LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

ANGULAR SPECIFISKA TYPESCRIPT PIRMKODA ĢENERĒŠANA AR ROSLYN .NET KOMPILATORA PLATFORMU

MAĢISTRA DARBS

Autors: **Kristers Zīmecs**
Stud. apl. Nr. kz11042
Darba vadītājs: Dr. dat. Sergejs Kozlovičs

RĪGA 2017

ANOTĀCIJA

Modernos tīmekļa risinājumos datu modelis tiek dublēts starp servera un klienta lietotnēm. Šo informāciju vajag uzturēt atbilstošu abos galos - mainoties datu modelim, izmaiņas vajag atspoguļot gan servera, gan klienta pusē (kas atbilst dažādām izpildes vidēm un dažādām programmēšanas valodām). Lai izvairītos no manuālās izmaiņu sinhronizācijas, darbā tiek aplūkota pirmkoda ģenerēšanas pieeja, kas ļauj šo procesu automatizēt.

Darbā identificēti dublētie informācijas vienumi tieši ASP.NET (servera pusē) un Angular (klienta pusē) tīmekļa risinājumos, izpētīti esošie problēmas risinājumi, kas veic pirmkoda ģenerēšanu, kā arī izpētītas .NET pirmkoda analīzes un TypeScript pirmkoda ģenerēšanas iespējas.

Darba praktiskajā daļā izstrādāts risinājums, kas spēj analizēt .NET servera puses lietotnes ar Roslyn un ģenerēt no servera lietotnes atkarīgās TypeScript komponentes Angular klienta puses risinājumiem.

Izveidotais risinājums ļauj automatizēt datu modeļa sinhronizāciju starp servera un klienta puses lietotnēm, tādējādi ietaupot izstrādātāju laiku un paaugstinot produktivitāti.

Atslēgvārdi: Roslyn .NET kompilatora platforma, TypeScript, Angular, pirmkoda ģenerēšana, pirmkoda analīze, tīmekļa tehnoloģijas.

ABSTRACT

Angular specific TypeScript source code generation using Roslyn .NET compiler platform.

In modern web solutions, data models are being duplicated in the server and client side applications. This information must be kept in sync between both parties – when data models change, that change must be reflected in both server and client side applications (which run in different execution environments and are implemented in different programming languages). To avoid manual synchronization of said changes, source code generation approaches are examined, which allow the automatization of this process.

In this paper the author identifies duplicated pieces of information in ASP.NET (server side) and Angular (client side) web solutions, analyzes existing solutions that do source code generation and examines .NET source code analysis and TypeScript source code generation options.

As a result, author presents a solution for analyzing .NET server side applications with Roslyn and generating server-dependent TypeScript components for Angular client side applications.

The solution allows developers to automate the synchronization of data models between server and client side applications, thus saving time and increasing productivity.

Keywords: Roslyn .NET compiler platform, TypeScript, Angular, source code generation, source code analysis, web technologies.

AUTOREFERĀTS

Darba praktiskās daļas rezultāts ir autora patstāvīgi izstrādāts risinājums servera un klienta puses komponenšu sinhronizācijai ASP.NET un Angular risinājumos. Risinājums ir labāks par esošajiem, jo spēj veikt datu modeļu, tīmekļa klientu un validācijas nosacījumu ģenerēšanu – radniecīgi risinājumi ir funkcionāli nepilnīgāki, neatbalsta nepieciešamās tīmekļa tehnoloģijas vai nav piemēroti darbā apskatīto problēmu risināšanai. Papildus tam, tiek izmantoti moderni un piemēroti rīki tehniskajai realizācijai – Roslyn .NET kompilatora platformu un TypeScript kompilatora API.

Darba teorētiskajā daļā ir izpētīti .NET rīki pirmkoda analīzei un ģenerēšanai, par pamatu ņemot .NET meta-programmēšanas grāmatu un tīmekļa resursus. Meklēti, analizēti un novērtēti publiski pieejamie rīki darba vai radniecīgu problēmu risināšanai. Darba problēmas rakstura dēļ, esošie risinājumi ir gana specifiski un ne ļoti apjomīgi projekti, tāpēc informācija un atsauksmes par tiem lielākoties atrodamas tīmekļa resursos kopā ar programmatūras pirmkodu.

Problēmas un risinājumi izklāstīti pietiekoši detalizēti, darba ievadā aprakstītas visas darbā izmantotās tehnoloģijas un koncepti. Lai labāk ilustrētu idejas, tehnoloģisko pielietojumu un ieguvumus, darbs satur grafiskus materiālus un pirmkoda fragmentus. Darba nodaļu, apakšnodaļu un punktu struktūra ir ar augstu detalizācijas pakāpi.

Galvenais darba rezultāts un lielākais darba apjoms ir ieguldīts programmatūras izstrādē, kas kalpo par darba problēmas risinājumu. Risinājuma praktiskai izstrādei veltīti aptuveni 3 mēneši. Maģistra darba dokumenta izstrādei veltīti aptuveni 2 mēneši.

Izstrādātais risinājums tiek izmantots praksē, vairāku tīmekļa projektu izstrādē. Ir plāns risinājumu turpināt attīstīt, kā arī iepazīstināt citas, ieinteresētas uzņēmuma izstrādes komandas ar to, uzlabot dokumentāciju un izplatīt uzņēmumā, izmantojot kādu programmatūras pārvaldības risinājumu. Risinājuma validācijai realizēta testu automatizācija, to ikdienā izmanto un validē vairāki izstrādātāji – rīka lietotāji. Praksē risinājums savu uzdevumu veic labi un ļauj ietaupīt būtisku laiku apjomu, kurš savādāk tiktu pavadīts manuāli sinhronizējot informāciju starp servera un klienta lietotnēm.

Darba teksts ir pārlasīts un atrastās kļūdas ir izlabotas. Darbā ir izmantota pieņemtā nozares terminoloģija un ir izpildīti metodisko norādījumu 6.pielikuma “Darba noformējuma kontrolsaraksts” minētie punkti. Darbs veikts patstāvīgi, visi aizguvumi no citiem autoriem ir atzīmēti ar literatūras atsaucēm. Maģistra darba dokuments izstrādāts balstoties uz Latvijas Universitātes Datorikas Fakultātes sagatavoto dokumentu “Maģistra darba izstrādes un aizstāvēšanas metodiskie norādījumi”.

SATURS

APZĪMĒJUMU SARAKSTS

Nodaļā identificēti specifiski apzīmējumi, kas tiek lietoti maģistra darbā. Apzīmējumi un to paskaidrojumi atrodami 1.1. tabula.

1.1. tabula

Apzīmējumu saraksts

Apzīmējums	Paskaidrojums
.NET	Programmatūras izstrādes ietvars.
ADO.NET	Datu piekļuves tehnoloģija .NET ietvaram, ko izmanto komunikācijai ar relāciju sistēmām, piemēram, SQL datubāzēm.
AJAX	No angļu valodas (*asynchronous JavaScript and XML*) – tīmekļa tehnoloģiju kopums asinhronu pieprasījumu veikšanai no klienta uz servera lietotnēm.
API	No angļu valodas (*Application Programming Interface*) – lietojumprogrammas saskarne.
ASP.NET	Tīmekļa lietotņu izstrādes ietvars .NET ietvaram.
AST	No angļu valodas (*abstract syntax tree*) – abstraktais sintakses koks.
BCL	No angļu valodas (*Base Class Library*) – bāzes klašu bibliotēka.
C#	Plaša pielietojuma, objektorientēta programmēšanas valoda, ko izstrādājis Microsoft. Viena no .NET ietvara programmēšanas valodām.
CIL	No angļu valodas (*Common Intermediate Language*) – zema līmeņa programmēšanas valoda .NET platformai.
CLI	No angļu valodas (*Command Line Interface*) – komandrindas saskarne.
CLR	No angļu valodas (*Common Language Runtime*) – .NET platformas komponente, kas nodrošina .NET programmatūras izpildi.
CPU	No angļu valodas (*central processing unit*) – centrālais procesors.
CRUD	No angļu valodas (*Create, Read, Update, Delete*) – apzīmē resursa izveides, lasīšanas, atjaunošanas un izdzēšanas operācijas.
CSS	No angļu valodas (*Cascading Style Sheets*) – notācija, ko lieto, lai aprakstītu HTML dokumentu stilu.
CTP	No angļu valodas (*Community Technology Preview*) – Microsoft adoptēts nosaukums programmatūras beta versijai.
ES	No angļu valodas (*ECMAScript*) – specifikācija JavaScript programmēšanas valodai.

F#	.NET programmēšanas valoda, kas aptver funkcionālās, imperatīvās un objektorientētās programmēšanas paradigmas.
HTML	No angļu valodas (*Hypertext Markup Language*) – hiperteksta iezīmēšanas valoda, kas izstrādāta tīmekļa lappušu un citas pārlūkprogrammā attēlojamas informācijas aprakstīšanai.
HTTP	No angļu valodas (*Hypertext Transfer Protocol*) – lietojumslāņa protokols, kas paredzēts datu apmaiņai starp tīmekļa serveriem un pārlūkprogrammām.
JSON	No angļu valodas (*JavaScript Object Notation*) – datu apmaiņas formāts JavaScript objektu notācijā.
LINQ	No angļu valodas (*Language Integrated Query*) – .NET ietvara komponente, kas pievieno datu atlases funkcionalitāti dažādiem objektiem.
REST	No angļu valodas (*Representational state transfer*) – tīmekļa arhitektūras pieeja, kas iekļauj resursus un pārejas starp tiem.
RTM	No angļu valodas (*Release to Manufacturing*) – programmatūras dzīves cikla posms.
SDK	No angļu valodas (*Software development kit*) – izstrādes rīku kopums programmatūras izstrādei noteiktam ietvaram vai platformai.
URL	No angļu valodas (*Uniform Resource Locator*) – atsauce uz tīmekļa resursu, kas identificē resursa atrašanās vietu tīklā un mehānismu tā izgūšanai.
VB.NET	.NET programmēšanas valoda, oriģinālās Visual Basic programmēšanas valodas turpinājums.
WCF	No angļu valodas (*Windows Communication Foundation*) – ietvars servisu orientētu lietotņu izstrādei. NET platformā.
WPF	No angļu valodas (*Windows Presentation Foundation*) – komponente lietotāja saskarnes renderēšanai Windows lietojumprogrammās.
XML	No angļu valodas (*eXtensible Markup Language*) – iezīmēšanas valoda datu uzglabāšanai un transportēšanai.

IEVADS

HTML (*HyperText Markup Language*) [1] ir populāra iezīmēšanas valoda statisku dokumentu aprakstīšanai. HTML sākotnēji netika paredzēts dinamisku tīmekļa lietotņu izstrādei. Google nolēma risināt dinamiska tīmekļa satura attēlošanas problēmu un piedāvāja AngularJS – JavaScript ietvaru un rīkus dinamisku HTML, CSS un JavaScript tīmekļa lietotņu izstrādei (CSS un JavaScript ir tīmekļa lietotņu izstrādes tehnoloģijas [2, 3]). Izstrādātāju kopiena deva lielu ieguldījumu Angular projektam un izstrādāja Angular specifiskus risinājumus bieži sastopamu tīmekļa lietotņu izstrādes problēmu risināšanai – lietotāja saskarnes elementu attēlošana, datņu augšupielāde, lokalizācija, interaktīvas datu tabulas u.c. Dēļ spējas labi risināt aprakstītās problēmas, AngularJS iemantoja milzīgu popularitāti un atzinību tīmekļa lietotņu izstrādātāju vidū. Angular tehnoloģijas mūsdienās darbina vairākas populāras tīmekļa lietotnes.

Angular ir viens no apspriestākajiem tematiem tīmekļa lietotņu izstrādes kontekstā arī šodien. Par iemeslu tam ir jaunākā Angular versija (Angular 2) – pilnībā pārrakstīta ietvara pirmā versija ar uzsvaru uz modularitāti, veiktspēju un lietotņu izstrādi vairākām platformām uzreiz, izmantojot pazīstamas tīmekļa tehnoloģijas un rīkus. Angular 2 ir izstrādāts TypeScript programmēšanas valodā, kuras lielākās pievienotās vērtības izstrādes laikā ir statiska datu tipu pārbaude, un objektorientētas programmēšanas principu (klases, saskarnes, mantošana, moduļi) izmantošana JavaScript lietotņu izstrādei [4]. Tīmekļa tehnoloģiju ekosistēmā ir nostiprinājušās arī citas klienta puses lietotņu izstrādes platformas. Populāra ir Facebook piedāvātā platforma ReactJS, un sava tirgus daļa ir arī tādiem ietvariem kā Ember.js un Vue.js, tomēr Angular mūsdienās ir viena no primārajam izvēlēm jaunu tīmekļa lietotņu izstrādei [5].

ASP.NET ir tīmekļa tehnoloģiju platforma servera puses lietotņu izstrādei Microsoft .NET ekosistēmā. Vēsturiski .NET tehnoloģijas ir bijušas pieejamas tikai Windows platformas lietotājiem priekš mitināšanas Windows serveros. Ar .NET Core un ASP.NET Core atvērtā pirmkoda projektu parādīšanos, ir iespējams izstrādāt tīmekļa lietotnes arī MacOS un Linux platformām. Papildus tam, Microsoft piedāvā no platformas neatkarīgus rīkus ASP.NET risinājumu izstrādei. Šādi pavērsieni ļauj arvien lielākai izstrādātāju grupai izmantot ASP.NET tehnoloģijas savu tīmekļa lietotņu izstrādei, neatkarīgi no konkrētas platformas, kurā tiek veikta izstrāde vai kurā plānots izmitināt un darbināt risinājumu.

Arvien vairāk izstrādātāju savos projektos izvēlas izmantot tieši ASP.NET un Angular tehnoloģijas, lai veiktu jaunu tīmekļa projektu izstrādi. Servera un klienta puses risinājumi nav tieši saistīti, jo tie atbilst dažādām izpildes vidēm, tehnoloģijām un programmēšanas

valodām, taču abas lietotnes dublē noteiktus informācijas vienumus. Servera puses risinājumi definē saskarnes datu apmaiņai, datu modeļu klases, datu modeļu validācijas nosacījumus un citu klienta puses risinājumam būtisku informāciju. Atkarībā no konkrētā lietojuma, iespējams arī kāda daļa no biznesa loģikas tiek dublēta servera un klienta lietotnēs. Minētās informācijas duplikācija apjomīgos tīmekļa risinājumos negatīvi atsaucas uz izstrādātāju produktivitāti, jo izmaiņas programmatūras specifikācijā un biznesa pieņēmumos nozīmē manuālu informācijas sinhronizāciju klienta un servera lietotnēs.

Ar TypeScript parādīšanos un popularitātes pieaugumu (gan Angular kontekstā, gan vispārīgi), problēmas aktualitāte ir strauji pieaugusi. TypeScript aicina izstrādātājus definēt no servera saņemto datu modeļus un HTTP saskarnes. Izstrādātāji ir pamanījuši, ka C# un TypeScript balstās uz objektorientētās programmēšanas principiem un problēmas abās valodās tiek risinātas līdzīgi, un izveidojuši publiski pieejamus rīkus, kas spēj ģenerēt TypeScript klases no vienkāršiem C# datu modeļiem. Autors uzskata, ka tā ir tikai daļa no problēmas risinājuma, jo klienta lietotne nav atkarīga tikai no servera saskarnes datu modeļiem. Izmantojot modernus pirmkoda analīzes rīkus, kas spēj izgūt pirmkoda sintaktisko un semantisko informāciju, problēmu būtu iespējams risināt labāk, efektīvāk un ar atzīstamākiem rezultātiem.

Pēdējos gados .NET platformā ir attīstījusies .NET kompilatora platforma jeb Roslyn. Roslyn ir atvērtā pirmkoda .NET platformas kompilatora serviss, kura galvenais mērķis ir dot izstrādātājiem piekļuvi tai informācijai, kuru kompilators iegūst pirmkoda kompilācijas laikā un ātri aizmirst pēc tam, kad darbs ir paveikts. Roslyn piedāvā rīkus, kas ļauj lasīt, kompilēt, analizēt un ģenerēt .NET platformas pirmkodu.

Pētījuma mērķis ir apvienot Roslyn pirmkoda analīzes iespējas, TypeScript un Angular zināšanas, lai izveidotu risinājumu, kas spētu ģenerēt tās klienta puses lietotnes komponentes, kas ir atkarīgas no servera puses lietotnes. Mērķa sasniegšanai ir nepieciešams padziļināti izpētīt Roslyn darbības principus, TypeScript pirmkoda ģenerēšanas iespējas un pielietot tās darba problēmas risināšanai.

Lai sasniegtu aprakstītos mērķus, tika izvirzīti sekojoši uzdevumi:

1. definēt dublētos informācijas vienumus ASP.NET un Angular tīmekļa risinājumos;
2. izpētīt un analizēt esošos pirmkoda ģenerēšanas risinājumus – definēt kritērijus risinājumu klasifikācijai un veikt novērtējumu;
3. izpētīt Roslyn darbības principus un iespējas pirmkoda analīzei;
4. piedāvāt savu risinājumu, definēt atbalstīto funkcionalitāti un izmantošanas iespējas;

5. realizēt savu risinājumu un aprakstīt tā realizācijas pieeju, pielietojumu, sniegtās iespējas un ierobežojumus.

Darbā izmantotās pētniecības metodes iekļauj pieejamās literatūras un risinājumu analīzi, kā arī unikāla risinājuma izstrādi.

Pirmajā nodaļā lasītājs tiek iepazīstināts ar darbā izmantotajām tehnoloģijām (.NET, TypeScript, Angular) un konkrētiem konceptiem, kas ir būtiski maģistra darba ideju izpratnei. Otrajā nodaļā detalizētāk aprakstīta darbā aplūkotā problēma un autora motivācija to risināt, definēts risināmās problēmas tvērums. Trešajā nodaļā apskatīti .NET rīki pirmkoda analīzei un ģenerēšanai, apskatīti esošie risinājumi līdzīgu problēmu risināšanai un doti pamatojumi, kāpēc tie neder (vai neder pilnībā) darbā aplūkoto problēmu risināšanai. Ceturtā nodaļa veltīta Roslyn .NET kompilatora platformas detalizētākam apskatam. Piektajā nodaļā formulēti piedāvātā risinājuma atbalstītie lietojumi. Sestajā nodaļā aprakstīta autora pieeja risinājuma realizācijai. Darba septītajā nodaļā aprakstīti darba rezultāti, dots risinājuma novērtējums – salīdzinājums ar radniecīgiem risinājumiem, pieminēts risinājuma praktiskais pielietojums autora izstrādātajos projektos. Darba izskaņā aprakstīti izdarītie secinājumi, gūtās atziņas un ieskicēts turpmākais darbs risinājuma attīstīšanai.

