā un regulāri tiek mainīts kods), kā rezultātā nākas bieži labot kodu, kurš veic klases instanču inicializāciju. Lai risinātu šīs problēmas, tika radīti DI ietvari jeb konteineri, kas šo uzdevumu spēj automatizēt un aiztaupīt programmētājam darbu.

DI ietvariem ļoti atšķiras progresīvās funkcijas, tomēr jebkurš ietvars spēj paveikt divas galvenās funkcijas, kas ir neatņemama sastāvdaļa un vispār definē DI ietvarus: tipu reģistrēšana (*register*) un atrisināšana (*resolve*). Neskatoties uz sarežģīto funkcionalitāti, ko sniedz DI ietvari, to visu var reducēt uz šīm divām funkcijām, kas ir pats pamats visam.

Tipa reģistrēšanas procesā tiek konfigurēts, kuram interfeisam atbilst kura klase. Dažādos ietvaros šai funkcijai ir dažādi nosaukumi, viens no biežāk izmantotajiem ir *register* (ietvaros *Windsor, Autofac*). Alternatīvi nosaukumi:

- *Bind/To - Ninject* ietvarā.
- *For/Use - StructureMap* ietvarā.
- *RegisterType - Unity* ietvarā.

Tipa atrisināšanas procesā DI konteinerim tiek pieprasīts atgriezt tipu, ko tas arī dara - vai nu atgriežot jaunu instanci, vai jau eksistējošu (atkarībā no konfigurācijas). Līdzīgā veidā tiek apstrādātas visas tipa atkarības - rekursīvi, līdz visas nepieciešamās instances ir pieejamas.

2.3. DI ietvara izvēle

Internetā meklējot informāciju par DI ietvariem, viens no biežākajiem jautājumiem ir - kuru ietvaru izvēlēties? Jautājums ir pamatots, jo reti kuram ietvaru veidam ir tik daudz izvēles variantu - .NET vidē šobrīd ir jau 30 dažādi ietvari. Bieži vien tiek salīdzināta veiktspēja [14], tomēr lielākajā daļā gadījumu DI ietvars ne tuvu nebūs vājā vieta. Ir citas, svarīgākas lietas:

- Lasāmība - cik pašsaprotama ir sintakse.
- Konfigurējamība - cik plašas iespējas piedāvā ietvars, cik ļoti tas ir papildināms.
- Progresīvā funkcionalitāte.

Attiecībā uz progresīvo funkcionalitāti, ir divas galvenās funkcijas, kas ir pieejamas ne visos ietvaros [14]:

- Pārķeršana (*interception*) - šī iespēja ir cieši saistīta ar AOP (aspektu orientētā programmēšana - starpnozaru problēmu atdalīšana no biznesa loģikas). Pārķeršanas funkcija dod iespēju izveidot klases, kas implementē kādu starpnozaru funkciju (piemēram, auditēšanu), un reģistrācijas veidā norādīt, kurām klasēm šo funkciju izmantot kā papildinājumu.
- Diagnostika - sniedz informāciju par iespējamajām problēmām ietvara konfigurācijā, balstoties uz labajām praksēm (piemēram, dažādu saistītu komponentu dzīves ciklu atšķirības). DI konfigurācija var sagādāt problēmas un tajā ir daudz iespēju kļūdīties, tāpēc diagnostikas funkcionalitāte var ļoti palīdzēt, apgūstot ietvara konfigurēšanu vai veicot atkļūdošanu.

Autora ieteikums ietvara izvēlei - sākotnēji izvēlēties kādu no populārākajiem ietvariem ar konkrētajam programmētājam šķietami vienkāršāko sintaksi. Lielākā daļa populāro ietvaru realizē praktiski visas funkcijas, kas standarta situācijās būs nepieciešamas. Līdz ar to nav vērts pavadīt laiku analizējot dažādu ietvaru iespējas. Ja ar laiku radīsies vajadzība pēc progresīvām funkcijām vai sarežģītākas konfigurācijas, var sākt domāt par DI ietvara maiņu. Tam nevajadzētu būt problemātiski, ņemot vērā idejiskās līdzības starp standarta funkcijām dažādos ietvaros.

2.4. Castle Windsor

Šajā darbā turpmāk tiks aprakstīts tikai viens DI ietvars - *Castle Windsor* (turpmāk tekstā - *Windsor*). Šis ir viens no vecākajiem DI ietvariem, tomēr tas joprojām tiek aktīvi izstrādāts un savu aktualitāti nav zaudējis. Ietvara galvenais pluss - plaši konfigurējamas reģistrācijas iespējas. Tā rezultātā šis ir viens no dažiem ietvariem, kurš var tikt izmantots automātiskai maketu veidošanai (skatīt apakšnodaļu 3.7). Autora izvēles gadījumā šis bija galvenais faktors. Tomēr, sakarā ar ietvara vecumu un plašajām iespējām, sintakse nav tik lasāma un vienkārši saprotama kā citiem, jaunākiem ietvariem.

Abas DI pamatfunkcijas (reģistrēšana un atrisināšana) *Windsor* ietvarā izskatās šādi:

```
container.Register.Component.For<IComponentInterface>.ImplementedBy<ComponentClass>();
container.Resolve<IComponentInterface>();
```

2.5. Konfigurācija un dinamiskā reģistrācija

Galvenais uzdevums, kas jāveic izmantojot DI ietvaru, ir konfigurācija, kas nozīmē definēt, kurš tips atbilst kuram interfeisam. *Windsor* ietvarā tas tiek panākts ar *IWindsorInstaller* instanču palīdzību [15]. Izmantojot primitīvāko variantu, komponenti tiek reģistrēti pa vienam. Individuāla tipu reģistrēšana prasa salīdzinoši daudz manuāla darba un nozīmē, ka katru reizi izveidojot jaunu klasi un interfeisu, tie ir jāpiereģistrē. Tomēr, šo procesu var atvieglot, izmantojot dinamisko reģistrāciju, kura ļauj atlasīt un filtrēt daļu no tipiem, un piereģistrēt uzreiz visus rezultātus, norādot veidu, kādā tam tiks piemeklēts interfeiss [16].

Galvenie filtrēšanas veidi ir šādi:

- *BasedOn<IService>()* - atlasa tipus, kas implementē norādīto interfeisu.
- *Where()* - ļoti elastīgs filtrs, jo parametrā var norādīt lambda izteiksmi. Piemēram:
    - *Where(t => t != typeof(SomeClass));*
    - *Where(Component.IsInNamespace("Namespace.Name")).*
- *Pick()* - vienkārši atlasa visus tipus.

Interfeisa veida norādīšanai ir šādas iespējas:

- *FirstInterface()* - izmanto pirmo no klases implementētajiem interfeisiem. Domāts klasēm, kuras implementē tikai vienu interfeisu un ir attiecībā 1:1, savādāk varētu rasties problēmas (kāda klase netiek reģistrēta u.tml.).

- *DefaultInterface()* - gadījumos, kad klase implementē vairākus interfeisus. Šī funkcija izvēlas to interfeisu, kurš atbilst klases nosaukumam (piemēram, klase "*Service*" un interfeiss "*IService*").
- *AllInterfaces()* - piereģistrē klasi visiem tās implementētajiem interfeisiem.

Piemēri, kā varētu izskatīties pilna konfigurācija (filtrs + interfeisa veids):

```
container.Register(Classes.FromAssemblyNamed("Assembly.Name").Pick().WithServi
ce.FirstInterface().LifestylePerWebRequest());
container.Register(Classes.FromThisAssembly().BasedOn<IBaseService>().WithServ
ice.DefaultInterface().LifestylePerWebRequest());
```

Reizēm var nākties mazliet pielabot vai pārrakstīt konfigurāciju pēc dinamiskās reģistrācijas. Tādos gadījumos ir trīs varianti [17]:

- Reģistrēt klasi atsevišķi, beigās pievienojot *.IsDefault()* izsaukumu. Ja tas netiks izdarīts, reģistrācijas laikā notiks izņēmuma gadījums, jo *Windsor* ietvarā ir paredzēts, ka katram interfeisam drīkst būt reģistrēta tikai viena klase.
- Veicot dinamisko reģistrāciju, beigās pievienot *.IsFallback()* izsaukumu. Tas nozīmē, ka ietvars ļaus šo komponentu pārreģistrēt.
- Piešķirot unikālu nosaukumu ar *.Named("NewComponentName").*

Jebkurā gadījumā, ietvars darbojas pēc sistēmas "pēdējā reģistrācija uzvar". No aprakstītajiem variantiem, autoraprāt, labākais ir *IsDefault()* izmantošana, kas skaidri norāda, ka komponenta reģistrācija tiek pārrakstīta. *IsFallback()* norāda tikai to, ka reģistrācija varētu tikt pārrakstīta, līdz ar to varētu būt problēmas noteikt vai tiešām kādā vietā tiek darīt un tieši kur. Savukārt, unikālu nosaukumu norādīšana tikai lieki apgrūtina reģistrācijas procesu.

2.6. Instanču dzīves cikls

Būtiska daļa no konfigurācijas ir instanču dzīves cikla pārvaldība. Dzīves cikls nosaka, kurā brīdī jārada jauna instance, un kurā brīdī tā ir jāatbrīvo (*dispose*). Biežāk izmantotie varianti ir [18]:

- *Singleton* - katru reizi pieprasot tipu, tiek atgriezta viena un tā pati instance.
- *Transient* - pretējs *singleton*, katru reizi atgriež jaunu instanci.
- *PerWebRequest* - izmanto tīmekļa lietotnēs, katram pieprasījumam tiek izveidota jauna instance. Piemēram, katrs pieprasījums saņem savu datubāzes savienojuma klasi.

Šis sadalījums ir diezgan vispārīgs un ir aktuāls praktiski jebkuram DI ietvaram. Ir pieejami arī daži ļoti reti izmantoti dzīves cikli - *PerThread* un *Pooled*.

2.7. Windsor izmantošana kopā ar ASP.NET MVC

Labākais vaids, kā izmantot DI konteineri, ir uzstādīt tā izmantošanu augstā līmenī. *ASP.NET MVC* sākot ar 3. versiju ir pieejams kontrolieru izveidošanas papildināšanas modelis, kas dod iespēju norādīt klasi, kas būs atbildīga par kontrolieru izveidošanu - *ControllerFactory*. Tas ir jādara pašā lietotnes dzīves cikla sākumā - *Application_Start()*, kas atrodas *Global.asax* failā. Lai to izdarītu jāizsauc *ControllerBuilder.SetControllerFactory*() funkcija, padodot tai *WindsorControllerFactory* klases instanci, kas satur loģiku, kā atrisināt kontrolierus [19].

```
public class WindsorControllerFactory : DefaultControllerFactory

{
    protected override IController GetControllerInstance(RequestContext
requestContext, Type controllerType)
    {
        return (IController)kernel.Resolve(controllerType);
    }
}
```

Tā rezultātā, rakstot kontrolieru klašu kodu, konstruktorā jānorāda nepieciešamās atkarības, un tālāk, kad tiks veikts *HTTP* pieprasījums, *MVC* ietvars katram pieprasījumam izveidos jaunu kontroliera instanci, aizpildot visas nepieciešamās atkarības atbilstoši DI konteinera konfigurācijai.

2.8. Service Locator anti modelis

Līdz ar to vienīgā vieta, kur visā lietotnes kodā parādās DI ietvara norādes un izsaukumi, ir augsta līmeņa sāknēšanas kods. Ietvars savā ziņā tiek apvīts apkārt lietotnei, līdz ar to ir viegli aizstājams.

Ir ļoti būtiski, lai DI ietvara izsaukumi neparādītos nekur citur, jo īpaši zemākos līmeņos (piemēram, biznesa loģikas slānī). Tipisks sliktais piemērs ir *Service Locator* anti-modelis [12 lpp. 154-160], kurš izpaužas kā DI konteinera instances izmantošana zemā līmenī, kur klases saņem konteineri kā atkarību (vai vēl sliktāk, konteineris ir statiska globāla klase), un tālāk pašas veic atkarību piemeklēšanu. Tā rezultātā klase teorētiski kļūst atkarīga no jebkuras citas klases, jo konteineris satur norādes uz jebkuru tajā reģistrēto klasi. Veidojas globālas atkarības, līdzīgi kā statiskas klases, kas ir tipisks piemērs grūti testējamam kodam.

Teorētiski šis modelis var tikt izmantots arī pareizā veidā, tomēr biežāk tas noved pie tā, ka zūd skaidrība par klases atkarībām, un ir ļoti viegli šo modeli izmantot nepareizā veidā.

3. VIENĪBTESTI

Vienībtests ir funkcija, kas darbina citu koda fragmentu izolācijā (parasti vienu funkciju) un pēc tam veic pārbaudes, lai noteiktu darbinātā koda pareizību.

Klasiskajā definīcijā, testējamā klase tiek absolūti izolēta no visām citām klasēm, no kā tā ir atkarīga. Tomēr, dažādos avotos diezgan būtiski atšķiras definīcija tam, cik tālu nepieciešams izolēt klases atkarības. Laikam ejot ir radusies progresīvāka definīcija, kas nosaka, ka testējamais vienums var būt ne tikai viena klase, bet arī klašu kopa [20 lpp. 4] [21]. Respektīvi, var neizolēt tās atkarības, kuras loģiski iederas testējamajā klasē un to iesaistīšana testā dod vairāk labuma, nekā izolēšana. Šajā definīcijā robeža starp vienībtestiem un zema līmeņa integrācijas testiem kļūst ļoti minimāla.

Protams, ir svarīgi arī integrācijas testi, lai pārbaudītu, kā klases strādā kopā un sadarbojas, jo īpaši ar ārējām atkarībām – datubāzi, failu sistēmu utt. Tomēr, šobrīd labā prakse ir apjoma ziņā likt uzsvaru uz vienībtestiem (attiecība ir apmēram 80:20). Tam ir vairāki iemesli:

- Vienībtesti izpildās ātrāk. Galvenokārt tāpēc, ka ir izolētas ārējās atkarības, kuru darbināšana parasti aizņem lielāko daļu no izpildes laika.
- Vienībtesti ir vieglāk uzturami. Tie parasti ir īsāki un skaidrāki. Kļūdu gadījumā var ātrāk identificēt problēmas cēloni, jo tiek darbināts mazāks koda apjoms.

Vienībtesti bieži vien tiek jaukti ar integrācijas testiem, jo neizolē atkarības vai arī darbina pārāk daudz koda un veic pārāk daudz pārbaudes. Šādi testi ir grūti veidojami un uzturami, ir grūti novērtēt to pareizību un saprast, kur ir problēma, kad tests atgriež negatīvu rezultātu. Tas parasti noved pie tā, ka vienā brīdī programmētājs vienkārši beidz šos testus uzturēt un veidot jaunus, jo ieguldītais laiks neatmaksājas. Tāpēc ir svarīgi veidot labus testus.

Labu vienībtestu raksturo šādas īpašības:

- Testam ir jābūt īsam un labi saprotamam. Garš tests, kura darbības nav skaidras acumirklī apskatot kodu, ir grūti uzturams un lasāms, kā arī kļūdu gadījumā ir grūti atrast koda daļu, kurā ir problēma.
- Testam ir jābūt ātri izpildāmam. Sarežģītās sistēmās testu skaits ir mērāms tūkstošos, tāpēc ir svarīgi, lai tie izpildītos ātri. Ja ir izolētās visas ārējās atkarības (datubāze utt.), tad tam tā arī vajadzētu būt. Bieži vien šādu sistēmu izstrāde notiek tādā veidā, ka testi (visi vai daļa, atkarībā cik liela sistēma) tiek automātiski izpildīti brīdī, kad tiek saglabāts pirmkoda fails, līdz ar to ir skaidrs, ka šeit visu testu izpildes laikam jābūt mērāmam sekundēs.

- Testam ir jābūt automātiski izpildāmam. Ja testu ir daudz, tad to manuāla izpilde ir pārāk laikietilpīga, lai testu veidošanā ieguldītais laiks atmaksātos.
- Testam ir jābūt atkārtojamam un konsekventam. Ja tests nav atkārtoti izpildāms, to tikpat labi var vienkārši nerakstīt, jo to vairs nevar automatizēt. Lai to varētu izdarīt:
    - Izpildes sākumā visam, no kā tests ir atkarīgs, ir jābūt paredzamā stāvoklī.
    - Izpildes beigās testa veiktās darbības nedrīkst atstāt paliekošas sekas.
    - Izpildes beigās rezultātam ir jābūt vienādam katrā izpildes reizē.
    - Testu izpildes secība nedrīkst būt svarīga.

3.1. Vienībtestu nozīme

Testu vadītas izstrādes efektivitāte ir pierādīta vairākos darbos [22] [23] – tā uzlabo gan koda kvalitāti (modularitāte, vāji saistītas klases, moduļi), gan samazina kļūdu skaitu (pētījumos minēti 40-90% uzlabojums). Tomēr, tā padara izstrādi lēnāku – pētījumos tiek minēta 15-35% atšķirība. Ne velti ir attīstījušies vairāki progresīvi izstrādes modeļi, kā piemēram TDD (*Test driven development* – izstrādes pieeja, kurā izstrāde apvienota ar vienībtestu veidošanu, fokusējoties uz klašu testēšanu izolācijā) un BDD (*Behaviour driven development* – izstrādes pieeja, kurā izstrāde apvienota ar vienībtestu veidošanu, fokusējoties uz testu scenārijiem, kas atbilst lietošanas piemēriem).

Varētu likties, kas tas ir pietiekami, lai šādi izstrādes modeļi būtu standarts, tomēr praksē tie netiek tik bieži izmantoti.

Viens no biežākajiem iemesliem ir vadības atbalsta trūkums. Vadība neizprot vienībtestu jēgu un redz tos kā lieku laika tērēšanu, jo ir grūti izvērtēt to efektivitāti, jo pozitīvs rezultāts ir redzams tikai pēc pāris iterācijām, kad programmatūrā kļūst arvien grūtāk pieļaut kļūdas, kā arī samazinās izstrādes laiks, jo nav tik daudz jāuztraucas par iespējām ieviest kļūdas jau esošajā kodā.

Attiecībā uz programmētājiem, galvenie iemesli ir laika trūkums un sarežģītība. Attiecībā uz sarežģītību, tīmeklī ir daudz informācijas un piemēru, kā testēt triviālu kodu kuram nav atkarību. Diemžēl, kompleksu un sarežģītu sistēmu testēšana ir tālu no šādiem piemēriem – ir iesaistīta datubāze, klasei ir atkarības vairākos līmeņos, netriviāla biznesa loģika, funkciju rezultāts nav novērojams tiešā veidā (stāvokļa izmaiņas, uzvedība), utt. Līdz ar to par testēšanas iespējamību ir jādomā jau arhitektūras līmenī.

Attiecībā uz izstrādes laiku, autora viedoklis ir tāds, ka sarežģītās sistēmās vienībtestu vadīta izstrāde var pat uzlabot izstrādes laiku. Iemesls ir tāds, ka šādās sistēmās testējot no lietotāja saskarnes, bieži vien jāpavada vairākas minūtes, lai tiktu līdz stāvoklim, kāds nepieciešams testpiemēra izpildei. Šādās situācijās vienībtesti palīdz izstrādi veikt dinamiskāk un ātrāk.

Bieži vien priekšroka tiek dota automātiskiem lietotāja saskarnes testiem. Kamēr tas ir pieņemams variants mazāk sarežģītām sistēmām un labs papildinājums lielākām sistēmām, tomēr no lietotāja puses bieži vien nav iespējams pārbaudīt sarežģītu biznesa loģiku. Turklāt, šie testi ļoti viegli "lūzt", arī pie minimālām lietotnes izmaiņām.

3.2. Modeļi

Konkrēta modeļa izmantošanas mērķis ir ieviest paredzamu, strukturētu veidu, kā izkārtot testpiemēra kodu. Standarta modeļa ievērošana palīdz strukturēt testus, padarīt tos lasāmus un vieglāk saprotamus gan testa autoram, gan citiem programmētājiem, kam var nākties šos testus darbināt.

Standarta modelis ir *Arrange Act Assert (*AAA) [20 lpp. 93-96]. Šajā modelī katrs testpiemērs tiek iedalīts 3 daļās:

- *Arrange* – testa sagatavošanas fāze, izveido testam nepieciešamā klases instanci un atkarības.
- *Act* – darbības fāze, izsauc testējamo funkciju.
- *Assert* – apgalvojuma pārbaudes fāze, veic pārbaudes pret izvirzīto apgalvojumu, lai noteiktu, vai apgalvojums ir patiess.

Iespējams atrast arī citus modeļus, tomēr, salīdzinot var secināt - vai nu tas pats nosaukts savādāk, vai arī neliela variācija. Citi pieejamie modeļi ir:

- *4 Phase Test* - testpiemērs sastāv no 4 daļām - *Setup, Exercise, Verify, Teardown*. Pirmās trīs ir praktiski identiskas, klāt nāk *Teardown*, kas būtībā varētu būt arī neobligāta daļa AAA modelī [24].
- *Given, When, Then* (GWT) – būtībā tas pats AAA, tikai šādi testus pieņemts saukt BDD izstrādes pieejā, kurā izstrāde apvienota ar vienībtestu veidošanu, fokusējoties uz testu scenārijiem, kas atbilst lietojumgadījumiem [25], nevis atsevišķu vienību testēšanu, līdz ar to tas vairāk atbilst akcepttestiem vai augsta līmeņa integrācijas testiem. Kā arī, GWT

pārbaudes fāzē iespējami vairāki pārbaudes apgalvojumi, atšķirībā no AAA modeļa, kur ieteicams izmantot tieši vienu apgalvojumu.

- *Arrange Assert Act Assert* (AAAA) - interesants papildinājums AAA modelim. Pazīstams arī ar nosaukumu "*Guard Assertion*". Pirmais *assert* veic pretēju pārbaudi otrajam, tādā veidā pārbaudot, ka *act* fāzē izpildītā funkcija tiešām ir veikusi nepieciešamās izmaiņas, kas noveda pie otrā *assert* pozitīva rezultāta. Būtībā modelis paredzēts, lai izvairītos no gadījumiem, kad jau uzstādīšanas posmā pārbaudāmais rezultāts jau ir ieguvis beigās vēlamo vērtību [26].

3.3. Pārbaudes

Ir daudz dažādu veidu, kā pārbaudīt testa iznākumu. Katrā ietvarā ir virkne funkciju, kas paredzētas testa iznākumu pārbaudei. Lai mazinātu sarežģītību un ieviestu struktūru definīcijās, to visu var iedalīt trīs kategorijās [27 lpp. 107-113]:

- Atgriezto vērtību pārbaudes - veic pārbaudi vērtībai, ko atgriež testējamā funkcija. Šis ir vienkāršākais un saprotamākais veids.
- Stāvokļa pārbaudes - *void* tipa funkcijām nepieciešams cits veids, kā veikt pārbaudes. Tā kā nekas netiek atgriezts, nākamā lieta ko pārbaudīt ir stāvoklis (atribūta vērtība) vai nu testējamajai klasei, vai citām klasēm, ko SUT (*System Under Test* - testējamās klases instance vienībtestā) izmanto (atkarībām).
- Uzvedības pārbaudes – aktuāli klasēm, kas pašas nerealizē biznesa loģiku, bet nodarbojas ar citu klašu darbību koordinēšanu (piemēram, *MVC* kontrolieris). Šādos gadījumos ir nepieciešams pārbaudīt, vai SUT korekti veic izsaukumus (padod pareizos parametrus utt.). Testa dubultnieku ietvariem ir funkcionalitāte, kas spēj ierakstīt visas ar dubultnieku veiktās darbības, lai testa beigās pārbaudītu to, kas nepieciešams. Pāris vienkāršākie pārbaužu piemēri:
    - Pārbaudīt, vai funkcija vispār tika izsaukta.
    - Pārbaudīt, vai funkcija tika izsaukta, specificējot parametrus.
    - Pārbaudīt, vai funkcija netika izsaukta.

3.4. Vienībtestu ietvari

Vienībtestu rakstīšanā parasti tiek izmantots ietvars, kas dod iespēju strukturēt un grupēt testus armatūrās (*test fixture* – testu klase, kas apvieno vairākus vienībtestus), kā arī veikt apgalvojumu (*assertions*) pārbaudes. Praktiski jebkurš daudz maz populārs testu ietvars

nodrošina šīs iespējas. Galvenās atšķirības ir sintaksē un ideoloģijā. Ietvari parasti tiek izmantoti kombinācijā ar testu izpildītājiem (*test runner*), kuri nodrošina testu automātisku izpildi, statusa (aktīvais tests) un rezultātu (veiksmīgie testi, izpildes laiki) attēlošanu.

3.5. Testu dubultnieki

Testu dubultnieks ir objekts, ar kuru testos tiek aizstāta klases atkarība, lai:

- Kontrolētu ievadu - datus, ko atgriež dubultnieks.
- Pārbaudītu izvadu - atgrieztās vērtības, stāvokli vai uzvedību.

Tā kā klašu atkarības ir interfeisi, nevis konkrētas implementācijas, tad testos uzdevums ir šīs atkarības aizvietot ar testu dubultniekiem, kas implementē tos pašus interfeisus, bet veic citas, kontrolējamas darbības, kas palīdz regulēt testpiemēra gaitu. Piemēram, ja SUT ir atkarīga no failu lasītāja klases (atkarība, ko vajag aizstāt), tad vienībtestā šīs klases instances vietā tiek izmantota citas klases instance, kura implementē to pašu interfeisu, bet tā vietā, lai lasītu kādu failu, tā atgriež jau iepriekš sagatavotu teksta fragmentu.

Testu dubultniekus parasti iedala dažādās kategorijās, atkarībā no to īpašībām un uzdevumiem. Kategoriju skaits atkarīgs no dažādiem autoriem, kā arī ir vērojamas būtiskas atšķirības definīcijās. Piemēram, salīdzinot divas no populārākajām testēšanas grāmatām, avots [20 lpp. 75-90] definē trīs kategorijas:

- Aizbāznis (*stub*) - izolējamā interfeisa implementācija, kas ļauj kontrolēt atgrieztās vērtības un stāvokli.
- Makets (*mock*) - pret šo objektu veic pārbaudes, lai noteiktu, vai SUT pareizi sadarbojas ar atkarību.
- Viltus objekts (*fake*) – vispārīgs apzīmējums testu dubultniekiem, iekļauj sevī gan aizbāžņus, gan maketus.

Savukārt, avots [27 lpp. 133-139] piedāvā 5 kategorijas un būtiski atšķirīgas definīcijas:

- Aizbāznis (*stub*) - ļauj kontrolēt atgrieztās vērtības un stāvokli. Šim terminam definīcijas sakrīt gan abos minētajos avotos, gan arī lielākajā daļa interneta resursu.
- Spiegs (*spy*) - darbojas kā novērotājs, saglabājot pret to veiktos izsaukumus, lai vēlāk veiktu pārbaudes, vai SUT korekti sadarbojas ar atkarību.
- Makets (*mock*) - apvieno aizbāžņa un spiega funkcionalitāti – gan kontrolē atgriežamās vērtības, gan veic izsaukumu pārbaudi.