1. DEFINĪCIJAS

Maģistra darbā tiek apskatīta un risināta problēma .NET, Angular un TypeScript tehnoloģiju kontekstā, tāpēc nodaļā dots virspusīgs izklāsts par minētajām tehnoloģijām, detalizētāk aprakstot tos faktorus, kas ir būtiski darba labākai izpratnei.

1.1. .NET

.NET ir tēma, kas caurvij būtisku darba apjomu. Darba galvenās problēmas un idejas saistītas ar .NET tīmekļa risinājumu pirmkoda analīzi, tāpēc ir svarīgi iepazīties ar tām .NET pamatidejām, kas nepieciešamas darbā aprakstīto problēmu un risinājumu izpratnei.

1.1.1. Vēsture

Microsoft uzsāka .NET ietvara izstrādi deviņdesmito gadu beigās zem nosaukuma *Next Generation Windows Services (NGWS).* 2000. gada nogalē tika izlaistas pirmās .NET 1.0 beta versijas. 2000. gada augustā Microsoft, Hewlett-Packard un Intel sāka darbu pie .NET CLI un C# standartiem, un 2001. gada decembrī abi tika apstiprināti kā ECMA standarti. ISO standartizācija sekoja 2003. gada aprīlī un tekošās ISO standarta versijas ir attiecīgi ISO-IEC 23271:2012 [6] un ISO/IEC 23270:2006 [7].

2007. gada 3. oktobrī, Microsoft paziņoja, ka .NET 3.5 pirmkods kļūs pieejams zem *Microsoft Reference Source License (Ms-RSL)* licences nosacījumiem. Licence paredz, ka pirmkods ir pieejams tikai atsauces nolūkiem un ir noderīgs programmatūras atkļūdošanas laikā. 2008. gada 16. janvārī pirmkoda repozitorijs kļuva pieejams un saturēja BCL, ASP.NET, ADO.NET, Windows Forms, WPF, un XML bibliotēkas. Tas bija pirmais solis pretī .NET platformas "atvēršanai" un atvērtā pirmkoda kultūras ieviešanai, kādu var novērot šodien.

2014. gada 12. novembrī Microsoft izziņoja .NET Core projektu, ar mērķi piedāvāt .NET ietvara funkcionalitāti (vai daļu no tās) lietotājiem ārpus Microsoft Windows platformas. .NET Core kļuva par atvērtā pirmkoda projektu, kas tika izvietots GitHub vietnē, kur tas turpināja savu attīstību.

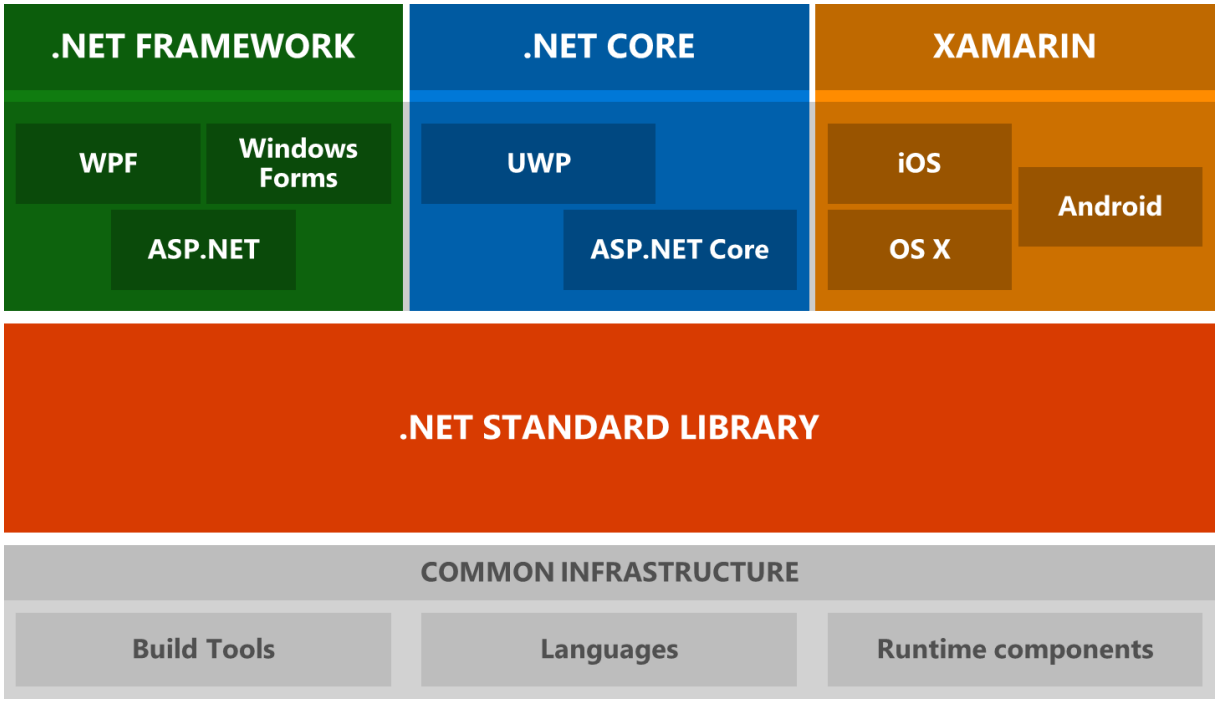
Microsoft paziņoja, ka turpmākie soļi saistībā ar .NET platformu tiks pilnībā saistīti ar atvērtā pirmkoda projektiem, modernu tehnoloģiju izstrādei. Pagaidām WPF un Windows Forms tehnoloģijas netiek plānots padarīt par atvērtā pirmkoda projektiem [8].

1.1.2. .NET komponentes

.NET ekosistēma mūsdienās sastāv no vairākām komponentēm. Būtiska .NET sastāvdaļa mūsdienās ir standarta bibliotēka (*.NET Standard Library*). Standarta bibliotēka

definē visu to API kopu, kas jārealizē katram .NET izpildlaikam (*runtime*). Pašlaik uzturētie izpildlaiki ir .NET pilnais ietvars, .NET Core un Mono. .NET programmēšanas valodas (pazīstamākās no tām ir C#, VB.NET, F#) spēj darboties katrā no minētajām izpildes vidēm. Papildus tam .NET ekosistēmā ir rīku kopa programmatūras izstrādei – rīki ir kopīgi visām platformām vai izpildlaikiem [9].

1.1. att. redzams visu iepriekš pieminēto komponenšu grafisks attēlojums.



1.1. att. **.NET platformas komponentes [9]**

Turpinājumā tiks īsi aprakstīta katra iepriekš minētā komponente.

1.1.3. .NET Standard

.NET Standard Library ir API kopa, kuru implementē jebkurš .NET izpildlaiks. Reizē ar vairāku .NET izpildlaiku parādīšanos, standarta bibliotēkas mērķis ir nodrošināt lielāku homogenitāti .NET ekosistēmā. Standarta bibliotēka definē mazāko kopīgo dalītāju visiem .NET izpildlaikiem, tādējādi izstrādātais pirmkods ir izmantojams vairākās platformās, bez nepieciešamības to mainīt vai speciāli pielāgot [10].

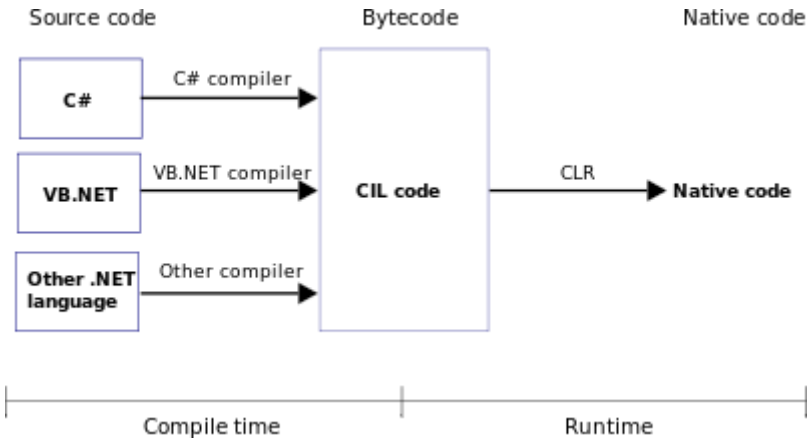
1.1.4. .NET izpildlaiki

Eksistē 3 primārie .NET izpildlaiki, kurus Microsoft aktīvi uztur un izstrādā – .NET pilnais ietvars, .NET Core un Mono. Detalizētāk apskatīti .NET pilnais ietvars un .NET Core. Mono ir .NET izpildlaiks Linux lietotājiem, kuru primāri uztur Microsoft iegādātais uzņēmums Xamarin, taču Mono maģistra darba kontekstā nav aktuāls [11].

1.1.4.1. .NET pilnais ietvars

.NET pilnais ietvars sastāv no CLR (*Common Language Runtime*) un .NET klašu bibliotēkas. CLR nodrošina .NET lietojumprogrammu pārvaldītu izpildi, bet klašu bibliotēka piedāvā testētu funcionalitāti, kuru izstrādātāji var izmantot savos risinājumos dažādu funkciju veikšanai. Tālāk aprakstīti galvenie ieguvumi no .NET izmantošanas [12].

- Atmiņas pārvaldība – viens no CLR uzdevumiem ir objektu dzīves cikla pārvaldība. Izstrādātājiem nav atbildīgi par lietojumprogrammai nepieciešamās atmiņas piešķiršanu un atbrīvošanu.
- Vienota tipu sistēma – tradicionālās programmēšanas valodās, bāzes tipus definē kompilators, bet .NET bāzes tipus definē .NET tipu sistēma, kas ir kopīga visām .NET programmēšanas valodām (C#, VB.NET, F#).
- Ekstensīva klašu bibliotēka – tā vietā, lai rakstītu milzīgu apjomu pirmkoda, vispārpieņemtu, zema līmeņa programmēšanas operāciju veikšanai, izstrādātāji var izmantot .NET klašu bibliotēkas piedāvātās klases un to metodes.
- Izstrādes ietvari un tehnoloģijas – .NET ietvars satur bibliotēkas dažādu specifisku lietotņu izstrādes vajadzībām kā piemēram ASP.NET priekš tīmekļa lietotņu izstrādes, ADO.NET priekš komunikācijas ar datu bāzēm, vai WCF priekš tīmekļa servisu bāzētām lietotnēm.
- Programmēšanas valodu savietojamība – .NET programmēšanas valodu kompilatori izdod CIL pirmkodu, ko tālāk CLR izpildes laikā kompilē uz konkrētās arhitektūras specifisko pirmkodu. Tādējādi programmatūra, kas uzrakstīta vienā .NET valodā var tikt izmantota arī citās .NET valodās. .NET programmēšanas valodu kompilācijas process redzams 1.2. att.



1.2. att. **CIL kompilācija [12]**

- Versiju savietojamība – ar retiem izņēmumiem, programmatūra, kas izstrādāta, izmantojot konkrētu .NET ietvara versiju var darboties bez modifikācijām ar jaunāku .NET ietvara versiju.
- Vairākas .NET versijas var eksistēt uz vienas operētājsistēmas, tādējādi dažādas lietotnes var izpildīties tām paredzētajās .NET ietvara versijās.
- Izstrādātāji var izstrādāt lietotnes, kas var izpildīties dažādās .NET platformās, piemēram, dažādas Windows darbstaciju versijas, Windows Phone vai Xbox sistēmās.

1.1.4.2. .NET Core

.NET Core ir izstrādes platforma, kas atbalsta Windows, Linux un MacOS operētājsistēmas. .Net Core var darboties gan uz iekārtām, gan mākoņdatņošanas risinājumos vai iegultās programmatūras scenārijos.

.NET Core vislabāk apraksta tālāk aprakstītās īpašības.

- Ērta piegāde – var tikt piegādāts lietotājam kopā ar lietotni vai arī tikt instalēts uz darbstacijas kopā ar citiem .NET izpildlaikiem vai citām .NET Core versijām.
- No platformas neatkarīgs – atbalsta Windows, Linux un MacOS operētājsistēmas. Atbalstīto platformu, CPU un lietojumu scenāriju skaitu plānots palielināt.
- Komandrindas rīku atbalsts – visus scenārijus var izpildīt, izmantojot komandrindas saskarni.
- Savietojamība – savietojams ar citiem .NET izpildlaikiem, jo implementē .NET standarta bibliotēku.
- Atvērtā pirmkoda projekts.
- Microsoft atbalsts.

Līdzīgi kā .NET pilnais ietvars, .NET Core sastāv no izpildlaika, klašu bibliotēkas, SDK rīkiem un valodu kompilatoriem. .NET Core var uzskatīt par .NET pilnā ietvara no platformas neatkarīgo versiju. Tas ir kompromiss starp bagātīgu API un platformas neatkarību – Windows specifiski scenāriji un klases nav pilnībā implementētas vai implementētas daļēji.

Microsoft piedāvā pietiekoši daudz informācijas, lai palīdzētu izstrādātājiem salīdzināt abus iepriekšminētos .NET ietvarus un izdarītu izvēli par labu vienam no tiem kāda konkrēta projekta izstrādei. Maģistra darba ietvaros ir pietiekoši saprast, ka .NET Core ir radīts no platformas neatkarīgiem risinājumiem, kam nepieciešama augsta veiktspēja un mērogojamība, kamēr .NET pilnais ietvars realizē lielāku tehnoloģiju un iespēju kopu Windows platformai [13].

1.1.5. Rīki

Visām .NET implementācijām ir kopīgi dažādi rīki, kas izmantojami risinājumu izstrādē dažādās platformās [9].

- .NET programmēšanas valodas un to kompilatori (C#, VB, F#).
- Izpildes laika komponentes, piemēram, JIT (*Just-In-Time*) kompilators un GC (*Garbage Collector*).
- .NET projektu sistēma (*csproj, vbproj, fsproj*).
- MSBuild – programmatūra kompilācijas veikšanai.
- NuGet – Microsoft rīks programmatūras pakotņu pārvaldībai .NET projektos.
- .NET CLI – no platformas neatkarīga komandrindas saskarne .NET projektu izstrādei.
- Visual Studio un Visual Studio Code integrētās izstrādes vides.

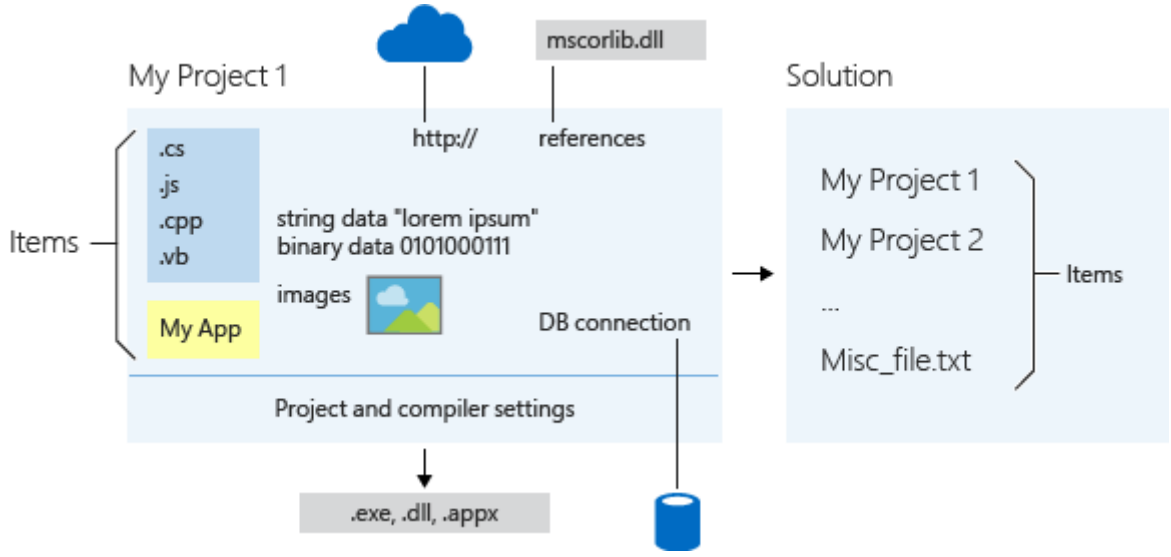
1.1.6. Projektu sistēma

Jebkāda veida programmatūras izstrāde Visual Studio izstrādes vidē vienmēr sākas ar jauna projekta izveidi. Projekts satur visas pirmkoda datnes, attēlus, ikonas un datus, kas nepieciešami, lai kompilētu izpildāmu programmatūru vai tīmekļa vietni. Projekts satur arī visus kompilatora uzstādījumus un citas konfigurācijas datnes, kas var būt nepieciešamas, lai lietotne varētu komunicēt ar citām risinājuma komponentēm vai servisiem.

Projektu definē XML datne ar paplašinājumu, kas atkarīgs no projektā izmantotās programmēšanas valodas (*.vbproj, .csproj, .vcxproj*). Projekta XML datne satur projekta kompilācijas uzstādījumus un projekta virtuālo datņu hierarhiju kopā ar ceļiem uz katru projektā iekļauto datni uz cietā diska. Visual Studio izmanto projekta datni, lai vizuāli attēlotu projekta saturu un uzstādījumus. MSBuild izmanto projekta datni, lai veiktu projekta kompilāciju un izveidotu izpildāmu lietotni.

Katrs projekts loģiskā izpratnē un arī fiziski datņu sistēmā ir iekļauts risinājumā (solution). Risinājums var saturēt vienu vai vairākus projektus un, veidojot jaunu projektu, vienmēr tiek automātiski izveidots ar noklusētiem uzstādījumiem. Risinājuma primārais mērķis ir uzturēt visu projektā iekļauto projektu kompilācijas uzstādījumus, jo projekti var būt atkarīgi viens no otra un, kompilējot visu risinājumu, ir svarīga secība, kādā tiek kompilēti tajā iekļautie projekti. Risinājums satur arī Visual Studio uzstādījumus un citas datnes, kas nav asociētas ar konkrētu projektu, piemēram, dokumentācija vai uzstādīšanas instrukcijas. Risinājuma informācija tiek glabāta teksta datnē ar paplašinājumu *.sln*. Datnes tekstuālā informācija tiek glabāta ar tai specifisku formātu un vispārīgā gadījumā datni nav paredzēts

manuāli koriģēt. Katram lietotājam, kas atver risinājumu Visual Studio, tiek izveidota risinājuma uzstādījumu datne ar paplašinājumu *.suo*, kas satur katra izstrādātāja specifiskos uzstādījumus un konfigurāciju. Uzstādījumu datni parasti neievieto pirmkoda repozitorijā, jo tā nav daļa no risinājuma, un ir specifiska katram izstrādātājam, kas strādā pie risinājuma [14]. 1.3. att. redzama projektu un risinājumu savstarpējā attiecības un vienumi, kurus tie loģiski satur.



1.3. att. **.NET projektu struktūra [14]**

Maģistra darbā .NET projektu struktūras izpratne ir nepieciešama .NET risinājumu analīzes kontekstā, kad tiks runāt par to, kā izstrādātāji strukturē savas datnes.

1.1.7. ASP.NET un ASP.NET Core

ASP.NET ir primāra platforma servera puses tīmekļa risinājumu izstrādei ar Microsoft tehnoloģijām priekš Windows. Kopš 2016. gada vasaras Microsoft piedāvā arī atvērtā pirmkoda platformu ASP.NET Core, kuras galvenie mērķi ir vairāku platformu atbalsts, augsta veiktspēja un modularitāte.

ASP.NET Core ir tiek uzturēts paralēli ASP.NET 4.6 un nav tā turpinājums, bet gan atsevišķs izstrādes ietvars – mazāks un modulārāks. ASP.NET Core ļauj izstrādātājiem izvēlēties nepieciešamās komponentes un iekļaut tās savā tīmekļa lietotnē. Papildus tam iespējams izmantot .NET Core un darbināt lietotnes arī Linux un Macintosh operētājsistēmās. [15]. Maģistra darbā tiks analizēti ASP.NET servera puses risinājumi.

1.1.8. C#

C# ir viena no .NET programmēšanas valodām ar saknēm C programmēšanas valodu ģimenē, kas to padara viegli apgūstamu izstrādātājiem ar C, C++, Java un JavaScript zināšanām [16].

Viena no darbā vairākkārt pieminētām C# īpašībām, kas nav plaši sastopama citās objektorientētās valodās, ir atribūti. Klases, lauki un citi objekti C# lietotnē atbalsta dažādus modifikatorus, kas kontrolē noteiktus to darbības aspektus. Piemēram, klases un to metožu pieejamību kontrolē `public`, `protected`, `internal`, un `private` modifikatori. C# paplašina šo funkcionalitāti, ļaujot izstrādātājiem definēt savus tipus ar deklaratīvu informāciju, asociēt tos ar C# objektiem un izgūt to informāciju lietotnes izpildes laikā. C# lietotnes definē šādu deklaratīvu informāciju, izmantojot konstrukciju, ko sauc par **atribūtu**.

1.4. att. piemērs deklarē `HttpGetAttribute` atribūtu, kas var tikt asociēts ar ASP.NET kontrolieru metodēm, lai instruētu ASP.NET maršrutēšanas mehānismu par atbilstoši HTTP metodi un URL fragmentu.

```
public class HttpGetAttribute : HttpMethodAttribute
{
    public HttpGetAttribute(): base(_supportedMethods) {}

    public HttpGetAttribute(string template): base(_supportedMethods, template)
    {
        if (template == null)
        {
            throw new ArgumentNullException(nameof(template));
        }
    }
}
```

1.4. att. **C# atribūta deklarācija**

Atribūtus var izmantot, atsaucoties uz to nosaukumu un norādot nepieciešamos parametrus kvadrātiekavās, bet izlaižot sufiksu *Attribute*, ja tāds ir norādīts. 1.5. att. redzama C# metode, kas atgriež sarakstu ar studentiem uz HTTP GET ‘api/students’ pieprasījumu.

```
[HttpGet(“api/students”)]
public IEnumerable<Student> Get() {}
```

1.5. att. **C# atribūta izmantošana**

Atribūti var realizēt arī kompleksu loģiku, piemēram, izpildīt kādu funkciju katru reizi, kad noteikta metode tiek izsaukta, tādējādi implementējot notikumu reģistrēšanu vai pārbaudot piekļuves tiesības lietotājam. .NET ietvars realizē lielu apjomu dažādu atribūtu,

taču darba kontekstā būtiski ir saprast atribūtu būtību un primāro pielietojumu, tāpēc detalizētāk tiks apskatīti tikai interesējošie atribūti nodaļās, kas apraksta darba problēmas risinājumu [17].

1.2. ECMAScript

Pirms tiek runāts par TypeScript, īsi apskatīta programmēšanas valoda, kas ir katras TypeScript lietotnes kompilācijas rezultāts un kas galu galā izpildās lietotāja tīmekļa pārlūkā, proti JavaScript un tā standarts – ECMAScript.

JavaScript ir standartizēta programmēšanas valoda un tās specifikāciju apraksta ECMAScript jeb ES standarts. ECMAScript specifikāciju nosaka ECMA-262 [18] un ISO/IEC 16262 [19] standartos, un par to atbild *Ecma International*. JavaScript ir populārākā ECMAScript implementācija, taču nebūt ne vienīgā – JScript un ActionScript ir dažas no mazāk izplatītām (un nu jau vairs neatbalstītām) ECMAScript implementācijām.

Kopš 1997. gada kopā ir publicēti septiņi ECMAScript specifikācijas izdevumi, no kuriem pēdējais tika pabeigts 2016. gada jūnijā. ECMAScript 5. izdevums (publicēts 2009. gada decembrī) ir pilnībā realizēts visos modernajos tīmekļa pārlūkos un platformās, savukārt 6. izdevums (publicēts 2015. gada jūnijā) ir implementēts tikai jaunākajās darbvirsmas pārlūku versijās (tipiski – Google Chrome un Mozilla Firefox). ECMAScript 6 pievieno vairākas, ļoti noderīgas iespējas kompleksu lietotņu izstrādei kā piemēram klases, moduļus, jaunus kolekciju datu tipus u.c., bet definē tās ES5 semantikas ietvaros. Lai izmantotu ES6 funkcionalitāti tīmekļa pārlūkos, kas attiecīgo specifikāciju vēl nav realizējuši, var izmantot tādus rīkus kā Babel, Closure vai Traceur, kas transformē ES6 saderīgu JavaScript pirmkodu ES5 saderīgā pirmkodā. Tādējādi izstrādātāji var izmantot jaunākās JavaScript valodas iespējas, atvieglojot izstrādi un uzlabojot savu produktivitāti, bet tai pašā laikā neuztraucoties par lietotājiem, kas izmanto tīmekļa pārlūku bez ES6 atbalsta.

ECMAScript 5 bija aktuālais standarts gandrīz sešu gadus, bet pēc tam tika publicēti divi standarti divu gadu laikā, tāpēc runājot par JavaScript, arvien biežāk tiek lietoti termini ECMAScript 5 un ECMAScript 6 (vai to saīsinājumi ES5 un ES6), lai apzīmētu konkrētu specifikācijas versiju.

1.3. TypeScript

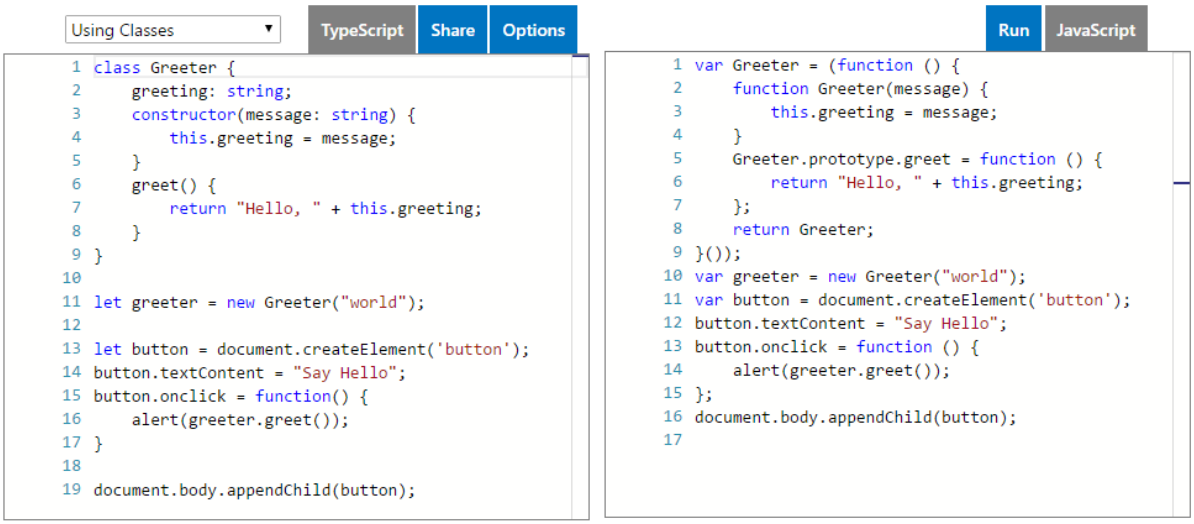
JavaScript kā programmēšana valoda sākotnēji tika paredzēta vienkāršu, nelielu tīmekļa lietotņu izstrādei. JavaScript (vismaz līdz ECMAScript 6. versijai) nav statiska datu tipu sistēma un tādas objektorientētās programmēšanas konstrukcijas kā klases, saskarnes un

moduļi, kas ievērojami atvieglo programmatūras diagnostiku un lietotnes pirmkoda strukturēšanu. Minētās konstrukcijas ir ļoti noderīgas apjomīgu klienta puses risinājumu izstrādei un uzturēšanai.

TypeScript ir atvērtā pirmkoda programmēšanas valoda no Microsoft, kas risina iepriekšminētās problēmas, paplašinot JavaScript ar statiskiem datu tipiem un objektorientētās programmēšanas paradigmām – moduļiem, klasēm, saskarnēm, mantošanu, u.c.

TypeScript pirmkods vienmēr tiek kompilēts uz JavaScript un nogādāts klienta tīmekļa pārlūkam, līdz ar to ieguvumus no TypeScript spēj izjust tikai izstrādātāji lietotnes izstrādes, kompilācijas un atkļūdošanas laikā. Jebkura JavaScript lietotne ir korekta TypeScript lietotne – lai sāktu izmantot TypeScript iespējas, nepieciešams vienkārši nomainīt datnes paplašinājumu no *.js* uz *.ts* (tiesa tad arī nepieciešams papildus kompilācijas solis, kas veiks TypeScript pirmkoda kompilāciju) [20].

1.6. att. redzams TypeScript pirmkoda piemērs ar kompilācijas rezultātā iegūto JavaScript, kas tam atbilst.



1.6. att. **TypeScript kompilācijas rezultāts [20]**

Tā kā būtiska praktiskā darba daļa ir TypeScript pirmkoda ģenerēšana, ir apskatītas fundamentālās TypeScript konstrukcijas un koncepti. Tālākajos punktos aprakstītas un, ar piemēriem, parādītas TypeScript labākās īpašības un ar tām saistītie ieguvumi.

1.3.1. Statiski datu tipi

TypeScript lielākais ieguvums ir statisko datu tipu realizācija un to pārbaude kompilācijas laikā. JavaScript ir dinamiska tipu sistēma – funkcija `function f(x) { return x * 2; }` pieņems jebkura tipa argumentus, bet neatgriezīs saturīgu rezultātu. Reizēm dinamisku tipu uzvedība palīdz sasniegt vēlamo rezultātu vieglāk, bet bieži vien tā rada grūti pamanāmas kļūdas – it īpaši apjomīgākos projektos vai projektos ar problemātisku pirmkoda

bāzi. TypeScript šo problēmu risina ar datu tipu anotācijām, proti iepriekš minētā funkcijas deklarācija pārtop par `function f(x: number): number { return x * 2; }`. Tas instruē TypeScript kompilatoru, ka funkcijas argumenta un atgriežamās vērtības datu tips ir *number* un tādi funkcijas izsaukumi kā *f("Hello, world")* vai *f([1,2,3])* rezultēsies kompilācijās laika kļūdās. TypeScript var izveidot nepieciešamās JavaScript datnes, pat ja kompilācija atgriež kļūdas, kas tādā gadījumā kalpo par brīdinājumu, ka lietotne visticamāk nestrādās kā paredzēts [21].

1.3.2. Klases un mantošana

Funkcijas ir JavaScript fundamentālie izstrādes bloki. Funkcijas palīdz realizēt lietotnē nepieciešamās abstrakcijas, atdarina klases, paslēpj informāciju un implementē moduļus. Izstrādātājiem ar C# vai Java pieredzi, kas vēlas apgūt JavaScript, bieži vien ir problēmas adaptēties un rakstīt idiomātisku JavaScript pirmkodu. TypeScript ļauj izstrādātājiem izmantot tradicionālu objektorientētu pieeju, kur klases iekapsulē funkcionalitāti un objekti tiek veidoti no tām. Sākot ar ECMAScript 2015 (ES6) specifikāciju, JavaScript atbalsta klases, bet TypeScript ļauj tās izmantot vēl pirms populārākie tīmekļa pārlūki realizē jaunāko specifikāciju – kompilējot pirmkodu uz ES5 saderīgu JavaScript, bet saglabājot lietotnes semantiku [22].

1.7. att. redzama TypeScript klašu deklarācijas. Sintakse ir ļoti līdzīga C# un Java klašu deklarācijas sintaksei, tādējādi palīdzot izstrādātājiem vieglāk pielāgoties jaunai valodai.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0): void {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters: number = 5): void {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
```

1.7. att. **TypeScript klases**

Arī citi klašu koncepti ir pārņemti no C# un radniecīgām valodām – abstraktas klases, ģenēriskas klases, mantošana u.c.

1.3.3. Moduļu eksports un imports

Sākot ar ECMAScript 2015, JavaScript eksistē tāds koncepts kā modulis un TypeScript implementē analoģisku moduļu konceptu. Moduļi izpildās savā kontekstā, neatkarīgi no globālā lietotnes konteksta – visi mainīgie, klases un funkcijas, kas deklarēti iekš moduļa, pēc noklusējuma nav pieejami ārpus šī moduļa. Attēlos redzami piemēri klases deklarācijai, kas tiek eksportēta un pēc tam importēta citā TypeScript datnē [23].

Moduļu eksporta deklarācija redzama 1.8. att.

```
export const numberRegexp: RegExp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string): boolean {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

1.8. att. **TypeScript moduļu eksports**

Imports, norādot relatīvu ceļu uz importējamo datni, attēlots 1.9. att.

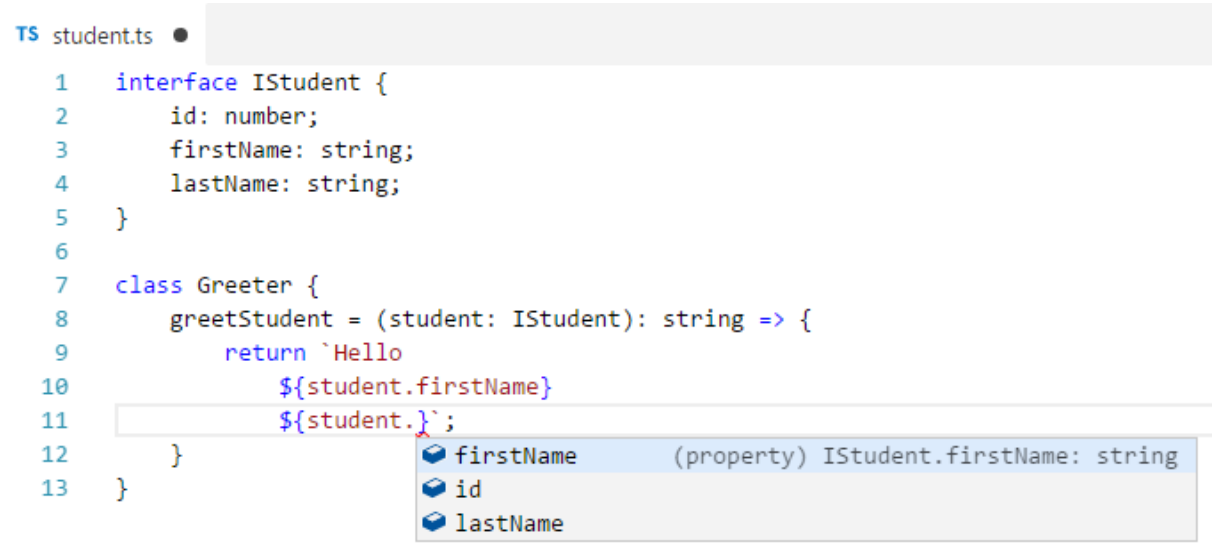
```
import { ZipCodeValidator } from "./ZipCodeValidator";
let myValidator: ZipCodeValidator = new ZipCodeValidator();
```

1.9. att. **TypeScript moduļa imports**

Modulis ir jebkura konstrukcija (klase, saskarne, funkcija, objekts), kas tiek eksportēta no TypeScript datnes un, lai to izmantotu ārpus attiecīgās datnes, moduli nepieciešams importēt, norādot relatīvu ceļu līdz datnei, kurā tas definēts. Viena datne var saturēt vairākus eksportējamus moduļus.