- Viltus objekts (*fake*) - izmanto, lai aizstātu daļu no funkcionalitātes ar vienkāršotu variantu. Izmanto gan testu nolūkiem, gan arī gadījumos, kad izstrādei nepieciešama funkcionalitāte, kas vēl nav gatava. Mazāk saistīts ar vienībtestiem, vairāk ar izstrādi kā tādu, tomēr šo objektu bieži iedala testu dubultnieku klasifikācijā.
- Manekens (*dummy*) – darbojas vienkārši kā vietas aizvietotājs, fiktīvi nepieciešama atkarība, kas testā vispār netiek izmantota, bet nepieciešama objekta izveidošanai (dēļ konstruktora signatūras).

Kā redzams, definīcijas būtiski atšķiras, un situācija kļūst vēl sarežģītāka, mēģinot salīdzināt ar dažādiem tīmekļa resursiem. Pēc autora pieredzes, tīmeklī visbiežāk tiek izmantots maketa termins, kurš bieži vien jau tiek izmantots kā vispārīgs apzīmējums visiem testu dubultnieku veidiem. Līdzīgu filozofiju izmanto arī lielākā daļa testu dubultnieku ietvaru, kuru izmanto terminu "makets", lai apzīmētu visas dubultnieku kategorijas.

3.6. Testu dubultnieku ietvari

Testu dubultniekus var rakstīt manuāli, tomēr ar laiku nākas secināt:

- Tas aizņem pietiekami daudz laika, jo dubultnieks ir jāveido visiem interfeisiem, ko vajag aizvietot.
- Testu dubultniekiem nav nepieciešama ļoti pārdomāta funkcionalitāte, parasti tas ir kaut kas ļoti primitīvs, vispārīgs.
- Bieži vien nepieciešama tikai daļa no dubultnieka funkcijām (pārējās testā vispār netiek izsauktas). Manuāli implementējot interfeisu nepieciešams definēt visas funkcijas, pat, ja tikai jāuzraksta funkcijas galvene, tas tā pat aizņem laiku.

Līdz ar to laika gaitā radusies vesela virkne ar ietvariem, kas risina šo problēmu, ļaujot programmētājiem koncentrēties uz testu dubultnieku maksimāli ātru veidošanu. Lai gan dubultnieks tiek izveidots ar vienas rindas palīdzību, joprojām ir iespēja tam piešķirt konkrētu funkcionalitāti tajās daļās, kur tas nepieciešams, lai kontrolētu testa gaitu. Līdz ar to objekta nesvarīgās daļas kļūst ļoti vispārīgas, savukārt svarīgās daļas joprojām ir iespējams kontrolēt.

Šobrīd populārākie varianti ir:

- *Moq*
- *NSubstitute*
- *FakeItEasy*

Līdzīgi kā vienībtestu ietvariem, arī šeit visi populārākie ietvari piedāvā daudz maz vienādu funkcionalitāti, atšķirības ir galvenokārt sintaksē. Līdz ar to izvēle šeit ir subjektīva un ne pārāk svarīga.

Autors sākotnēji ilgu laiku izmantoja *Moq* ietvaru, tomēr laika gaitā nonāca pie secinājuma, ka *NSubstitute* ietvaram ir daudz labāka (lasāmāka) sintakse, kas ir ļoti svarīgi, jo testpiemēram ir jābūt labi saprotamam.

Salīdzinājumam attēlotas standarta funkcijas abos ietvaros – izveidošana, atribūta uzstādīšana, funkcionalitātes definēšana un izsaukumu pārbaude, piekļūšana objektam.

```
// maketa izveidošana
var mock = Substitute.For<IAnyClass>(); // NSubstitute
var mock = new Mock<IAnyClass>(); // Moq

// atribūta uzstādīšana, definē, ka  SomeAttribute = 1
mock.SomeAttribute = 1; // NSubstitute
mock.Setup(m => m.SomeAttribute).Returns(1); // Moq

// funkcionalitātes definēšana - izsaucot funkciju DoSomething ar parametru 1,
tiks atgriezta vērtība 2
mock.DoSomething(1).Returns(2); // NSubstitute
mock.Setup(m => m.DoSomething(1)).Returns(2); // Moq

// izsaukumu pārbaude - pārbauda, vai makets saņēmis izsaukumu funkcijai
DoSomething ar jebkādu int tipa parametru
mock.Received().DoSomething(Arg.Any<int>()); // NSubstitute
mock.Verify(m => m.DoSomething(It.IsAny<int>())); // Moq

// piekļūšana objekta instancei un tā atribūtiem apgalvojuma pārbaudes
veikšanai
Assert.AreEqual(1, mock.SomeAttribute); // NSubstitute
Assert.AreEqual(1, mock.Object.SomeAttribute); // Moq
```

Līdz ar to, turpmāk darbā tiks aprakstīts tikai *NSubstitute* ietvars. Līdzīgi kā lielākajā daļā populāro ietvaru, arī šis izlīdzina robežas starp dubultnieku kategorijām. Līdz ar to nav jādomā, kādu dubultnieka instanci veidot (aizbāzni, maketu, spiegu), pietiek vienkārši pateikt, ka nepieciešams dubultnieks interfeisam, izmantojot:

```
var testDouble = Substitute.For<IAnyClass>();
```

kur *IAnyClass* vietā var būt jebkurš interfeisa tips. Tālākā objekta izmantošana noteiks, kurā kategorijā tas ietilpst, bet būtībā tas nav svarīgi. Piemēram, ja pret šo objektu veicam pārbaudi, vai tika izsaukta funkcija, varam šo dubultnieku uzskatīt par maketu.

3.7. DI ietvara izmantošana vienībtestos

Parasti DI ietvarus testos nemēdz izmantot, bet autors ir saskāries ar dažām problēmām, kad DI ietvars var būt noderīgs:

- Tipiska problēma testos ir to "salūšana", izmainoties klases konstruktora signatūrai, precīzāk, parametru skaitam vai tipiem. Piemēram, ja tiek testēta klase, kura ir izstrādes stadijā, tad ļoti iespējams, ka tā var bieži mainīties, piemēram, ja izrādās, ka tai nepieciešama vēl kāda atkarība, ko padod caur konstruktoru.
- Intensīvi rakstot testus, ar laiku varētu rasties secinājums, ka pārāk daudz laika un koda rindu aizņem testa sagatavošanas fāze - daudz koda tiek rakstīts, radot testējamās klases atkarību instances. Praktiski tas nozīmē viena rinda koda katras atkarības izveidošanai. Un tas ir katrā testpiemērā.

Risinājuma būtība ir izmantot DI konteineri mazliet citam mērķim, nekā tas oŗģināli bija paredzēts – dinamiski aizstāt visas testējamās klases atkarības ar dubultniekiem. Šo tehniku sauc par automātisko maketu veidošanu (*auto-mocking*) [28].

Lai to izdarītu, testos izmantotajai konteinera instancei nepieciešama specifiska konfigurācija. Ja šāda konfigurācija ir uzstādīta, tad, veicot tipa atrisināšanu, tiek atgriezta instance, kurai visas atkarības aizstātas ar dubultnieku instancēm. Tādā veidā tiek aiztaupītas tik daudz koda rindas, cik atkarību ir testējamajai klasei. Ņemot vērā, ka šis ietaupījums ir katram testa piemēram, tad ieguvums ir ļoti būtisks.

Lai realizētu šo metodi ar darbā izmantotajiem ietvariem (*Windsor* un *NSubstitute*), ir divi varianti.

3.7.1. ILazyComponentLoader

Pirmais variants ir *ILazyComponentLoader* interfeisa izmantošana (daļa no *Windsor* ietvara). Oŗģināli šis interfeiss ir paredzēts, lai varētu reģistrēt komponentus pēc tam, kad jau ir izveidota DI konteinera instance [29]. Parasti to izmanto gadījumos, kad konteinera inicializēšanas brīdī vēl nav zināms, ar kādu tipu būs realizēts kāds interfeiss.

Vienībtestu kontekstā tam ir alternatīvs pielietojums. Šo interfeisu ir iespējams izmantot, lai panāktu efektu, kad konteineris, atrisinot klasi, atgriež patieso implementāciju testējamajai klasei, bet dubultnieku instances visām šīs klases atkarībām.

Lai to izdarītu, nepieciešams izveidot *ILazyComponentLoader* instanci, kura dinamiski atgriež *NSubstitute* objektus, veicot tipa atrisināšanu [30].

```
public class LazyAutoMocker : ILazyComponentLoader
{
    public IRegistration Load(string key, Type service, IDictionary arguments)
    {
        return Component.For(service).Instance(
            Substitute.For(new[] { service }, null));
    }
}
```

Pēc tam, vienībtestos reģistrējot konteinera komponentus, jāpiereģistrē *LazyAutoMocker* instance:

```
var container = new WindsorContainer();
container.Register(Component.For<LazyAutoMocker>());
container.Register(Component.For<TClassUnderTest>());
```

Kā rezultātā, pēc tam testā veicot SUT atrisināšanu no konteinera, tiek atgriezta testējamās klases instance, kur visas atkarības ir aizstātas ar testu dubultniekiem objektiem:

```
var sut = container.Resolve<TClassUnderTest>();
```

3.7.2. ISubDependencyResolver

Iepriekšējās metodes izmantošanai ir viena problēma - katras testu armatūras uzstādīšanas posmā jāveic reģistrēšana gan *LazyAutoMocker*, gan SUT tipiem. Ja tas jādara katrā armatūrā, tas aizņem pietiekami daudz laika, lai būtu vērts meklēt labāku risinājumu. Ērtāk būtu veikt konteinera uzstādīšanu citā līmenī, bet joprojām saglabājot principu, kad klasei visas atkarības tiek automātiski aizstātas ar testu dubultniekam.

Windsor ietvara *ISubDependencyResolver* interfeiss ir paredzēts, lai konfigurētu, kā tiks atrisinātas tipa atkarības, bet ne pats tips. Vienībtestu nolūkos šo iespēju var ļoti labi izmantot, pievienojot konteinerim *ISubDependencyResolver* konfigurāciju [28]:

```
public class AutoNsubMockResolver : ISubDependencyResolver
{
      private IKernel _kernel;
```

```
        public AutoNsubMockResolver(IKernel kernel)
        {
              _kernel = kernel;
        }

        public bool CanResolve(...) { ... }
        public object Resolve(...) { ... }
}
```

Šī konfigurācija liek dinamiski atgriezt dubultnieka instanci katrai tipa atkarībai. Lai to izdarītu, *ISubDependencyResolver* funkcija *Resolve()* jāimplementē šādā veidā:

```
public object Resolve(CreationContext context,
  ISubDependencyResolver contextHandlerResolver,
  ComponentModel model,
  DependencyModel dependency)
        {
            var targetType = dependency.TargetType;
            return Substitute.For(new[] { targetType }, null);
        }
```

Tālāk atliek tikai piereģistrēt konteinerim izveidoto *ISubDependencyResolver* instanci, un vēlamais efekts ir panākts.

```
container.Kernel.Resolver.AddSubResolver(
    new AutoNsubMockResolver(container.Kernel));
```

3.7.3. Dubultnieku modificēšana

Izmantojot šo metodi, var rasties vajadzība piekļūt testu dubultnieku instancēm. Parasti tam ir trīs iemesli.

- Kādu no dubultniekiem nepieciešams aizstāt ar īstu instanci (ja SUT nepieciešams testēt integrācijā ar patieso realizāciju, nevis dubultnieku).
- Kādam no dubultniekiem nepieciešams piešķirt daļēju funkcionalitāti. Visbiežāk tas ir tāpēc, ka SUT no dubultnieka izsauc funkcijas, kas atgriež datus un tālāk ar tiem veic darbības. Ja dubultniekam netiks piešķirta funkcionalitāte, tas šādos gadījumos atgriezīs *null*, kas rezultēsies ar tūlītēju testa pārtraukšanu dēļ *NullReferenceException*.
- Pret kādu no dubultniekiem nepieciešams veikt pārbaudes, vai nu stāvokļa (lauka vērtība) vai uzvedības (vai funkcija tika izsaukta).

Lai piekļūtu dubultnieku instancēm, ir divi varianti.

Pirmais ir vienkārši konkrētā tipa atrisināšana no konteinera. Šis variants strādā tikai gadījumos, kad atrisināmā tipa dzīves cikls ir *singleton*. Tā kā vienībtestos parasti tiek izmantots *singleton* dzīves cikls, tad šis ir saprātīgs pieņēmums. Tātad testos, pieprasot dubultnieka tipu konteinerim, tiks atgriezta aktuālās atkarības instance, kurai tad būs iespējams piešķirt funkcionalitāti.

```
var sut = _container.Resolve<SomeClass>();
// atgriež to pašu instanci, kas ir SomeClass atkarība
var dep = _container.Resolve<SomeDependency>();
// definē funkcionalitāti
dep.DoSomething(1).Returns(2);
```

Viena būtiska problēma šai metodei ir testu lasāmības sarežģīšana. Vienkārši apskatot kodu, varētu uzreiz nelikties pašsaprotami, ka atrisinot tipu, tiek atgriezta tā pati instance, kas tiek izmantota kā atkarība SUT klasei. Lai to izdarītu, ir jāpēta konteinera konfigurācija.

Lai to risinātu, ir otrs variants - atkarību instanču publiskošana testējamajā klasē. Parasti atkarības tiek padotas kā konstruktora parametri, un tiek saglabātas kā mainīgie klases iekšpusē. Ja šie mainīgie būtu publiski, tad testā izveidojot SUT instanci tiem varētu piekļūt ļoti vienkārši, turklāt saglabājot testa lasāmību. Šī metode krietni uzlabo testa lasāmību, jo uzreiz ir skaidrs, no kurienes nāk atkarību instances, piemēram, šādi izskatās kods, kurš piekļūst atkarības instancei un papildina funkcionalitāti:

```
sut.someDependency.DoSomething(1).Returns(2);
```

Šādā veidā ir uzskatāmi redzams, ka piekļūst testējamās klases atribūtam, kas ir injicētā atkarība.

4. STARPNOZARU PROBLĒMAS

Starpnozaru problēmas ir funkcionalitāte, kas neattiecas uz kādu konkrētu biznesa loģiku, bet gan uz visu sistēmu, un tiek izmantota vairāku biznesa sadaļu realizācijā. Tipiski piemēri ir:

- Auditēšana. Piemēram, audita tabulā pierakstīt, pieprasījuma izpildi, datus, laiku, lietotāju utt.
- Kļūdu apstrāde. Audita tabulā veikt ierakstus par noteikušajām kļūdām, to laikiem, lietotājiem.
- Drošība. Autentifikācija un autorizācija. Pirms pieprasījuma apstrādes pārbaudīt, vai lietotājs drīkst izmantot šo resursu.
- Transakciju kontrole.
- Veiktspējas skaitītāji. Piemēram, funkcijas izpildes sākumā uzņemt laiku, beigās apstādināt un saglabāt vēlākai analīzei. Šis vairāk attiecas uz ne-augstākā līmeņa funkcijām, jo augšējā slāņa izpildes laiku reģistrēšanu parasti veic tīmekļa serveris savos audita žurnālos.
- Kešatmiņa. Pieprasījumu rezultātu saglabāšana jau tīmekļa servera līmenī.

Aspektorientētā programmēšana (AOP) ir tehnika, kas palīdz šīs problēmas realizēt ārpus biznesa loģikas klasēm, lai palīdzētu saglabāt vienas atbildības principu - *single responsibility principle* no SOLID (skatīt apakšnodaļu 1.1). Biznesa loģikas klasei nevajadzētu neko "zināt" par auditēšanas funkcionalitāti – tas neattiecas uz biznesa funkcijām un biznesa cilvēkiem (sistēmas lietotājiem) šāds termins vispār neinteresē. Tādējādi, modelējot sistēmas klases atbilstoši dzīvei, arī šeit vajadzētu nodalīt lietas.

Ir vairāki veidi, kā ASP.NET MVC lietotnē realizēt funkcionalitāti, izmantojot AOP tehnikas. Daži no tiem ietver projektējuma modeļu izmantošanu, viens izmanto ASP.NET iespējas, savukārt vēl viens paļaujas uz DI ietvaru piedāvātajām iespējām. Visi veidi balstās uz vienu ļoti vienkāršu principu – injicēt papildus loģiku pirms un pēc funkcijas izsaukuma.

4.1. AOP realizācija ar projektējuma modeļiem

Ir pieejami dažādi projektējuma modeļi, ar kuru palīdzību bez papildus ietvaru vai konkrētu tehnoloģiju izmantošanas var realizēt AOP principus. Līdz ar to arī iespēju un elastīguma ir krietni mazāk, toties nav nepieciešami smagi uzstādīšanas darbi, kā tas ir abos pārējos risinājumos (ASP.NET filtri un pārķeršana).

4.1.1. Hole in the middle

Šis modelis ir ļoti atbilstošs savam nosaukumam. Būtībā tā ir funkcija, kas kā parametru saņem citu funkciju, lai pirms un pēc parametra funkcijas izsaukuma izpildītu papildus kodu [31]. Piemēram, šādi izskatītos transakciju metode:

```
public T UseTransaction<T>(Func<T> fnToCall)
{
      var scope = new TransactionScope();
      try
      {
            var result = fnToCall();
            transactionScope.Complete();
            return result;
      }
      catch
      {
            transactionScope.Dispose();
            throw;
      }
}
```

Būtisks pluss – šo metodi var izmantot jebkurā slānī, jebkurā vietā. Atšķirībā, piemēram, no kontrolieru bāzes vai ASP.NET filtriem, kuri ir pieejami tikai pašā augšējā – kontrolieru līmenī.

4.1.2. Kontrolieru funkciju pārrakstīšana

ASP.NET kontrolieru klasēm ir pieejamas divas funkcijas - *OnActionExecuting* un *OnActionExecuted,* kuras var pārrakstīt, lai ievietotu funkcionalitāti pirms un pēc [32]. To var izmantot viena kontroliera ietvaros, vai arī globāli, izveidojot kontrolieru bāzes klasi, no kuras mantos visi pārējie kontrolieri. Būtisks mīnuss – metodes elastīgums. Tā attiecas vai nu uz visām kontroliera metodēm, vai nevienu. Līdz ar to ne visas starpnozaru problēmas var risināt šādā veidā. Piemēram, ne visām kontroliera metodēm var būt nepieciešams izmantot transakcijas.

4.2. ASP.NET filtri

ASP.NET piedāvā savu risinājumu AOP realizēšanai tīmekļa lietotnēs. Filtri ļauj injicēt loģiku pirms un pēc pieprasījuma apstrādes. Filtri dalās vairākos tipos, bet, autoraprāt, elastīgākais un biežāk izmantotais ir darbības filtrs (*action filter*). Lai izveidotu darbības filtru, jāizveido klase atbilstoši šādiem nosacījumiem [33]:

- Manto no klases *ActionFilterAttribute*.
- Pārraksta (*override*) funkciju *OnActionExecuting*, kura izpildīsies pirms pieprasījuma apstrādes.
- Pārraksta funkciju *OnActionExecuted*, kura izpildīsies pēc pieprasījuma apstrādes.

Piemēram, auditēšanas filtrs varētu izskatīties šādi:

```
public class AuditAttribute : ActionFilterAttribute, IPropInjectable
{
    public IAuditLog _auditLog { get; set; }

    public override void OnActionExecuted(ActionExecutedContext context)
    {
      // POST dati no pieprasījuma
      var req = context.RequestContext.HttpContext.Request.Form;
      _auditLog.log(JsonConvert.SerializeObject(req));
    }
}
```

Sasaisti ar pārējo sistēmu šeit nodrošina *ActionExecutedContext* objekts, ko kā parametru saņem pārrakstītā *OnActionExecuted* funkcija. Tas satur informāciju par pieprasījumu, MVC kontroliera instanci, kontroliera atgrieztos datu, informāciju par izņēmuma situāciju (ja tāda bija) utt.

Kad filtra klase ir radīta, to var piereģistrēt trīs dažādos veidos [33]:

- Globāli. Failā *FilterConfig.cs* statiskajai funkcijai *RegisterGlobalFilters* pievienojot izveidoto filtru.
- Visam kontrolierim. Pievienojot kā atribūtu (filtra nosaukums kvadrātiekavās) kontroliera galvenei.
- Funkcijai. Pievienojot kā atribūtu funkcijas galvenei.

Reti kad filtrs spēs veikt nozīmīgu darbību bez citu klašu līdzdalības. Ņemot vērā, ka filtru instanču veidošana notiek ASP.NET MVC iekšienē, praktiski nav kontroles pār šo procesu. Līdz ar to rodas problēma – kā padot nepieciešamās atkarības filtriem.

4.2.1. ControllerActionInvoker

Labākais variants šeit ir *ControllerActionInvoker* izmantošana. Tā ir ASP.NET paplašināšanas (*extensibility*) vieta, kura dod iespēju iesprakties ASP.NET pieprasījumu apstrādes procesā. Šajā gadījumā to var izmantot, vai jau gatavām filtru instancēm injicētu

atkarības. Tā kā instances jau ir izveidotas, tad konstruktora injekcija šeit nav iespējama. Šis ir viens no retajiem gadījumiem, kad noderīga un ieteicama ir atribūtu injekcija, jo būtībā citu variantu nav.

Lai injicētu atkarības, nepieciešams *ControllerActionInvoker* klasei pārrakstīt pāris funkcijas – *InvokeActionMethodWithFilters*, *InvokeActionResultWithFilters*, *InvokeExceptionFilters*. Šīs funkcijas atbild par pieprasījuma apstrādes izsaukšanu ar filtriem. Rezultāts izskatās apmēram šādi [34]:

```
protected override ActionExecutedContext
InvokeActionMethodWithFilters(ControllerContext controllerContext,
IList<IActionFilter> filters, ActionDescriptor actionDescriptor,
IDictionary<string, object> parameters)
{
   // iet cauri jau izveidotajām filtru instancēm
   foreach (IActionFilter actionFilter in filters)
   {
       // Filtru klases, kurām nepieciešams veikt injicēšanu,
       // implementē IPropInjectable interfeisu
       if (actionFilter is IPropInjectable)
       {
            _kernel.InjectProperties(actionFilter);
       }
    }
    return base.InvokeActionMethodWithFilters(controllerContext, filters,
actionDescriptor, parameters);
}
```

Windsor ietvars pēc noklusējuma nesatur atribūtu injicēšanu jau gataviem objektiem, tomēr var atrast realizācijas piemērus [34]. Idejas būtība – ciklā iet cauri visiem objekta atribūtiem (izmantojot C# reflekciju) un mēģina atrisināt tipu no konteinera un uzstādīt vērtību.

4.2.2. Filtru kešdarbe

Diemžēl, lielākajā daļā internetā atrodamo pamācību nav pieminēts būtisks fakts. ASP.NET MVC sākot ar 3. versiju pēc noklusējuma veic filtru kešdarbi [35]. Tas nozīmē, ka viena un tā pati filtra instance var tikt (visdrīzāk arī tiks) izmantota vairākos pieprasījumos. Ņemot vērā, ka filtros tiek implementēta transakciju loģika, DB (datubāze) savienojums u.c. svarīgas lietas, sekas šādu objektu izmantošanai starp pieprasījumiem ir neparedzamas, grūti atkļūdojamas un sliktākajā gadījumā var radīt mistiskas kļūdas.

Ir pārsteidzoši grūti atrast risinājumu šai problēmai, autoram ir izdevies atrast tikai vienu nelielu ierakstu par šo tēmu [36]. Risinājums pats par sevi gan ir diezgan vienkāršs – aizvietot ASP.NET noklusēto filtru veidotāju ar jaunu instanci, kurai konstruktora parametrā var norādīt, ka nedrīkst veikt kešdarbi filtriem (parametrs *cacheAttributeInstances*), respektīvi, katram pieprasījumam izveidot jaunas filtru instances.

Filtru veidošanā ir ļoti vienkārši kļūdīties:

- Neliela kļūda DI konfigurācijā.
- Netiek izmantots filtru radītājs ar atslēgtu kešdarbi.
- Jaunas izmaiņas ASP.NET ietvarā, līdzīgi kā pārejot no 2. uz 3. versiju.

Kļūdas rezultāts ir grūti identificējams un var tikt nepamanīts ļoti ilgi. Autora gadījumā veicot izstrādi vairākas nedēļas nebija nekādu būtisku pazīmju, tās parādījās, kad sistēmu testēšanas nolūkos mēģināja izmantot vairāki lietotāji. Ņemot vērā, ka filtri realizē transakcijas un DB savienojumu, nekādas kļūdas šeit nav pieļaujamas.

Līdz ar to, autors iesaka izmantot aizsargājošās programmēšanas (*defensive programming*) tehniku kombinācijā ar ātru kļūdu atrašanu (*fail fast*). Idejas būtība – veikt pārbaudes sistēmas darbības laikā, lai nekavējoties atklātu situāciju, ja tomēr filtra instance tiek saglabāta un pielietota vairākas reizes. Šī ir situācija, kuru automātiskajos testos praktiski nevar atkārtot.

```
public class AuditAttribute : ActionFilterAttribute, IPropInjectable
{
   public IAuditLog _auditLog { get; set; }
   private int _callCount = 0;
   public override void OnActionExecuted(ActionExecutedContext context)
   {
      // šeit ir filtra funkcionalitāte, pēc tās pārbauda izsaukumu skaitu
      AssertThatNotMultipleCalls();
   }
   private void AssertThatNotMultipleCalls()
   {
      _callCount++;
      if (_callCount > 1)
      {
         throw new Exception("Filtra atribūts saņēmis vairākus izsaukumus!");
      }
   }
}
```

Neilgi pēc šī aizsardzības mehānisma ieviešanas, autoram izdevās pārliecināties par tā lietderīgumu. Pateicoties tam tika atrasta nepilnība iepriekš aprakstītajā risinājumā vai pareizāk kļūda ASP.NET – globālie filtri gluži vienkārši neņem vērā *cacheAttributeInstances* parametru, līdz ar to globālajiem filtriem pēc noklusējuma tiek veikta kešdarbe pat uzstādot parametru. Līdz ar to globālajos filtros nevar realizēt funkcionalitāti, kas prasa atkarības, kurām nedrīkst veikt kešdarbi. Autora gadījumā tas nozīmē, ka globālie filtri vispār nav izmantojami. Atliek vien izmantot kontrolieru līmeņa filtrus, galvenais neaizmirst pievienot filtra atribūtu, veidojot jaunus kontrolierus.

4.2.3. Alternatīvi varianti

Ir pieejami arī daži citi varianti atkarību padošanai.

IFilterProvider ir vēl viens veids, kā iespraukties ASP.NET pieprasījuma apstrādes procesā, idejiski līdzīgs iepriekš aprakstītajam variantam, būtībā tas pats, tikai realizēts mazliet savādāk. Pārrakstot funkciju *GetActionAttributes* var piekļūt jau izveidotajām filtru instancēm un injicēt tajās atkarības [37]. Lai lietotne izmantotu šo filtru veidotāju kā noklusēto, nepieciešams atreģistrēt veco un piereģistrēt jauno, līdzīgi kā sadaļā 4.2.2, kur notiek vecā filtru veidotāja aizstāšana ar jaunu, kurš neveic kešdarbi. Salīdzinot ar pirmo variantu, mīnusu nav, jo pieejas ir stipri līdzīgas.