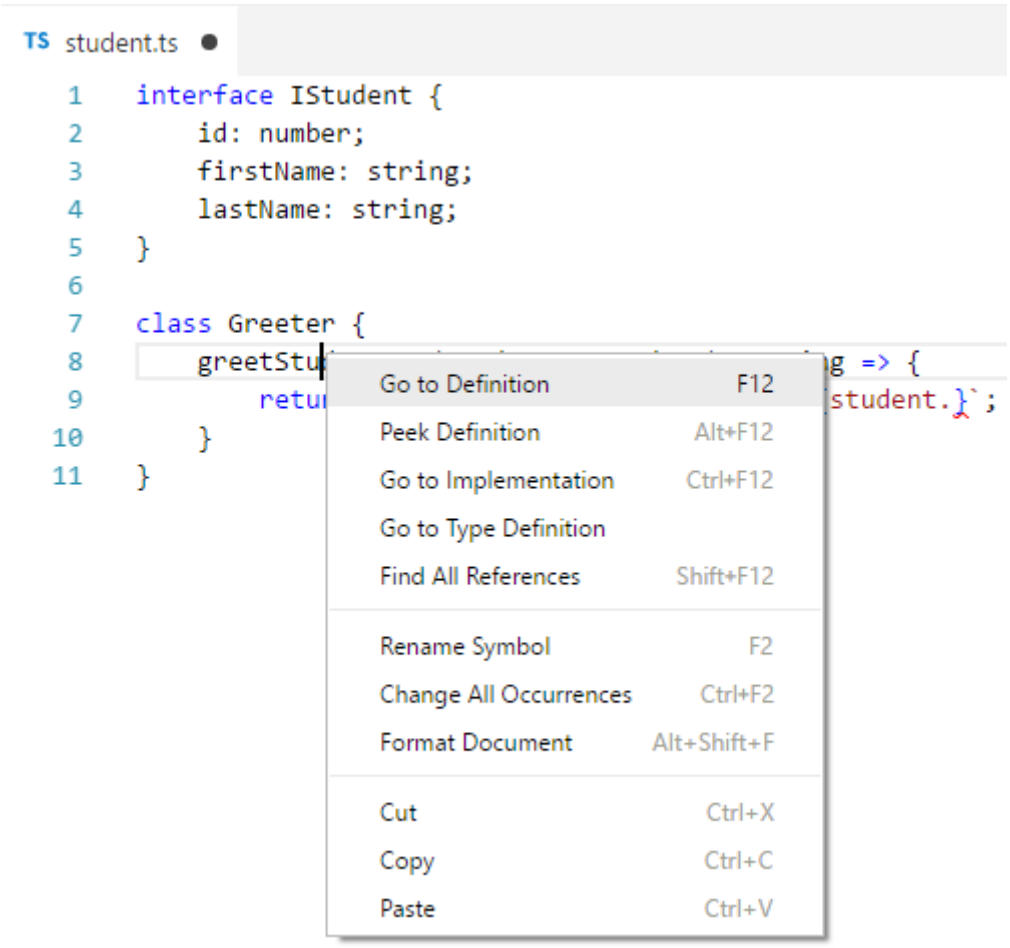
1.3.4. Izstrādes vides atbalsts

Kā izstrādes laika tehnoloģija, TypeScript var lepoties ar to rīku kopumu, kas ir pieejams TypeScript izstrādātājiem. Populāras izstrādes vides nodrošina teicamu rīku komplektu un iespējas TypeScript lietotņu izstrādei. Visual Studio un Visual Studio Code TypeScript iespējas ir iebūvētas, kamēr tādās izstrādes vidēs kā WebStorm, Eclipse, NetBeans un Atom paļaujas uz trešās puses spraudņiem. Kontekstuāla pirmkoda izstrādes palīdzība, pirmkoda analīzes, navigācijas un uzlabošanas iespējas mūsdienās ir kritiski faktori izstrādātāju produktivitātes nodrošināšanai. Izstrādātāji aizvien vairāk un vairāk paļaujas uz minētajām iespējām pirmkoda izstrādei. Tīmekļa lietotņu izstrādei klienta pusē tādas iespējas pāris gadus atpakaļ nebija pieejamas vispār. 1.10. att. redzamas TypeScript iespējas Visual Studio Code izstrādes vidē.



1.10. att. **TypeScript izstrādes palīdzība Visual Studio Code izstrādes vidē**

1.11. att. redzamas iespējas TypeScript analīzei Visual Studio Code izstrādes vidē.



1.11. att. **TypeScript pirmkoda analīzes iespējas Visual Studio Code izstrādes vidē**

Izstrādes vides atbalsts šādā līmenī ir iespējams dēļ Microsoft lēmuma padarīt TypeScript kompilatoru un valodas servisus pieejamus visiem. Vairāk par TypeScript kompilatora API un tā pielietojumu maģistra darbā tiks apspriests turpmākajās nodaļās.

1.3.5. Alternatīvas

TypeScript ir dažas alternatīvas, kas vienā vai otrā veidā veic līdzīgu uzdevumu – viena no tām ir CoffeeScript. CoffeeScript ir vienkārša programmēšanas valoda, kas kompilējas uz JavaScript un piedāvā alternatīvu sintaksi JavaScript konstrukcijām. CoffeeScript nav statiskas tipu sistēmas un, kopš tīmeklī parādījās risinājumi jaunākās ES sintakses kompilācijai uz vecāku (tīmekļa pārlūku atbalstītu), CoffeeScript strauji zaudēja popularitāti, jo ES6 klases, moduļi u.c. ir neatsverams ieguvums.

Google Dart ir vēl viena TypeScript alternatīva. Dart izmanto C# stila sintaksi un to iespējams kompilēt uz JavaScript. Dart sākotnēji tika paredzēts izpildei virtuālajā mašīnā uz servera un sākotnējie JavaScript kompilatori neieguva lielu popularitāti. Vēlāk Google piedāvāja savu JavaScript kompilatoru, taču TypeScript jau bija nostabilizējies kā primārais risinājums, jo piedāvā plašāku rīku atbalstu [24].

1.4. Angular

Angular ir ietvars vienas lappuses klienta puses lietotņu izstrādei JavaScript programmēšanas valodā. Sākotnējā ietvara versija savu debiju piedzīvoja 2010. gada 20. oktobrī. Maģistra darbs satur tādu jēdzienu kā "Angular specifisks TypeScript pirmkods". Tas nav "īpašs" TypeScript pirmkods. Angular specifisks TypeScript pirmkods implicē to, ka tam ir atkarības no Angular specifiskiem moduļiem, tas izmanto Angular specifiskus dekoratorus (līdzīgi C# atribūtiem) un tas tiek izstrādāts un strukturēts atbilstoši Angular vadlīnijām un labajām praksēm. Respektīvi, Angular specifisku pirmkodu bez modifikācijām iespējams izmantot tikai Angular risinājumos. Sadaļā virspusīgi aprakstīta Angular ekosistēma.

1.4.1. Angular 2+

Angular 2 ir AngularJS ietvara jaunākā versija. Salīdzinājumā ar iepriekšējām versijām, Angular 2 ir solis jaunā virzienā – ietvars ir pilnībā pārrakstīts, paturot labākās īpašības un gūtās mācības iepriekšējo versiju izstrādē. Angular 2 ir vislabāk piemērots jaunu projektu izstrādei, jo pirmkoda migrācija no iepriekšējām versijām var izrādīties gana problemātiska, lai gan ir pieejami Google izstrādāti rīki lietotņu migrācijai no AngularJS. Angular 2 ir jauni koncepti, sintakse, metodoloģijas un viedokļi.

Jaunā ietvara uzturēšanai, Angular komanda izmanto semantisko versiju sistēmu [25]. Tas nozīmē, ka ietvars izmanto x.y.z formāta versiju numurus, kur attiecīgie numuri tiek palielināti sekojošos gadījumos:

- x – tiek mainīts programmatūras API (nav saderībās ar iepriekšējo versiju);

- y – tiek pievienota funkcionalitāte, kas ir saderīga ar iepriekšējo versiju;
- z – tiek labotas programmatūras kļūdas, kas ir saderīga ar iepriekšējo versiju;

Tātad pēc Angular 2.0.0 var iznākt Angular 3.0.0 un tas nebūt nenozīmētu fundamentālas ietvara izmaiņas, kā tas notika Angular 1.x.x gadījumā. Minēto iemeslu dēļ Google aicina izstrādātājus ietvaru dēvēt vienkārši par "Angular" un tikai nepieciešamības gadījumā izmantot versijas numuru, lai norādītu konkrētu ietvara versiju. Ir vērts pieminēt, ka Angular 2017. gada 24. martā nonāca līdz 4. versijai, pilnībā izlaižot 3.0.0. Šāds solis veikts, lai izveidotu homogenitāti Angular komponentu vidū (Angular maršrutēšanas komponente atradās vienu versijas numuru priekšā visām pārējām komponentēm). Līdzīgs princips pieņemts arī maģistra darbā – vārds "Angular" vienmēr attiecināms uz ietvara pēdējo versiju, kamēr specifiskas versijas, kur nepieciešams, tiek apzīmētas ar versijas numuru, piemēram, Angular 2.0 vai Angular 4.1.0.

1.4.2. Programmēšanas valodas

Angular komanda sākotnēji paziņoja, ka Angular jaunās versijas izstrādei izmantos AtScript – programmēšanas valodu, kas paplašina TypeScript ar papildus iespējām. Neilgu laiku pēc tam, sekojot ciešākai Google sadarbībai ar Microsoft, nepieciešamā funkcionalitāte tika implementēta TypeScript valodā un TypeScript kļuva par galveno programmēšanas valodu Angular izstrādei. Lai gan Angular ir izstrādāts TypeScript programmēšanas valodā, Angular lietotņu izstrādātājiem nav obligāti jāizmanto TypeScript. Angular lietotnes var izstrādāt arī JavaScript programmēšanas valodā, konkrēti izmantojot ES5, ES6 vai Babel. Izstrādātāji, kas ir pazīstami ar Google Dart var izmantot arī to. Jāpiemin, ka nospiedoši lielākais apjoms resursu (ieskaitot oficiālo dokumentāciju) par Angular izstrādi satur TypeScript piemērus, tāpēc lēmumam neizmantot TypeScript jābūt vairāk kā pamatotam.

1.4.3. Izstrādes rīki

Angular lietotņu izstrādē nepieciešams izmantot dažādus rīkus, kas palīdz pārvaldīt projekta atkarības, sagatavo un iepako lietotni pēc iespējas mazākā izmērā un dod iespēju automatizēt bieži izpildāmus uzdevumus. Izstrādes rīkus var iekļaut vairākās kategorijās, kuras apskatītas zemāk. Problēmas, ko risina minētie rīki, nav ļoti būtiskas maģistra darba ietvaros, taču ir svarīgi zināt, ka tādi ir nepieciešami Angular lietotņu izstrādei, un kādu uzdevumu tie veic izstrādes procesā.

1.4.3.1. Pakotņu pārvaldība

Teju katrs projekts izmanto kādu trešās puses bibliotēku, jebkurš nopietns projekts izmanto daudz šādu bibliotēku – *Bootstrap, jQuery, moment* un daudzas citas. Bibliotēkas

atrodas dažādos repozitorijos, katram no tiem ir vairākas versijas, katrai versijai ir atkarības no citām pakotnēm. Pakotnēm ir dažāds saturs, dažādi arhivēšanas formāti. Šīs problēmas risina pakotņu pārvaldnieki, tādējādi atvieglojot projekta pakotņu instalāciju un atjaunošanu. Populārākie pakotņu pārvaldnieki – *NPM, Bower, Yarn, jspm*.

1.4.3.2. Moduļu ielādēšana un komplektēšana

Jebkura mēroga projektos izstrādātāji savu JavaScript pirmkoda bāzi sadala pa vairākām fiziskām datnēm, lai nodrošinātu lielāku modularitāti. Katru no individuālajām datnēm iespējams iekļaut HTML lapā ar <*script*> birku, taču tādā veidā katra skripta nogāde klientam prasīs individuālu HTTP savienojumu un mazām datnēm savienojuma izveidi patērē vairāk laika nekā pati datu pārsūtīšana klientam. Lejupielādes laika problēmu parasti atrisina savienojot atsevišķo datņu saturu vienā vai vairākās datnēs un piegādājot klientam vienu vai dažas JavaScript datnes. Sapakojot visu pirmkodu dažās datnēs, tiek zaudēts elastīgums – var būt problēmas ar lietotnes savstarpējo moduļu atkarībām. Datori moduļu atkarību problēmu var risināt daudz efektīvāk un ir pieejami vairāki rīki, kas to dara. Populārākie piemēri – *RequireJS, Browserify, Webpack, SystemJS*.

1.4.3.3. Uzdevumu automatizācija

Lai sagatavotu un iepakotu Angular lietotni efektīvam darbam tīmekļa pārlūkā, nepieciešams veikt virkni darbību – TypeScript kompilācija uz JavaScript, JavaScript un CSS datņu saspiešana un komplektēšana. Maksimālai efektivitātei lietotnes izstrādes laikā, to nepieciešams darīt katru reizi, kad mainās kāda no pirmkoda datnēm, lai pēc tam varētu automātiski pārlādēt tīmekļa pārlūku un darbināt lietotni. Tas ir tipisks klienta puses lietotnes izstrādes scenārijs, jo tīmekļa lietotnes izstrāde ir cikls starp pirmkoda rakstīšanu, kompilāciju un tīmekļa lietotnes atkārtotu palaišanu pārlūkā. Minēto uzdevumu automatizāciju piedāvā virkne rīku, no kuriem populārākie ir *Grunt, Gulp* un *Webpack*.

1.4.3.4. Projekta ģeneratori

Jauna tīmekļa projekta izveide konkrētam ietvaram parasti aizņem būtisku laika daudzumu – izstrādes rīku ir daudz, nepieciešamo datņu daudzums ir augsts un visu vēl nepieciešams konfigurēt. Projekta ģeneratori ļauj izveidot projekta sākotnējo struktūru ar nepieciešamajām mapēm, datnēm un konfigurāciju. Populāri piemēri – *Slush* un *Yeaoman.*

1.4.3.5. Angular CLI

Angular komanda, reaģējot uz izstrādes rīku lielo piesātinājumu tirgū, ir nākuši pretī Angular kopienai un izveidojuši komandrindas saskarni, kas veic visas iepriekšminētās funkcijas (vai vismaz tās, kuras ir nepieciešamas lielākai daļai izstrādes scenāriju). Rīks ir

bāzēts uz *Webpack* rīku un būtiski atvieglo izstrādes procesu, ļaujot izstrādātājiem koncentrēties uz biznesa loģikas implementāciju, nevis lietotnes konfigurāciju [26].

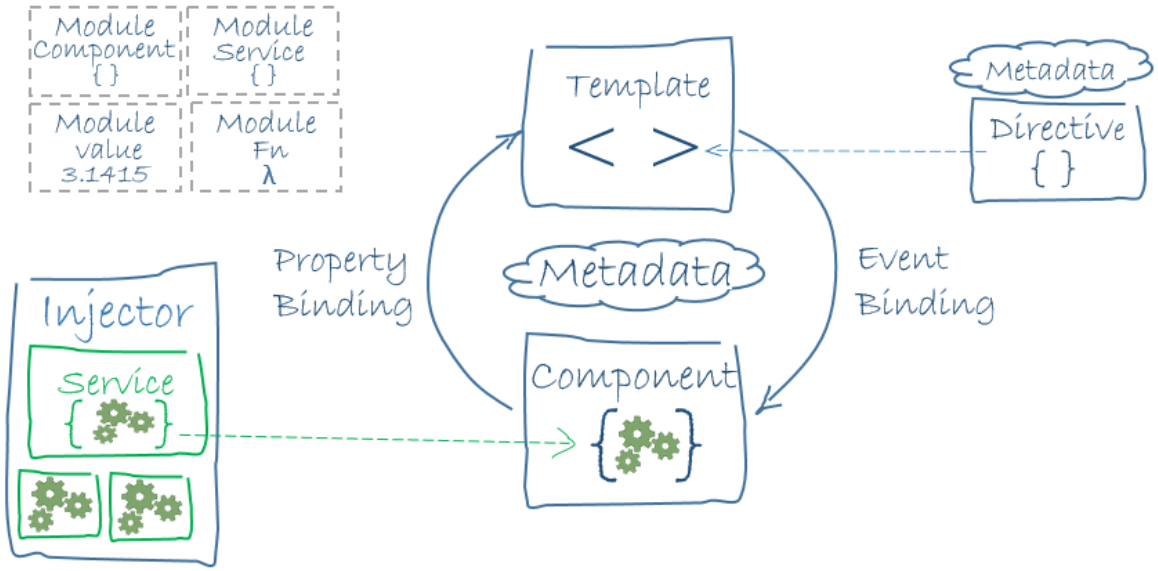
1.4.4. Arhitektūra

Angular lietotnes pamata uzbūves elementi ir komponentes. Komponente sevī iekļauj grupu ar savstarpēji saistītiem resursiem – JavaScript (loģika), CSS (stils) un HTML (struktūra). Komponentes ir savstarpēji izolētas, katrai komponentes instancei ir savs stāvoklis un stingri noteikti ievades un izvades dati. 1.12. att. redzama Angular komponentes deklarācija.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  title = 'app works!';
}
```

1.12. att. **Angular komponentes deklarācija**

Komponentes izmanto, lai iekapsulētu kādu lietotnes sastāvdaļu, kas tiek izmantota vairākās vietās, piemēram datņu augšupielādes iespēja vai saraksts ar komentāriem. 1.13. att. redzama Angular komponentes pamatuzbūve – komponentē tiek iekapsulēta TypeScript loģika, kas kontrolē noteiktu HTML apgabalu. Komponente var izmantot citus servisus, kuru instances, kad nepieciešams, nodrošina pirmkoda injekcijas iespējas [27].



1.13. att. **Angular arhitektūra [27]**

Lai nebūtu lieki jālēkā pa darba nodaļām, Angular specifiskie datu modeļi, HTTP servisi, direktīvas un formu validācijas mehānismi tiks paskaidroti turpmākās nodaļās, pirmkoda ģenerēšanas kontekstā.

2. PROBLĒMAS APRAKSTS

Nodaļā detalizētāk aprakstīta autora motivācija risināt darba problēmu, identificēti dublētie informācijas vienumi .NET un TypeScript tīmekļa risinājumos, kā arī definēts problēmas tvērums – tā problēmas daļa, kas tiks pētīta un risināta maģistra darbā.

2.1. Motivācija

Tīmekļa risinājumi savu popularitāti ir iemantojuši un saglabājuši vairāku iemeslu dēļ:

- atvērti tīmekļa standarti un specifikācijas;
- neatkarību no platformas realizē tīmekļa pārlūki;
- plaši pazīstami un izmantoti izstrādes rīki klienta puses lietotnes izstrādei – HTML, CSS, JavaScript (realitātē saraksts ir daudz garāks, bet visa pamatā ir šie);
- populāras programmēšanas valodas un ietvari realizē atbalstu tīmekļa risinājumu izstrādei (HTTP pieprasījumu apstrāde un servera puses implementācija).

Tīmekļa risinājumus izmanto ne tikai publiski pieejamu tīmekļa vietņu izstrādei. Arī iekšēju resursu pārvaldībai, uzņēmumi un organizācijas bieži vien realizē tīmekļa risinājumus, jo tos ir viegli izplatīt klientiem (darbiniekiem) un klientiem ir viegli tos patērēt, neatkarīgi no izmantotās platformas vai iekārtas.

Tipisks tīmekļa risinājuma izstrādes scenārijs ir tāds, kur viena projekta ietvaros servera un klienta puses lietotnes realizē viens izstrādātājs, komanda vai uzņēmums. Piemēram, uzņēmumam nepieciešams uzskaitīt darbinieku patērēto laiku – servera puses lietotne realizē HTTP/REST saskarni pieprasījumu apstrādei, klienta puses lietotne tīmekļa pārlūkā attēlo datu ievades formas, sarakstus un citus elementus datu ievadei un prezentācijai. Izņēmums minētājam scenārijam ir tikai vienas komponentes – servera vai klienta lietotnes – izstrāde. Piemēram, Google Maps API nodrošina HTTP saskarnes bagātīgas ģeogrāfiskas informācijas iegūšanai, tai pašā laikā eksistē klienta puses lietotnes, kas patērē Google Maps API sniegto informāciju un piedāvā lietotāja saskarnes informācijas attēlošanai.

Darba kontekstā, autoram interesē pirmais scenārijs, kurā viens un tas pats izstrādātājs implementē gan servera, gan klienta puses lietotnes. Šajā gadījumā izstrādātājam visa izstrādes procesa garumā ir jāuztur noteikts kontrakts starp abām lietotnēm – kā minimums, servera un klienta puses lietotnes komunicē savā starpā, izmantojot noteiktu datu formātu un struktūru. Pirms TypeScript parādīšanās, tā nebija ļoti aktuāla problēma – klients veic HTTP pieprasījumu pēc noteikta resursa, saņem atbildē kaut kādu patvaļīgu datu modeli un veic

darbības ar to, balstoties uz savām zināšanām par saņemto datu struktūru. 2.1. att. redzams piemērs AJAX izsaukumam ar *jQuery* bibliotēku.

```
$.get('api/get-data')
    .done(function(data) {
        console.log(data);
    })
    .fail(function(data) {
        console.log('Error: ' + data);
    });
```

2.1. att. **jQuery AJAX izsaukums**

Kā jau tika minēts iepriekšējā nodaļā, JavaScript ir dinamisku tipu valoda, izstrādātājs var atsaukties uz patvaļīgu *data* objekta atribūtu vai metodi (piemēram, *data.id* vai *data.save()*) un kļūdas fakts parādīsies vien lietotnes izpildes laikā – gala lietotājam darbinot tīmekļa vietni. Ar TypeScript, savukārt, iespējams definēt saskarni objektam, kas tiek saņemts no servera, un ar datu tipa anotācijas palīdzību norādīt, kāds konkrēti datu modelis tik saņemts no katras servera API metodes. 2.2. att. redzams iepriekšējais koda fragments, kas modificēts ar TypeScript.

```
interface IServerData {
    id: number;
    name: string;
}

$.get("api/get-data")
    .done(function(data: IServerData): void {
        console.log(data);
    })
    .fail(function(data: any): void {
        console.log("Error: " + data);
    });
```

2.2. att. **jQuery AJAX izsaukums ar TypeScript**

Šāda prakse neizbēgami noved pie informācijas duplikācijas – servera puse definētie C# datu modeļi, tiek kopēti klienta pusē un pielāgoti TypeScript programmēšanas valodai. Risinājumos, kuros norit aktīva izstrāde un kuri satur vairākus simtus API metožu un datu modeļu, to sinhronizācija patērē būtisku laika daudzumu. Laikam ritot, mainās arī prasības un pieņēmumi, modeļi neizbēgami mainās un laiks tiek zaudēts. Laiks, kurš varētu tikt patērēts realizējot biznesam kritisku funkcionalitāti, nevis veicot manuālu sinhronizāciju un zaudējot produktivitāti. Iespējams varētu rasties jautājums, kāpēc kopēt saskarnes klienta pusē, ja var to nedarīt? Darba 1.3. nodaļā jau tika ilustrēts, kādus ieguvumus TypeScript dod izstrādes

procesam, definējot saskarnes saņemtajiem datu modeļiem. Apjomīgos projektos tas ir vienīgais veids, kā iespējams uzturēt izstrādāto klienta puses programmatūru, līdz ar to šis jautājums praktiski nav aktuāls. Papildus tam, datu modeļu saskarnes nav vienīgais dublētais informācijas vienums tīmekļa risinājumos.

Darba autors praksē ir bieži novērojis šo problēmu un tās negatīvo ietekmi uz tīmekļa bāzētu sistēmu izstrādes procesu. TypeScript idiomas ir ļoti tuvas C# idiomām – abās programmēšanas valodās problēmas tiek risinātas līdzīgi, līdz ar to pirmkoda ģenerēšanu var uzskatīt par derīgu problēmas risinājumu. Aplūkojot esošos risinājumus un neatrodot risinājumu, kas spētu pilnībā apmierināt visus informācijas duplikācijas scenārijus, kā arī novērojot .NET pirmkoda analīzes iespēju paplašināšanos ar Roslyn.NET kompilatora platformu, darba autoram rādās ideja par problēmas risinājumu, veicot klienta puses pirmkoda ģenerēšanu.

2.2. Informācijas duplikācijas gadījumi

Iepriekšējā sadaļā, kopā ar problēmas motivāciju jau daļēji aprakstīta datu modeļu duplikācija un pieminēts fakts, ka tas nav vienīgais informācijas duplikācijas scenārijs tīmekļa risinājumos. Šajā sadaļā aprakstīti visi autora sastaptie veidi, kā tiek dublēta informācija starp klienta un servera lietotnēm ASP.NET un Angular risinājumos.

2.2.1. Datu modeļi

Primārie duplikācijas vienumi ir datu modeļi. Datu modeļi (vai skatu modeļi, no angļu valodas vārda – *viewmodel*) tipiski tiek saņemti klienta lietotnē informācijas prezentācijai, bet tie var tikt izmantoti jebkādas informācijas apmaiņai starp serveri un klientam un darba kontekstā tas nav tik būtiski. Datu modeļi var būt dažādas sarežģītības – tie var saturēt tikai primāros vērtību tipus, var saturēt kompleksus objektus, tiem var būt tipu parametri un bāzes klases, tie var saturēt dažādas atkarības. Visus minētos gadījumus un daudzus citus nepieciešams manuāli apstrādāt. 2.3. att. redzami piemēri analoģiskiem datu modeļiem C# un TypeScript programmēšanas valodās.

```
using System;
using Samples.DotNetFull.Common.Enums;

namespace Samples.DotNetFull.ViewModels
{
    public class Person
    {
        public long Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime DateOfBirth { get; set; }
        public Address Address { get; set; }
        public Gender Gender { get; set; }
    }
}
```

```
import { Address } from './address.model';
import { Gender } from '../enums/gender.enum';

export class Person {
    id: number;
    firstName: string;
    lastName: string;
    dateOfBirth: Date;
    address: Address;
    gender: Gender;
}
```

2.3. att. **C# un TypeScript datu modeļi**

Galvenā duplikācijas problēma – regulāri mainot datu modeļu definīcijas, tas vienmēr ir jādara divās vietās – klienta un servera pusē.

2.2.2. API klienti

ASP.NET servera puses risinājumos tiek definētas publiskas API metodes, kas, izvietojot risinājumu tīmeklī, ir pieejamas, izmantojot HTTP transporta protokolu. Klienta puses risinājumi izmanto šo API, lai komunicētu ar servera lietotni, nosūtot un saņemot serializētu informāciju – tipiski XML vai JSON formātā. Abos galos šī informācija tiek deserializēta jau minētajos datu modeļos jeb datu struktūrās. API metodēm ir vairāki parametri, kas tās viennozīmīgi identificē:

- URL (piemēram *https://myapi.com/api/students/5*);
- parametri (piemēram *https://myapi.com/api/students?name=chris*);
- atgriežamais tips;
- HTTP metode (GET, POST, PUT, PATCH, DELETE).

Mainot kādu no šiem parametriem servera pusē, nepieciešams attiecīgi pielāgot arī API izsaukumu klienta pusē, līdz ar to šīs komponentes ir tieši atkarīgas no servera lietotnē definētās saskarnes. Angular risinājumos, API klientu komponentes konkrētam resursam tipiski izolē no pārējās loģikas, līdz ar to šādas komponentes iespējams ģenerēt automātiski.

2.2.3. Datu modeļu validācija

Informācijas duplikācija eksistē arī datu modeļa validācijas kontekstā. Praktiski jebkurā ASP.NET tīmekļa lietotnē, no klienta saņemtā informācija tiek validēta servera pusē, lai nodrošinātos pret SQL injekcijām vai cita tipa ļaunprātīgu darbību – nekad nevar uzticēties no klienta saņemtajai informācijai. Validāciju var realizēt ļoti daudz dažādos veidos, bet parasti cenšas izveidot vai izmantot datu validācijas mehānismus, kurus var izmantot neatkarīgi no

konkrēta datu modeļa un kurus parametrizējot var atkārtoti izmantot. ASP.NET piedāvā izmantot validācijas atribūtus, kas ļauj iekapsulēt validācijas loģiku un atkārtoti izmantot to uz dažādiem modeļa laukiem, kā redzams 2.4. att..

```
public class Address
{
    public int Id { get; set; }

    [Required]
    [StringLength(100)]
    public string Country { get; set; }

    [Required]
    [StringLength(100)]
    public string City { get; set; }

    [Required]
    [RegularExpression(@"(?<!\d)(?!0000)\d{4}(?!\d)", ErrorMessage = "Invalid Postal Code")]
    public string PostalCode { get; set; }

    public string Street { get; set; }

    public string HouseNumber { get; set; }
}
```

2.4. att. **C# datu modelis ar validācijas atribūtiem**

Plaši pieņemta prakse jebkurā klienta-servera tīmekļa risinājumā ir veikt to pašu validāciju arī klienta pusē. To dara, lai izvairītos no liekas informācijas apmaiņas ar serveri un lai nodrošinātu labāku lietotāja saskarnes pieredzi lietotājam, laicīgi informējot lietotāju par ievaddatu kļūdām (pretēji tam, lai ļautu lietotājam aizpildīt 20 formas laukus un tikai pēc datu nosūtīšanas uz serveri, paziņot, ka 18 no tiem ir ar kļūdainām vērtībām). Acīmredzami, ka klienta puses validācijas loģikai jābūt saskaņā ar servera puses validācijas nosacījumiem. Mainoties validācijas nosacījumiem vai to parametriem servera pusē, nepieciešams atjaunot arī klienta pusē realizēto loģiku.

Šis scenārijs ir nedaudz komplicētāks par diviem iepriekšminētajiem, taču izpētot Angular dinamisku formu un to validācijas iespējas, to ir iespējams realizēt.

2.2.4. Konfigurācija un citi vienumi

Daļa servera puses konfigurācijas var tikt dublēta klienta pusē, piemēram, autentifikācijas servisa sniedzēja informācija, ja risinājums tādu izmanto. Konfigurācija parasti tiek glabāta statiskos objektos servera pusē, vai kāda populāra datu formāta datnēs, parasti JSON vai XML. Kā arguments pret šīs informācijas sinhronizācijas automatizāciju droši vien ir fakts, ka tā tik bieži nemainās un negatīvā ietekme uz izstrādātāju produktivitāti ir zema.

2.3. Tvērums

Apskatītajai problēmai tiek definēts tvērums, lai skaidri noteiktu, kas tiek un kas netiek risināts maģistra darba ietvaros.

Tehnoloģiskā ziņā tiek apskatīti tikai tie tīmekļa risinājumi, kas izmanto .NET, Angular un TypeScript tehnoloģijas. Citas servera puses tehnoloģijas, kas potenciāli sastopas ar līdzīgām informācijas duplikācijas problēmām, piemēram, Java un Scala, netiek apskatītas. Ģenerētais TypeScript pirmkods primāri domāts priekš Angular risinājumiem, lai gan ģenerētie datu modeļi varētu tikt izmantojami jebkurā patvaļīgā TypeScript lietotnē (jo ir parastas TypeScript klases, kas reizē ir arī parastas JavaScript klases). Atbalsts citiem klienta puses izstrādes ietvariem, piemēram, React vai Ember, maģistra darba ietvaros netiek iekļauts, taču to var uzskatīt par labu darba turpinājumu.

TypeScript pirmkoda ģenerēšana tiek veikta tikai tādā līmenī, lai novērstu minētos informācijas duplikācijas scenārijus. Darba mērķis ir izveidot minimālu darbspējīgo produktu, kas ļautu izstrādātājiem automatizēt dublētās informācijas sinhronizāciju un būtu lietojams praksē. Darba mērķis nav izveidot risinājumu, kas spētu pārnest patvaļīgi izvēlētu pirmkoda fragmentu (vai veselu risinājumu) no vienas programmēšanas valodas uz otru – tādi risinājumi pastāv, tiem ir savs pielietojums, bet darbā aprakstītās problēmas risināšana nav viens no tiem.

3. ESOŠO RISINĀJUMU APSKATS

Darba problēmas būtība ir informācijas pārnešana no vienas tehnoloģijas/programmēšanas valodas uz otru un apskatītais risinājums tai ir pirmkoda ģenerēšana. Pirmkoda ģenerēšanas uzdevums dalās divās daļās:

- avota (.NET) risinājuma analīze lai izgūtu nepieciešamos metadatus un informāciju;
- pirmkoda ģenerēšana mērķa risinājumam (darba kontekstā – Angular specifiska TypeScript ģenerēšana).

Uzdevumu risināšanai, autors apskata iespējas .NET risinājumu analīzei, kā arī esošos risinājumus TypeScript pirmkoda ģenerēšanai no .NET/C# risinājumiem. Nodaļas beigās dots izvērtējums apskatītajiem .NET risinājumu analīzes rīkiem, kā arī pamatots, kāpēc esošie TypeScript pirmkoda ģenerēšanas risinājumi nerisina vai daļēji risina darbā apskatīto problēmu.

3.1. .NET rīki pirmkoda analīzei

Meta-programmēšanai ir vairākas definīcijas, bet vispārīgi to var uzskatīt par programmēšanas tehniku, kur programmatūra spēj lasīt, ģenerēt, analizēt vai transformēt citas programmas, vai pati sevi, tās izpildes laikā. Meta-programmas spēj izmantot citas programmas kā savus datus. Meta-programmēšanai .NET platformā ir vairāki izpausmes veidi, bet darba kontekstā interesē apskatīt tos rīkus, kas ļauj piekļūt .NET risinājumu metadatiem, pirmkoda strukturālajai un funkcionālajai informācijai [28].

3.1.1. Refleksija

Refleksijas koncepts daudzās programmēšanas valodās ir pastāvējis vēl ilgu laiku pirms .NET. Refleksijas būtība ir dot izstrādātājam iespēju lasīt programmas saturu un izpildīt tās pirmkodu. Refleksijas API .NET platformā ir bijis pieejams jau kopš pirmās versijas. Refleksijai ir vairāki praktiski pielietojumi, taču autora interesēs ir vispārīgi apskatīt .NET refleksijas API, tā iespējas un ierobežojumus darba problēmas kontekstā.

Ar refleksiju asociētās klases atrodas .NET klašu bibliotēkas `System.Reflection` vārdu telpā. Eksistē divi galvenie ieejas punkti darbam ar refleksijas API – klašu bibliotēka (*assembly*) vai tips/klase (*type*). Gan klašu bibliotēkai, gan tipam ir vairākas pieejas attiecīgā objekta – `Assembly` vai `Type` iegūšanai. 3.1. att. redzami dažādi veidi bibliotēkas un tipu instanču iegūšanai.

```
var type = Type.GetType("System.Random");
var type2 = typeof(Random);
var type3 = new Random().GetType();

var assembly = Assembly.Load(new AssemblyName()
    { Name = "mscorlib", Version = new Version(4, 0, 0, 0) });
var assembly2 = Assembly.Load("mscorlib, Version=4.0.0.0");
var assembly3 = Assembly.LoadFrom(
    @"file:///C:/Windows/Microsoft.NET/Framework/v4.0.30319/mscorlib.dll");
```

3.1. att. **Tipu un bibliotēku atlase ar refleksiju**

Eksistē arī notācija ģenērisku tipu iegūšanai, piemēram, `System.Lazy<T>` vai `Tuple<T1, T2, T3>`. 3.2. att. redzams piemērs šādu tipu atlasei.

```
var lazyType = randomAssembly.GetType("System.Lazy`1");
var lazyType = typeof(Lazy<>);
var threeTupleType = typeof(Tuple<,,>);
```

3.2. att. **Ģenērisku tipu atlase ar refleksiju**

Kad iegūta atsauce uz noteiktu tipu, iespējams apskatīt to metodes, kā redzams 3.3. att. Metožu pārlādēšanas gadījumā iespējams norādīt parametru skaitu vai atlasīt tikai publiskās metodes. Ar `BindingFlags` opcijām iespējams veikt sarežģītāku filtrēšanu.