Gadījumā, ja nu tomēr ļoti nepieciešams veikt kešdarbi filtriem, variants ir atkarību instances glabāt *HttpContext.Items* vārdnīcā [38], kurai var piekļūt caur *OnActionExecuted* padoto kontekstu. Būtībā tā ir datu/objektu glabātuve, kas attiecas uz konkrēto pieprasījumu. Filtra instances nebūs unikālas, toties *Items* vārdnīca saturēs konkrētās sesijas mainīgos, tātad unikālus. Mīnuss tāds, ka jāatrod kāds labs brīdis, kad instances ievietot vārdnīcā. Kā arī, instancēm piekļūst kā jau vārdnīcai (ar *string* atslēgu), kas nozīmē, ka nebūs nekādas kompilēšanas laika drošības. Arī mainīgo nosaukumu izmaiņu gadījumā var rasties problēmas.

Ļoti nepareizs, bet bieži izmantots variants ir statiska DI konteinera izmantošana, kas būtībā ir *Service Locator* anti-modelis (skatīt apakšnodaļu 2.8). Ņemot vērā visas filtru problēmas, šis būtu pieļaujams variants, bet tikai kā pēdējais, ja kaut kādu iemeslu dēļ pārējie varianti nestrādā vai arī nav iespējams tos izmantot.

4.3. DI ietvari, pārķeršana un starpniekklases

Šī ir funkcionalitāte, ko atbalsta tikai daļa no DI ietvariem. Piemēram, no populārākajiem ietvariem *Windsor, Autofac, StructureMap* atbalsta, savukārt *Ninject* un Spring.NET – ne [14].

Pārķeršana (*interception*) ir veids, kādā DI ietvaros tiek realizēta AOP tehnika. *Windsor* ietvarā tas notiek, izmantojot starpniekklases (*dynamic proxy*) [39]. Izveidojot pārķērēju, iespējams definēt, kādas darbības veikt pirms un pēc pārķertās funkcijas izsaukuma. Lai pievienotu klasei pārķērēju, ir divi varianti [39]:

- Norādīt to kā atribūtu klases galvenē.
- Norādīt, veicot klašu reģistrāciju (konteinera instalācijā), izmantojot *Interceptors*() konfigurācijas funkciju.

Otrais variants ir elastīgāks, tomēr ne pārāk uzskatāms. Nav uzreiz skaidri redzams, ka klase izmanto kādu pārķērēju, tas viss atrodams tikai vietā, kur klase tiek piereģistrēta DI konteinerī. Pirmais variants lasāmības un saprotamības ziņā ir daudz pārāks.

Ir divas būtiskas atšķirības no ASP.NET darbības filtru varianta.

ASP.NET filtru iespējams reģistrēt tikai pašā augšējā izpildes līmenī, respektīvi – globāli (visiem kontrolieriem), kontrolierim (visām funkcijām) vai konkrētai kontroliera funkcijai. Zemākos līmeņos filtru nav iespējams pievienot. Pārķeršana savukārt dod iespēju filtru pievienot jebkurai klasei.

Lai gan pārķeršanu var izmantot jebkurai klasei, tā ir paredzēta izmantošanai vienmēr visām klases funkcijām, kas ne vienmēr ir vēlams. To var apiet, bet risinājums nav elegants.

Lai izmantotu pārķērēju specifiskai funkcijai, nevis visām klases funkcijām, ir divi varianti:

- Filtrēšana pēc funkcijas nosaukuma pārķērēja klasē, pārrakstot *CanIntercept()* funkciju [39]. Mīnuss – saprotamība. Informācija par pārķertajām funkcijām atrodas filtrā, nevis pie klases.
- Atribūtu izmantošana. Viens atribūts, lai klasei norādītu pārķērēju. Otrs, lai filtrētu funkcijas, kurām izmantot filtru. Izmanto to pašu *CanIntercept(),* bet nevis pēc funkcijas nosaukuma, bet gan pēc atribūta esamības [40].

Diskutējams ir jautājums, kas ir atbildīgs par pārķērēja izsaukšanu, respektīvi, vai pareizāks būtu pirmais vai otrais variants. Skatoties no OOP un SOLID principiem, biznesa funkcionalitātes klasei nebūtu nekas jāzina par starpnozaru problēmām, piemēram, auditēšanu. Līdz ar to, pareizāk būtu ļaut pašam pārķērējam izlemt, ko vajadzētu auditēt, tātad – pārķeramās klases un funkcijas definēt, konfigurējot konteineri. No otras puses, otrais variants daudz skaidrāk un saprotamāk norāda, ka un kur tiek izmantota pārķeršana. Rezumējot – izvēle starp SOLID principu ievērošanu vai lasāmību. Autors uzskata, ka abi varianti ir pieņemami, galvenais – ievērot konsistenci jeb izmantot vienmēr vienu variantu.

4.4. Secinājumi

Diskutējams ir jautājums par to, vai visas starpnozaru problēmas vienmēr var atrisināt ar AOP palīdzību – bieži vien ir nepieciešama specifiskāka informācija un nepietiek tikai ar koda "ievietošanu" pirms un pēc funkcijas izsaukuma. Piemēram, ar auditēšanu ne vienmēr viss ir tik vienkārši. Ir lietas, kuru auditēšana ir praktiski identiska, piemēram, pieprasījuma beigās saglabāt lietotājvārdu, datumu, laiku, pieprasījuma un atbildes datus. Tomēr bieži vien ir vajadzība veikt auditēšanu specifisku funkciju iekšienē un tādos gadījumos katra situācija ir atšķirīga un nevar paļauties uz vienotu mehānismu, ka saglabāt nepieciešamo informāciju.

Autora ieteikums – izmantot virspusīgu auditēšanu filtru līmenī, kas ieraksta pieprasījuma izpildi, lietotāju, pieprasījuma/atbildes datus un citas vispārīgas lietas. Specifiskākiem datiem izmantot atsevišķus izsaukumus pa tiešo no biznesa klasēm. Ja nu tiešām principiāli vajag saglabāt SOLID principu ievērošanu, var izmantot DI piedāvātās starpnieku (*proxy*) un pārķeršanas (*interception*) iespējas AOP kontekstā.

5. DATU PIEKĻUVES SLĀNIS

ORM jau sen vairs nav nekas jauns, tomēr, autoraprāt, pieejams pārāk maz informācijas par to izmantošanu dažādās sarežģītās situācijās, jo īpaši kombinācijā ar mantotām sistēmām un eksistējošu datubāzi. Šajā sadaļā tiks aprakstītas dažādas problēmas un risinājumi, ar ko autors saskārās darba praktiskās daļas izstrādes gaitā.

Pamācībās ORM kā tādi (tai skaitā EF) izskatās ļoti eleganti – pilnīga datubāzes ignorance, relāciju datubāze esošie dati kļūst apstrādājami OOP manierē un vēl citas labas lietas. Autora pieredzētais ir tālu no aprakstītā. Situāciju ļoti sarežģīja šādi apstākļi:

- *Oracle* datubāze. Bez šaubām, EF visbiežāk tiek izmantots kopā ar *MSSQL Server*, līdz ar to lielākā daļa no informācijas internetā ir par šo kombināciju. Tomēr, kā jau visi ORM, EF tiek veidots kā maksimāli neatkarīgs no DBVS (datubāzes vadības sistēma) tehnoloģijām, līdz ar to tam vajadzētu pietiekami labi strādāt arī ar citām DBVS.
- Mantotā sistēma un eksistējoša datubāze. Lielākā daļa EF aprakstu, piemēru, pamācību koncentrējas uz svaigas datubāzes veidošanu no EF izveidotajām klasēm. Eksistējošas datubāzes izmantošana ir krietni sarežģītāka – esošās tabulas, relācijas un citas lietas jāmēģina iemānīt EF modelī. DB shēma ir tāda, kāda ir, un iespējas to izmainīt dēļ ORM vajadzībām ir stipri minimālas. Atliek cerēt, ka ORM spēs piemēroties esošajai datubāzei. Kā izrādās, ir vairākas lietas, kas, pēc autora domām, ir absolūti nepieciešamas, tomēr nav realizētas EF.

Tiks aprakstītas EF 5. un 6. versija – tās abas ir pietiekami jaunas (6. ir pagaidām jaunākā) un autoram ir pieredze darbā ar šīm abām versijām kombinācijā ar Oracle DB.

5.1. EF modeļa stratēģijas

EF piedāvā divas stratēģijas, kā veidot modeli.

- *Model First*. Modelis tiek veidots vizuālajā saskarnē. Būtībā XML fails.
- *Code First*. Modelis tiek veidots kā C# klases.

Katrai no šīm stratēģijām ir divi varianti – ģenerēt modeli no eksistējošas DB, vai arī sākumā izveidot modeli, un tad uzģenerēt datubāzi.

Šajā darbā tiks apskatīts tikai *Code First* modelis ar eksistējošu datubāzi. Pirmkārt tāpēc, ka praktiskā darba ietvaros tiks izmantota esoša datubāze, uz to balstās šis darbs. Otrkārt, Code First

ir jaunāks un modernāks veids, kā pārvaldīt EF modeli. Treškārt, *Model First* ģenerē EDMX failu, kas ir grūti pārvaldāms. Grūtības rodas tajā brīdī, kad automātiski uzģenerētais modelis nav līdz galam korekts (piemēram, nav atrasta primārā atslēga) vai nepieciešams veikt izmaiņas (piemēram, mainīt datu tipus, lai būtu iespējams izveidot relāciju attiecību). Tādā gadījumā nākas vērt vaļā EDMX failu nevis vizuālajā režīmā, bet gan kā XML failu. Šis fails ir salīdzinoši liels un nepārskatāms. Vēl sliktāk – daži izmaiņu veidi ir jāveic vairākās vietās, piemēram, ja vajag samainīt datu tipu kādai kolonnai, tas nozīmē veikt izmaiņas 2-4 vietās, atkarībā no situācijas. Ļoti neērti, bet ir paveicams.

Pēdējā problēma, kas padara *Model First* praktiski nelietojamu ar sarežģītu DB shēmu – modeļa pārģenerēšana. Ja sistēma attīstās un DB shēma mainās, neglābjami nāksies kādā momentā sinhronizēt jaunākās izmaiņas no datubāzes uz EF modeli. To darot, manuāli veiktās izmaiņas tiek aizvietotas. Varētu mēģināt saglabāt veco modeli, ģenerēt atsevišķu modeli ar jaunajām tabulām vai laukiem un tos pievienot, bet pāris tūkstošu rindu garā XML failā veikt šīs darbības ir ļoti neērti, it īpaši ja tas jādara regulāri.

5.2. Ģenerētā modeļa koriģēšana

Autora pieredze rāda, ka, izmantojot *Code First* ar eksistējošu modeli, bieži vien nāksies piekoriģēt uzģenerēto modeli, lai pielāgotos dažādām EF prasībām, kas neatbilst DB shēmai.

Pāris iemesli modeļa koriģēšanai:

- Ģenerators vispār neatrod vai nekorekti atrod entītiju atslēgas. Visbiežāk šī situācija rodas skatījumiem. Risinājums – modelī manuāli pielikt *Key* atribūtu.
- Relāciju savienojuma lauki nav ar vienādu datu tipu. Datubāzē nav problēmu veikt savienošanu pēc dažādu tipu laukiem, tomēr EF to kategoriski aizliedz un nemaz nekompilējas. Risinājums – modelī manuāli samainīt datu tipus. Diemžēl, ne visi varianti šeit strādā. Piemēram, dažādu skaitlisko tipu (*short, int, decimal*) maiņa uz *string* nestrādā. Toties strādā maiņas starp dažādiem skaitliskiem tipiem, piemēram, *int* uz *decimal, short* uz *int,* un tamlīdzīgas maiņas, kurās nezūd precizitāte. Tas nav daudz, tomēr dod mazliet iespējas variēt un pielabot neatbilstības. Autora gadījumā tas bija pietiekami, lai novērstu pāris būtiskas problēmas.
- Nenosakās relācijas. Risinājums – modelī manuāli definēt relācijas entītiju klasēm. Relāciju definēšana modelī ir plaši aprakstīta dažādās pamācībās, autoram nav šeit īsti ko piebilst.

- Atslēgt automātisko atslēgu ģenerēšanu. Reizēm EF dažām entītiju klasēm izlemj, veicot datu ievietošanu izmantot atslēgu ģenerēšanu DB pusē. Lai to nedarītu, nepieciešams entītijai EF modelī pievienot atribūtu *DatabaseGenerated(DatabaseGeneratedOption.None).*

5.3. Relācijas, kas savienojas ne pēc atslēgas

Situācija – relācija ar galveno entītiju savienojas nevis pēc atslēgas lauka, bet kāda cita. Piemērs:

```
public class SCHOOL
{
      [Key]
      public string OBJECTID { get; set; }
      public string NAME { get; set; }
      public virtual ICollection<STUDENT> STUDENTS { get; set; }
}

public class STUDENT
{
      [Key]
      public string OBJECTID { get; set; }
      public string NAME { get; set; }
      public string SCHOOLNAME { get; set; }

      // šī relācija nebūs pieļaujama,
      // jo STUDENT.SCHOOLNAME attiecas uz SCHOOL.NAME kolonnu
      // bet SCHOOL.NAME nav atslēgas kolonna
      [ForeignKey("SCHOOLNAME")]
      public virtual SCHOOL SCHOOL { get; set; }
}
```

EF šādu situāciju absolūti neatbalsta – visās relācijas drīkst savienoties tikai caur atslēgas lauku, bet atslēgas nevar būt divas (izņemot salikto atslēgu, bet tas ir savādāk) [41]. Risinājums tiekot solīts EF 7. versijā un tas sastāvēs no divām daļām. Pirmkārt, būs pieejams *Unique* atribūts, kas ļaus kolonnu norādīt kā unikālu. Otrkārt, pēc *Unique* kolonnām varēs veikt savienošanu.

Autors problēmas apiešanai izmanto šādu risinājumu – saistītu ierakstu manuāla pielasīšana datu piekļuves klasē un pievienošana galvenajai entītijai. Obligāti relācijas parametram jāpievieno atribūts *NotMapped*, lai EF ignorētu šo relāciju gan atlasot, gan saglabājot datus. Saistīto ierakstu fiziska pievienošana objektam nodrošina to, ka šī problēma paliek datu piekļuves slāņa līmenī un neietekmē pārējos – no datu atlases funkcijas tiek atgriezts korekts objekts ar visām nepieciešamajām relācijām un saistītajiem datiem, pārējie slāņi neko nezin par EF modeļa un datubāzes nesakritību.

Piemērs, kā varētu izskatīties datu piekļuves klase iepriekš aprakstītajai *SCHOOL-STUDENT* situācijai:

```
public class SchoolQueryObject
{
  private CodeFirstDbContext _context;
  public FellQueryObject(CodeFirstDbContext context)
  {
    _context = context;
  }

  public SCHOOL ByOid(int oid)
  {
    var school = _context.SCHOOL.Single(s => s.OBJECTID == oid);
    school.STUDENTS = this.GetStudents(school);
    return school;
  }

  private List<STUDENT> GetStudents(SCHOOL school)
  {
     return _context.STUDENT.Where(s => s.SCHOOLNAME == school.NAME).ToList();
  }
}
```

Ir viens mīnuss. Šādai relācijai nav pieejamā slinkā datu atlase (*lazy load* - projektējuma modelis, attiecībā uz ORM nozīmē, ka dati tiek atlasīti nevis uzreiz, bet tajā brīdī, kad tiem pirmoreiz mēģina piekļūt). Līdz ar to, autoraprāt, būtu korekti to atslēgt visam EF modelim, galvenokārt lai ievērotu konsistenci attiecībā uz to, kā tiek atlasīti dati. Pretējā gadījumā biznesa slānī var rasties mulsinoša situācija, kad dažas relācijas ir iespējams atlasīt vēlāk pēc vajadzības,

bet dažās ne. Tas notiek, jo relācijai ir *NotMapped* atribūts, kas nozīmē, ka EF ģenerējot starpniekklasi entītijai neizveidos datu atlases funkciju šai relācijai.

5.4. Atslēgu kolonnas

Būtiskus sarežģījumus rada situācija, kad DB īpatnību dēļ nav iespējams izmantot DB ģenerētas atslēgu vērtības. Piemēram, autora gadījumā jāizsauc īpaša procedūra ar parametriem atslēgas ģenerēšanai, kura atgriež jauno atslēgu.

Lai risinātu šo situāciju, pirmkārt nepieciešams EF modelī atslēgt atslēgas ģenerēšanu DB pusē, to var izdarīt, entītijas klasē atslēgas kolonnai pievienojot atribūtu:

```
[Key]
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public int OBJECTID { get; set; }
```

Ja šis parametrs nav pievienots, tad *Insert* komandā atslēgu vērtības gluži vienkārši netiek padotas, līdz ar to tās nav iespējams uzstādīt no lietotnes puses.

Galvenā problēma sākas šajā brīdī. Normālā situācijā ar EF, objektus varam veidot vienkārši ar *new()* operatoru un neuztraukties par atslēgu ģenerēšanu.

Šajā situācijā atslēgas vērtība ir jāpiešķir manuāli, un pareizākais brīdis kad to darīt ir objekta izveidošanas laikā. Konstruktorā šādu funkcionalitāti realizēt nevar – entītija nevar pati sev uzstādīt atslēgu, it īpaši ja tam ir jāizmanto datubāze pieprasījums.

Autoraprāt, pareizākā vieta, kur šo realizēt, būtu *DbContext* vai kāds domēna serviss (skatīt punktu 6.3.5). Autors izmanto šādu palīgklasi, kuru pēc tam izmanto arī *DbContext*:

```
public class EntityCreator : IEntityCreator
{
  public T Create<T>() where T : BaseEntity
  {
      // izmanto reflekciju, lai izveidotu tipu T
      var entity = (T)Activator.CreateInstance(typeof(T));
      var tableName = entity.GetType().Name;
      // izsauc DB funkciju lai atlasītu jauno atslēgu
      var nextOid = _oracle.getNextOID(tableName);
      // izmanto reflekciju, lai uzstādītu OBJECTID kolonnu
      entity.SetValue("OBJECTID", nextOid);
      return entity;
   }
}
```

DbContext šo klasi izmanto, lai uzģenerētu entītiju ar gatavu atslēgu, un pēc tam pievieno to pie novērotajām entītijām jeb pie attiecīgā *DbSet*, kas ir konkrēta tipa entītiju kopa:

```
public T Create<T>() where T : BaseEntity
{
   var entity = _entityCreator.Create<T>();
   this.Set(typeof(T)).Add(entity);
   return entity;
}
```

5.5. Noklusētās shēmas uzstādīšana

Pēc noklusējuma, EF izmanto DBO shēmu. EF6 gadījumā, to diezgan vienkārši var mainīt – jāpārraksta *OnModelCreating* funkcija, kas dod piekļuvi *DbModelBuilder* instancei:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
     modelBuilder.HasDefaultSchema("MYSCHEMA");
}
```

Šīs funkcijas izsaukšana uzstāda shēmu uzreiz visām tabulām, kas atrodas modelī.

EF5 gadījumā ir krietni sarežģītāk. Shēma jāuzstāda katrai tabulai atsevišķi. *DbModelBuilder* funkcija *ToTable* ir domāta, lai norādītu, kura entītiju klase atbilst kurai tabulai. Viena no tās versijām (*overload*) dod iespēju norādīt arī shēmu:

```
modelBuilder.Entity<PERSON>().ToTable("PERSON", "MYSCHEMA");
```

Autoraprāt, tabulu reģistrēšana pa vienai nav uzturams risinājums. Izmantojot C# reflekciju, var panākt praktiski automātisku risinājumu [42]:

```
private void SetDefaultSchemaEf5(DbModelBuilder modelBuilder)
{
   var types = GetEntityTypes();
   foreach (Type t in types)
   {
     var method = modelBuilder.GetType().GetMethod("Entity");
     var genericMethod = method.MakeGenericMethod(t);
     var entTypConfig = genericMethod.Invoke(modelBuilder, null);

     entTypConfig.GetType().InvokeMember("ToTable", BindingFlags.InvokeMethod,
     null, entTypConfig, new object[] { t.Name, "MYSCHEMA" });
   }
}
```

5.6. EF un ADO.NET vienlaicīgi

EF ne vienmēr būs vienīgais datu piekļuves mehānisms, jo īpaši mantotās sistēmās. Piemēram, autora gadījumā, daļa funkcionalitātes realizēta ar EF, bet daļa – ar ADO.NET un SQL pieprasījumiem caur datu lasītāju. Tas nozīmē, ka tiek izmantota viens DB savienojums, kas tiek padots gan ADO.NET, gan EF klasēm.

Par laimi, EF *DbContext* konstruktorā ir pieejams variants, kad tiek padota eksistējoša *DbConnection* instance. Papildus vēl jāpadod parametrs *contextOwnsConnection*, kurš jāuzstāda uz *false*, lai EF norādītu, ka nedrīkst šo savienojumu aizvērt – tas tiek kontrolēts no ārpuses.

Diemžēl, EF 5. versijā ir vairākas problēmas ar savienojuma padošanu no ārpuses. Pirmkārt, drīkst padot tikai aizvērtu savienojumu, tas saistīts ar šo kļūdu [43]. Līdz ar to, situācijā, kad vienlaicīgi datu piekļuvei tiek izmantots gan EF gan ADO.NET, rodas neliels konflikts. EF nepieciešams aizvērts savienojums, toties ADO.NET – atvērts. Ja pirmais datu piekļuves brīdis ir caur ADO.NET, tas nozīmē, ka savienojums jāatver, savukārt tas nozīmē, ka EF šo savienojumu vairs nevarēs izmantot. Autors situāciju risināja tā, ka savienojums tiek atvērts pašā pieprasījuma apstrādes sākumā, jau ASP.NET filtru līmenī. Lai EF neradītu izņēmuma situāciju, savienojums jāatver pēc padošanas EF, turklāt izmantojot EF specifisko funkciju. Pareizais veids, kā to darīt ir [43]:

```
((IObjectContextAdapter)context).ObjectContext.Connection.Open();
```

ObjectContext būtībā ir tas pats *DbContext*, tikai ar daudz plašāku, zemāka līmeņa piekļuvi EF funkcionalitātei. *DbContext* ir tikai apvalks, kas piedāvā ierobežotu, saprotamāku interfeisu konteksta pārvaldīšanai.

```
public class ConnectionAttribute : ActionFilterAttribute, IPropInjectable
{
  public CodeFirstDbContext _context { get; set; }
  public override void OnActionExecuting(ActionExecutingContext filterContext)
  {
     _context.OpenConnection();
  }
}
```

Par laimi, EF 6. versijā šī kļūda ir novērsta, un *DbContext* spēj normāli darboties arī tad, ja tam padod atvērtu DB savienojumu.

5.7. Secinājumi

Sākotnēji EF tika izvēlēts kā labākais rīks šim uzdevumam, jo darba sākumā tas bija populārākais rīks ar plašākajām iespējām un lielāko potenciālu. *NHibernate* tajā brīdī strauji kritās popularitāte un izstrādātāju aktivitāte pie kļūdu labošanas un jaunu versiju izdošanas bija zema. Popularitāte sākotnēji tika uzskatīta par plusu – vairāk informācijas nozīmē, ka problēmu gadījumā būs vieglāk atrast risinājumus. Diemžēl, šāda tipa risinājumiem (eksistējoša, netriviāla datubāze) tas vairāk nostrādāja kā mīnuss – ņemot vērā EF popularitāti, tas tiek veidots, lai izpatiktu visiem lietotājiem. Kas nozīmē, ka rīks ir veidots vidējam lietotājam, līdz ar to ir ļoti vispārīgs un labi strādā tipiskos, triviālos scenārijos. Sarežģītākās situācijās tam gluži vienkārši pietrūkst elastības. Daudz maz visas problēmas, ar ko saskārās autors, ir apejamas. Tomēr, tas atstāj būtisku iespaidu uz koda lietojamību, lasāmību un saprotamību. Datu saglabāšanas un datubāzes rūpes ietekmē biznesa slāni un loģiku, to izolēšana prasa pārāk daudz darba, lai tas būtu tā vērts.

Diemžēl šīs problēmas atklājās ļoti lēni un tikai darba gaitā – pirms tam būtu grūti ko tādu paredzēt. Ja būtu vēlreiz kas tāds jādara, apsvērtu iespēju izmantot kādu "vieglāku" un elastīgāku ORM. Šobrīd ļoti populāri ir tā sauktie mikro ORM, piemēram *Dapper, Massive, PetaPoco*. Šeit gan uzreiz jārēķinās ar to, ka krietni vairāk funkcionalitātes nāksies realizēt pašam. Autoraprāt, galvenais mīnuss šiem rīkiem ir vājais atbalsts darbā ar objektu grafiem. Datu atlasīšana ir salīdzinoši vienkārša, toties labošanā būtiska problēma ir izmaiņu izsekošana (*change tracking*), jo īpaši dziļākajos grafa līmeņos.

6. BIZNESA LOĢIKAS SLĀNIS

Arhitektūras un programmēšanas labās prakses nosaka, ka biznesa loģikas slānim būtu jābūt pilnībā nodalītam no datu piekļuves. Pāris veidi, kā tas izpaužas:

- Domēna entītijas neko "nezin" par datu piekļuves slāni un nesatur tiešas atkarības uz datu piekļuves klasēm.
- Biznesa loģikas slānis nesatur norādes uz datu piekļuves slāni.
- Domēna entītiju modelis un pastāvīguma (*persistance*) modelis ir divas dažādas lietas. Biznesa loģika tiek realizēta ar domēna entītijām, kuras pēc tam tiek kartētas (*mapping*) uz pastāvīguma entītijām.

Teorētiski šie nosacījumi izklausās labi, bet reālajā dzīvē bieži nākas atrast balansu starp pragmatismu un teorētiski pareizo metožu izmantošanu.

6.1. EF izolēšana no biznesa loģikas slāņa

Neskatoties uz to, ka ORM palīdz izolēties no datubāzes, aktuāla tēma ir izolēšanās no ORM, respektīvi – no datu piekļuves slāņa. Šajā ziņā praktiski par standartu ir kļuvis repozitorija modelis (*repository pattern*) [44]. Modeļa mērķis ir atsaistīt biznesa loģiku no datu piekļuves loģikas. Repozitorija klases darbojas kā starpnieks pa vidu datu piekļuves slānim un biznesa loģikas slānim [45 lpp. 280-284]. Izmantojot kopā ar EF, repozitorijs darbojas kā starpnieks pa vidu biznesa loģikai un EF.

Realizācija parasti izskatās apmēram šāda – klase saņem *DbContext* (klase, caur kuru notiek datubāzes saskarne EF ietvarā) kā atkarību, un implementē metodes datu atlasei, entītiju izveidošanai, labošanai un dzēšanai. Stipri vienkāršojot situāciju un atstājot tikai svarīgās detaļas, tas izskatās šādi:

```
public class PersonRepository : IPersonRepository
{
      public Person GetById(int id) { return context.Person.Find(id); }
      public List<Person> GetAll() { return context.Person.ToList(); }
      public void Insert(Person person) { context.Person.Add(entity); }
      public void Delete(Person person) { context.Person.Remove(person); }
      public void Save() { context.SaveChanges(); }
}
```

Kā redzams, *Insert, Delete* un *Save* vienkārši izsauc attiecīgo *DbContext* funkciju. Savukārt atlases funkcijas (*GetById, GetAll*) tiek veidotas ļoti vispārīgas, neizmantojot konkrētā ORM speciālo funkcionalitāti, lai saglabātu neatkarību no izvēlētā ORM.

Mazliet paeksperimentējot ar šo modeli, autors sāka stipri apšaubīt tā noderīgumu, tieši kombinācijā ar EF. Pats par sevi šis modelis ir korekts, vērtīgs un vajadzīgs. Sākotnēji tas tika plaši izmantots, veidojot abstrakciju pār datu piekļuves loģiku gadījumos, kad tiek izmantoti SQL pieprasījumi, tīmekļa serviss vai kāds cits datu avots, kam būtu vērts ieviest vienotu interfeisu datu piekļuvei. Tā lietošana kopā ar EF ir apšaubāma, pirmkārt tāpēc, ka EF (būtībā jebkurš ORM) pats par sevi realizē repozitorija modeli. Neskatoties uz to, šis modelis joprojām bieži tiek rekomendēts izmantošanai kombinācijā ar EF. Pārsvarā tiek izmantoti šādi argumenti:

- EF izolēšana no biznesa loģikas, tādā veidā panākot pilnīgu datu piekļuves ignoranci (gan attiecībā uz DB, gan ORM) domēna slānī.
- Izolēšanās no EF, gadījumā, ja nepieciešams mainīt ORM uz kādu citu.
- Testējamība.

Autora viedoklis šajā jautājumā būtiski atšķiras iepriekš minētā. Pirmkārt, repozitorija izmantošana nedod nekādu būtisku uzlabojumu attiecībā uz testējamību. Sākot ar EF6 ir ļoti atvieglota *DbContext* viltošana un testa objektu atgriešana testēšanas nolūkos (skatīt punktu 7.1.2). Arī iepriekšējās versijās to bija iespējams izdarīt, lai gan mazliet sarežģītāk. Turklāt vēl ir variants izmantot *Effort* bibliotēku (skatīt punktu 7.1.3). Diskutējams ir jautājums par to, vai vispār ir vajadzīgs izolēties no datubāzes, veicot biznesa slāņa testēšanu (skatīt punktu 7.4). Līdz ar to, arguments par testējamību attiecībā uz repozitorija modeli ir lieks.

Nākamais arguments – izolēšanās no EF, gadījumā, ja to nepieciešams aizvietot. Pirmkārt, ORM maiņa nav gluži tipisks scenārijs. Veidot lieku slāni (kas nozīmē papildus darbu) tikai tāpēc, ka varbūt kādreiz varētu būt nepieciešams mainīt ORM, nav vērts. Pat ja tas notiks, repozitorija modelis procesu padarīs mazliet vieglāku, tomēr galvenās problēmas radīsies citur, jo ORM maiņa nav gluži triviāla. ORM ir daudz un dažādi, to funkcionalitāte izskatās līdzīga, tomēr tie katrs satur daudz specifisku funkcionalitāti un to nianses ir stipri atšķirīgas. Autora aprakstītās problēmas (skatīt nodaļu 5) tikai pierāda, ka maiņas gadījumā nāktos saskarties ar virkni jaunām, savādākām problēmām, un maz ticams, ka repozitoriju klašu implementācijas pārrakstīšana būtu grūtākais un laikietilpīgākais darbs.

Par datu piekļuves ignoranci – autors uzskata, ka lielākajā daļā gadījumu nav vērts mēģināt to darīt. Datu piekļuve ir pārāk būtisks faktors, un neizbēgami pāris nianses "ielaužas" arī biznesa

loģikas slānī, līdz ar to izolēšanās kļūst ļoti fiktīva. Mēģinot izolēties, parasti nākas datu piekļuves slāni veidot ļoti vispusīgu, tādā veidā neizmantojot izvēlētas tehnoloģijas stiprās puses. Ja reiz šī tehnoloģija ir izvēlēta, kāpēc gan neizmantot to visā pilnībā, bet mēģināt to ierobežot?

Pēc autora domām, repozitorija modelis ir lieks abstrakcijas slānis, kas rada daudz lieka koda, bet nerada būtisku labumu. Autors mēģināja izmantot šo modeli, tomēr tā uzturēšana bija pārāk neērta un prasīja pārāk daudz darba. Ja papildus abstrakcijas slānis nedod nekādu labumu, bet tikai rada papildus darbu un lieki sarežģī arhitektūru, zūd jēga to izmanto, kaut vai tas skaitās labā prakse un tiek rekomendēts kā modelis, ko izmantot kopā ar ORM.

Ņemot vērā visus šos mīnusus, autors centās atrast alternatīvu risinājumu. Pirmais ir vienkāršs un komentārus neprasa – izmantot EF tieši, nemēģinot abstrahēties. Rezultātā tiek ietaupīts laiks, kas tiktu pavadīts izveidojot un uzturot visas nepieciešamās repozitoriju klases.

Otrs variants ir vaicājumu objekti jeb *QueryObject* modelis [46] [47]. Pamatdoma ir datu atlases vaicājumus (tikai atlases, datu labošana netiek apspriesta) (EF gadījumā tas būtu LINQ izteiksme) ievietot atsevišķās klasēs. Šis modelis saistās ar CQRS arhitektūru (skatīt apakšnodaļu 1.5), kurā par *QueryObject* tiek sauktas klases, kas atbild par datu atlasi. Šis modelis ir labi piemērots sarežģītu vaicājumu iekapsulēšanai, un tas rekomendē izmantot pilnu izmantotās datu piekļuves tehnoloģijas funkciju klāstu, lai veiktu sev nepieciešamās darbības. Ievietojot pieprasījumus atsevišķā klasē, datu atlases nianses tiek nodalītas no pārējā risinājuma. Ja nepieciešams, konkrētu klasi var izolēt, ieviešot pa vidu interfeisu.

Piemērs:

```
public class ComplexAggregateQueryObject
{
  public COMPLEXENTITY GetFullGraph(int oid)
  {
      var query = _context.COMPLEXENTITY
        .Include(f => f.RELATION1 // 1. līmeņa relācija
          .Select(c => c.SUBRELATION1 // 2. līmeņa relācija
            .Select(x => x.SUBSUBRELATION2) // 3. līmeņa relācija
        ))
        .Include(f => f.RELATION2) // 1. līmeņa relācija
        .Where(f => f.OBJECTID == oid);
        return query.Single();
  }
}
```

Šeit ir divi būtiski plusi – LINQ izteiksmju ievietošana vaicājumu objektos nozīmē, ka tādā veidā šie vaicājumi kļūst atkalizmantojami. Vienkāršākus vaicājumus var izmantot pa tiešo ar *DbContext*. Attiecībā uz datu labošanu ir divi varianti – izmantot *DbContext* metodes (*Save, Delete, Remove*) vai arī sekot CQRS taktikai, un datu labošanas loģiku ievietot komandu (*commands*) objektos [47].

6.2. Domēna modeļa izolēšana no pastāvīguma modeļa

Ņemot vērā labās prakses, domēna modelim (entītijām, kas veido biznesa funkcionalitāti) būtu jābūt nodalītām no pastāvīguma modeļa. Respektīvi – nevar izmantot vienu un to pašu modeli datu saglabāšanai un biznesa loģikas realizēšanai. DDD šajā ziņā ir īpaši strikts, nosakot, ka biznesa loģika būtu jāveido, vispār aizmirstot par datubāzi un koncentrējoties tikai uz biznesa objektu un situāciju modelēšanu [10 lpp. 56-59].

Attiecībā uz ORM, šajā sakarā rodas jautājums – vai drīkst izmantot EF ģenerētās entītiju klases kā domēna objektus, lai realizētu biznesa loģiku? Tīra DDD atbalstītāji to aizliedz un uzskata par absolūtu anti-modeli. Skatoties no praktiskās puses, ir vairākas lietas, ko var mēģināt realizēt izmantojot EF entītijas:

- Privātie vērtību uzstādītāji (*private setters*). Tādā veidā varam liegt no ārpuses labot laukus, kas attiecas tikai uz datubāzi.
- Loģikas realizēšana entītiju klasēs.
- Validācijas un biznesa ierobežojumi. Vērtību validēšana uzstādītāju (*setter*) līmenī.

Izmantojot EF entītijas kā domēna modeli, neglābjami datubāzes rūpes ielauzīsies arī domēna modelī, tādā veidā padarot to vairāk vai mazāk saistītu ar datubāzi. Tomēr, vai šī nelielā saistība ir pietiekami liels trūkums, lai realizētu divus atsevišķus modeļus – vienu biznesa loģikai, otru datubāzes piekļuvei? Jārēķinās, ka atsevišķu modeļu uzturēšana prasīs vairāk laika un sarežģīs arhitektūru. Visdrīzāk nāksies izmantot kādu kartēšanas rīku (piemēram, *AutoMapper*), lai vērtību uzstādīšana no viena modeļa uz otru nebūtu jāveic manuāli.

Pēc autora domām, lielākajā daļa gadījumu var mierīgi iztikt ar EF ģenerētajām entītijām. Ja gadījumā datubāzes rūpes sāk pārāk traucēt un aizņemt būtisku daļu no domēna modeļa, ir vērts domāt par atsevišķa modeļa veidošanu un vairāk DDD centrētu pieeju. Piemērs šim būtu sadaļā 5.3 aprakstītā problēma par relācijām, ko nav iespējams normāli savienot ar objektu grafa sakni, līdz ar to rodas būtiska nesakritība starp datubāzes relāciju modeli un EF modeli.

6.3. Domēna modelis un atkarības

Grūti iedomāties, ka biznesa funkcionalitāte būt tik vienkārša, ka to visu varēs ievietot tikai entītiju klasēs. Bieži vien nākas veikt sarežģītākas darbības, kas prasa dažādu atkarību izmantošanu, ārējus servisus u.tml. Pieaugot funkciju skaitam, ko veic entītija, vienā brīdi tai var kļūt pārāk daudz atbildību un tai sāk zust vienots mērķis, funkcionalitāte. Tādos gadījumos būtu nepieciešams daļu funkcionalitātes atdalīt. Virspusīgi apskatot problēmu, jautājums ir – vai injicēt nepieciešamās atkarības entītiju klasēs, vai tomēr mēģināt realizēt šo loģiku kādā augstākā līmenī.

6.3.1. Service Locator

Kā jau parasti, vienmēr ir variants izmantot statisku DI konteineri, lai piekļūtu vajadzīgajām atkarībām. Kā jau iepriekš tika minēts, statisks konteineris ir anti-modelis (skatīt apakšnodaļu 2.8) un ir izmantojams tikai galējas nepieciešamības gadījumā.

6.3.2. Atribūtu injicēšana

Lai injicētu atkarības, izmantojot DI ietvaru, nepieciešams pieāķēties EF dzīves ciklam momentā, kad pēc DB pieprasījuma dati tiek kartēti uz entītijām [48]. Šajā brīdī iespējams pārķert jau izveidotu entītijas instanci, un injicēt tajā nepieciešamās atkarības, izmantojot atribūtu injekciju (skatīt apakšnodaļu 2.1). Ņemot vērā, ka entītiju tipi ir dažādi, būtu nepieciešams procesu automatizēt. *Windsor* ietvara gadījumā, varam izveidot funkciju, kas dinamiski iet cauri visiem objekta atribūtiem, un ievieto atkarības (skatīt punktu 4.2.1).

Lai gan atribūtu injicēšanas ir ērts variants no programmēšanas viedokļa, tomēr tas piesārņo domēna modeli ar DB piekļuves klasēm, tādā veidā radot ciešu saistību.

6.3.3. Domēna notikumi

Notikumus parasti izmanto, lai mazinātu ciešo saistību starp klasēm, starp kurām pēc būtības nav nekādas saistības. Būtisks trūkums ir tāds, ka notikumus var izmantot tikai gadījumos, kad nav nepieciešams atpakaļ saņemt vērtību. Pretējā gadījumā tiek pārkāpta notikumu ideja kā tāda. Tātad būtībā notikumus var izmantot tikai funkcionalitātei, kas ir pilnīgi nesaistīta. Piemēram, mēs nevar izmantot šo modeli, lai veiktu slinko datu atlasi entītijas relācijām, jo tas nozīmētu vērtības atgriešanu.

Tipisks piemērs, kur var izmantot notikumus, ir e-pasta nosūtīšana [49]. Tā vietā, lai entītiju klasei injicētu *IEmailSender* atkarību un izsauktu *Send()* funkciju pēc biznesa funkcijas

izsaukšanas, klase izraisa notikumu, uz kuru ir parakstījies *IEmailSender,* kurš tad tālāk veic attiecīgās darbības.

6.3.4. Double Dispatch

Krietni savādāks variants ir tā sauktais *double dispatch* modelis [49] [50]. Nepieciešamā atkarība (kā interfeiss) tiek padota kā parametrs funkcijai. Tādā veidā entītija tiešā veidā nav atkarīga no tā. No programmēšanas principu viedokļa šī metode ir ļoti korekta – tiek saglabāta vāja saistība starp entītiju un atkarību, turklāt atkarība tiek padota tikai tajās funkcijās, kur tā ir nepieciešama. Padodot atkarības konstruktorā vai injicējot kā atribūtus, nāktos entītiju padarīt cieši saistītu ar atkarībām, kas nepieciešamas varbūt tikai vienai funkcijai. Šis modelis palīdz izvairīties no anēmiskā domēna, jo rezultātā entītiju klasēs var ievietot funkcijas, kas savādāk tur nebūtu ievietojamas dēļ atkarībām no ārējiem servisiem.

Pēc autora domām, praktiskai lietošanai šis modelis ir ļoti nepiemērots. Pirmkārt, atkarību padošana caur funkciju nozīmē, ka šī atkarība tiek padota no augstāka līmeņa klases (piemēram, domēna servisa), kas nozīmē, ka šis serviss pats kļūst cieši saistīts ar šo atkarību tikai tāpēc, lai to varētu padot entītijai, lai gan pats varbūt šo atkarību neizmanto.

Otrkārt, uzturēšana ir sarežģīta un kods ir grūti lasāms. Kamēr atkarību padošana notiek divu klašu ietvaros (piemēram, domēna serviss padod vienai entītijai), *double dispatch* izskatās pieņemami. Kad tiek iesaistītas vairākas klases, atkarības no domēna servisa nokļūst entītijā, tad tālāk jāpadod nākamajai tālāk un tālāk. Rezultātā funkciju signatūras kļūst garākas, jo vietu aizpilda atkarību parametri. Un tas savā ziņā novērš uzmanību no biznesa funkcionalitātes.

6.3.5. Domēna servisi

DDD arhitektūra nosaka, ka sarežģīta biznesa loģika ir realizējama domēna servisos (kas būtībā ir bezstāvokļa klases), ja tā [10 lpp. 75-78]:

- Neiederas viena agregāta (entītiju grafa sakne) ietvaros.
- Ir pārāk sarežģīta, lai realizētu entītijā.
- Realizācijai pieprasa ārējās atkarības.

Salīdzinoši no programmēšanas viedokļa, šis variants ir daudz ērtāks, nekā *double dispatch*. No vienas puses, domēns satur norādēs uz DB piekļuves klasēm, bet vismaz tas viss ir koncentrēts domēna servisos, nevis izkaisīts pa domēna slāni.

6.4. Secinājumi

Pēc ilgiem eksperimentiem ar šajā nodaļā pieminētajām tehnikām, autoram nācās atzīt – datubāzes ietekme konkrētajā lietotnē ir pārāk liela, lai būtu vērts no tās izolēties. Patērētie resursi nebūtu tā vērti. Autors uzskata, ka pārāk intensīva DDD principu ievērošana attiecībā uz datubāzes ignorēšanu un domēna slāņa izolēšanu, bieži vien rezultējas ar veiktspējas problēmām un/vai grūti lasāmu un saprotamu kodu (*double dispatch*). Liela daļa no funkcionalitātes izmanto *Oracle* specifiskas lietas, kā rezultātā izolēšanās tīri funkcionāli varētu nebūt optimāls risinājums.

7. BIZNESA LOĢIKAS TESTĒŠANA

Viena no galvenajām testējamajām lietām, protams, ir biznesa loģika. Neatkarīgi no tā, cik tālā abstrakcijas līmenī biznesa loģikas slānis ir iznests un cik slāņus zem tā notiek reāla sasaiste ar datubāzi, tas tomēr beigu beigās rezultējas ar pieprasījumiem uz datubāzi - slānis vairāk vai mazāk ir atkarīgs no datubāzes. Tas nozīmē, ka viena no galvenajām problēmām ir panākt, lai testus būtu iespējams atkārtot.

Šī ir viena no sarežģītākajām lietām, ko testēt tīmekļa lietotnē. Un šeit jo īpaši izpaužas princips - vai nu darīt lietas atbilstoši labajām praksēm, kas minētas nodaļā par testēšanu, vai nedarīt vispār. Testos viena no svarīgākajām īpašībām ir iespēja to atkārtot un konsekventi vienmēr iegūt vienu un to pašu rezultātu. To ir īpaši grūti panākt, testējot komponentus, kas izmanto datubāzi.

Svarīga izvēle šajā jautājumā ir - vai izolēties no datubāzes (vienībtesti), vai tomēr testēt biznesa loģiku kopā ar datubāzes loģiku datu atlasē un labošanā (integrācijas testi)? Virspusēji apskatot izvēli, uzreiz var teikt:

- Integrācijas testi būs lēnāki un grūtāk atkārtojami, toties tie pārbaudīs krietni vairāk funkcionalitātes.
- Vienībtesti būs ātrāki, bet tie var palaist garām vairākas svarīgas kļūdas, kas tieši saistītas ar datubāzi.

Vai nu viens no variantiem vai abi, bet ir skaidrs, ka bez biznesa loģikas testēšanas iztikt nav iespējams - tas ir viens no galvenajiem mērķiem, kāpēc vispār veidot arhitektūru un domāt par testēšanu.

Katra no izvēlēm rada vairākas problēmas, kuras nepieciešams risināt.

Vienībtestu gadījumā:

- Kas jāizdara, lai vispār varētu izolēties no datubāzes?
- No kurienes ņemt datus?

Ja testus veic integrācijā ar datubāzi:

- Ja izmanto koplietošanas datubāzi - kā panākt, ka testpiemērus iespējams atkārtot, respektīvi - uzsākot katru testu, datubāze ir paredzamā stāvoklī, nav iespējas, ka kāds testa datus ir mainījis, dzēsis?
- Kā panākt, ka testpiemēru beigās datubāzes stāvoklis nav mainīts, respektīvi, lai testu varētu atkārtot atkal un atkal.

- Testi ir salīdzinoši lēni, jo veic darbības (vaicājumus, datu ievietošanu) datubāzē. Pat, ja darbību nav daudz, joprojām ir jāizveido savienojums ar datubāzi, kas rada tīkla noslodzi utt.
- Loģikai kļūstot sarežģītākai, testpiemērus kļūst grūti atkārtot, tie prasa apjomīgus uzstādīšanas darbus - jāpanāk, lai datubāze ir testa izpildei nepieciešamajā stāvoklī.
- Grūti pārbaudīt darbību iznākumu. Jāveido vaicājumi, lai atlasītu datus, kas pārbauda, vai biznesa aprēķini veikti korekti.

7.1. Testi ar izolēšanos no datubāzes

Šajos testos oriģinālā datubāze vispār netiek izmantota. Ir dažādas metodes kā panākt, lai testos tiktu atgriezti viltoti entītiju objekti ar testam nepieciešamajiem datiem.

7.1.1. Repozitorija modelis

Diezgan universāla metode, kas nav atkarīga no EF versijām un būtībā vispār no datu piekļuves mehānisma. Cieši saistīta ar DI tehniku un testu dubultniekiem. Balstās uz to, ka visa datu piekļuve ir strikti nodalīta un notiek caur speciāli tam paredzētām klasēm - repozitorijiem. Būtībā tas ir vēl viens abstrakcijas slānis EF/ORM vai citai datu piekļuves metodei. Kā rezultātā testos iespējams šīs klases aizstāt ar dubultniekiem, kuriem tiek definēts atgriezt testiem specifisku objektu, kas iekļaujas noteiktā testu scenārijā .

Piemēram, klase, kas izmanto repozitorija klasi datu atlasei un tālāk veic darbu ar atlasītajiem datiem:

```
public class ClassToBeTested()
{
      public ClassToBeTested(IPersonRepository personRepository) { }

      public void FunctionToBeTested()
      {
            var data = personRepository.GetData();
            // tālāk funkcija izmanto iegūtos datus ...
      }
}
```

Testā *IPersonRepository* tiek aizstāts ar viltus implementāciju, izmantojot dubultnieku ietvaru (šajā gadījumā *NSubstitute*).

```
var mock = Substitute.For<IPersonRepository>();
mock.GetData().Returns(new List<Person>() {
      { Name = "Jānis" },
      { Name = "Juris" },
      { Name = "Valdis" }
};

// testējamajai klasei padodam viltus instanci
var sut = new ClassToBeTested(mock);
```

Autors uzskata, ka repozitorija modelis kombinācijā ar EF (ar ORM vispār) nav labs risinājums, jo būtībā tā ir abstrakcija uz jau esošas abstrakcijas - EF pats par sevi implementē abus - *Repository* un *Unit of Work* (skatīt apakšnodaļu 6.1). Ir pieejami labāki, ātrāk ieviešami varianti, kas neprasa veidot atsevišķu slāni, kas paredzēts tikai testēšanas nolūkos.

Tomēr, kombinācijā ar citām datu piekļuves metodēm (piemēram, ADO.NET) , šis ir vērā ņemams risinājums.

7.1.2. EF datubāzes konteksta viltošana

Šī ir EF nodrošināta iespēja, kas ir pieejama no EF 6. versijas. Šajā versijā ieviestās izmaiņas *DbContext* un *DbSet* klasēs dod iespēju vienkāršāk izolēties no datubāzes, nekā izmantojot repozitorija modeli [51]. Līdzīgi kā repozitorija modelī, arī šeit datu piekļuve tiek aizstāta ar speciāli sagatavotiem dubultniekiem.

Iepriekšējais piemērs izskatītos šādi:

```
var data = new List<Person>() {
      { Name="Jānis" },
      { Name="Juris" },
      { Name="Valdis" }
};

var mockSet = Substitute.For<DbSet<Person>>();
mockSet.GetEnumerator().Returns(data.GetEnumerator());

var mockContext = Substitute.For<MyDbContext>();
mockContext.Persons = mockSet;
```

Galvenais pluss ir ērtība – nav nepieciešams veidot speciālas klases datu piekļuvei tikai tāpēc, lai varētu tās izmantot testēšanas nolūkos. Mīnuss – risinājums ir EF specifisks.

Liels mīnuss abiem iepriekšējiem risinājumiem ir testu datu veidošana – tas jādara manuāli, turklāt tie ir C# objekti, un kodā tas neizskatās pārskatāmi. Nākamais risinājums piedāvā labāku veidu, kā pārvaldīt šo problēmu.

7.1.3. Effort bibliotēka

Nākamais evolūcijas solis EF testēšanā. *Effort* bibliotēka [52] ir speciāli paredzēta izmantošanai EF testos, lai izolētos no oriģinālās datubāzes, bet tajā pat laikā veiktu darbības ar apjomīgiem objektiem. Tā izmanto operatīvās atmiņas datubāzi (*in memory database*), kurā glabā datu objektus, simulējot īsto datubāzi.

Labākais veids datu glabāšanai ir CSV faili, kur katrs fails attēlo vienu datubāzes tabulu. Formāts gan ir mazliet specifisks , kas varētu radīt problēmas. Katras testu sesijas sākumā faili tiek ielasīti atmiņā, lai simulētu datubāzi.

CSV failu ielasīšana notiek šādi – iet cauri *DbContext* reģistrētajiem *DbSet*. katram no tiem mēģinot atrast atbilstošo CSV datu failu. Ja tas izdodas, tad tālāk iet cauri visām *DbSet* kolonnām, mēģinot pielasīt entītijai vērtības no CSV rindas. Būtiska šeit ir secībā (pielasa vērtības entītijai, nevis otrādi), kas nozīmē, ka DB shēmas maiņa vairs nav tik kritisks pasākums un trūkstošas kolonnas uzreiz nesabojās visus testpiemērus.

Plusi:

- Tā kā dati glabājas atmiņā nevis uz cietā diska, testpiemēru izpilde ir būtiski ātrāka nekā tad, ja tiktu izmantota īstā datubāze.
- Datu glabāšana CSV failos (atšķirībā no testu objektu manuālas veidošanas) dod lielisku iespēju tos viegli pārskatīt, mainīt, pievienot klāt pašam projektam, kā rezultātā tie kļūst par daļu no testu bibliotēkas, nevis kā ārējs faktors, kā tas ir gadījumā, kad tie glabājas kaut kur datubāzē. Līdz ar to testpiemēri ir pilnīgi izolēti no ārējām atkarībām, galvenokārt – no datubāzes shēmas maiņas.
- Var izveidot tikai to datubāzes daļu, kas nepieciešama. Ļoti būtiski, ja ir liela datubāze, kuras pilna izveidošana ir laikietilpīga. Dod iespēju testus dalīt grupās un izveidot tikai to datubāzes daļu, kas nepieciešama.
- Nekādas sasaistes ar datubāzi un datiem tajā. Izejas kodu var ātri iedot citam izstrādātājam, kurš bez problēmām varēs izpildīt visus testus. Ir ļoti būtiski, lai testi būtu viegli pārnesami uz citām vidēm, pretējā gadījumā ir maza jēga no tiem, ja izpildīšanas iespēja ir tikai vienam programmētājam uz viena konkrēta datora.

- Dinamiski piemērojas EF *Code First* stratēģijai (skatīt apakšnodaļu 5.1). Līdz ar to ir vieglāk turēties līdzi datubāzes shēmas izmaiņām.

Mīnusi:

- Datubāzes inicializācija var būt lēna, kas var traucēt, ja bieži grib izpildīt tikai vienu testu.
- Nav pieejamas DBVS specifiskās EF funkcijas, piemēram, *SqlFunctions.*
- Tā kā *Effort* ir ne gluži datubāze, bet glabā atmiņā būtībā C# objektus, tad nav pieejamas standarta DBVS komandas kā piemēram vaicājumi un komandas, nemaz nerunājot par procedūrām un funkcijām. Pietiekami bieži nopietnās sistēmās nevarēs iztikt tikai ar EF datu piekļuvei. Pirmkārt, tā ir veiktspēja, kuras nolūkos bieži vien izdevīgāk ir SQL vaicājumus rakstīt manuāli, nevis izmantot *Linq-to-Entities*. Autora gadījumā būtisks faktors ir mantotā sistēma, kurā būtiska daļa no datu apstrādes notiek ar tiešiem SQL vaicājumiem, kurus nav izdevīgi pārrakstīt uz EF. Līdz ar to šis ir nopietns trūkums, jo ļauj testēt tikai daļu no sistēmas.

7.2. Testi bez izolēšanās no datubāzes

Šajos testos biznesa loģikas slānis tiek testēts kombinācija ar datu piekļuves slāni, kurš savukārt izmanto īstu datubāzi datu iegūšanai. Protams, nevis produkcijas, bet gan testa, vai izstrādes - kā nu kurš to sauc.

Datubāzes izmantošana rada divas ļoti sarežģītas problēmas saistībā ar testu konsistenci un atkārtojamību:

- Tests nedrīkst atstāt paliekošas, būtiskas sekas datubāzē. Droši vien var neskaitīt audita ierakstus, palielinātus atslēgu skaitītājus, DB statistikas u.tml., jo tas nav tik būtiski.
- Uzsākot testpiemēra vai testu sesijas (atkarīgs kādā stilā raksta testus) izpildi, datubāzei jābūt konkrētā, paredzamā stāvoklī.

7.2.1. Testpiemēra atstāto seku novēršana

Pirmais punkts ir salīdzinoši vienkāršāks – to var atrisināt ar sadalītajām transakcijām vai DB rezerves kopijām vai momentuzņēmumiem.

7.2.1.1. Sadalītās transakcijas

Transakciju izmantošana testos ir diezgan ātrs veids, kā panākt to, ka testi neatstāj paliekošas sekas datubāzē, tātad - testi būs atkārtojami (vismaz tik ilgi, kamēr sākuma dati paliks nemainīgi). Pirmais un vienkāršākais veids ir transakciju izmantošana - uzsākam transakciju, uzstādām, veicam, pārbaudām testu un beigās veicam atriti (*rollback*) [53]. Tas nozīmē, ka testa veiktās darbības nav atstājušas nekādas būtiskas sekas datubāzē, līdz ar to varam šo testu atkārtot pēc vajadzības tik ilgi, kamēr testam nepieciešamās rindas datubāzē paliek nemainīgas.

Lai šis risinājums būtu ieviešams, lietotnes transakciju kontrolei ir jānotiek augstākā līmenī, nekā tiek veikti testi - transakcijas nedrīkst būt biznesa loģikas slānī vai, vēl sliktāk - domēna objektos.

Līdzīgi kā pašā lietotnē, arī šeit būtu labi transakcijas kontrolēt globālā līmenī, nevis katram testam atsevišķi. Ērts veids, kā selektīvi atzīmēt testpiemērus, kuriem nepieciešamas transakcijas (jo ne vienmēr visi testi veiks saglabāšanu datubāzē), ir izmantot anotācijas. *NUnit* to var panākt, izmantojot *TestAttribute* anotācijas, kas dod iespēju izveidot kodu, ko izsaukt pirms un pēc katra testpiemēra, šajā gadījumā:

- Pirms - atveram transakciju.
- Pēc - veicam atriti.

Izveidojot *Rollback* anotāciju, varam norādīt, ka testpiemēra beigās jāveic atrite [54].

Šādi izskatās *Rollback* anotācija:

```
public class RollbackAttribute : Attribute, ITestAction
{
   private TransactionScope _transaction;
   public void BeforeTest(TestDetails testDetails)
   {
     var transactionOptions = new TransactionOptions();
     transactionOptions.IsolationLevel = IsolationLevel.ReadCommitted;
     _transaction = new TransactionScope(
         TransactionScopeOption.Required, transactionOptions);
   }
   public void AfterTest(TestDetails testDetails)
   {
     _transaction.Dispose();
   }
}
```

Šādi tiek norādīts, ka testpiemērs izmantos atriti:

```
[Test, Rollback]
public void SomeTestCase() { … }
```

Attiecībā uz transakciju kontroli, ir divi varianti - izmantot parastās transakcijas, vai arī spēcīgāko *TransactionScope*, kas būtībā ir sadalīto transakciju kontrolieris - .NET klase, kas ļauj veikt transakcijas vairākās datubāzēs vienlaicīgi. Sadalītās transakcijas ir vieglāk ieviešamas, jo neprasa datubāzes savienojuma injicēšanu filtros vai ko tamlīdzīgu.

Neliels mīnuss – netiek testēts transakcijas *commit*, bet droši vien nebūs daudz situācijas, kad tas varētu būt būtisks traucēklis. Autoram nav nācies saskaties ar situāciju, kad tieši *commit* daļa radītu problēmas, un strādājošs testpiemērs neatklātu kļūdu dēļ šī apstākļa.

7.2.1.2. Rezerves kopijas paņemšana testu sākumā

Otrs risinājums ir krietni masīvāks. Testu sākumā tiek paņemta rezerves kopijā, tad tests veic savu scenāriju un pārbaudes, un tad datubāze tiek atjaunota no kopijas [53]. Uzreiz rodas vairākas problēmas:

- Jāpārdomā rezerves kopēšanas taktika.
- Jāizdomā, kuras tabulas kopēt, kuras ne.
- Šis risinājums pavisam noteikti ir sarežģītāks un krietni lēnāks.

Autoraprāt, vienīgais gadījums, kad šī taktika varētu būt izmantojama rezerves kopēšana – ja testu scenāriji ir milzīgi un iet secībā, katrs nākamais ir atkarīgs no iepriekšējo izpildes. Šajā gadījumā gan tos vairs īsti nevar saukt pat par integrācijas testiem.

Otrs gadījums varētu būt tas, ka *TransactionScope* metode netestē pašu *commit*. Praktiski visos citos gadījumos pirmais variants būs krietni ātrāk ieviešams un krietni vieglāk uzturams.

7.2.2. Konsistentu datu nodrošināšana testpiemēra sākumā

Transakcijas ļauj ļoti ātri iegūt spēcīgu testēšanas funkcionalitāti, respektīvi, testi neietekmē datubāzē esošos datus, jo tiek veikta atrite. Tomēr, joprojām paliek problēma - kā panākt, ka uzsākot testu, datubāzē ir mums nepieciešamie dati? Šī problēma prasa zināmu piepūli un stratēģiju. Pieaugot datubāzes izmēram un sarežģītībai, otrā punkta atrisināšana kļūst arvien neērtāka.

7.2.2.1. Datubāzes radīšana no jauna testu sesijas sākumā.

Pietiekami sarežģīts un grūti realizējams variants. Bez šaubām lēns. No SQL skriptiem veic datubāzes izveidošanu un testiem paredzēto datu ievietošanu. Katra testa vai sesijas sākumā tiek izsaukti visi skripti (pamatā standarta SQL), lai izveidotu datubāzi.

7.2.2.2. Atsevišķas datubāzes izmantošana

Protams, ir iespējams izmantot atsevišķu datubāzi tikai testiem [53], tomēr, tam ir vairāki būtiski trūkumi:

- Jāuztur atsevišķa datubāze tikai testiem bez cita reāla pielietojuma.
- Mainoties produkcijas datubāzes shēmai, jāveic sinhronizācija, kas prasa laiku, jo kādam tas viss ir jāuztur.
- Datu pārcelšana var būt ļoti ilgs process

Ir skaidrs, ka šis nav pats ērtākais variants. Daudz ērtāk būtu izmantot jau esošu datubāzi (piemēram, testa vidi), kurā šādas sinhronizācijas jau tiek veiktas.

7.2.2.3. Rezerves kopiju atjaunošana (backup)

Testa sākumā kods izsauc komandas, kas veic rezerves kopijas atjaunošanu [53]. Parasti tas nozīmē, ka ir saglabāta kāda kopija, kurā ir testiem nepieciešamie dati. Tomēr, ir skaidrs, ka šī metode varētu būt pārāk lēna, lai to varētu izmantot testos. Turklāt ir grūti iedomāties, ka sarežģītā sistēmā ir tik vienkārši un ātri veikt kopijas atjaunošanu, lai tos varētu izdarīt katra testa sākumā, turklāt saprotamā, automatizējamā veidā.

Autors uzskata, ka šie varianti ir pārāk lēni. Tos varētu darbināt, piemēram, kā paketes procesu ārpus darba laika, kad īsti vairs nav svarīgi, ka testu izpildes laiks mērāms stundās. Ikdienas izstrādei, kad atgriezenisko saiti par veiktajām izmaiņām nepieciešams saņemt uzreiz, šī metode vienkārši ir pārāk lēna.

7.2.2.4. Code First datubāzes inicializācija

Ļoti specifisks EF risinājums ir datubāzes inicializācija no EF *Code First* izveidotā koda. Tā kā EF *Code First* metodes pamatā ir iespēja datubāzes shēmu izveidot no entītiju klasēm, šo iespēju ļoti vienkārši var izmantot, lai izveidotu datubāzi tieši testēšanai, katra testa vai testu sesijas sākumā [55].

Teorētiski šis risinājums izskatās ļoti labi, bet ir grūti iedomāties, ka tas strādā sarežģītākās situācijās, jo īpaši kopā ar eksistējošu datubāzi. Ja nu tomēr *Code First* tiek izmantots jau no paša

sākuma un datubāze tiek pilnīgi veidota no modeļa, šis varētu būt ļoti labs risinājums. Jo īpaši tāpēc, ka, salīdzinot ar iepriekšējām metodēm, shēmas izmaiņas ir aprakstītas EF *Code First* modelī, līdz ar to testos vienmēr tiks izmantota jaunākā shēma.

Autoram šis risinājums neder, jo datubāze satur pārāk daudz specifisku lietu, kuras EF *Code First* modelī vienkārši nav iespējams ne aprakstīt, ne ievietot.

7.3. Autora piedāvātais risinājums

Autora situācijā iepriekšējie risinājumi nebija lietojami - vai nu nebija ieviešami dēļ datubāzes sarežģītības, vai arī tie bija pārāk lēni. Atsevišķas datubāzes izmantošana tikai testiem nav variants, jo tās izveidošana un uzturēšana prasa pārāk daudz laika. Tomēr, ir iespēja izmantot testu datubāzi, bet tā ir koplietošanas, tātad galvenā problēma būs panākt, ka testu sākumā datubāzē ir nepieciešamie dati. Problēmas sagādāja datubāzes shēmas maiņa, kas nozīmē, ka rezerves kopijas arī nav variants, jo sistēma būtībā tiek testēta ar novecojušu konfigurāciju.

Autora gadījumā testējamā sistēma ir mantotā sistēma. Attiecībā uz datubāzes piekļuvi tas nozīmē, ka:

- Lielākā daļa no datu piekļuves slāņa tiek realizēta ar ADO.NET un manuāli rakstītiem SQL pieprasījumiem.
- Biznesa loģika bieži vien ir cieši saistīta ar datu piekļuves loģiku, kas nozīmē, ka DB izolēšanas gadījumā testpiemērs nespēs pārbaudīt praktiski svarīgāko daļu no testējamā komponenta.

Būtu vēlams, lai testi būtu daļa no projekta. Respektīvi, testiem nepieciešamie dati glabājas daudz maz lasāmā formātā, ir pievienojami projektam un ievietojami versiju kontroles sistēmā.

Nevajadzētu būt tā, ka katrs testpiemērs veic pilnu inicializāciju datubāzei, tas ir pārāk ilgs process. Testpiemēram vajadzētu spēt inicializēt tikai sev nepieciešamās daļas, piemēram, ievietot nepieciešamos ierakstus.

Ņemot vērā iepriekšējo variantu mīnusus un autora specifiskos ierobežojumus, kā arī izmantojot un kombinējot vairākus iepriekš aprakstītos risinājumus, tika izveidots savs risinājums, kas ir labāk piemērots specifiskajai situācijai (jo īpaši mantotām sistēmām), tajā pat laikā ir pietiekami universāls.

Šajā risinājumā testpiemēram nepieciešamie dati tiek glabāti JSON failos un ievietoti datubāzē katra testa sākumā. Par to atbild katrs testpiemērs pats un tas notiek *Arrange* fāzē.

Būtiski situāciju apgrūtina tas, ka testiem ir nepieciešamas saliktas entītijas ar relācijām. Šādos gadījumos datu izgūšanai un ievietošanai lieliski ir piemērots EF. Pieņemot, ka mums ir korekti izveidots *Code First* modelis, process ir sekojošs.

7.3.1. Testa datu sagatavošana

Lai sagatavotu testa datus (tiek darīts ārpus paša testpiemēra kā atsevišķs process):

1. Izvēlēties no datubāzes kādu objektu testiem.
2. Izmantot EF, lai atlasītu objektu ar saistītajiem objektiem.
3. Serializēt datus JSON formātā un saglabāt kā failu projekta mapē.

Piemērs kodā:

```
[Test]
public void QueryDataForTest()
{
      var oid = 1234567;

      var context = _container.Resolve<CodeFirstDbContext>();

      var entityObj = context.COMPLEXENTITY
            .Include(f => f.RELATION1) // pirmā līmeņa relācija
            .Include(f => f.RELATION2 // pirmā līmeņa relācija
                  .Select(c => c.SUBRELATION2)) // otrā līmeņa relācija
            .Include(f => f.RELATION3
                  .Select(c => c.SUBRELATION3))
            .Include(f => f.RELATION4)
            .Single(f => f.OBJECTID == oid);

      // izmanto JSON.NET serializatoru
      Console.Write(JsonConvert.SerializeObject(entityObj));
}
```

7.3.2. Testa datu ievietošana

Tālāk, lai izmantotu sagatavotos JSON datus testā:

1. Nolasa failu un deserializē uz entītijas klasi.
2. Ievieto deserializēto objektu datubāzē. To var ļoti vienkārši izdarīt ar EF *DbContext.Add()* komandu. EF šeit paveic milzīgu darba apjomu, jo visi saistītie objekti (relācijas) tiks ievietoti līdz ar galveno objektu. Izņemot situācijas, kad DB

shēma nesakrīt ar EF nepieciešamo struktūru, lai varētu atpazīt un definēt relācijas (skatīt apakšnodaļu 5.3). Autoram tas ir ļoti būtiski, jo praktiski lielākā daļa funkcionalitātes (līdz ar to arī lielākā daļa testpiemēru) izmanto saliktus objektu grafus ar relācijām vairākos līmeņos.

3. Šajā brīdī testa sagatavošanas fāze, kas attiecas uz datu sagatavošanu, ir noslēgusies. Tālāk var veikt testā nepieciešamās darbības.

Piemērs, kā tas izskatās kodā:

```
[Test, Rollback]
public void Test()
{
      // nolasa no faila un deserializē uz atbilstošo entītijas tipu
      var objToSave =
TestDataProvider.GetType<EntityType>("/TestData/EntityType/1.json");

      // saglabā
      var context = _container.Resolve<CodeFirstDbContext>();
      context.Add<EntityType>(objToSave);
      context.SaveChanges();

      // tālāk veic pārējās testa darbības
}
```

Ļoti būtiski - lai testpiemērs neatstātu paliekošas sekas un testa datu ievietošanu varētu veikt atkal un atkal, viss testpiemērs tiek apvīts ar *Rollback* atribūtu, jeb būtībā *TransactionScope* (skatīt apakšpunktu 7.2.1.1.) Šīs tehnikas izmantošana nozīmē to, ka testam nepieciešamie ievietoti dati būs pieejami testa izpildes gaitā, bet pēc tam tiks veikta atrite un dati būs jāievieto atkal no jauna nākamajā testpiemērā.

Testa datu ievietošanai pat vienkāršākajos gadījumos autors iesaka izmanto palīgklasi, kad speciāli paredzētu testa datu atlasīšanai, serializēšanai un saglabāšanai. Viena no sarežģītībām rodas, kad DB shēmas īsti neatbalsta relāciju veidošanu. Tādos gadījumos ieteicama šāda taktika:

- Atlasīt un saglabāt tās relācijas, kuras atbalsta konkrētā DB shēma kombinācijā ar EF.
- Pārējās relācijas pielasīt speciāli tām sagatavotās funkcijās.

7.3.3. Testa datu primārās atslēgas

Serializētās entītijas sākotnēji satur oriģinālās atslēgas un ārējās atslēgas. Skaidrs, ka šādus datus nebūs iespējams atkārtoti ievietot. Autora piedāvājums ir izvēlēties jaunas atslēgu vērtības, izmantojot pietiekami lielas vērtības, kuras vēl kādu laiku nekonfliktēs ar DB esošajām vērtībām (ņemot vērā, ka atslēgu skaitītājs pieaugs).

7.3.4. EF līmeņa kešdarbe

Būtiska nianse, par ko viegli aizmirst vai nepamainīt – EF veic kešdarbi. Tātad, ja objekts jau atrodas EF kontekstā, tas netiks atkārtoti pieprasīts no datubāzes, bet gan tiks izmantota *DbContext* esošā versija.

Attiecībā uz autora piedāvāto testu risinājumu, tas rada būtisku trūkumu – noteiktās situācijās netiks notestēta datu atlases loģika. Piemēram, situācija:

- Veic testpiemēra uzstādīšanu - nolasa failu, deserializē uz entītiju klasi, ievieto datubāzē.
- Testa darbības fāzē izpilda vaicājumu, kurš atgriež daļu no datiem, kas tikko tika ievietoti. Šajā brīdī EF izmanto kontekstā jau esošās instances, jo pēc uzstādīšanas fāzē ievietotie dati joprojām atrodas tajā pašā *DbContext* instancē.

Ir skaidrs, ka šāda situācija atkārtotos pietiekami bieži, un šis scenārijs ir pietiekami svarīgs. Bez tā šis risinājums ir diezgan nepilnīgs.

Viens variants, kā risināt situāciju, būtu izmantot jaunu *DbContext* instanci. Tas gan nozīmē, ka pēc testa datu ievietošanas būtu jāveic transakcijas *commit*. Un tas padara sarežģītāku testpiemēra "notīrīšanas" (*cleanup*) fāzi – atrite būtu jāveic manuāli, kas rada iespēju lieki kļūdīties. Daudz labāk tomēr būtu turpināt izmantot *TransactionScope* piedāvātās iespējas un neuztraukties par izmaiņām, ko datubāzē varētu atstāt testpiemērs.

Kā izrādās, EF ir iespējams novērst vai apiet kešdarbi dažādos veidos un manuāli izņemt entītijas no konteksta. Tas gan prasa piekļūšana kontekstam mazliet zemākā līmenī – jāizmanto *ObjectContext*, kas būtībā ir tas pats *DbContext*, tikai ar plašākām iespējām.

Pirmais variants – *Refresh()* funkcija, norādot, ka datubāzei ir priekšroka:

```
(_context as IObjectContextAdapter)
      .ObjectContext.Refresh(RefreshMode.StoreWins, entity);
```

Iespējams var noderēt, tomēr trūkums ir tas, ka entītija tiek pārlādēta uzreiz. Labāks risinājums būtu, ja pārlādēšana notiktu pirmoreiz, kad nepieciešams piekļūt datiem, nevis uzreiz pēc testa datu ievietošanas.

Otrais variants – *Detach()* funkcija. Kā parametru padod konkrētu entītiju.

```
((IObjectContextAdapter)_context).ObjectContext.Detach(entity);
```

Svarīgs faktors, kas varētu nebūt pašsaprotams, ņemot vērā funkcijas nosaukumu – no konteksta tiek izņemts tikai galvenā entītija, bet nevis relācijas. Autoram šāds risinājums neder, jo praktiski jebkurš testpiemērs izmantos objektu grafu vairākos līmeņos. Manuāli iet cauri visām relācijām un tās atvienot pa vienai ir pārāk neērti un viegli palaist kādu garām. Būtu vajadzīgs kāds automātiskāks risinājums.

Trešais variants pilnībā atbilst autora prasībām. *ObjectStateManager* objekts dod iespēju piekļūt visām entītijām, kas atrodas kontekstā, tai skaitā relācijām.

```
var objContext = ((IObjectContextAdapter)this).ObjectContext;
var objectStateEntries = objContext
     .ObjectStateManager
     .GetObjectStateEntries(EntityState.Unchanged);