```
var randomType = new Random().GetType();
var nextMethod = randomType.GetMethod("Next");
var nextWithTwoArguments = randomType.GetMethod("Next",
    new Type[] { typeof(int), typeof(int) });
var nextWithTwoArguments = randomType.GetMethod("Next",
    BindingFlags.Instance | BindingFlags.Public, null,
    new Type[] { typeof(int), typeof(int) }, null);
```

3.3. att. **Metožu atlase ar refleksiju**

Kad iegūta atsauce uz meklēto metodi, iespējams iegūt parametru informāciju ar `GetParameters` metodi. Līdzīgi iespējams atlasīt tipa/klases atribūtus (`GetProperties`), laukus (`GetFields`) un notikumus (`GetEvents`). Katra no metodēm atgriež masīvu ar attiecīgajiem objektiem, piemēram, `MethodInfo`, `PropertyInfo`, `FieldInfo` – katrs no tiem satur attiecīgā objekta metadatus un citas metodes.

Vēl viena būtiska refleksijas sastāvdaļa, kas nav aktuāla maģistra darba kontekstā, ir pirmkoda izpilde. Refleksija ļauj veidot pieejamo tipu instances, izsaukt izveidoto instanču metodes un piešķirt laukiem vērtības.

Rezumējot, refleksija ļauj apskatīt klašu, metožu, lauku un citu objektu metadatus, bet ar to nevar veikt dekompilāciju vai lasīt CIL pirmkodu. Tādējādi nav iespējams apskatīt konkrētu metožu implementāciju vai citu pirmkodu, kas atrodas iekš C# objektiem. Refleksiju var kombinēt ar kādu no CIL analīzes risinājumiem, piemēram, Mono *Cecil*[1], lai spētu veikt detalizētāku analīzi [28, 41.-62.lpp].

3.1.2. Roslyn

Roslyn API dod iespēju analizēt C# un VB.NET risinājumus, "atverot" kompilatoru un padarot to par servisu. Iespējams analizēt pirmkoda datņu sintakses kokus, veikt kompilācijas un strādāt ar simbolu semantisko informāciju. Roslyn ļauj analizēt .NET risinājumus arī projektu līmenī, tādējādi paverot plašas iespējas meta-programmēšanas aspektā. Detalizēts Roslyn apskats veikts darba 4. nodaļā.

3.2. .NET rīki pirmkoda ģenerēšanai

Daļa no meta-programmēšanas definīcijas ir arī pirmkoda ģenerēšana. Lai gan TypeScript dala vairākas objektorientētās programmēšanas paradigmas un sintaktiskos elementus ar C#, galu galā tā ir pavisam cita programmēšanas valoda un daudzi .NET pirmkoda ģenerēšanas risinājumi ir paredzēti specifiski .NET pirmkoda ģenerēšanai. Tomēr eksistē tāds risinājums kā T4, kas ir izmantojams jebkuras tekstuālas informācijas datņu ģenerēšanai. Reizē autors apskata arī citus risinājumus un pieejas .NET pirmkoda ģenerēšanai, lai izprastu un, iespējams, aizgūtu to galvenās idejas.

3.2.1. Text Template Transformation Toolkit

Text Template Transformation Toolkit (turpmāk tekstā izmantots vispārpieņemtais saīsinājums T4) ir rīks, kas izmanto teksta veidnes, lai ģenerētu teksta datnes ar jebkuru paplašinājumu. T4 teksta veidnes satur teksta šablonus un kontroles loģiku. Kontroles loģika tiek rakstīta kā C# vai Visual Basic pirmkoda fragmenti. Izšķir izpildes un izstrādes laika T4 teksta veidnes. Veicot pirmkoda ģenerēšanu ar T4, izstrādātāji tipiski paļaujas uz `System.Reflection` bibliotēkas sniegtajām iespējām, lai iegūtu informāciju par kompilētās bibliotēkās definētajiem tipiem, to atribūtiem un metodēm (teorētiski tas var būt jebkurš cits metadatu avots, bet refleksija ir viens no visērtāk sasniedzamajiem risinājumiem) [28, 65.-100.lpp].

Izpildes laika veidnes izmanto programmatūras izpildes laikā teksta fragmentu ģenerēšanai. 3.4. att. redzama daļa no dinamiski ģenerēta HTML vai JavaScript pirmkoda.

[1] http://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil

```
<html>
 <body>
  The date and time now is: <#= DateTime.Now #>
 </body>
</html>
```

3.4. att. **T4 izpildes laika veidne**

Izstrādes laika veidnes izmanto izstrādes laikā, lai ģenerētu kādu daļu no programmatūras pirmkoda vai lietotnē izmantojamiem resursiem. 3.5. att. redzama izpildes laika veidne C# pirmkoda fragmenta ģenerēšanai.

```
<#@ output extension=".txt" #>
<#@ assembly name="System.Xml" #>
<#  System.Xml.XmlDocument configurationData = ...;  #>
namespace Fabrikam.<#= configurationData.SelectSingleNode("jobName").Value #>  { }
```

3.5. att. **T4 izstrādes laika veidne**

T4 ir teicams rīks, lai veiktu veidņu bāzētu pirmkoda ģenerēšanu jebkurai programmēšanas valodai (vai citam teksta bāzētam izteiksmes veidam).

Argumenti par labu T4:

- Visual Studio izstrādes vides integrācija;
- var ģenerēt TypeScript pirmkoda veidnes;

Argumenti pret T4:

- liela daļa loģikas teksta šablonos;
- veidņu bāzēta pirmkoda ģenerēšana nav derīga kompleksas loģikas ģenerēšanai.

3.2.2. CodeDOM

Tīmekļa pārlūkiem ir dokumentu objektu modelis lapu izveidei un navigācijai starp tām. HTML ir dokumentu objektu modelis šo lapu satura un struktūras aprakstīšanai. JavaScript ir dokumentu objektu modelis, lai automatizētu abus iepriekš pieminētos. Minētie modeļi attiecīgi tiek saukti par dokumentu objektu modeļiem, jo tie ir specifiski paredzēti globālajam tīmeklim, kas lielākoties ir uz dokumentiem bāzēta sistēma. Arī Microsoft jau .NET pirmsākumos izstrādāja tehnoloģiju ar nosaukumu CodeDOM, kas ļauj veikt sekojošas aktivitātes:

- aprakstīt pirmkodu lielākoties no valodas neatkarīgā datu struktūrā;
- ģenerēt pirmkodu dažādām programmēšanas valodām;
- kompilēt .NET pirmkodu.

CodeDOM ir vairāku kompleksu klašu kopa, kas atrodama .NET `System.CodeDom` un `System.CodeDom.Compiler` vārdu telpās. CodeDOM eksistē .NET klašu bibliotēkā jau kopš 1. versijas izlaišanas 2002. gada februārī. Turpmāko divu versiju laikā, CodeDOM piedzīvoja nopietnas pārmaiņas, lai atbalstītu delegātu modeļa un ģenērisku tipu ieviešanu .NET. Kopš 2005. gada, CodeDOM nav piedzīvojis būtiskas pārmaiņas, daļēji tāpēc, ka *LINQ* parādīšanās pieprasīja izteiksmju koku (*expression trees*) ieviešanu .NET, kas ir fundamentāli atšķirīgi no veida, kā CodeDOM izsaka pirmkodu kā datus – ar koda grafiem. Neatkarīgi no tā, CodeDOM ir savi pielietojumi un priekšrocības pār izteiksmju kokiem arī mūsdienās – izteiksmju kokus nevar izmantot, lai ģenerētu jaunus .NET tipus.

CodeDOM ir realizēta kompleksa klašu struktūra, kas ļauj aprakstīt tipisku .NET lietotni. Ir tādas klases kā `CodeNamespace, CodeStatement, CodeExpression, CodeTypeMember` u.t.t. Izmantojot minētās bāzes klases un klases, kas no tām manto vairākos līmeņos, iespējams izveidot jau pieminēto pirmkoda grafu jebkurai atbalstītai .NET programmēšanas valodai. CodeDOM atbalsta C#, Visual Basic, C++, J#, un JScript programmēšanas valodas. Lieki piebilst, ka atbalstītās iespējas starp programmēšanas valodām atšķiras – ja starp C# un VB.NET eksistē zināma paritāte, tad JScript un C++ realizēto iespēju skaits ir mazāks. Katrai programmēšanas valodai ir savs koda nodrošinātājs (*code provider*) – klase, kas spēj komunicēt ar attiecīgo kompilatoru un valodas servisiem. Nodrošinātājiem ir vairākas noderīgas instances metodes, kas ļauj kompilēt uzdotu koda grafu vai ģenerēt tam atbilstošu pirmkodu. 3.6. att. redzama C# metodes deklarācija un tās CodeDOM reprezentācija.

```
int Foo(int bar)
{
    int i = 0;
    if ( bar == 1 ) i = 1;
    return i;
}

method foo
    declaration
    if (expression)
        assignment
    return
```

3.6. att. **C# metode un tās CodeDOM reprezentācija**

CodeDOM piedāvā modeli kā attēlot .NET programmu kā datu struktūru, kā arī ļauj ģenerēt un kompilēt .NET pirmkodu dinamiski [28, 101.-138.lpp]. CodeDOM piedāvātā ideja

ir gana interesanta un tā realizētā datu struktūra vismaz daļēji ir izmantojama arī TypeScript programmas struktūras aprakstīšanai.

3.2.3. Reflection.Emit

`System.Reflection.Emit` vārdu telpa satur klases, kas ļauj kompilatoram vai citam klientam dinamiski veidot un izpildīt CIL pirmkodu. *Reflection.Emit* iespējas parasti izmanto skriptu lietotnes un kompilatori. *LINQPad* izmanto *Reflection.Emit* funkcionalitāti, lai ģenerētu datu kontekstus lietotnes izpildes laikā[2]. Šī ir zema līmeņa tehnoloģija, kuras izmantošanai nepieciešamas padziļinātas CIL zināšanas un kurai maģistra darba kontekstā nav pamatota pielietojuma [28, 139.-170.lpp].

3.2.4. Expression API

Expression API ir daļa no LINQ iekš C#. Tas ļauj veidot izteiksmes, kuras var dinamiski kompilēt un izpildīt programmatūras izpildes laikā. Savā būtībā, izteiksme (*expression*) ir pirmkoda reprezentācija ar datu struktūras palīdzību. Piemēram, 3.7. att. redzamais pirmkods filtrē kolekciju ar objektiem:

```
var filteredResults =
    from container in containers
    where container.Value.Contains("a")
    select container;
```

3.7. att. **LINQ kolekcijas filtrs**

Izstrādātājs ar klasiskām LINQ zināšanām teiktu, ka izteiksmi lietotnes izpildes laikā nav iespējams mainīt un tā vienmēr atfiltrēs vienumus, kuru vērtība satur simbolu “a”, taču ar padziļinātākām *Expression* API zināšanām to ir iespējams izdarīt. 3.8. att. redzamais pirmkoda fragments parāda, kā iespējams dinamiski izveidot un kompilēt analoģisku filtra nosacījumu, kuru attiecīgi pēc nepieciešamības iespējams dinamiski pielāgot lietotnes izpildes laikā [28, 171.-198.lpp].

[2] https://www.linqpad.net/HowLINQPadWorks.aspx

```
var argument = Expression.Parameter(typeof(Container));
var valueProperty = Expression.Property(argument, "Value");
var containsCall = Expression.Call(valueProperty, typeof(string).GetMethod(
    "Contains", new Type[] { typeof(string) }),
    Expression.Constant("a", typeof(string)));

var wherePredicate = Expression.Lambda<Func<Container, bool>>(containsCall, argument);
var whereCall = Expression.Call(typeof(Queryable), "Where",
    new Type[] { typeof(Container) },
    containers.AsQueryable().Expression, wherePredicate);

var expressionResults = containers.AsQueryable()
    .Provider.CreateQuery<Container>(whereCall);
```

3.8. att. **Dinamisks filtrs ar *Expression* API**

Arī šim paņēmienam nav lielas vērtības vispārīgā pirmkoda ģenerēšanas uzdevumā, jo tas ir specifisks C#, bez iespējas izdot pirmkodu uz diska.

3.2.5. CIL pārrakstīšana

Liela daļa izstrādātāju uzskata, ka pēc tam, kad C# pirmkods tiek kompilēts uz CIL un saglabāts uz diska, tas ir "iesaldēts" un nav rediģējams. Tā gluži nav, jau iepriekš pieminētais Mono *Cecil* projekts, ļauj ielādēt bibliotēku, veikt nepieciešamās CIL pirmkoda izmaiņas un saglabāt tās uz diska. Tas ļauj nodrošināt dažādus pirmkoda injekcijas scenārijus. Piemēram, 3.9. att. redzama metode ar implementētu notikumu reģistrēšanu.

```
public static int Divide(int x, int y)
{
    Console.Out.WriteLine("Divide started");
    Console.Out.WriteLine("x = " + x);
    Console.Out.WriteLine("y = " + y);

    if(y == 0)
    {
        Console.Out.WriteLine("Divide threw an ArgumentException");
        throw new ArgumentException();
    }

    var result = x / y;

    Console.Out.WriteLine("Divide finished - return = " + result);

    return result;
}
```

3.9. att. **Manuāli realizēta notikumu reģistrēšana**

Ne visiem gadījumiem tas ir nepieciešams un tas pievieno lielu apjomu pirmkoda, kas nav saistīts ar pašas metodes implementāciju, tāpēc ar pirmkoda injekcijas palīdzību ir iespējams identificēt visas nepieciešamās metodes ar `Trace` atribūtu un pievienot nepieciešamo pirmkodu pēc kompilācijas, balstoties uz lietotnes metadatiem [28, 199.-219.lpp]. Metode tad izskatītos, kā redzams 3.10. att.

```
[Trace]
public static int Divide(int x, int y)
{
    if(y == 0)
    {
        throw new ArgumentException();
    }
    return x / y;
}
```

3.10. att. **Notikumu reģistrācija ar pirmkoda injekciju**

Arī šis paņēmiens darbojas tikai ar CLR programmēšanas valodām un nav izmantojams TypeScript pirmkoda ģenerēšanai.

3.2.6. Roslyn

C# pirmkoda ģenerēšana ar Roslyn strādā sintakses līmenī. Iespējams izveidot sintakses kokus no jau gatavām C# pirmkoda datnēm un tad darboties ar to – atlasīt mezglus, aizvietot tos ar citiem, pievienot vai noņemt specifiskus sintakses mezglus. Iespējams arī veidot pavisam jaunus sintakses kokus no nulles – Roslyn piedāvā metodes visu iespējamo sintaktisko konstrukciju izveidei. Ir pieejams arī rīks (un tā pirmkods) ar nosaukumu Roslyn *Quoter*[3], kas ļauj ievadīt C# pirmkodu un izgūt Roslyn API pieprasījumus analoģiska pirmkoda fragmenta izveidei [28, 287.-315.lpp]. Piemēram, tukšas klases deklarācijas sintakses koku, `public class C {}`, iespējams iegūt ar 3.11. att. redzamo pirmkodu.