foreach (var objectStateEntry in objectStateEntries)
{
    objContext.Detach(objectStateEntry.Entity);
}
```

Līdz ar to, jāatceras šo funkciju izsaukt katru reizi pēc testa datu ievietošanas. Tā rezultātā EF tos dzēš no konteksta, vairs neglabā informāciju par tikko ievietotajiem datiem, un pie nākamās vajadzības tos atlasīs no datubāzes. Tādā veidā tiek iegūta praktiski jauna *DbContext* instance, un ir iespējams pārbaudīt būtisko datu atlases loģiku un funkcionalitāti.

7.4. Secinājumi

Viens no galvenajiem plusiem vienlaicīgi ir arī galvenais mīnuss – datubāzes izolēšana. Testu ātrums un vienkāršība rezultējas ar būtisku problēmu – operācijas ar objektiem no atmiņas krietni atšķiras no operācijām ar datubāzes objektiem [56]. Lai vai cik ļoti LINQ censtos abstrahēties un apakšā esošā datu avota, tomēr *LINQ-to-Objects* (kas būtībā izmantojas testos, kad tiek izmantoti viltus dati) nav tas pats, kas *LINQ-to-Entities* (kurš savukārt rezultējas ar SQL pieprasījumiem uz DB).

Testpiemēri un jo īpaši datu sagatavošana pavisam noteikti prasīs vērā ņemamus laika resursus. Līdz ar to rodas jautājums, vai ir vērts tērēt laiku, lai izolētos no datubāzes, kā rezultātā lielākā daļa no kļūdām testos netiktu identificētas, jo noteiktajā vidē šīs kļūdas gluži vienkārši neatkārtojas. Pēc autora pieredzes, vairāk nekā 90% no kļūdām datu piekļuves un biznesa loģikas slāņos ir tieši saistītas ar datubāzes tehnoloģiju un ORM. Lai vai cik ļoti ORM ietvari mēģinātu sasaisti ar datubāzi padarīt nemanāmu, tomēr reālajā dzīvē ir pārāk daudz dažādu izņēmuma situāciju.

Arī vienībtestos reizēm var nonākt tādā situācijā, kad atkarības tiek izolētas tik tālu, ka beigu beigās testpiemērs pārbauda praktiski neko – viss ir viltots un reālās funkcionalitātes testā praktiski nav. Līdzīgi ir ar datubāzes izolēšanu. Autors uzskata, ka sarežģītās sistēmās, jo īpaši mantotās, kuras daudz izmanto parastos SQL pieprasījumus un ne tikai EF, izolēšanās no datubāzes prasa pārāk daudz resursu, bet testi nedot gaidīto pārliecību par sistēmas korektumu.

Aprakstīto metodi autors kādu laiku sekmīgi izmanto sistēmā. Metode pagaidām ir pietiekami uzturama un ir palīdzējusi novērst vairākas kļūdas sakarā ar datu piekļuves slāni, tai pat laikā ļaujot pietiekami efektīvi pārbaudīt arī biznesa slāņa funkcionalitāti. Būtībā visas aprakstītās metodes ir pietiekami sarežģītas un to ieviešana prasa daudz laika, tāpēc būtu vēlams izvēlēties vienu metodi un pie tās arī pieturēties. Izņēmums varētu būt gadījums, kad integrācijas testi kļūst pārāk lēni. Tādā gadījumā var zemāka līmeņa biznesa loģikas komponentus testēt izolācijā, izmantojot kādu no ātrāk ieviešamajām metodēm, piemēram, *DbContext* viltošanu. Savukārt augstāka līmeņa testos turpināt izmantot datubāzi un pilnu datu piekļuves slāni.

8. PĒTĪJUMA GAITĀ IZVEIDOTĀ ARHITEKTŪRA

Darba praktiskās daļas ietvaros tika veikta reālas lietotnes pārnešana uz jaunām tehnoloģijām. Kā rezultātā daļa no moduļiem tika pārrakstīta pilnībā, daļa tika būtiski modificēti, bet daļai nācās atrast pareizo balansu starp koda kvalitātes uzlabošanu un moduļa ātru, nesāpīgu pārnešanu. Visas galvenās problēmas, ar ko autors saskārās praktiskās daļas izstrādē, ir aprakstītas šajā darbā. Visi secinājumi par iepriekšējās nodaļās aprakstītajām metodēm, tehnikām un dizaina modeļiem ir radušies, mēģinot tos praktiski ieviest šajā sistēmā, kādā modulī, sadaļā vai funkcijā.

Pārrakstītās lietotnes nosaukums ir "GeoWeb". Tā ir ĢIS (ģeogrāfiskā informācijas sistēma) lietotne, kas nozīmē, ka liela daļa funkcionalitātes norisinās ap kartēm, ģeometrijām u.c. telpiskām lietām. Biznesa nozare ir mežkopība, kas nozīmē, ka galvenie objekti sistēmā ir cirtes, nogabali, pievešanas ceļi, krautuves utt. Īsumā par galvenajām sadaļām (kas arī tiks apskatītas koda metriku un veiktspējas testos), lai piešķirtu lietotnei mazliet vairāk konteksta:

- Labošana (*edit*)– vispārīga ģeometrisko objektu labošana ar saistītajiem ierakstiem.
- Cirsmas skiču ievade (*CSI*) – cirsmas, nogabalu, pievešanas ceļu, krautuves un citu saistīto datu un ģeometrijas ievade.
- Cirsmu vērtēšana (*assesment*) – nogabalu/sugu/sortimentu informācijas ievadīšana, virkne aprēķinu un algoritmu.
- Mežizstrādes norādījumi (*instruction*) – dažādu kritēriju ievade attiecībā uz mežizstrādi. Būtisku daļu sastāda izdruku veidošana.

Lietotnes pārrakstīšana, pārnešana, tehnoloģiju migrācija utt. nekad nav viegls pasākums, un lielākajā daļā gadījumu nodara vairāk posta nekā labuma. Tomēr, šajā gadījumā no tā nevarēja izvairīties. Galvenais migrācijas iemesls bija pāreja uz *ArcGIS* (uzņēmumā izmantotais ĢIS serveris) *Server* 10, jo vecā 9.3 versija oficiāli skaitās novecojusi un brīžiem vairs nespēj korekti veikt savus pienākumus. Diemžēl, ĢIS servera versijas maiņa rada nepārvaramas problēmas ar lietotnes koda atpakaļsaderību, kā rezultātā būtiska daļa koda ir jāpārveido. Līdz ar to tika pieņemts lēmums veikt vispārēju arhitektūras uzlabošanu, lai paātrinātu migrācijas procesu, kā arī atvieglotu tālāko izstrādi. Viens no galvenajiem mērķiem bija ieviest automātiskos testus, iepriekš tādu praktiski nebija. Pēc autora domām, šī sistēma ir daudz par sarežģītu, lai to varētu normāli un kvalitatīvi izstrādāt bez automātisko testu palīdzības.

Pārrakstot funkcionalitāti, bieži nācās izvēlēties starp ātrāku, bet mazāk korektu risinājumu, un pareizāku, bet krietni lēnāku risinājumu. Sākotnēji tika plānots pilnīgi pārrakstīt krietni vairāk funkcionalitātes, nekā tas beigās tika izdarīts. Iemesls tam – lai arī funkcija ir ļoti neuzturama, sarežģīta un slikta visos iespējamajos veidos, tomēr tā strādā un ir pārbaudīta reālā darbībā – produkcijā. Pārrakstot to, tiktu uzlabots kods, bet noteikti rastos jaunas kļūdas. Vai tas ir vajadzīgs? Lielākajā daļā gadījumu nē, vismaz ne līdz pirmajām nepieciešamajām izmaiņām.

Pētījuma gaitā autors vairākas reizes pārdomāja būtiskus lēmumus attiecībā uz arhitektūru. Noslēgumā izvēlētais piegājiens izteikti virzījās uz pragmatisko pusi.

Izveidotā arhitektūra balstās uz pāris vienkāršiem principiem, aizgūstot vairākas idejas no šajā darbā apskatītajiem arhitektūras veidiem (*Onion,* CQRS, DDD).

- Neizmantot repozitorija modeli kā vēl vienu slāni virs EF.
- Necensties izlikties, ka datubāzes un ORM nav, tā vietā izmantot to stiprās puses, tajā pat laikā maksimāli samazinot saskarsmes punktus datu piekļuves funkcionalitātei.
- Funkcionalitātes dalīšana pirmkārt moduļos, nevis slāņos. Ideja aizgūta no DDD, kurš definē, ka lietotnei augot, nepieciešams to dalīt mazākās daļās (ne tikai slāņos), tādā veidā palielinot kohēziju un grupējot ciešāk saistīto funkcionalitāti [10 lpp. 79-84]. Šāds kods mērogojās daudz labāk. Funkcionalitātei augot, tā vietā, lai katrs slānis kļūtu masīvāks, vienkārši rodas jauns modulis, vai arī lielāki moduļi tiek sašķelti daļās. Katra moduļa ietvaros notiek mini slāņošana – ir domēna, datu piekļuves u.c. slāņi. Bet tie nav strikti nodalīti un netiek savā starpā izolēti.
- Domēna servisu izmantošana biznesa loģikas realizēšanai tā vietā, lai satrauktos par anēmiskā domēna modeļa esamību. Entītijas, protams, satur arī funkcionalitāti, bet tikai pašu primitīvāko un centrālāko. Mēģināt ievietot visu funkcionalitāti tikai entītijās nozīmētu, ka stipri mazinātos to kohēzija.
- Lai maksimāli varētu darboties ar atmiņā esošiem datiem (nevis visu laiku pielasīt klāt no datubāzes), vaicājumu objektu uzdevums ir atgriezt visus konkrētajai darbībai nepieciešamos datus, bet tikai tos, kurus darbības sākumā ir iespējams paredzēt. Ir pietiekami daudz dinamisko datu, kas atkarīgi no mainīgajiem izpildes gaitā, un to pielasīšana gan ir jāuztic kādam citam, šajā gadījumā domēna servisiem.

9. KODA METRIKU SALĪDZINĀJUMS

Lielāko daļu no labas arhitektūras plusiem ir grūti aprakstīt skaitļos. Tomēr, šajā nodaļā autors mēģinās paveikto darbu novērtēt, vairāk balstoties uz skaitļiem un statistiku. Pirmkārt, tiks salīdzinātas dažādas koda metrikas – cikliskā sarežģītība, klašu saistība utt. Otrkārt, tiks izvērtēta lietotnes ātrdarbība, salīdzinot ar veco.

Koda metrikas ir dažādi mērījumi par kodu, kas vairāk vai mazāk mēģina izmērīt koda kvalitāti. Ir pieejami dažādi jaudīgi maksas rīki, bet autors ērtības nolūkos izvēlējās sev vieglāk pieejamo variantu - *Visual Studio 2012* iebūvēto koda metriku mērītāju. Tas piedāvā veikt šādus mērījumus [57]:

- *Maintainability Index* – uzturamības indekss. Aprēķinās, izmantojot visus pārējos parametrus. Nosaukums "uzturamības indekss" ir mazliet maldinošs, jo šis parametrs apzīmē ne tikai uzturamību, bet koda kvalitāti vispār. Jo kods kvalitatīvāks, jo vieglāk tas ir uzturams. Vērtību skaidrojums īsumā:
  - 0-9 = slikti.
  - 10-19 = vajadzētu uzlabot.
  - 20-100 = labi.
- *Cyclomatic Complexity* – cikliskā sarežģītība. Cik daudz dažādas kontroles plūsmas ir iespējamas vienas funkcijas ietvaros.
  - *0*-9 = labi.
  - *10*-25 = vajadzētu uzlabot.
  - >25 = slikti, praktiski neuzturams, liela iespējamība kļūdām.
- *Depth of Inheritance* – cik līmeņos notiek mantošana.
- *Class Coupling* – saistība. No cik klasēm ir atkarīga jeb cik klases izmanto konkrētais modulis/klase/funkcija. Augšējais limits tiek minēts 9 klases [58].
- *Lines of Code* – loģiskās koda rindas, nevis fiziskās.

Sākumā abas lietotnes (vecā un jaunā) tika salīdzinātas, vienkārši izpildot metriku mērīšanu un apskatot rezultātus bez īpašas iedziļināšanās. Respektīvi - skats no augšas. Vecais kods tiek grupēts pa vārdtelpām (jo iepriekš viss bija vienā projektā), savukārt jaunais – pa projektiem. Summārā rinda tiek rēķināta tāpat, kā to dara *Visual Studio* – visiem summa, izņemot uzturamības indeksu, kuram tiek rēķināts vidējais.

Tabula 9.1

Vispārējās metrikas - pirms

Namespace	***Maint. Index***	***Cycl. Compl.***	***Depth of Inherit.***	***Class Coupling***	***LLOC***
lvm.Models.Restriction	87	32	2	5	44
lvm.Models.Group	81	62	1	6	119
lvm.Models.Geometry	89	28	1	12	53
lvm	80	5	4	14	13
PDF_Tests	81	28	2	16	56
lvm.Models.User	80	38	2	16	91
lvm.Models.Assortment	83	62	2	18	112
lvm.Models.Audit	84	46	2	19	100
lvm.Models.Medus	68	33	2	19	99
lvm.Models.Clasific	81	78	2	23	161
lvm.Models.Felling	65	400	1	75	1541
lvm.Models.Report	69	961	2	99	3382
lvm.Models.Print	41	572	7	100	3590
lvm.Models	79	2163	3	204	4660
lvm.Controllers	64	1667	6	244	6189
Sum/avg	***75***	***6175***		***870***	***20210***

Tabula 9.2

Vispārējās metrikas - pēc

Project	***Maint. Index***	***Cycl. Compl.***	***Depth of Inherit.***	***Class Coupling***	***LLOC***
GeoWeb2.Restriction	66	4	1	5	20
GeoWeb2.ConfirmationRequestPdf	41	35	2	26	233
GeoWeb2.ArcGISREST	89	130	1	36	235
GeoWeb2.Confirmation	78	383	1	42	813
GeoWeb2.ReportBase	62	462	2	56	1593
GeoWeb2.Core	92	1178	2	68	1338
GeoWeb2.Instruction	68	424	2	83	2563
GeoWeb2.Assesment	73	186	3	84	437
GeoWeb2.Infrastructure	74	270	2	121	729
GeoWeb2.MVC4	79	201	4	190	500
SOE.Common	82	327	1	89	657
SOE.CSI	64	306	1	109	1147
SOE.Edit	71	118	1	113	379
Sum/avg	***85***	***4024***	***-***	***1022***	***10644***

Tā kā koda struktūra ir būtiski mainījusies, grūti šeit vilk paralēles. Jāņem gan vērā, ka ir pārnesti apmēram 60% no funkcionalitātes un izdarīts 80% no kopējā darba. Tas nozīmē, sākotnēji tika pārrakstīta sarežģītākā funkcionalitāte, kas noteikti prasīs vairāk darba. Tomēr, var izdarīt pāris vispārīgus secinājumus par kopējo ainu:

- Uzturamības indekss ir mazliet audzis, bet kopumā bez būtiskām izmaiņām.
- Cikliskā sarežģītība pagaidām ir divreiz mazāka.
- Koda rindu skaits pagaidām ir divreiz mazāks, pārrakstot visu funkcionalitāti, tas varētu būt tikai mazliet mazāks, autora prognoze – 15 000 pret 20 000.
- Klašu saistība jau tagad ir lielāka. Tam tā jābūt, jaunajā kodā ir daudz vairāk klašu, un tā kā rezultāts ir summa nevis vidējais, tad tam būtu jābūt lielākam.

Lai labāk novērtētu rezultātus, autors centās sīkāk izprast uzturamības indeksu. Uzturamības indekss izklausās pēc noderīgas metrikas, bet ko patiesībā tas nozīmē? Šī metrika radusies 1992. gadā, bet *Visual Studio* izmanto modificētu formulu, lai iekļautos vērtībās no 0 – 100 [59].

```
Maintainability Index =
  MAX(0, (171 - 5.2 * ln(Halstead Volume)
            - 0.23 * Cyclomatic Complexity
            - 16.2 * ln(Lines of Code)
          ) * 100 / 171)
```

Attēls 9.1 – uzturamības indeksa aprēķina formula [59]

Cyclomatic Complexity un *Lines of Code* ir tie paši iepriekš aprakstītie lielumi, bet *Halstead Volume* ir atkarīgs no operandu un operatoru daudzuma.

Apskatot rezultātus funkciju līmenī un sīkāk tos izpētot, autoram radās vairāki secinājumi, tieši attiecībā uz to, kā tiek mērīti rezultāti:

- Autors absolūti nepiekrīt *Visual Studio* vērtību skalai attiecībā uz uzturamības indeksu. 20 noteikti nav tas rezultāts, kuru varētu saukt par labu, uzturamu un saprotamu. Izpētot atsevišķi funkcijas un to uzturamības rezultātus, autora personiskais vērtējums būtu, ka 40-50 ir minimālā robeža, kas skaitās pieņemami. Savukārt laba funkcija sākas no indeksa 70.
- Mazliet nepamatoti šķiet tas, ka katrs klases mainīgais rada +1 ciklisko sarežģītību. Piemēram, jaunajā projektā ir daudz EF ģenerēto entītiju klašu, dažas ar vairāk nekā

100 atribūtiem (būtībā tās ir DB kolonnas). Šajā gadījumā tas dod apmēram +500 ciklisko sarežģītību. Ņemot vērā, ka summa visam kodam ir 4000, ietekme ir pārāk liela.

- Autoram par pārsteigumu izrādās, ka vidējais projekta uzturamības indekss tiek rēķināts pat pārāk primitīvi – vidējais no visiem failiem [59]. Nevis svērtais, ņemot vērā, piemēram, rindu skaitu vai tamlīdzīgi. Tas nozīmē, ka šo metriku ir ļoti vienkārši apiet. Piemēram:
    - Ir viena svarīga klase ar 1000 rindām, kas realizē galveno biznesa funkcionalitāti ar uzturamību 0.
    - 9 mini klases ar pāris rindām un uzturamību > 90.
    - Rezultātā iegūstam kopējo uzturamību ~90, lai gan tas ir vairāk nekā neobjektīvi. Ja reiz 99% no koda rindām ir slikti uzturamas, rēķināt vidējo tikai no failiem ir nepareizi un rezultāts neattēlo patieso situāciju.
    - Tas lieliski atspoguļojas vecajā lietotnē. Tikai 11 funkcijas ir ar indeksu zem 20, tātad viss pārējais kods pēc *Visual Studio* standartiem ir labs. Diemžēl, šīs 11 funkcijas sastāda 25% no visa koda (5000 loģiskās rindas no 20 000)!
    - Lai novērstu šo netaisnību, turpmāk autors šo metriku rēķinās kā vidējo svērto pēc rindu skaita. Tas varbūt nav optimālākais un objektīvākais veids, bet tomēr krietni labāks par *Visual Studio* standarta veidu.
- Uzturamības indekss pats pat sevi ir diezgan labs veids, kā salīdzināt divas funkcijas vai noteikt, vai funkcija ir salīdzinoši uzturama. Izpētot un salīdzinot dažādus rezultātus, nākas secināt, ka lielākajā daļā gadījumu, augstāks rezultāts tiešām ir labāks. Varbūt ne pāris punktu ietvaros, bet tomēr – 80 ir labāk nekā 60, kas ir labāk nekā 40, kas ir labāk nekā 20 utt.
- Mantošanas dziļums nav vērā ņemama metrika, vismaz ne šajā projektā, jo mantošana tiek izmantota tikai pāris gadījumos. Līdz ar to šī metrika vairs netiks apskatīta, komentēta un salīdzināta.
- Mērot klašu saistību, ir diezgan bezjēdzīgi rēķināt summu (kā to dara *Visual Studio*), jo tādā veidā noteicošais ir kopējais klašu skaits projektā. Daudz noderīgāk būtu rēķināt vidējās vērtības un skatīties, vai klase nepārsniedz noteikto limitu.

Skatoties no šāda, salīdzinoši augsta skatpunkta, grūti redzēt kādas pozitīvas izmaiņas. Skaidrs ir tas, ka skatīties šīs metrikas projekta līmenī īsti nav jēgas – patiesie rezultāti pazūd,

aprēķinot vidējās vērtības. Tas pats attiecas uz problēmātisko vietu meklēšanu kodā – nevar paļauties uz kopējiem rezultātiem. Labākais veids būtu speciāli filtrēt funkcijas, kur uzturamības indekss ir zemāks par kādu vērtību.

Turpinājumā tiks apskatīti dati dažādos griezumos. Lai iegūtu reālāku skatījumu uz situāciju, turpmākajos salīdzinājumos autors veiks šādas korekcijas:

- Detalizēti tiks apskatītas pāris svarīgākās, lielākās un kļūdām bagātākās biznesa funkcijas, kas tika pārrakstītas vai pārstrādātas.
- Uzturamības indekss, cikliskā sarežģītība – tiks apskatīti funkciju līmenī, sagrupēti dažādos veidos.
- Vidējais uzturamības indekss tiks rēķināts kā vidējais svērtais, pēc loģisko rindu skaita.

9.1. Grupēšana

Autors pamēģināja sagrupēt uzturamību un ciklisko sarežģītību vērtību grupās ar soli 10. Tika grupētas tikai funkcijas (respektīvi filtrs "*where scope=Member*"). Tika iegūti šādi rezultāti:

Tabula 9.3

Uzturamības indekss sagrupēts ar soli 10

	Pirms	***Pēc***
0-9	8	1
10-19	3	1
20-29	15	10
30-39	58	25
40-49	88	32
50-59	183	104
60-69	218	194
70-79	247	199
80-89	212	159
90-100	969	1667
Kopā	***2001***	***2392***

Pozitīva tendence – būtisks samazinājums funkcijām ar vērtību no 0 līdz 60. Tas nozīmē, ka samazinās grūtāk uzturamo funkciju skaits. "90-100" grupu visdrīzāk var ignorēt – tur pārsvarā atrodas *geteri*/*seteri*, kurus tā vienkārši nevar izfiltrēt, līdz ar to grūti objektīvi novērtēt šo grupu.

Tabula 9.4

Cikliskā sarežģītība sagrupēta ar soli 5

	Pirms	***Pēc***
0-4	1728	2285
5-9	161	66
10-14	44	17
15-19	26	15
20-24	19	3
25-29	7	2
>=30	16	4
Kopā	***2001***	***2392***

Arī šeit vērojama pozitīva tendence – būtiski samazinājies "sarežģīto" funkciju skaits. Iepriekš anti-rekords bija 174, kurš tika pārrakstīts uz vairākām mazākām funkcijām ar sarežģītību 1-10. Jaunajā kodā anti-rekords ir 59.

9.2. Atsevišķu funkciju izpēte

Lai paskatītos uz datiem no cita leņķa, autors mēģināja atsevišķi apskatīt lielākās funkcijas. Stāsts visām šīm funkcijām ir diezgan līdzīgs – sākotnēji realizētas kontrolierī kā viena 1000-2000 rindu gara funkcija (ar varbūt dažām palīgfunkcijām), kas sevī saturēja gan augstāka gan zemāka līmeņa kodu. Jaunajā kodā šī funkcija ir sadalīta vairākās klasēs, funkcijās. Trīs no četrām salīdzinātajām funkcijām nodarbojas ar datu saglabāšanu, kas arī ir sarežģītākā daļa. Datu atlases funkcijas šajā lietotnē pārsvarā ir krietni īsākas un vienkāršākas, izņemot varbūt telpisko datu atlasi.

Tabula 9.5

CSI saglabāšana - pirms

Type	***Member***	***Maint. Index***	***Cycl. Compl.***	***Class Coupling***	***LLOC***
CSIController	*Save()*	0	142	77	575
CsiDamage		54	35	22	116
MArchive		63	21	14	74
MFellCompartment	*saveAreas()*	37	67	2	23
Sum/Avg		***14***	***198***	***113***	***765***

Tabula 9.6

CSI saglabāšana - pēc

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
CompSaveCmd		61	15	15	57
CsiDamage		57	16	12	72
CsiDamageArchive		61	8	16	29
CsiSaveCmd		55	61	43	157
CsiSaveCommon		63	14	22	38
DeliveryRoadSaveCmd		55	34	28	144
Sum/Avg		***57***	***148***	***136***	***497***

Tabula 9.7

Cirsmu vērtēšanas saglabāšana - pirms

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
MFelling	*SaveAssesment()*	21	49	27	129
MFellCompartment	*SaveAssesment()*	54	21	2	11
MFellSpecies	*Save()*	34	38	4	51
MFellCompartment	*SaveSpecifityRow()*	63	5	2	6
MFelling	*createHistoryRow()*	57	3	7	13
Sum/Avg		***29***	***116***	***42***	***210***

Tabula 9.8

Cirsmu vērtēšanas saglabāšana - pēc

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
AssesmentSaveService		60	62	39	129
FELLCOMPARTMENTS	*SaveAssesment()*	62	1	2	9
FellAssesmentExtensions	*SaveAssesment()*	62	5	5	8
FELLCOMPARTMENTSPECIES	*Save()*	55	1	2	15
FELLS	*CreateHistoryRow()*	61	4	5	10
Sum/Avg		***60***	***73***	***53***	***171***

Tabula 9.9

Mežizstrādes norādījumu izdruka - pirms

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
MPrintInstruction		24	166	65	1700
MFelling	*updateAutomaticQualityActs()*	52	8	3	18
MFelling	*getIntersect()*	34	21	21	64
FellingAreaCalc		69	18	7	40
MFellCompartment	*getHeightForMaxCrosCutLeft()*	54	9	2	16
MFellCompartment	*FellModelDifferenceKKC()*	59	4	5	11
MFellCompartment	*GetNewMinCCValues()*	54	7	6	15
MFellCompartment	*FellModelDifferenceOther()*	47	10	7	24
Sum/Avg		***27***	***243***	***116***	***1888***

Tabula 9.10

Mežizstrādes norādījumu izdruka - pēc

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
PrintInstructionService		38	136	54	1453
UniqueSpecieService		59	17	14	52
PrintImageService		50	20	24	98
FellingAreaCalc		72	12	9	23
FellCompartmentCalc		64	18	9	50
Sum/Avg		***41***	***203***	***110***	***1676***

Tabula 9.11

Objektu saglabāšana ar relācijām - pirms

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
EditController	*Save() : JsonResult*	4	95	60	288
Sum/Avg		***4***	***95***	***60***	***288***

Tabula 9.12

Objektu saglabāšana ar relācijām - pēc

Type	*Member*	*Maint. Index*	*Cycl. Compl.*	*Class Coupling*	*LLOC*
EditSaveCmd	*GetGeometry()*	70	5	5	6
EditSaveCmd	*GetOidFromAttributes()*	66	3	0	7
EditSaveCmd	*PerformRelationAction()*	38	19	22	47
EditSaveCmd	*Save()*	42	8	14	41
EditSaveCmd	*SetAuditFields()*	59	6	4	11
RelationManager	*GetRelationClass()*	52	6	6	20
Sum/Avg		***46***	***47***	***51***	***132***

Šeit jāizceļ vairākas lietas. Pirmkārt, būtiski uzlabojies uzturamības indekss. Cikliskā sarežģītība ir mazliet samazinājusies. Pēc autora domām, ciklisko sarežģītību ir grūti samazināt, respektīvi, funkcija no biznesa puses ir tik sarežģīta, cik ir. Koda pārkārtošana sarežģītību īpaši nemazinās. To var izolēt, nodalīt, pārnest uz citu vietu, bet gala rezultātā summa īpaši nemainās.

Klašu saistība ir lielāka, bet tā jābūt – jaunais kods ir modulārāks, kas nozīmē, ka katras darbības veikšanā būs iesaistītas daudz vairāk klases, nekā iepriekš. Koda rindu skaits ir samazinājies – tas ir pozitīvi. Jo mazāk koda, jo vieglāk to uzturēt, jo grūtāk kļūdīties, jo ātrāk to var izlasīt un saprast.

9.3. Secinājumi

Salīdzināt metrikas starp tik dažādiem projektiem bez iedziļināšanās un analīzes ir diezgan bezjēdzīgi. Nepieciešams datus pagriezt un apskatīt dažādos leņķos, lai iegūtu patieso skatu. Patiesās vērtības ir apslēptas zem vidējām, līdz ar to tās gluži vienkārši ir "jāizrok". Situāciju sarežģīt *Visual Studio* taktika, kad metriku rezultāti tiek pasniegti kā summa. Respektīvi, klases rezultāts ir visu funkciju summa, moduļa rezultāts ir visu klašu summa utt. Diemžēl, nav pieejamas vidējās vērtības vai kas tamlīdzīgs.

Autors ikdienā noteikti nemēģinātu vēlreiz šādi veikt dažādu projektu salīdzināšanu. Tomēr, metrikas var būt ļoti noderīgas, lai vienas lietotnes ietvaros kodā identificētu vājās vietas – slikti uzturamas funkcijas/klases/moduļus, ar pārāk lielu sarežģītību, pārāk daudz koda rindām u.tml.

Koda metriku uzlabošana pavisam noteikti nebija mērķis – galu galā tie ir tikai skaitļi, kas paši par sevi neko nemaina. Tomēr, bija cerība, ka tie spētu attēlot izmaiņas un norādīt uz vietām,

kur nepieciešami vēl papildus uzlabojumi. Tas arī izdevās. Ar nelielu piebildi, ka autors gaidīja lielāku atšķirību rezultātos starp veco un jauno kodu.

Autoraprāt, *Visual Studio* pietrūkst vairākas būtiskas metrikas:

- Funkciju skaits, kur pārsniegts konkrēts rindu skaits.
- Kohēzija – ļoti būtisks parametrs, lai izvairītos no klasēm, kas dara pārāk daudz dažādu lietu un satur savā starpā maz saistītas funkcijas.
- Vidējā funkciju sarežģītība. Kopējo sarežģītību var savākt dažādos veidos, tomēr pie vienāda rezultāta var būt liela atšķirība – vai tās ir dažas, sarežģītas funkcijas, vai tomēr daudz bet mazas, kas summā dod vienādu rezultātu.

10. VEIKTSPĒJAS TESTI

Šajā nodaļā jaunā lietotne tiks salīdzināta ar veco, veicot veiktspējas testus. Datubāze būs vienota, bet *ArcGIS* serveris vecajai lietotnei tiks izmantots 9.3, savukārt jaunajai – 10. Diemžēl savādāk nevar un tas atstās ietekmi uz veiktspēju. Tomēr, *ArcGIS* serveris nebūs iesaistīts visu funkciju izpildē, tāpēc ĢIS specifiskās funkcijas tiks apzīmētas ar zvaigznīti (*).

Testi tika veikti, izmantojot *JMeter* rīku. Rīka uzdevums šajā gadījumā ir vienkārši caur HTTP protokolu veikt pieprasījumu, lai izsauktu testējamo lietotnes funkciju, tādā veidā simulējot lietotāja darbību. Testēšanai tika izvēlētas pašas "smagākās" funkcijas, pārsvarā gan datu atlasei un ievietošanai paredzētās. Datu labošanas funkcijas prasa pārāk daudz un pārāk sarežģītus uzstādīšanas darbus.

Katrai funkcijai tika izvēlēti 10 testpiemēri. Tabulā attēlotas vidējais atbildes laiks pirms un pēc (sekundēs).

Tabula 10.1

Vidējais atbildes dažādām funkcijām laiks pirms un pēc

Funkcija	*Apraksts / komentārs*	*Pirms (s)*	*Pēc (s)*
Assesment / Felling	cirtes datu atlase	1.8	0.5
Assesment / Felling	tas pats, tikai sarežģīti piemēri ar daudz relācijām	39.7	0.64
*Edit / Relation *	objekta saistīto datu atlase	1.5	0.1
*CSI / AreaInfo *	nogabalu informācijas atlase	1.7	0.2
*CSI / AreaDetect *	nogabalu telpiskā noteikšana	5.2	1.2
Instruction / Save	mežistrādes norādījumu saglabāšana	6.3	4.3
Instruction / Index	mežistrādes norādījumu datu atlase	7	3.6
Instruction / Print	mežistrādes norādījumu izdruka ar tipu 3	10.5	6.5
Instruction / Print	mežistrādes norādījumu izdruka ar tipu 2	15.2	15.6
*Edit / Save *	Bojājumu slāņa saglabāšana, jauns ieraksts	5.6	3.5
*Edit / Save *	Mežsaimniecisko darbu slāņa saglabāšana, jauns ieraksts	1.5	0.5
*Edit / Save *	Mežsaimniecisko darbu slāņa saglabāšana, eksistējošs ieraksts	1.4	0.7
*CSI/Save *	Cirtes saglabāšana, eksistējoša ieraksta atjaunošana	3.7	0.2

Īstumā – lielākā daļa no testētajām funkcijām ir būtiski ātrākas. Dažas ir uz pusi ātrākas. Viena tomēr izrādījās lēnāka.

Vides, kurās tika testētas abas lietotnes, nebija vienādas, tāpēc rezultāti jāuztver mazliet skeptiski, jo nav zināms, cik lielu daļu no rezultāta sastāda jaunais *ArcGIS* 10 serveris. Jebkurā gadījumā, liela daļa no šiem uzlabojumiem ir pateicoties tehnoloģijām, nevis arhitektūrai.

Piemēram, EF izmantošana. EF pats par sevi ir lēns, salīdzinot ar citiem ORM, tomēr šajā gadījumā tas ir krietni ātrāks par iepriekš izmantoto datu piekļuves tehnoloģiju – *ArcObjects*, kas ir ĢIS bibliotēka.

Tomēr, daļu no rezultātiem sastāda izmaiņas, kas tiešā veidā saistītas ar arhitektūru.

- Datu piekļuves slānis. Iepriekš datu piekļuve notika ar statiskām klasēm, kas parasti nozīmē to, ka katrs pieprasījums veido jaunu datubāzes savienojumu. Jaunā arhitektūra ļāva savienojumu regulēt globālā līmenī un izmantot tikai vienu savienojumu katram pieprasījumam.
- Iepriekš tika izmantots manuāli veidots ORM, kas diezgan slikti realizēja slinko datu atlasi – netika izmantota kešdarbe un katru reizi saistītie objekti tika pieprasīti no jauna. Savukārt augstāki koda līmeņi šo niansi ignorēja, un izmantoja datu piekļuves slāni tā, it kā dati glabātos operatīvajā atmiņā. Datu piekļuves nodalīšana un vaicājumu objektu izmantošana ļāva ielādēt visus nepieciešamos datus uzreiz.
- Sakārtota biznesa loģika un mazas, pārskatāmas funkcijas ļauj daudz labāk atrast vietas, kur tiek veikti lieki datubāzes pieprasījumi pēc datiem, kuri jau iepriekš ir pieprasīti, vai arī, kurus varētu pieprasīt, veicot savienošanu (*join*) nevis atsevišķus pieprasījumus.
- Datu piekļuves izdalīšana vaicājumu objektos ļauj ļoti labi pārskatīt, kur, kad, kādi dati tiek pieprasīti, kas tieši ir vajadzīgs pieprasījuma izpildei utt. Iepriekš datu pieprasījumi bija paslēpti atribūtos (*get/set*), līdz ar to bija viegli palaist garām, kad netīšam tiek pieprasīti dati.

Arhitektūrai (ne šajā darbā, ne vispār) tiešā veidā nav nekāda sakara ar veiktspēju. Visu papildus ietvaru – *Entity Framework, Windsor* - izmantošana tikai palēnina lietotni. Tomēr, salīdzinot veco ar jauno lietotni, autors vēlējās parādīt, ka sakārtojot kodu, var labāk pārvaldīt datu piekļuves slāni, tādā veidā izveidoties no liekiem pieprasījumiem un optimizējot esošos.

11. KLIENTA PUSES ARHITEKTŪRA UN TESTĒJAMĪBA

Šajā sadaļā tiks rakstīts par standarta tīmekļa tehnoloģijām – HTML, CSS, JS (*Javascript* – tīmekļa programmēšanas valoda). Ņemot vērā, ka par šo tēmu ir pieejams ļoti daudz informācijas, autors nemēģinās veikt informācijas apkopošanu, ietvaru salīdzināšanu vai ko tamlīdzīgu, tā vietā apskatot problēmu no savas pieredzes – moderna ietvara integrāciju ar mantoto kodu. Klienta puses ietvari un tehnoloģijas attīstās pārāk ātri, līdz ar to rodas problēma – kā visas šīs jaunas lietas apvienot ar jau esošu kodu, kurš ir rakstīts primitīvā JS stilā, kurš bija aktuāls pirms gadiem 5-7? Šai tēmai, pēc autora domām, tiek pievērsts pārāk maz uzmanības, jo koda pārrakstīšana ne vienmēr ir variants, un jaunie ietvari cenšas uzspiest savu arhitektūras stilu, pārāk maz uzmanības pievēršot to integrācijai ar citu kodu. Šajā nodaļa tiks apskatītas divas tēmas:

- *Angular* ietvara (1.3 versija) integrācija ar mantotu lietotni.
- Klienta puses koda automātiskā testēšana.

Angular ir klienta puses ietvars, kurš koncentrējas uz DOM (*Document Object Model* – funkcijas HTML piekļuvei un apstrādei) manipulāciju nodalīšanu no biznesa loģikas, atkarību pārvaldīšanu un testējamību [60]. Īsumā par galvenajiem uzbūves elementiem šajā ietvarā:

- Sfēra (*scope*) – modelis, satur datus, kas tiks attēloti skatā.
- Kontrolieris (*controller*) – klases, kas satur biznesa loģiku un uzstāda sfēras vērtības, kas tiks attēlotas skatā. Ja biznesa loģikas ir pārāk daudz, tad kontrolieris tikai orķestrē darbības, biznesa loģiku uzticot servisa klasēm.
- Serviss – klases, kas satur biznesa loģiku vai arī kādu citu, atkalizmantojamu loģiku.
- Skats – HTML fails ar mainīgajiem, ko uzstādīs kontrolieris.

Angular integrācija ar mantotu lietotni tiks aprakstīta, jo tā bija viena no lietām ar ko autors veica praktiskā darba ietvaros. Nepieciešamība pēc testējama koda bija liela, tāpēc tika pieņemts lēmums izmantot *Angular*:

- Visas jaunās sadaļas veidot kā *Angular* moduļus/kontrolierus.
- Eksistējošos moduļus maksimāli nodalīt DOM apstrādes funkcijas no algoritmiskām funkcijām.
    - DOM apstrādei veidot speciālus testus, kas izmanto DOM.
    - Pārējos komponentus izolēt no DOM un servera, un testēt izolācijā.

Autors ilgu laiku centās veidot testus mantotajai lietotnei, un tas parasti beidzās tā, ka pēc lielākām izmaiņām testi regulāri "salūza" un patērētais laiks bija pietiekami liels, lai sāktu apšaubīt šo testu lietderību. Problēmas sakne bija DOM manipulācijas – vecais kods bija rakstīts standarta *jQuery* (JS bibliotēka DOM manipulāciju atvieglošanai) stilā, ar brīvu piekļuvi visam HTML, kas radīja būtiskas problēmas sakarā ar testu izolāciju.

11.1. Testēšana

Mantotās lietotnēs pavisam noteikti nevarēs iztikt tikai ar vienībtestiem. DOM manipulācijas apgrūtina šo situāciju. Autors pagaidām izmanto 3 dažādus testu veidus (vēlāk, iespējams, radīsies vajadzība izdalīt vēl kādu):

- UNIT – vienībtesti *Angular* izpratnē. Izolēti no servera un DOM. Tīri algoritmiskas funkcijas.
- INT – integrācijas testi *Angular* izpratnē. Izolēti no DOM, bet izmanto serveri, lai testētu integrāciju ar to. Dažas darbības ir pārāk sarežģītas, lai tās viltotu klienta pusē, vienkāršāk ir samierināties ar testu sarežģītības pieaugumu, bet izmantot serveri kā dalībnieku testos, pie reizes pārbaudot arī integrāciju.
- DOM – mantotās lietotnes testi. Izmanto DOM, bet ne serveri. Prasa daudz vairāk padomāt par testpiemēru notīrīšanu, lai neatstātu būtiskas, traucējošas pēdas DOM kokā.

Ar vienīb/integrācijas testiem viss ir skaidrs – *Angular* ir pietiekami daudz pamācību, kur viss ir izstāstīts. Toties mantotu sistēmu DOM testi nav īpaši labi aprakstīts temats, tāpēc autors šajā apakšnodaļā aprakstīs savu risinājumu.

Pirmā problēma – kā nodrošināt, lai no servera puses nākošie dati būtu konstantā, paredzamā stāvoklī, kas nepieciešams testiem? Jāizmanto JS viltošanas ietvars, turklāt tāds, kas māk pārķert AJAX pieprasījumus. Ir pieejami vairāki ietvari (šajā gadījumā tika izvēlēts *Sinon* bibliotēka [61]), tomēr to standarta iespējams nebija pietiekamas, lai autors varētu realizēt savu funkcionalitāti.

Lietas, kas pietrūkst:

- Tā kā praktiski visi testpiemēri izmanto lielu datu apjomu, nepieciešams veids, kā lasīt datus no failiem vai tamlīdzīgi.

- Lai katru reizi nelasītu failu par jaunu, būtu labu izmantot kešdarbi. Ņemot vērā testa datu apjomu, pieprasījumi pēc testa datu failiem aizņem pārāk daudz laika un lieki kavē procesu.
- Vienas funkcijas izpildē bieži vien notiek vairāki pieprasījumi uz serveri. Pēc noklusējuma *Sinon* nepiedāvā ērtu veidu, kā norādīt, kuram pieprasījumam nepieciešama kura atbilde.

11.1.1. Kešdarbe

Līdzīgi kā nodaļā par biznesa loģikas testēšanu, autors testam nepieciešamos datus glabā JSON failos. Testa datu pieprasīšanas loģika nodalīta atsevišķā klasē – *testDataProvider.js*, turklāt, tā izmanto kešdarbi, lai lieki nepieprasītu vienu failu divreiz.

```
var testDataProviderCache = {};
myApp.factory('testDataProvider', function (core) {
    var cache = testDataProviderCache;
    function get(path) {
        var arr = path.map(function (p) {

            if (cache[p]) {
                var clone = core.arr.deepCopy(cache[p]);
                return Promise.resolve(clone);
            }
            return core.ajax.callAsync({url: "/tests/testData/" + p });
        });

        return asyncAll(arr).then(function(respArr) {
            // cache save
            for (var i=0; i<respArr.length; i++) {
                if (!cache[path[i]]) {
                    var clone = core.arr.deepCopy(respArr[i]);
                    cache[path[i]] = clone;
                }
            }
            return respArr;
        });
    }
```

Tātad, testa datu fails pie pirmās vajadzības tiek pieprasīts no failu sistēmas (testa datu faili ir pievienoti projektam). Nākamajā reizē tiek izmantota saglabātā kopija. Ļoti svarīgi ir neatgriezt saglabāto datu instanci tieši, bet kopēt, turklāt veikt dziļo kopēšanu (rinda *core.arr.deepCopy()*). Tādā veidā tiek panākta testpiemēru izolācija gadījumos, kad testpiemērs maina testa datus.

11.1.2. Viltus servera uzstādīšana

Viltus serveris ir domāts, lai pārķertu AJAX pieprasījumus un atgrieztu testam speciāli sagatavotos datus. Servera uzstādīšana *Sinon* notiek ļoti vienkārši [61]:

```
var server = sinon.fakeServer.create();
```

Problēmas sagādā pieprasījumu un atbilžu konfigurēšana. Standartā tas notiek šādi:

```
server.respondWith(url, response);
```

Url ir filtrs pieprasījumam, savukārt *response* ir divi varianti [61]:

- Masīva veidā norāda atbildes statusu (standarta tīmekļa atbildes statusi, piemēram, 200 ir veiksmīgs, 500 kļūda).
- Funkcija kas sagatavo atbildi pirmā varianta formātā. Šis ir elastīgs variants un tiks izmantots problēmas risināšanā.

Problēmas rodas tajā brīdī, kad uz vienu identisku URL ir vairāki pieprasījumi, un testā uz katru no tiem nepieciešamas dažādas atbildes.

Ņemot vērā šīs nepilnības, kā arī specifisko uzstādīšanas procedūru, autors visu šo funkcionalitāti iznesa atsevišķā klasē, kas būtībā ir vēl viens neliels slānis virs *Sinon*.

Lai risinātu iepriekš aprakstīto problēmu, klases iekšpusē katram pieprasījuma URL tiek izveidots masīvs, kas secīgi glabā atbildes. Tā kā *server.respondWith()* kā otro parametru var padot arī funkciju, tas dod papildus elastīgumu, ar ko pietiek, lai šajā situācijā varētu sameklēt atbilstošo atbildi pieprasījumam.

```
var fakeServer = (function () {

    var server = null;

    // key-arr, glabās url un atbilstošo atbilžu masīvu
    var serverResponses = {};

    // path - daļējs, piemēram Edit/Save
    function create() {
```

```
        server = sinon.fakeServer.create();
        server.respondImmediately = true;
        server.respondWith(function (xhr) {
            var url = core.str.replaceAll(xhr.url, "//", "/");
            var response = getNextResponse(url);
            xhr.respond(response[0], response[1], response[2]);
        });
    }

    // responseArr ir masīvs, [0] ir statuss, [1] ir pati atbilde
    function addSingle(path, responseArr) {

        if (!serverResponses[path]) {
            serverResponses[path] = [];
        }

        var status = responseArr[0];
        var response = JSON.stringify(responseArr[1]);

        serverResponses[path].push([status, { "Content-Type":
"application/json" }, response]);
    }

    function restore() {
        serverResponses = {};
        server.restore();
    }

    function getNextResponse(url) {
        for (var key in serverResponses) {
            if (url.indexOf(key) >= 0) {
                return serverResponses[key].shift();
            }
        }
    }
    return { … };
})();
```

Svarīga šeit ir *getNextResponse()* funkcija, kas pēc atbildes atgriešanas izmet to no masīva. Katra testa sākumā tiek izsaukts *create()* un beigās *restore()*. Lai nerakstītu katru reizi, var izmantot *beforeEach()* un *afterEach()*, attiecīgi uzreiz konfigurējot uzstādīšanu visai testu klasei, nevis katram testpiemēram atsevišķi.

Piemērs testam, kur ar AJAX tiek izsaukta servera funkcija *Edit/Save*, un viltus serveris pirmo reizi atgriež virkni "smth", bet otro reizi – "smthElse".

```
it("fakeServer.addSingle2x", function (done) {

   fakeServer.addSingle("Edit/Save", [200, { "smth": "smth" }]);
   fakeServer.addSingle("Edit/Save", [200, { "smth": " smthElse" }]);

   core.ajax.callAsync({url: "Edit/Save"}).then(function (resp) {
      expect(resp).toEqual({ "smth": "smth" });
   }).then(function () {
      return core.ajax.callAsync({ url: "Edit/Save" });
   }).then(function (resp) {
      expect(resp).toEqual({ "smth": "smthElse" });
   }).then(done);
});
```

Lai testā izmantotu datus no JSON failiem, nepieciešams tos vispirms pieprasīt ar *testDataProvider* palīdzību, un tad ievietot viltus servera atbilžu masīvā.

```
testDataProvider.get("path/to/test/file.json").then(function (resp) {
      fakeServer.addSingle("Edit/LoadData", [200, resp]);
});
```

11.2. Angular integrācija ar mantoto JS kodu

Tā kā viss *Angular* kods tiek rakstīts servisos un kontrolieros, tas ir cieši saistīts ar *Angular* ietvaru un tā uzbūvi. Šī koda izsaukšana no ārpuses nav pārāk eleganta. Vēl svarīgāk – ir būtiski pārvaldīt tieši savienojumu starp *Angular* un ne-*Angular* kodu, lai starp tiem nerastos cieša saistība. Būtībā ir jāatrisina divas lietas – kā no *Angular* izsaukt mantoto kodu, un otrādi.

11.2.1. Angular inicializācija

Problēma – *Angular* pēc noklusējuma veic ietvara inicializāciju uzreiz pēc lapas ielādes. Autora gadījumā (un droši vien lielākajā daļā citu mantoto sistēmu) ir nepieciešams pirms tam

veikt virkni citu darbu, un tikai tad veikt *Angular* inicializāciju. Jo īpaši tas attiecas uz DOM līmeņa testpiemēriem. Respektīvi, nepieciešams atlikt inicializācijas procesu.

Lai to izdarītu, manuāli jāizsauc *angular.bootstrap()* funkcija lietotnes sāknēšanas kodā [62]:

```
startupQueries.init() // notiek visi ārpus angular sagatavošanas darbi
   .then(function() { // visi dati ielādēti
    angular.bootstrap($("body").get(0), ["myApp"]);  // angular inicializācija
});
```

Ar sāknēšanas kodu domāts kāds speciāli izdalīts fails, kas veic dažādus uzstādīšanas darbus, lai "iedarbinātu" lietotni. Piemēram, autoram ir fails boot.js, kurš veic sākotnējo datu pieprasīšanu, globālo mainīgo ietīšanu servisos, kartes slāņu pievienošanu, *Angular* inicializāciju, kontroļu inicializāciju, utt. Ir svarīgi, lai lietotnei būtu kopīga inicializācija, nevis katrs modulis inicializētu pats sevi, jo tādos gadījumos ir grūti pēc tam apvienot šos notikumus vai izpildīt tos secībā. Savā ziņā var vilkt paralēles ar DI ietvariem, kur klases pašas neveido sev atkarību instances, bet tās tiek padotas no ārpuses.

11.2.2. Angular izsaukšana no mantotā koda

Angular dod iespēju no ārpuses piekļūt iekšējiem servisiem, kontrolieriem un citiem objektiem [63].

- *angular.element(domElement).injector()* – būtībā DI menedžeris, izsaucot atgrieztajam objektam *get(name)* funkciju, pēc vārda var piekļūt servisa instancei.
- *angular.element(domElement).controller()* – piekļūst kontroliera instancei.

Pēc autora domām, izkaisīt šādus izsaukumus pa visu kodu nav pārāk eleganti, jo funkcijas kā parametru pieņem DOM elementu. Ērtāk būtu piekļūt pēc nosaukuma. Tāpēc autors izveidoja tam apvalku:

```
var myAng = function () {
    function getCtrlByViewId(id) {
        return angular.element(document.getElementById(id)).scope();
    }

     function getCtrl(ctrlId) {
         var $div = $("[ng-controller=" + ctrlId + "]").ensure();
         var id = $div.prop("id");
         return getCtrlByViewId(id);
```

```
    }
    function getSrv(name) {
        return angular.element("body").injector().get(name);
    }
}
```

Lai katru reizi nebūtu jāraksta *myAng.getSrv(myServiceName)* ar labu iespēju kļūdīties servisa nosaukumā, autors izveidoja papildus moduli, kas automātiski reģistrē visus pieejamos servisus.

```
var ng = {};
ng.srvArr = [ ... ]; // masīvs ar servisu nosaukumiem

ng.init = function () {
    ng.srvArr.forEach(function (e) {
        var srv = myAng.getSrv(e);
        ng[e] = srv; // lai varētu piekļūt pēc ng[serviceName]
    });
};
```

Pēc *ng.init()* funkcijas izsaukšanas (to ieteicams darīt sāknēšanas failā), visi *ng.srvArr* norādītie servisi ir pieejami caur globālu parametru *ng* kā šī objekta atribūti. Šādā veidā tiek ieviesta zināma struktūra, jo visi servisi un kontrolieri ir pieejami tikai caur *ng* mainīgo.

11.2.3. Mantotā koda izsaukšana no Angular

Autors šādām situācijām iesaka strikti pieturēties pie *Angular* labās prakses attiecībā uz servisu/kontrolieru izolēšanu un atkarību norādīšanu:

- Nekādas tiešās DOM darbības nedrīkst tikt izsauktas no servisiem un kontrolieriem.
- Visas atkarības tiek padotas *Angular* stilā, kā atkarību masīvs objekta inicializācijā. Tas nozīmē, ka nenotiek nekāda globālu objektu izsaukšana, tiek izmantoti tikai tie objekti, kas tika padoti kā atkarības. Tas nozīmē, ka visi parastie JS objekti, kas būs nepieciešami kādam *Angular* objektam, ir "jāietin" servisos. Šie objekti joprojām būs globāli, tomēr tie būs pieejami no *Angular* caur vienotu mehānismu. Tam ir milzīga nozīme testējamībā.

Piemēram, parasts JS modulis, kas satur mantoto kodu un veco funkcionalitāti:

```
var someModule = {
    // 6000 rindas ar funkcijām, mainīgajiem un DOM manipulācijām …
};
```

Visi šāda stila moduļi, kuri būs nepieciešami arī *Angular* pusē, tiek ietīti servisos (ieteicams to darīt sāknēšanas kodā):

```
myApp.value("someModule", someModule);
```

Lai pēc tam tos varētu padot kā atkarības:

```
myApp.controller("SomeController", ["$scope", "someModule"
   , function($scope, someModule) {
   // kontroliera kods
}]);
```

Tādā veidā skaidri tiek norādītas atkarības. Ieteicams būtu servisus nosaukt tāpat kā vecos JS moduļus. Varbūt varētu mēģināt likt kādu frāzi, lai skaidri norādītu, ka tie ir vecie, mantotie, ne-*Angular* moduļi.

11.2.4. Notikumi (events)

Tīrā *Angular* lietotnē, saziņa starp moduļiem/kontrolieriem notiek ar *Angular* iebūvētajiem notikumiem *($broadcast, $emit, $on*). Starp mantoto kodu un *Angular* moduļiem šādā veidā sazināties nevar. Lai risinātu šo situāciju, autors izmanto paštaisītu noteikumu pārvaldnieku jeb publicēšanas/pierakstīšanās (*publish/subscribe*) moduli (iedvesmojoties no šī piemēra [64]):

```
core.events = (function (core) {
  // saturēs sarakstu ar kanāliem, piemēram, csi, edit, search utt.
  var channels = {};
  function publish(channelName, eventName, args) {
    channels[channelName] = channels[channelName] || {};
    channels[channelName][eventName] = channels[channelName][eventName] || [];

    for (var i = 0; i < channels[channelName][eventName].length; i++) {
        channels[channelName][eventName][i](args);
    }
  };
  function subscribe(channelName, eventName, func) {
    channels[channelName] = channels[channelName] || {};
    channels[channelName][eventName] = channels[channelName][eventName] || [];
    channels[channelName][eventName].push(func);
  };
  return { … };
})(core);
```

Autors no *Angular* standarta notikumiem ir atteicies vispār, tā vietā izmantojot augstāk aprakstīto variantu. Šis variants dod vienotu saskarni, kā sazināties moduļiem, kas savā starpā ir pārāk atšķirīgi, lai sarunātos tieši. Notikumi arī palīdz veidot vāji saistītus moduļus, tomēr jāatceras, ka nevajadzētu tos izmantot gadījumos, kad nepieciešama atgriežamā vērtība. Tādā veidā tiek radītas slēptas atkarības, kas ir vēl sliktāk.

11.3. Veiktspēja

Agrāk tēma par klienta puses veiktspēju bija lieka – normālā sistēmā, ja netiek pieļautas rupjas kļūdas, klienta puses veiktais darbs nav tik vērā ņemams, lai būtu jāsatraucas par veiktspēju. Tomēr šobrīd tēma ir kļuvusi aktuāla dēļ veida, kādā jaunie JS ietvari veic vērtību izmaiņu noteikšanu un skatu atjaunošanu. Mazām lietotnēm tā galīgi nav problēma, tomēr normāla izmēra lietotnēs, kritiska kļūst *Angular* veiktspēja. Lielākā daļa internetā esošo pamācību cenšas nodalīties no *Angular* iekšējās uzbūves, ķeroties uzreiz klāt pie funkcionalitātes veidošanas, neizskaidrojot, kā tas reāli strādā.

Praktiski par standarta ieteikumu attiecībā uz *Angular* veiktspēju ir kļuvis šāds nosacījums (bez sīkākiem paskaidrojumiem) – kamēr vērotāju skaits nepārsniedz 2000, nav problēmu [63]. Autors tam nepiekrīt, un ir nācies nonākt ar vairākās situācijās, kad šis apgalvojums nav patiess. Daudzums ir svarīgs, bet vēl svarīgāk – cik bieži vērotājam nākas atjaunot visus skatus? Pilnīgi pietiek ar 50 vērotājiem, kas izsaucas 50 reizes sekundē, lai katra skata atjaunošana prasītu sekundes, kas ir absolūti nepieņemami mūsdienās. Turklāt, viena vērotāja veiktais darba apjoms var būt dažāds. Lielākajā daļā gadījumu tā būs vienkārša divu mainīgo salīdzināšana lai noteiktu, vai tas ir mainījies. Tomēr, nav problēma izveidot sarežģītu vērotāju, kurš, piemēram, salīdzina divus masīvus, veic tīmekļa pieprasījumus utt. Nesaprotot, kā strādā *Angular*, šādas lietas vienkārši tiek palaistas garām.

Autors uzskata, ka nopietnās lietotnēs nedrīkst ķerties klāt pie *Angular* ietvara, neizprotot:

- Kas ir vērotāji?
- Kādos gadījumos tie tiek izveidoti?
- Kā noskaidrot, cik daudz to ir lietotnē?
- Kad notiek skatu atjaunošanas cikls (*digest cycle*)?

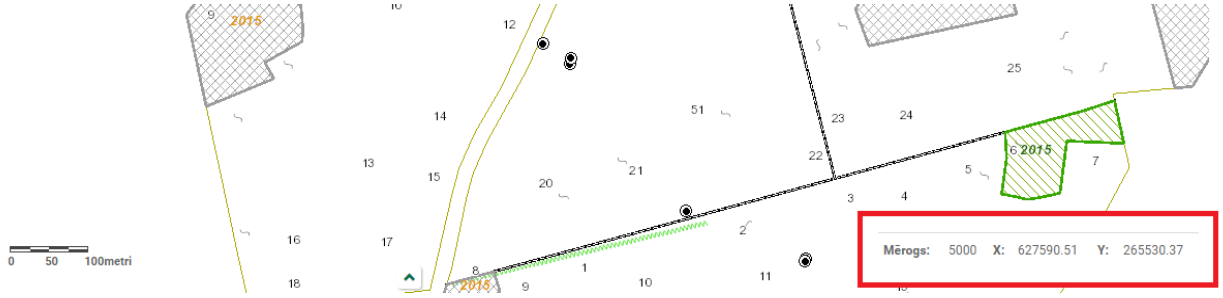
Angular izmanto netīro pārbaudi jeb *dirty checking*. Tēma ir vairāk nekā labi aprakstīta [65], tāpēc pavisam īsi – katru reizi, kad tiek izsaukts *$scope.$apply()*, ciklā tiek iets cauri visiem vērotajiem mainīgajiem un tiek salīdzināta vecā un jaunā vērtība. *Angular* vidē *$scope.$apply()*

izsaukšanu veic pats ietvars dažādos stratēģiski svarīgos brīžos, piemēram, pogas klikšķis, ievadlauku vērtību maiņa, AJAX pieprasījumu utt., bet ārpus *Angular* šo funkciju var izsaukt arī pats.

Ir vairākas tipiskas *Angular* veiktspējas problēmas un risinājumi, kas ir bieži aprakstīti – *ng-repeat* izmantošana lieliem sarakstiem, vienpusējā saistīšana (*one way data binding*), filtri, nemainīgās (*immutable*) kolekcijas utt. Turpinājumā autors pievieno pāris jaunas problēmas un risinājumus no savas pieredzes.

11.3.1. Mouseover notikumi

Noteikti ir jāuzmanās no šiem notikumiem *Angular* kontrolieros. Autoram bija situācija – lietotnes interfeisā attēlota karte, kurai uz *mouseover* notikumu formā mainās *x* un *y* vērtības.



Attēls 11.1 – apakšējā labajā stūrī *x* un *y* koordinātas

Vienā brīdi lietotnes interfeiss sāka nopietni "bremzēt". Uzstādot atkļūdotāju, tika noskaidrots, ka lietotnē veicot peles kursora vilkšanu pāri kartei (*mouseover* notikums) sekundē notiek pāris simti notikumu un katrs no tiem izsauc visu skatu pārģenerēšanu. Loģiski, ka tīmekļa pārlūks ar tādu darba apjomu netiek galā.

Būtībā ir divi risinājumi:

- Izmantot *debounce* parametru, kurš ļauj norādīt notikuma aizturi. Jeb norādīt laiku ms, cik ilgi pēc pēdējā notikuma veikt skata atjaunošanu.
- Vienkārši pārrakstīt moduli uz ne-*Angular* kodu.

Debounce parametrs bieži vien būs labs risinājums. Piemēram, *keyup* notikumi un meklēšanas lauks – veikt meklēšanu tikai tad, kad lietotājs beidzis rakstīt vai ievadījis noteiktu skaitu simbolu. Bet ir gadījumi, kad no gluži vienkārši no lietojamības šis nebūs pieņemams variants. Kaut vai ņemot vērā autora gadījumu – pirmo reizi *x* un *y* koordinātas tiks atjaunotas tikai tad, kad lietotājs novirzīs kursoru nost no kartes, kas būtībā ir diezgan mulsinoši un nez vai lietotājam liksies pašsaprotami.

Autors iesaka stipri uzmanīties no šādiem, biežiem notikumiem. Jo tādos brīžos nav vairs svarīgi, cik vērotāju ir lietotnē, svarīgs ir kopējais skaits, kas notiek sekundē.

11.3.2. Ng-include

Viena problēma, ar ko autoram nācās saskarties, bija *ng-include* izmantošana. Šī direktīva ir domāta, lai nebūtu viss HTML jāraksta vienā failā, bet varētu to norādīt pa daļām, kuras ielādēt asinhroni brīdī, kas lietotājs atver lietotni. Ielādētais saturs tiek ievietots DOM kokā. Autors izmantoja šo direktīvu tam domātajam mērķim, bet pietika ar pāris desmitiem direktīvu, lai lietotnes ielādēs izskatītos saraustīta - attēlu skaits sekundē bija zem acij tīkamā. Iemesls tam – katrs *ng-include* ielādes notikums beigās izsauc skatu pārlādes ciklu, kā rezultātā tikko atverot lietotni notiek vairāki desmiti (autora gadījumā) šo notikumu.