```
CompilationUnit()
.WithMembers(
    SingletonList<MemberDeclarationSyntax>(
        ClassDeclaration("C")
        .WithModifiers(
            TokenList(
                Token(SyntaxKind.PublicKeyword)))))
.NormalizeWhitespace()
```

3.11. att. **C# pirmkoda ģenerēšana ar Roslyn**

[3] http://roslynquoter.azurewebsites.net

Roslyn piedāvā labākas pirmkoda ģenerēšanas iespējas par CodeDOM, jo pirmkoda ģenerēšanai tiek izmantoti tie paši objekti, ko C# kompilators izmanto pirmkoda parsēšanai. Tas nozīmē, ka tiek atbalstīti visas C# valodas iespējas.

Roslyn ir pārliecinoši ērtākais rīks C# un VB.NET pirmkoda ģenerēšanai, bet diemžēl tā sintakses koki ir specifiski konkrētai programmēšanas valodai un nav vispārīgi attiecināmi uz TypeScript (lai gan specifiskas konstrukcijas ir identiskas).

3.3. Rīki TypeScript pirmkoda ģenerēšanai

Maģistra darba ietvaros apskatīti pieejamie rīki TypeScript pirmkoda ģenerēšanai. Maza daļa no apskatītājiem rīkiem ir komerciāli, vairums rīku ir atvērtā pirmkoda rīki, kuru pirmkods pieejams publiskos repozitorijos, parasti – *GitHub*.

3.3.1. Rīku klasifikācija

Apskatīto rīku kopsaucējs ir TypeScript pirmkoda ģenerēšana, taču to mērķi, realizācijas pieeja un pielietojums krasi atšķiras, tāpēc nebūtu saprātīgi aplūkot katru no tiem savā kontekstā. Tā vietā rīki klasificēti un novērtēti darbā apskatītās problēmas kontekstā, proti pēc to spējām ģenerēt pirmkodu identificētajiem informācijas duplikācijas vienumiem.

Detalizētāks kritēriju apraksts redzams 3.1. tabula.

3.1. tabula

Esošo risinājumu novērtējuma kritēriji

Kritērijs	**Jā**	**Daļēji**	**Nē**
Modeļi	Atbalsta CLR datu tipus, ģenēriskus datu tipus, bāzes klases, *Nullable* tipus, mantošanu, enumerācijas, masīvus un kolekcijas.	Atbalsta tikai daļu no uzskaitītajiem.	Scenārijs netiek atbalstīts.
API klienti	Ģenerē servera implementācijai atbilstošu HTTP klientu, atbalsta drošības implementācijas.	Neatbalsta drošības implementācijas.	Scenārijs netiek atbalstīts.
Validācija	Ģenerē validāciju atribūtus, gan iebūvētos, gan izstrādātāja implementētos. Papildus atbalsts kādai trešās puses validāciju bibliotēkai.	Neatbalsta izstrādātāja implementētos validāciju mehānismus.	Scenārijs netiek atbalstīts.

ASP.NET	Analizē .NET risinājumus, konkrēti C# un ASP.NET tehnoloģijas.	-	Scenārijs netiek atbalstīts.
Angular	Ģenerē Angular specifisku pirmkodu, konkrēti Angular 2+ versijām.	Atbalsta tikai vecākas Angular versijas.	Scenārijs netiek atbalstīts.

3.3.2. Rīku novērtējums

Kopā apskatīti 12 dažādi risinājumi. Rīku novērtējums redzams 3.2. tabula.

3.2. tabula

Esošo risinājumu novērtējums

Nosaukums	Modeļi	API klienti	Validācija	ASP.NET	Angular
Bridge.NET[4]	jā	nē	nē	jā	nē
DuoCode[5]	jā	nē	nē	jā	nē
Netjs[6]	jā	nē	nē	jā	nē
Rosetta[7]	daļēji	nē	nē	jā	nē
Script#[8]	jā	nē	nē	jā	nē
TypeLITE[9]	daļēji	nē	nē	jā	daļēji
TypeScriptDefinitionGenerator[10]	daļēji	nē	nē	jā	daļēji
TypeScriptSyntaxPaste[11]	daļēji	nē	nē	jā	nē
TypeSharp[12]	daļēji	nē	nē	jā	nē
TypeWalker[13]	jā	daļēji	nē	jā	nē
TypeWriter[14]	daļēji	nē	nē	jā	nē
webapiclientgen[15]	jā	jā	nē	jā	jā

[4] http://bridge.net
[5] http://duoco.de
[6] https://github.com/praeclarum/Netjs
[7] https://github.com/andry-tino/Rosetta
[8] https://github.com/nikhilk/scriptsharp
[9] http://type.litesolutions.net
[10] https://github.com/madskristensen/TypeScriptDefinitionGenerator
[11] https://github.com/nhabuiduc/TypescriptSyntaxPaste
[12] https://github.com/davemckeown/TypeSharp
[13] https://github.com/stevecooperorg/TypeWalker
[14] https://github.com/frhagn/Typewriter
[15] https://github.com/zijianhuang/webapiclientgen

Apskatītie rīki dalās divās daļās – kompilatori, kas nodrošina JavaScript/TypeScript lietotņu izstrādi ar .NET izstrādes līdzekļiem un specifiski rīki, kas izstrādāti ar mērķi ģenerēt specifiskas JavaScript/TypeScript komponentes no C# risinājumiem. Tālāk īsi apskatīti abas no kategorijām.

3.3.3. C# uz JavaScript/TypeScript kompilatori

Šādu rīku galvenais mērķis ir pārnest C# pirmkodu vai lietotni uz tai atbilstošu JavaScript/TypeScript pirmkodu vai lietotni. To galvenā mērķa auditorija ir izstrādātāji, kas vēlas pielietot savas C# zināšanas klienta puses tīmekļa lietotņu izstrādei. No apskatītajiem rīkiem tādi ir Bridge.NET, DuoCode un Netjs. Var pieminēt arī vecākus rīkus – Project V un JSIL. Rīku lielākā problēma ir rezultējošā JavaScript pirmkoda milzīgais apjoms un vāja lasāmība, lai nodrošinātu dažādu kompleksu .NET konstrukciju iespējas.

3.3.4. TypeScript ģenerēšanas utilītas

Pārējie no apskatītajiem rīkiem ir specifiska lietojuma utilītas, kas ģenerē TypeScript pirmkodu no .NET risinājumiem. Šādi rīki tipiski ir ar minimālu dokumentāciju un piemēriem, bez nopietnām pielāgošanas iespējām un tikai viens no apskatītajiem bija spējīgs ģenerēt HTTP klientus no ASP.NET kontrolieriem. Neviens no apskatītajiem rīkiem pilnībā neapmierināja visus kritērijus, tāpēc nepieciešamība pēc autora izstrādāta risinājuma tiek apstiprināta.

3.4. TypeScript kompilatora API

Līdzīgi Roslyn pieejai izstrādāt C# kompilatoru C# programmēšanas valodā, arī TypeScript kompilators ir izstrādāts TypeScript programmēšanas valodā un pieejams kā atvērtā pirmkoda projekts. Papildus tam, TypeScript valodas izstrādātāji ir izveidojuši vairākus kompilatora un valodas servisu API. Šāds solis ievērojami atvieglo izstrādes rīku ražotāju darbu un ļauj radīt izstrādes rīkus, kas lasa, analizē, ģenerē un izdod TypeScript pirmkodu. Teicamu izstrādes rīku esamība ir viena no TypeScript lielākajām vērtībām, kas ievērojami atvieglo izstrādes procesu un uzlabo produktivitāti. Visual Studio Code izstrādes vide ir labs piemērs iebūvētiem TypeScript rīkiem un dažādiem trešās puses izstrādātāju radītiem paplašinājumiem[16].

TypeScript kompilatora API piedāvā iespējas veikt dažādas funkcijas:

- kompilēt TypeScript pirmkoda datnes uz JavaScript;

[16] https://code.visualstudio.com/docs/languages/typescript

- transformēt TypeScript pirmkoda fragmentus uz JavaScript;
- iegūt sintakses koka reprezentāciju no TypeScript pirmkoda;
- apstaigāt TypeScript sintakses kokus un veikt izmaiņas tajos;
- veidot TypeScript sintakses kokus un iegūt to pirmkoda reprezentāciju;
- analizēt simbolu informāciju.

Kompilatora API pastāvīgi attīstās un to dokumentācija, diemžēl, nevar lepoties ar augstu detalizācijas pakāpi, tāpēc API apguve lielā mērā balstās uz praktisku darbu.

Kompilatora API esamība paver zināmas iespējas arī maģistra darba kontekstā. API piedāvā stingri definētu saskarni abstrakta sintakses koka izveidei un pirmkoda iegūšanai no tā, līdz ar to būtu lieki izstrādāt kādu savu mehānismu TypeScript pirmkoda struktūras formālai aprakstīšanai un parsēšanai. It īpaši tāpēc, ka šo API uztur Microsoft – TypeScript izstrādātājs – un tas attīstās līdz ar pašu programmēšanas valodu, tādējādi sintaktiskās struktūras izmaiņas vienmēr tiks atspoguļotas iekš kompilatora API un autoram to nevajadzēs darīt manuāli [29, 30, 31]. Tīmeklī ir atrodami rīki, kuru mērķis ir piedāvāt ērtāk izmantojamus un labāk dokumentētus API, kas iekapsulē TypeScript kompilatoru API[17][18]. Jāsaka gan, ka to tekošajā stāvoklī, minētie rīki vēl nav izmantojami sintakses koku izveidei un pirmkoda iegūšanai.

3.5. Kopsavilkums

Esošie pirmkoda ģenerēšanas risinājumi nepiedāvā pietiekoši lielu automatizāciju vai arī pilnībā koncentrējas uz JavaScript/TypeScript aizstāšanu ar C#, kas nav darba apskatītās problēmas risinājums. Darbā apskatītajai problēmai nepieciešams risinājums, kas apvieno konkrētas Angular un ASP.NET ietvaru zināšanas, lai ģenerētu metodes API izsaukumiem, spētu analizēt .NET validācijas mehānismus un tulkot tos uz Angular analoģiskām validācijas direktīvām. Tāpat nepieciešams analizēt projektus un dokumentus risinājuma līmenī, ko apskatītie rīki nespēj.

[17] https://github.com/dsherret/ts-type-info
[18] https://github.com/dsherret/ts-simple-ast

4. ROSLYN .NET KOMPILATORA PLATFORMA

Roslyn .NET kompilatora platformas izpētei un pielietošanai maģistra darbā atvēlēta pietiekoši liela loma, tāpēc vesela nodaļa veltīta Roslyn darbības principu apskatei.

4.1. Ievads

Kompilatori tradicionāli ir bijuši melnās kastes – programmatūras pirmkods tiek pārvērsts izpildāmās datnēs vai klašu bibliotēkās. Kompilācijas kļūdu gadījumā, tiek izdoti kļūdu paziņojumi un tā arī ir vienīgā informācija, kas seko no kompilācijas procesa. Kompilācijas laikā kompilatori iegūst dziļu izpratni par kompilējamo kodu, taču iegūtā informācija ārpus kompilatora procesiem nevienam nav pieejama un pēc kompilācijas beigām tiek ātri aizmirsta.

Gadiem ilgi šāds melnās kastes princips ir bijis pietiekams, taču arvien vairāk izstrādātāji paļaujas uz kompleksām integrētās izstrādes vides iespējām. Lai uzlabotu savu produktivitāti un pirmkoda kvalitāti, izstrādātāji izmanto dažādus pirmkoda analīzes un ģenerēšanas rīkus, kas spēj atrast references, inteliģenti refaktorēt pirmkodu, palīdz pārvaldīt apjomīgus risinājumus u.t.t. Izstrādes vides rīkiem kļūstot inteliģentākiem, aizvien vairāk nepieciešams iegūt padziļinātas pirmkoda zināšanas, kas ir pieejamas tikai kompilatoram. Roslyn galvenais mērķis ir atvērt šīs kompilatoru melnās kastes un ļaut izstrādes rīkiem un izstrādātājiem piekļūt kompilācijas procesiem un informācijai, ko tie satur. Tādējādi kompilators kļūst par platformu, ko lietotāji var izmantot ar pirmkoda analīzi saistītu uzdevumu veikšanai savos izstrādes rīkos un lietotnēs.

Roslyn platformas izveide ļauj daudz plašākai izstrādātāju kopai nodarboties ar pirmkoda analīzes rīku un lietotņu izveidi, kā arī tā rada inovācijas iespējas tādās jomās kā meta-programmēšana, pirmkoda ģenerēšana, analīze un transformācija.

Roslyn ir segvārds, kas sākotnēji tika izmantots projekta identifikācijai kopš pirmās CTP versijas 2011. gada oktobrī. Līdz ar produkta pirmo RTM versiju, kas tika izlaista kopā ar Visual Studio 2015, Roslyn ieguva oficiālu nosaukumu – ".NET kompilatora platforma" (*.NET compiler platform*). Lai gan oficiālais nosaukums pastāv jau labu laiku, Roslyn nosaukums ir saglabājis savu popularitāti un dažādi informācijas avoti (elektroniskie un drukātie) vēljoprojām to izmanto (un visticamāk turpinās izmantot). Maģistra darbā lielākoties izmantots "Roslyn" nosaukums – tas ir īsāks un labskanīgāks variants, taču abi nosaukumi ir savstarpēji apmaināmi.

4.2. Roslyn kompilatora API

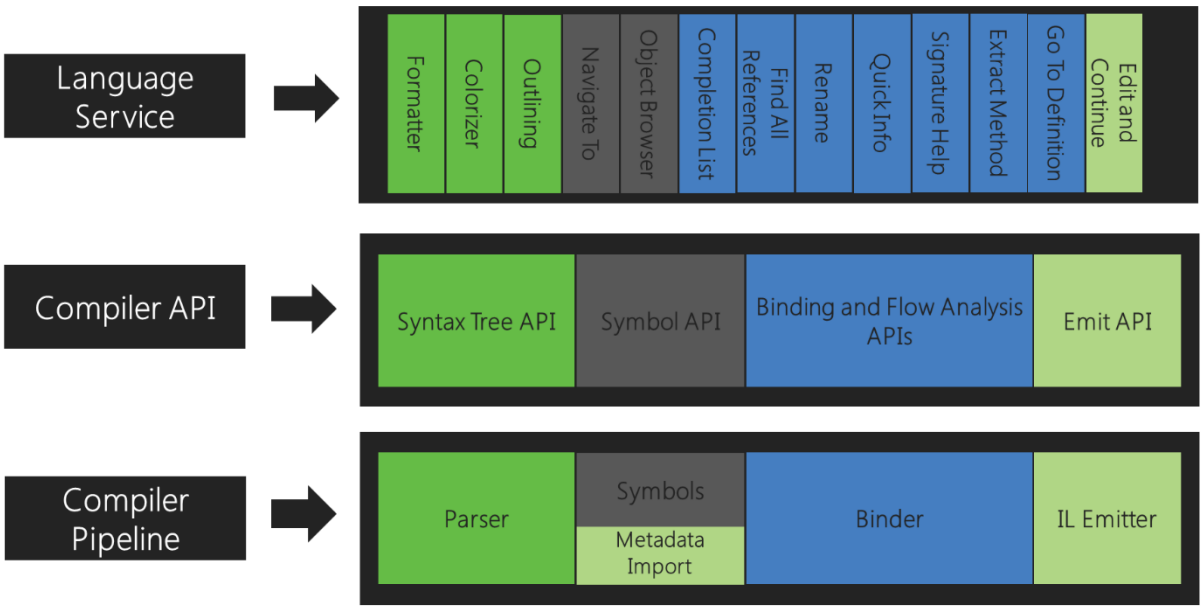
Roslyn API dublē tradicionāla kompilatora funkciju apgabalus. Atbilstoši katram apgabalam, Roslyn dod piekļuvi apgabala modeļa reprezentācijai. 4.1. tabula redzama katrs kompilatora funkciju apgabals un attiecīgās informācijas Roslyn piedāvātā reprezentācija.

4.1. tabula

Kompilatora fāzes un Roslyn reprezentācija

Fāze	Kompilators	Roslyn reprezentācija
Parsēšana	Pirmkods tiek sadalīts loģiskās sintaktiskās vienībās, kas atbilst valodas gramatikai.	Sintakses koks.
Deklarācija	Pirmkoda un metadatu deklarācijas tiek analizētas, lai formētu simbolus.	Hierarhiska simbolu tabula.
Saistīšana	Identifikatori tiek saistīti ar simboliem.	Semantiskās analīzes modelis.
Izvade	Visa kompilatora uzkrātā informācija tiek izmantota izpildkoda izvadei.	Saskarne, kas izdod CIL baitkodu.

4.1. att. var redzēt, kā minētās fāzes sasaistās ar Roslyn valodu servisiem.



4.1. att. **Roslyn API un kompilatora fāzes [32]**

Roslyn valodu servisi darbina Visual Studio 2015+ pirmkoda analīzes un transformācijas iespējas. Valodu servisus var izmantot Visual Studio iespēju paplašināšanai un jaunu rīku izstrādei.

4.3. Sintakse

Sintakses koks ir fundamentāla datu struktūra, ko piedāvā Roslyn kompilatora API. Sintakses koki reprezentē pirmkoda leksisko un sintaktisko struktūru. Roslyn kontekstā sintakses koki galvenokārt pilda divas funkcijas.

1. Tie ļauj programmatūras izstrādes rīkiem, piemēram, integrētām izstrādes vidēm, spraudņiem, pirmkoda analīzes un transformācijas rīkiem apstrādāt pirmkoda sintaktisko struktūru .NET projektā.
2. Tie ļauj izstrādes rīkiem veidot, rediģēt un pārkārtot pirmkodu dabiskā manierē, bez nepieciešamības tieši rediģēt pirmkoda tekstu. Veidojot un manipulējot sintakses kokus, rīki var viegli rediģēt pirmkodu pēc saviem ieskatiem.

4.3.1. Sintakses koki

Sintakses koki ir primārā struktūra, kas tiek izmantota pirmkoda kompilācijas, analīzes, transformācijas un ģenerēšanas procesos. Sintakses koki tiek plaši izmantoti arī izstrādes vides iespēju nodrošināšanai. Kompilācijas pirmais solis ir pirmkoda parsēšana – simbolu virknes vispirms tiek identificēts un kategorizēts kādā no daudziem, zināmiem valodas struktūras elementiem.

Datu struktūrai, kas reprezentē Roslyn sintakses kokus, ir trīs galvenās īpašības. Pirmkārt, sintakses koki satur visu pirmkoda datnē ietverto informāciju (angļu valodas avotos šo īpašību sauc par *full fidelity*). Tas ir pats pirmkoda teksts, katra gramatiskā konstrukcija, katrs leksiskais vienums un viss pārējais starp tiem, ieskaitot komentārus, tukšuma simbolus un kompilatora direktīvas. Sintakses koki satur arī pirmkodā identificētās sintaktiskās kļūdas.

No pirmās īpašības tieši seko otrā – no pirmkoda iegūtais sintakses koks ir pilnībā transformējams atpakaļ uz sākotnējo pirmkodu. Katram sintakses mezglam kokā ir iespējams iegūt pilnu apakškoka teksta reprezentāciju. Šī īpašība ļauj izmantot sintakses koka datu struktūru pirmkoda teksta izveidei un rediģēšanai. Proti, izveidojot jaunu sintakses koku, netieši tiek izveidots ekvivalents pirmkoda teksts, bet rediģējot esošu koku, tiek rediģēts reprezentētais pirmkoda teksts.

Treškārt, sintakses koki ir nemainīgi (*immutable*). Kad sintakses koks tiek izveidots, tas ir pirmkoda tekošā stāvokļa momentuzņēmums un attiecīgā sintakses koka instance nekad netiek mainīta. Šī īpašība ļauj vairākiem lietotājiem vienlaicīgi manipulēt vienu un to pašu sintakses koku dažādos procesos, neizmantojot duplikāciju vai bloķēšanu. Sintakses koki neatļauj tiešas modifikācijas tajos, tāpēc Roslyn piedāvā metodes sintakses koku veidošanai un rediģēšanai.