```
<div id="tab-legend" class="tab-pane">
    <div id="legend-div" class="panel panel-default no-margin"></div>
</div>

<div id="tab-search" class="tab-pane">
    <div id="panel-search" class="panel panel-default no-margin" ng-include="'js/search/search.html'"></div>
</div>

<div id="tab-drawing" class="tab-pane">
    <div id="panel-drawing" class="panel panel-default no-margin" ng-include="'js/drawing/drawing.html'" </div>
</div>

<div id="tab-measure" class="tab-pane">
    <div id="panel-measure" class="panel panel-default no-margin" ng-include="'js/measure/measure.html'" </div>
</div>
```

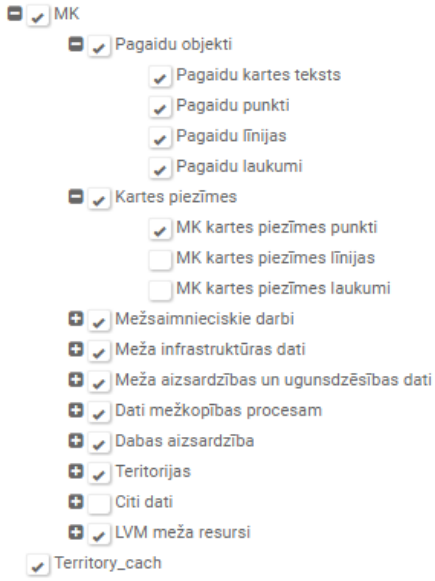
Attēls 11.2 – tabulatora kontrolis un *ng-include* izmantošana

Vēl viena problēma – *ng-include* neļauj normāli parakstīties uz direktīvas ielādes notikumu, tas netiek nekādā veidā publiskots vai padarīts pieejams. Kā rezultātā nākas meklēt dažādus neelegantus apejas risinājumus, lai varētu pierakstīties un izpildīt kodu tad, kad viss lapas kods ir ielādēts.

Pēc autora domām, vienīgais pieņemamais risinājums ir izmantot kādu no JS būves sistēmām (*build system*), piemēram *Gulp, Grunt,* un ielādēt visas *ng-include* direktīvas jau būvēšanas posmā, nevis katru reizi, kad lietotājs atver lapu. Piemēram, autors izmanto *Gulp* spraudni *gulp-file-include*, kurš var paveikt šo uzdevumu. Tas noteikti atstāj pozitīvu iespaidu arī uz lapas ielādes ātrumu, jo visas direktīvas jau uzreiz tiek pievienotas HTML kodam. Ir gan viena būtiska problēma – šis variants nestrādā pie dinamiskiem *ng-include*, piemēram, ja tie ir rekursīvi un atkarīgi no kāda mainīga lieluma. Šai problēmai autors pagaidām nav atradis risinājumu.

11.3.3. Rekursīvās struktūras

Rekursīvās struktūras ir ļoti vienkāršs veids, kā nemanot sasniegt augstu vērotāju skaitu un tādējādi padarīt lietotāja saskarni daudz lēnāku. Autora piemērs ir sekojošs – kartes slāņu saraksts, kur katram slānim var būt vairāki apakšslāņi. Katram slānim ir astoņi mainīgie, kas ir iekļauti skatā. Rezultātā pie 100 slāņiem ir jau 800 vērotāju un praktiski tikai viena lietotnes sadaļa jau rada pietiekami lielu noslodzi.



Attēls 11.3 – rekursīva struktūra – kartes slāņu saraksts

Pāris varianti, kā risināt situāciju:

- Reizēm 2-3 skata vērtību vietā var izmantot vienu, pirms tam sarēķinātu.
- Kur vien iespējams, izmantot vienpusējo vērtību savienošanu. Piemēram, autora gadījumā bija skaidrs, ka slāņa nosaukums ir nemainīgs, līdz ar to tā ir vienpusēja vērtība, respektīvi, skatu atjaunošanas ciklā šo vērtību nav nepieciešams vairs pārbaudīt.
- *ng-if* izmantošana. Šī direktīva nevis paslēpj, bet vispār aizvāc struktūru no DOM, ja nosacījums neizpildās. Labi pielietojams pie paslēpjamiem elementiem. Piemēram, autora gadījumā virsējie slāņi ir sakļaujami, kas nozīmē, ka visus zemākos var vienkārši noņemt no DOM, tādā veidā ietaupot daudz vērotājus. Mīnuss – elementu atkalpievienošana ir ilgāka, nekā izmantojot parasto slēpšanu. Līdz ar to, šis variants pielietojams situācijām, kad elementi netiek bieži slēpti, rādīti.

11.4. Secinājumi

Klienta puses kodā par moduļu/komponentu izolāciju ir jādomā vēl vairāk, nekā servera. Tā kā JS valodā ir ļoti viegli izveidot globālus mainīgos un brīvi manipulēt ar visu DOM, ir ļoti jācenšas izolēties un nodalīt moduļus. Ja par šīm lietām netiek piedomāts, iegūtajam risinājumam ir praktiski nereāli izveidot saprotamus un uzturamus automātiskos testus.

JS ietvaru fragmentācija kļūst arvien izteiktāka, kas rada problēmas tādā ziņā, ka ne visi ietvari un bibliotēkas labi iet kopā. Jaunākie ietvari ātri vien noveco un to vietā nāk jauni, savienojamības iespējas un atpakaļsaderība ir diezgan sliktā līmenī.

Attiecībā uz biznesa loģikas realizāciju, ir svarīgi nedublēt datus JS objektos un DOM kokā jeb nemēģināt manuāli veidot kaut ko līdzīgu ietvaram, kas veic skatu un JS objektu sasaistīšanu. Tas vienmēr beidzas vienā veidā – neliela kļūda atjaunojot datus vai nu DOM vai JS objektos, kā rezultātā tas ko redz lietotājs, un dati, kas tiek nosūtīti uz saglabāšanu, ir pilnīgi dažādi. Šādas kļūdas ir ļoti grūti identificējamas – lietotājs saka, ka saglabājis vienu, bet servera audita ierakstos ir pavisam kas cits, un nevar saprast – kam ticēt? Vai nu tad izmantot DOM datu glabāšanai, vai izmantot kādu stabilu ietvaru (*Angular, Backbone, Knockout* - vienalga). Tikai ne kaut ko pa vidu.

Angular ir lielisks ietvars, tomēr tam ir savas robežas. Īpaši tas izpaužas sakarā ar veiktspēju. Tādos brīžos ir labi apsvērt variantu būvēt ne visas lietas *Angular* stilā. Piemēram, autors *Angular* uztic tikai atsevišķu moduļu funkcionalitāti, kodols ir rakstīts tīrā JS. Kaut gan, tas vairāk saistīts ar to, ka lietotne ir mantota un satur daudz kodu, kas veic tiešas DOM manipulācijas.

REZULTĀTI

Atsaucoties uz ievada aprakstītajiem mērķiem, rezultāti pa nodaļām:

- Arhitektūra - Aprakstīti dažādi arhitektūras modeļi, kuri koncentrējas uz risinājuma nodalīšanu slāņos, modularitāti, testēšanu, vāju saistību starp slāņiem, moduļiem, klasēm, ātrdarbību un biznesa loģikas pārvaldīšanu
- Atkarību injicēšana - aprakstīti trīs injicēšanas pamatveidi, dažādi scenāriji *Windsor* DI ietvara izmantošanā, konfigurēšanā un integrēšanā ASP.NET MVC veida lietotnēs.
- Vienībtesti – aprakstītas testēšanas labās prakses, testu dubultnieku iedalījums, DI ietvara izmantošana testu uzstādīšanas fāzē.
- Starpnozaru problēmas – visas būtiskās problēmas bija atrisināmas. Izmantojot ASP.NET filtrus, var ieviest labu risinājumu starpnozaru problēmām MVC kontrolieru līmenī.
- Datu piekļuves slānis – atrasts risinājums vairākām problēmām, kas izriet no EF un eksistējošas datubāzes izmantošanas. Daudz ko var apiet, tomēr EF pietrūkst elastīguma.
- Biznesa loģikas slānis – aprakstīta izolēšana no datu piekļuves slāņa. To var paveikt, bet ļoti diskutējams ir jautājums, vai to vajag? Autors sliecās uz to, ka lielākajā daļā gadījumu tam būs vairāk negatīvas sekas.
- Biznesa loģikas testēšana – aprakstīts risinājums, kurš izmanto testa objektu grafu glabāšanu JSON failos, ievietošanu datubāzē testa sākumā ar atriti beigās. Mantota sistēma un eksistējoša datubāze rada daudz problēmu. Beigās nākas izvēlēties starp testu konsistenci un patērēto laiku.
- Pētījuma gaitā izveidotā arhitektūra – aprakstīta pētījumā izmantotā lietotne un izveidotās arhitektūras principi.
- Koda metriku salīdzinājums – apskatīti *Visual Studio* koda metriku dati dažādos griezumos, pierādot, ka jaunā arhitektūra ir modulārāka, testējamāka, vienkāršāka un uzturamāka nekā iepriekšējā ne tikai pēc autora domām, bet arī pēc statistikas.
- Veiktspējas testi – arhitektūra kā tāda neietekmē veiktspēju, tomēr šajā sadaļā parādīts, ka sakārtota sistēma tomēr strādā ātrāk.
- Klienta puses arhitektūra – aprakstīts risinājums mantotas lietotnes (JS) testēšanai un integrācijai ar *Angular* ietvaru.

SECINĀJUMI

- Attiecībā uz biznesa loģikas testēšanu - sarežģītās sistēmās, jo īpaši mantotās, kuras daudz izmanto parastos SQL pieprasījumus un ne tikai EF, izolēšanās no datubāzes prasa pārāk daudz resursu, bet testi nedod gaidīto pārliecību par sistēmas korektumu, jo lielākā daļa kļūdu saistītas ar datubāzi vai ORM.
- Gandrīz vienmēr, ne visa sistēma būs labi modelēta un ar skaistu arhitektūru. Tomēr, jācenšas labās un sliktās daļas maksimāli nodalīt.
- Lietotnēm ar sarežģītu datu shēmu un lielām prasībām pret ORM elastīgumu, būtu nepieciešams apsvērt mikro ORM izmantošanu, nevis lielo, funkcijām bagāto EF. Šajā ziņā autors, ļoti iespējams, sākotnēji pieņēma nepareizu lēmumu.
- Attiecībā uz izolēšanos no datu piekļuves, nepieciešams stipri padomāt, vai tas vispār vajadzīgs.
- Pētījuma sākumā autors koncentrējās vairāk uz vienībtestiem, tomēr praktiskās izstrādes gaitā saprata, ka šāda tipa sistēmām labāk piemēroti zema līmeņa integrācijas testi, veidoti atbilstoši BDD tehnikām.
- Rakstīt vienībtestus nav grūti, grūti ir rakstīt kodu, kuram var uzrakstīt vienībtestus, turklāt tādus, kas joprojām darbosies arī pēc mēneša un "nesalūzīs" pie minimālām testējamās klases izmaiņām.
- Lai izveidotu testējamu lietotni, par risinājumu jāsāk domāt arhitektūras veidošanas brīdī.
- Praktiski obligāts nosacījums ir arhitektūras dalīšana slāņos un pienākumu nodalīšana. Pretējā gadījumā biznesa loģikas kods saplūst ar prezentācijas kodu, kas ir tipisks piemērs nenotestējamam kodam.
- Arhitektūra jādala ne tikai slāņos, bet vispirms moduļos. Tālāk katru moduli var dalīt slāņos.
- Mūsdienīgas arhitektūras koncentrējas uz vāji saistītiem slāņiem un klasēm, kā rezultātā iespējams izolēt jebkuru atkarību, jo īpaši ārējās atkarības (datubāze utt.).
- Progresīvas arhitektūras cenšas izolēties no datubāzes, nodalot datu piekļuves slāni kā ārēju faktoru.

- Testpiemēri jāveido maksimāli īsi un saprotami, citādāk nepatiesa apgalvojuma gadījumā ir grūti saprast problēmas cēloni. Tas noved pie grūti lasāmiem un uzturamiem testiem.
- Vienībtestam jābūt ātram un automātiskam, lai koda izmaiņu gadījumā ātri varētu izpildīt visus vienībtestus.
- Vienībtestiem un integrācijas testiem jābūt atkārtojamiem un konsekventiem. Tāpēc ir svarīgi testus veidot izolētus vienu no otra, lai tiem nebūtu kopīgs stāvoklis.
- Zemāka līmeņa algoritmiskiem komponentiem efektīvāk ir izmantot TDD tipa testpiemērus, savukārt augstāka līmeņa – BDD tipa, mēģinot realizēt kādu lietošanas gadījumu un pārbaudot visus funkcijas izvadus.
- Labākais laiks, kad rakstīt testus ir izstrādes laikā, paralēli ar testējamās funkcionalitātes ieviešanu. Atstājot testus uz vēlāku laiku, bieži vien nākas secināt, ka klases struktūra nav piemērota testēšanai, kā rezultātā nākas veikt izmaiņas, kas ietekmē arī citas klases, līdz ar to rada lieku darbu, no kura varēja izvairīties.
- DI ietvaram jābūt nemanāmam – tikai maza daļa no lietotnes koda drīkst izmantot tiešos izsaukumus ietvara funkcijām. Parasti tam vajadzētu notikt augsta līmeņa sāknēšanas kodā. Pretējs piemērs ir *Service Locator* modelis, kā rezultātā tiek pārkāpts strikti nodalīto atkarību princips, un klase iegūst globālas atkarības.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Martin, Robert C. objectmentor.com - Principles and Patterns. [Tiešsaiste] [Citēts: 15.12.2014.] http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf.

2. msdn.microsoft.com - MVC. [Tiešsaiste] [Citēts: 25.01.2015.] https://msdn.microsoft.com/en-us/library/dd381412(v=vs.108).aspx.

3. mvctemplate.com. [Tiešsaiste] [Citēts: 25.01.2015.] http://www.mvctemplate.com/Documentation/Introduction/Architecture.html.

4. eohmicrosoft.blogspot.com. [Tiešsaiste] [Citēts: 25.01.2015.] http://eohmicrosoft.blogspot.com/2012/08/laying-it-out-onion-architecture.html.

5. Palermo, Jeffrey. jeffreypalermo.com - The Onion Architecture. [Tiešsaiste] [Citēts: 20.12.2014.] http://jeffreypalermo.com/blog/the-onion-architecture-part-1/.

6. Hanselman, Scott. hanselman.com - MultiLayer Architecture. [Tiešsaiste] [Citēts: 25.01.2015.] http://www.hanselman.com/blog/AReminderOnThreeMultiTierLayerArchitectureDesignBroughtToYouByMyLateNightFrustrations.aspx.

7. Cockburn, Alistair. alistair.cockburn.us. [Tiešsaiste] [Citēts: 27.01.2015.] http://alistair.cockburn.us/Hexagonal+architecture.

8. Martin, Robert C. blog.8thlight.com. [Tiešsaiste] [Citēts: 27.01.2015.] http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html.

9. Young, Greg. cqrs.files.wordpress.com. [Tiešsaiste] [Citēts: 25.01.2015.] https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.

10. Evans, Eric. *Domain Driven Design: Tackling Complexity in the Heart of Software.* bez viet. : Addison-Wesley, 2003.

11. newmedialabs.com. [Tiešsaiste] [Citēts: 26.01.2015.] http://newmedialabs.com/.

12. Seemann, Mark. *Dependency Injection in .NET.* bez viet. : Manning Publications, 2011.

13. Fowler, Martin. martinfowler.com - DI. [Tiešsaiste] [Citēts: 20.12.2014.] http://martinfowler.com/articles/injection.html.

14. Palme, Daniel. Palmmedia. [Tiešsaiste] [Citēts: 13.01.2015.] http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison.

15. Koźmic, Krzysztof. Windsor Installers. [Tiešsaiste] [Citēts: 27.01.2015.] http://docs.castleproject.org/Default.aspx?Page=Installers&NS=Windsor&AspxAutoDetectCookieSupport=1.

16. —. Windsor registering components by conventions. [Tiešsaiste] [Citēts: 27.01.2015.] http://docs.castleproject.org/Windsor.Registering-components-by-conventions.ashx.

17. stackoverflow.com - Override Existing Registration. [Tiešsaiste] [Citēts: 12.01.2015.] http://stackoverflow.com/questions/9253388/in-castle-windsor-3-override-an-existing-component-registration.

18. docs.castleproject.org - Standard Lifestyles. [Tiešsaiste] [Citēts: 15.01.2015.] http://docs.castleproject.org/Windsor.LifeStyles.ashx#Standard_lifestyles:_the_common_ones_14.

19. docs.castleproject.org - Plugging Windsor In. [Tiešsaiste] [Citēts: 07.01.2015.] http://docs.castleproject.org/Windsor.Windsor-tutorial-part-two-plugging-Windsor-in.ashx.

20. Osherove, Roy. *The Art of Unit Testing, Second Edition.* bez viet. : Manning Publications, 2014.

21. Fowler, Martin. martinfowler.com/bliki/UnitTest.html. [Tiešsaiste] [Citēts: 27.01.2015.] http://martinfowler.com/bliki/UnitTest.html.

22. George, Boby un Williams, Laurie. An Initial Investigation of Test Driven Development in Industry. [Tiešsaiste] http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf.

23. Williams, Laurie, Kudrjavets, Gunnar un Nagappan, Nachiappan. On the Effectiveness of Unit Test Automation at Microsoft. [Tiešsaiste] http://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf.

24. xunitpatterns - Four Phase Test. [Tiešsaiste] [Citēts: 05.01.2015.] http://xunitpatterns.com/Four%20Phase%20Test.html.

25. North, Dan. dannorth.net - Introducing BDD. [Tiešsaiste] [Citēts: 05.01.2015.] http://dannorth.net/introducing-bdd/.

26. stackoverflow.com - AAAA. [Tiešsaiste] [Citēts: 05.01.2015.] http://stackoverflow.com/questions/1021007/should-it-be-arrange-assert-act-assert.

27. Meszaros, Gerard. *XUnit Test Patterns.* bez viet. : Addison-Wesley, 2007.

28. Seemann, Mark. blog.ploeh.dk - Auto Mocking Container. [Tiešsaiste] [Citēts: 15.01.2015.] http://blog.ploeh.dk/2013/03/11/auto-mocking-container/.

29. Koźmic, Krzysztof. kozmic.net - Castle Windsor Lazy Loading. [Tiešsaiste] [Citēts: 13.01.2015.] http://kozmic.net/2009/11/.

30. Davis, Matt. bronumski - Auto Mocking with NSubstitute. [Tiešsaiste] [Citēts: 12.01.2015.] http://bronumski.blogspot.com/2013/02/auto-mocking-with-nsubstitute-and.html.

31. Needham, Mark. markhneedham.com. [Tiešsaiste] [Citēts: 25.04.2015.] http://www.markhneedham.com/blog/2009/04/04/functional-c-the-hole-in-the-middle-pattern/.

32. Allen, Scott. odetocode.com. [Tiešsaiste] [Citēts: 25.04.2015.] http://odetocode.com/blogs/scott/archive/2010/06/28/action-filter-versus-controller-base-class.aspx.

33. Filtering in ASP.NET MVC. [Tiešsaiste] [Citēts: 25.04.2015.] https://msdn.microsoft.com/en-us/library/gg416513(VS.98).aspx.

34. Alekseev, Igor. ialekseev.blogspot.com. [Tiešsaiste] [Citēts: 25.04.2015.] http://ialekseev.blogspot.com/2012/10/dependency-injection-in-aspnet-mvc-3.html.

35. asp.net. [Tiešsaiste] [Citēts: 25.04.2015.] http://www.asp.net/whitepapers/mvc3-release-notes#RTM-BC.

36. Radanovic, Ivan. gist.github.com. [Tiešsaiste] [Citēts: 25.04.2015.] https://gist.github.com/ivanra/9019273.

37. hwyfwk.com. [Tiešsaiste] [Citēts: 25.04.2015.] http://hwyfwk.com/blog/2013/10/06/mvc-filters-with-dependency-injection/.

38. stackoverflow.com. [Tiešsaiste] [Citēts: 26.04.2015.] http://stackoverflow.com/questions/8937200/are-actionfilterattributes-reused-across-threads-how-does-that-work.

39. Kozmic, Krzysztof. Introduction to AOP With Castle. [Tiešsaiste] [Citēts: 26.04.2015.] http://docs.castleproject.org/Default.aspx?Page=Introduction-to-AOP-With-Castle&NS=Windsor&AspxAutoDetectCookieSupport=1.

40. Roubíček, Aleš. Castle Windsor Interceptor Attribute. [Tiešsaiste] [Citēts: 26.04.2015.] http://stackoverflow.com/questions/21148285/castle-windsor-interceptor-not-working-with-method-level-attribute.

41. Unique Constraint (i.e. Candidate Key) Support. [Tiešsaiste] [Citēts: 20.03.2015.] http://data.uservoice.com/forums/72025-entity-framework-feature-suggestions/suggestions/1050579-unique-constraint-i-e-candidate-key-support.

42. How to pass a System.Type into a generic method using reflection. [Tiešsaiste] [Citēts: 20.03.2015.] http://stackoverflow.com/questions/7488438/how-to-pass-a-system-type-into-a-generic-method-using-reflection.

43. Vega, Diego. EntityConnection can only be constructed with a closed DbConnection. [Tiešsaiste] [Citēts: 20.03.2015.] http://blogs.msdn.com/b/diego/archive/2012/01/26/exception-from-dbcontext-api-entityconnection-can-only-be-constructed-with-a-closed-dbconnection.aspx.

44. Dykstra, Tom. Implementing the Repository and Unit of Work. [Tiešsaiste] [Citēts: 01.04.2015.] http://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application.

45. Fowler, Martin. *Patterns of Enterprise Application Architecture.* bez viet. : Addison-Wesley Professional, 2002.

46. Rahien, Ayende. ayende.com. [Tiešsaiste] [Citēts: 01.04.2015.] http://ayende.com/blog/3955/repository-is-the-new-singleton.

47. Conery, Rob. rob.conery.io. [Tiešsaiste] [Citēts: 01.04.2015.] http://rob.conery.io/2014/03/04/repositories-and-unitofwork-are-not-a-good-idea/.

48. Alsing, Roger. Entity Framework 4 – Entity Dependency Injection. [Tiešsaiste] [Citēts: 10.05.2015.] http://rogeralsing.com/2009/05/30/entity-framework-4-entity-dependency-injection/.

49. Pobiega, Szymon. Dependency inversion patterns and the domain model. [Tiešsaiste] [Citēts: 07.05.2015.] http://simon-says-architecture.com/2010/03/31/dependency-inversion-patterns-and-the-domain-model/.

50. Bogard, Jimmy. Strengthening your domain: The double dispatch pattern. [Tiešsaiste] [Citēts: 07.05.2015.] https://lostechies.com/jimmybogard/2010/03/30/strengthening-your-domain-the-double-dispatch-pattern/.

51. Testing with a mocking framework (EF6 onwards). [Tiešsaiste] [Citēts: 10.04.2015.] https://msdn.microsoft.com/en-us/data/dn314429.aspx.

52. effort.codeplex.com/. [Tiešsaiste] [Citēts: 11.04.2015.] http://effort.codeplex.com/.

53. Bogard, Jimmy. Isolating database data in integration tests. [Tiešsaiste] [Citēts: 10.04.2015.] https://lostechies.com/jimmybogard/2012/10/18/isolating-database-data-in-integration-tests/.

54. Vasilyev, Max. Rollback attribute for NUnit and Entity framework. [Tiešsaiste] [Citēts: 15.04.2015.] http://tech.trailmax.info/2013/04/rollback-attribute-for-nunit-and-entity-framework/.

55. —. How We Do Database Integration Tests With Entity Framework Migrations. [Tiešsaiste] [Citēts: 10.04.2015.] http://tech.trailmax.info/2014/03/how-we-do-database-integration-tests-with-entity-framework-migrations/.

56. Mrnka, Ladislav. Fake DbContext of Entity Framework 4.1 to Test. [Tiešsaiste] [Citēts: 14.04.2015.] http://stackoverflow.com/questions/6904139/fake-dbcontext-of-entity-framework-4-1-to-test.

57. Code Metrics Values. [Tiešsaiste] [Citēts: 17.05.2015.] https://msdn.microsoft.com/en-us/library/bb385914.aspx.

58. Code Metrics – Class Coupling. [Tiešsaiste] [Citēts: 20.05.2015.] http://blogs.msdn.com/b/zainnab/archive/2011/05/25/code-metrics-class-coupling.aspx.

59. Deursen, Arie van. Think Twice Before Using the "Maintainability Index". [Tiešsaiste] [Citēts: 17.05.2015.] http://avandeursen.com/2014/08/29/think-twice-before-using-the-maintainability-index/.

60. Guide to AngularJS. [Tiešsaiste] [Citēts: 22.05.2015.] https://docs.angularjs.org/guide.

61. Johansen, Christian. Sinon Fake Server. [Tiešsaiste] [Citēts: 22.05.2015.] http://sinonjs.org/docs/#server.

62. Which method should I use to manually bootstrap my AngularJS? [Tiešsaiste] [Citēts: 20.05.2015.] http://stackoverflow.com/questions/16537783/which-method-should-i-use-to-manually-bootstrap-my-angularjs.

63. Hevery, Misko. Call Angular JS from legacy code. [Tiešsaiste] [Citēts: 20.02.2015.] http://stackoverflow.com/questions/10490570/call-angular-js-from-legacy-code/10508731#10508731.

64. Osmani, Addy. Understanding the Publish/Subscribe Pattern for Greater JavaScript Scalability. [Tiešsaiste] [Citēts: 20.05.2015.] https://msdn.microsoft.com/en-us/magazine/hh201955.aspx.

65. Kras, Alex. 11 Tips to Improve AngularJS Performance. [Tiešsaiste] [Citēts: 22.05.2015.] http://www.alexkras.com/11-tips-to-improve-angularjs-performance/.

Maģistra darbs: **Modulāras un testējamas arhitektūras veidošana tīmekļa lietotnēm**

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: _____
(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____
(Vadītāja paraksts)

Darbs iesniegts maģistrantūras sekretariātā _____.
(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: _____.
(Metodiķes paraksts)

Recenzents: _____
(Akad. amats, zin. grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____
(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____
(Sekretāra paraksts)