Katrs sintakses koks tiek implementēts kā koka datu struktūra, kas sastāv no mezgliem (*nodes*), tekstvienībām (*tokens*) un papildinformācijas (*trivia*). Visual Studio izstrādes vidē ir iespējams lejupielādēt rīkus, kas ļauj apskatīt sintakses koka vizualizāciju jebkurai pirmkoda datnei. 1. pielikumā redzams sintakses koka vizualizācija C# pirmkoda datnei. Klikšķinot uz pirmkoda teksta fragmentiem, iespējams redzēt to reprezentāciju un atrašanās vietu sintakses kokā. 2. pielikumā redzami sintakses grafi, kurus iespējams iegūt ar to pašu rīku iekš Visual Studio. Apstaigājot sintakses koku iespējams identificēt jebkuru priekšrakstu, izteiksmi, tekstvienību vai tukšuma simbolu.

4.3.2. Sintakses mezgli

Sintakses mezgls ir viena no primārajām sintakses koka sastāvdaļām. Mezgli reprezentē sintaktiskas konstrukcijas – deklarācijas, priekšrakstus, klauzulas un izteiksmes. Katru minēto sintakses mezgla kategoriju reprezentē atsevišķa klase, kas manto no `SyntaxNode` klases.

Katrs sintakses koka mezgls nekad nav lapa sintakses kokā un tas vienmēr ir vecāks citiem mezgliem un tekstvienībām. Katrs mezgls, izņemot sakni, var piekļūt savam vecākam caur `Parent` atribūtu. Mezgla vecāks nekad nemainās, jo sintakses koki ir nemainīgi.

Katram mezglam ir `ChildNodes` metode, kas atgriež kolekciju ar visiem sintakses mezgliem – bērniem, tādā secībā, kādā tie parādās avota tekstā. Minētā kolekcijas nesatur tekstvienības. Katram mezglam ir arī `DescendantNodes, DescendantTokens` un `DescendantTrivia` metodes, kas attiecīgi satur visus elementus, kas ir pakārtoti attiecīgajam mezglam.

Katra sintakses mezgla apakšklase satur atribūtus, kuru tipi ir stingri definēti, piemēram, `BinaryExpressionSyntax` mezgla klasei ir trīs papildus atribūti, kas ir specifiski binārajiem operatoriem: `Left, OperatorToken` un `Right`, kuru tipi respektīvi ir `ExpressionSyntax` un `SyntaxToken`. Dažiem sintakses mezgliem ir neobligāti bērni, piemēram `IfStatementSyntax` ir neobligāts `ElseClauseSyntax` atribūts, kas atgriež `null`, ja `if` blokam neseko `else` bloks.

4.3.3. Sintakses tekstvienības

Sintakses tekstvienības ir sintakses koka un valodas gramatikas terminālās vienības, kas reprezentē pirmkoda mazākos sintaktiskos fragmentus – tādus elementus, kurus var tieši reprezentēt pirmkoda tekstā kā simbolus. Tekstvienības sintakses kokā vienmēr ir lapas un nekad nav vecāks citam koka mezglam. Sintakses tekstvienības iekļauj atslēgvārdus, identifikatorus, literāļus un punktuācijas.

Efektivitātes nolūkos, `SyntaxToken` tips ir CLR vērtības tips (tāds, kuram vērtību piešķir tieši, piemēram – `int`, `char`, `float`, `decimal`, `bool`), tāpēc pretēji sintakses mezgliem, visus tekstvienības veidus reprezentē tikai viena veida struktūra, kurai ir vairāki atribūti, kas nosaka kāda tipa tekstvienība tiek reprezentēta. Piemēram, vesela skaitļa literāļa tekstvienību reprezentē skaitliska vērtība. Struktūrai, kas reprezentē vesela skaitļa literāli ir `Value` un `ValueText` atribūti, kas attiecīgi satur veselā skaitļa vērtību (objekts) un tā attēlojumu tekstā (simbolu virkne).

4.3.4. Sintakses papildinformācija

Sintakses papildinformācija reprezentē tās avota teksta daļas, kas no kompilatora perspektīvas ir mazsvarīgas tipiskai pirmkoda izpratnei, bet var būt ļoti būtiskas rīkam, kas veic pirmkoda formatēšanu vai rediģēšanu. Tie ir tukšuma simboli, komentāri un kompilatora direktīvas.

Papildinformācija nav daļa no valodas sintakses un var atrasties starp jebkurām divām tekstvienībām, tāpēc papildinformācija nav iekļauta sintakses kokā kā sintakses mezgla bērns. Tomēr, papildinformācija ir nepieciešama informācija pirmkoda apstrādes kontekstā, tāpēc tā ir iekļauta sintakses kokā kā `LeadingTrivia` un `TrailingTrivia` kolekcijas sintakses tekstvienībām – piemēram tukšuma simboli pirms un pēc atslēgvārda. Avota teksta parsēšanas laikā, papildinformācija tiek asociēta ar tekstvienībām un vispārīgajā gadījumā tekstvienības tiek asociētas ar jebkādu papildinformāciju, kas atrodas tajā pašā pirmkoda rindiņā līdz nākamajai tekstvienībai. Attiecīgi pirmā tekstvienība avota datnē tiek asociēta ar visu sākotnējo papildinformāciju (atrodas pirms tās) un pēdējā tekstvienība datnē tiek asociēta ar visu noslēdzošo papildinformāciju (atrodas pēc tās).

Pretēji mezgliem un tekstvienībām, papildinformācijai nav vecāka, taču tā kā katra papildinformācija ir daļa no sintakses koka un ir asociēta ar tieši vienu tekstvienību, no papildinformācijas ir iespējams piekļūt asociētajai tekstvienībai, izmantojot `Token` atribūtu. Tāpat kā tekstvienības, papildinformācija ir vērtības tips un viena `SyntaxTrivia` struktūra tiek izmantota, lai reprezentētu visus papildinformācijas veidus.

4.3.5. Teksta apgabali

Katrs elements sintakses kokā sākas noteiktā pozīcijā avota tekstā un aizņem noteiktu simbolu skaitu. `TextSpan` objekts satur divus vesela skaitļa atribūtus – apgabala sākuma pozīciju un simbolu skaitu. Katram mezglam kokā ir divi `TextSpan` atribūti:

- `Span` – teksta apgabals no pirmās līdz pēdējai tekstvienībai apakškokā;

- `FullSpan` – teksta apgabals no pirmās līdz pēdējai tekstvienībai apakškokā, iekļaujot papildinformāciju.

4.3.6. Sintakses elementu klasifikācija

Katram sintakses elementam ir `RawKind` atribūts, kas viennozīmīgi to identificē. `RawKind` vērtību var attiecināt uz `SyntaxKind` enumerāciju, kas uzskaita visus iespējamos sintakses elementus C# programmēšanas valodā. Piemēram, iepriekš apskatītās `BinaryExpressionSyntax` klases gadījumā, `RawKind` nosaka konkrētas binārās operācijas veidu – `AddExpression`, `SubtractExpression`, vai `MultiplyExpression`.

4.3.7. Sintakses kļūdas

Ja pirmkoda parsēšanas laikā tiek identificētas sintaktiskas kļūdas, sintakses koks tiek izveidots, un no tā var iegūt sākotnējo, kļūdaino programmatūras tekstu (sintakses koku 2. īpašība). Kad pirmkoda parseris nonāk līdz konstrukcijai, kas neatbilst valodas gramatikai, tas izmanto divas metodes, lai izveidotu sintakses koku.

1. Ja parseris noteiktā vietā sagaida noteiktu tektsvienību, bet to neatrod, tas var automātiski to ievietot programmatūras tekstā. Ievietotā tekstvienība satur tukšu `Span` apgabalu un tai tiek uzstādīts `IsMissing` atribūts.
2. Parseris izlaiž visas tekstvienības, kas seko kļūdai, līdz tas atrod tektsvienību, no kuras tas var turpināt parsēt programmas tekstu atbilstoši valodas sintaksei. Izlaistās tekstvienības sintakses kokā tiek reprezentētas kā papildinformācija ar tipu `SkippedTokens`.

4.4. Semantika

Sintakses koki reprezentē pirmkoda leksiskos un sintaktiskos aspektus, un lai gan ar šo informāciju ir pietiekami, lai aprakstītu visas deklarācijas un loģiku, ar to ir par maz, lai viennozīmīgi noteiktu visas atsauces uz dotu tipu vai mainīgo. Piemēram, daudzi tipi, lauki, metodes un lokālie mainīgie ar vienādu nosaukumu var būt izkaisīti pa visu pirmkoda tekstu. Katrs no tiem teorētiski var būt unikāls un, lai noskaidrotu, uz kuru objektu katrs no tiem ir attiecināms, ir nepieciešams padziļinātas valodas likumu zināšanas. Šo iemeslu dēļ, papildus sintakses kokam, Roslyn piedāvā arī pirmkoda semantisko modeli, kas reprezentē programmēšanas valodas likumus un dod veidu, kā šo informāciju izmantot.

4.4.1. Kompilācija

Kompilācija ir tips, kas reprezentē visus vienumus, kas nepieciešami, lai kompilētu C# vai Visual Basic programmatūru – bibliotēku references, kompilatora iestatījumus un pirmkoda teksta datnes. Kompilācija satur katru deklarēto tipu, atribūtu vai mainīgo kā simbolu. Kompilācijas objekts satur dažādas metodes, kas palīdz atrast referencētos simbolus, kas deklarēti pirmkoda tekstā vai importēti no citas bibliotēkas. Līdzīgi kā sintakses koki, kompilācijas instances ir nemainīgas.

4.4.2. Simboli

Simbols reprezentē atsevišķu elementu, kas deklarēts pirmkodā vai importēts no referencētas bibliotēkas. Katrs modulis, tips, metode, atribūts, notikums, parametrs vai mainīgais ir simbols. Simboli no kompilatora skatupunkta ir entītijas, uz kurām atsaucas citi nosaukumi un izteiksmes. Process, kurā nosaukumi un izteiksmes tiek asociēti ar simboliem tiek saukts par saistīšanu (*binding*).

Kompilācijas satur vairākas metodes un atribūtus, kas palīdz atrast simbolus, piemēram, ir iespējams atrast simbolu deklarētam tipam pēc tā vārda metadatos, vai ir iespējams piekļūt visai simbolu tabulai.

Simboli satur papildus informāciju, ko kompilators spēj noteikt, izmantojot tam pieejamo informāciju, piemēram, referencētos simbolus. Katru simbola tipu reprezentē atsevišķa saskarne, kas manto no `ISymbol` saskarnes, katra ar saviem atribūtiem un metodēm, kas nepieciešamas kompilatora iegūtās informācijas detalizēšanai. Daudzi no saskarņu atribūtiem tieši referencē citus simbolus, piemēram, `IMethodSymbol` klases atribūts `ReturnType` tieši referencē metodes deklarācijā ietvertā atgriežamā tipa simbolu.

Simboli, kas deklarēti pirmkodā un tiek importēti no referencētajām bibliotēkām, nekādā veidā neatšķiras un tiek reprezentēti vienādi, piemēram, metodes, kas tika deklarētas pirmkodā vai tika ārēji referencētas, abas reprezentē `IMethodSymbol` klase ar vienādiem atribūtiem.

Roslyn simboli ir līdzīgs koncepts jau apskatītajiem `System.Reflection.Type` objektiem, taču Roslyn simboli modelē arī vārdu telpas un lokālos mainīgos, ne tikai tipus. Roslyn simboli ir piesaistīti konkrētās programmēšanas valodas, nevis CLR konceptiem.

4.4.3. Semantiskais modelis

Semantiskais modelis reprezentē semantisko informāciju vienai pirmkoda teksta datnei. Daži piemēri informācijai, kuru var noskaidrot, izmantojot semantisko modeli:

- simbolus un atsauces uz tiem specifiskās vietās pirmkodā;

- jebkuras izteiksmes rezultāta tipu;
- pirmkoda diagnostiska (kļūdu un brīdinājumu paziņojumi);
- mainīgo pārvietošanās starp dažādiem pirmkoda blokiem.

4.5. Darbs ar risinājumiem

Līdz šim apskatīti Roslyn darbības principi vienas datnes kontekstā, taču eksistē dažādi scenāriji, kuros ir nepieciešams analizēt pirmkodu arī projektu un risinājumu līmenī. Roslyn piedāvā `Workspace` saskarni, kas palīdz organizēt visu risinājumā ietilpstošo projektu informāciju vienā objekta modelī, bez nepieciešamības manuāli apstrādāt projektu atkarības un iestatījumus.

4.5.1. Darba vide

Darba vides objekts reprezentē risinājumu, kā kolekciju ar projektiem, kur katrs projekts ir kolekcija ar dokumentiem. Darba vide ir sasaistīta ar resursdatora vidi, kas nepārtraukti mainās lietotāja ietekmē.

Darba vides saskarne piedāvā pieeju tekošajam risinājuma modelim. Resursdatora izmaiņas rezultātā, darba vide apziņo par izmaiņām tajā, atjaunojot `CurrentSolution` atribūtu. Piemēram, kad lietotājs veic izmaiņas kādā no teksta dokumentiem projektā, darba vide izmanto programmas notikumu, lai signalizētu par izmaiņām risinājumā kopējā modelī un konkrētā dokumentā. Lietotājs vai rīks tādējādi var reaģēt uz izmaiņām, analizēt jauno modeli, validēt tā pareizību un paziņot par iespējamām problēmām vai ieteikt, ko pamainīt – atkarīgs no tā, kādu funkciju konkrētais rīks pilda.

4.5.2. Risinājumi, projekti un dokumenti

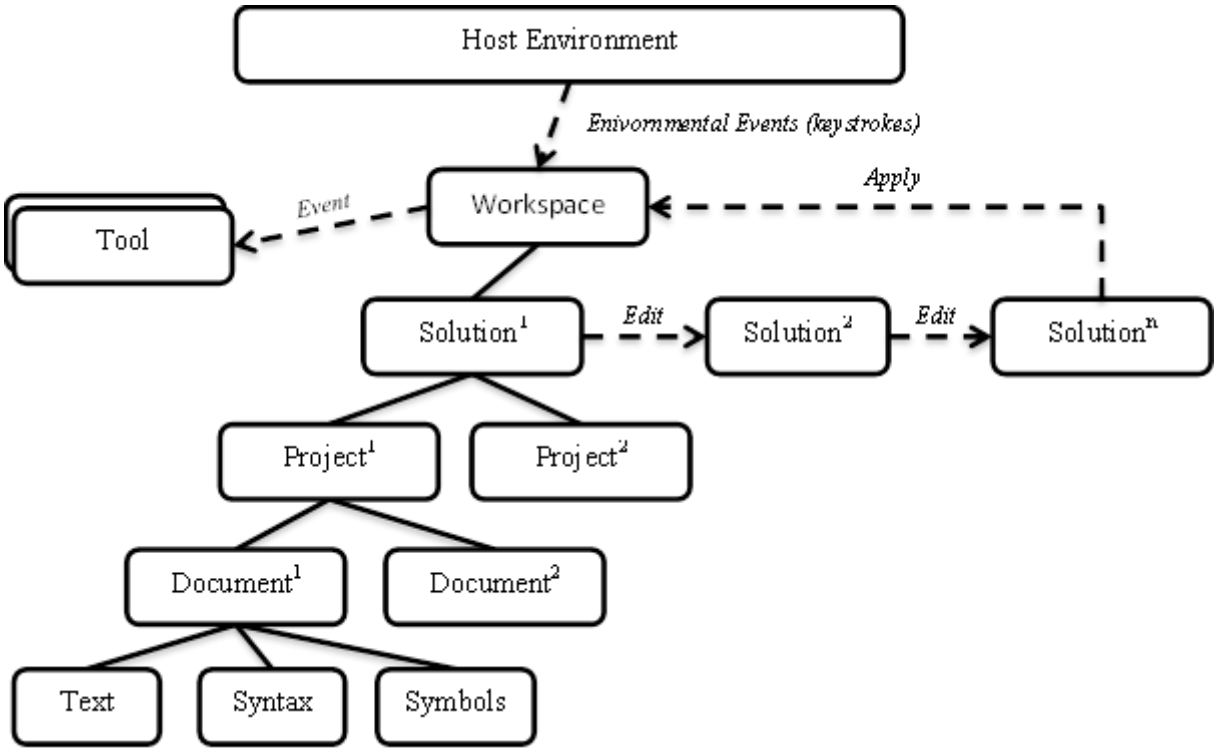
Lai gan risinājums var mainīties katru reizi, kad lietotājs nospiež taustiņu uz klaviatūras, Roslyn ļauj strādāt ar darba vides modeli izolācijā. Risinājuma modelis ir nemainīgs tajā iekļauto projektu un dokumentu momentuzņēmums. Līdzīgi kā ar sintakses kokiem, nemainīguma īpašība ļauj dažādiem procesiem vienlaicīgi strādāt ar modeli bez duplikācijas vai bloķēšanas. Pēc tam, kad risinājuma instance tiek iegūta no darba vides `CurrentSolution` atribūta, tā nekad nemainīsies. Izmaiņas tiek veiktas, veidojot jaunas instances ar veiktajām izmaiņām un piemērojot jauno risinājuma modeli darba videi.

Projekts ir daļa no kopējā nemainīgā risinājuma modeļa. Projekta modelis reprezentē visus tajā ietveros pirmkoda teksta dokumentus, parsēšanas un kompilācijas iestatījumus, atsauces uz ārējām bibliotēkām un citiem tajā pašā risinājumā ietvertajiem projektiem.

Izmantojot projekta modeli, ir iespējams piekļūt attiecīgajai kompilācijai, nav nepieciešams manuāli noteikt projekta atkarības vai parsēt pirmkoda datnes.

Arī dokuments ir daļa no kopējā nemainīgā risinājuma modeļa. Dokuments reprezentē tieši vienu pirmkoda datni, no kura iespējams piekļūt programmas tekstam, sintakses kokam un semantiskajam modelim [32, 33, 34].

4.2. att. redzamā diagramma attēlo darba vides relāciju ar resursdatora vidi un rīkiem, kā arī to, kā tiek veiktas izmaiņas risinājumos.



4.2. att. **Darba vide [32]**

4.6. API apguve

Nodaļa aprakstīti Roslyn koncepti un darbības principi, bet viens no veiksmīga risinājuma izstrādes nosacījumiem ir arī praktiska Roslyn API apguve. Lai gan Roslyn nav oficiālas dokumentācijas, ir pieejama tīmekļa vietne ar Roslyn pirmkoda repozitoriju[19], kurā iespējams pārlūkot pieejamās klases, piemēram, `SyntaxNode` un redzēt tās implementāciju, pieejamās metodes un atribūtus, Iespējams arī pārvietoties pa pirmkodu, klikšķinot uz asociēto tipu nosaukumiem un tādā veidā vizuāli izpētot Roslyn API. 3. pielikumā redzams ekrānuzņēmums no Roslyn pirmkoda avota tīmekļa vietnes.

[19] http://source.roslyn.io

5. PIEDĀVĀTAIS RISINĀJUMS

Nodaļā aprakstīts autora piedāvātais risinājums augstā līmenī.

5.1. Lietojuma scenāriji

Darba 2. nodaļā, autors vērš uzmanību uz bieži sastopamām problēmām, kas saistītas ar informācijas duplikāciju servera-klienta tīmekļa risinājumos. Izejot no minētajām problēmām, ir identificēti un aprakstīti konkrēti lietojuma scenāriji, kuru izpildi ir plānots nodrošināt izstrādātajai programmatūrai. Nodaļā aprakstīti katra scenārija darbības principi augstā līmenī, ieejas un izejas informācija, izmantotās datu struktūras un metodes plānotā rezultāta sasniegšanai. Katrs no scenārijiem nav vispārīgs un konkrēti definē atbalstīto funkcionalitāšu kopu.

5.1.1. Datu modeļu ģenerēšana

Modeļu sinhronizācija starp klienta un servera lietotnēm ir primārā informācijas duplikācijas problēma teju ikvienam projektam, kas klienta lietotnes izstrādei izmanto TypeScript, tāpēc modeļu ģenerēšanas lietojumam ir augsta prioritāte risinājuma atbalstīto scenāriju sarakstā.

Modeļu ģenerēšanas lietojuma uzdevums ir identificēt visu interesējošo projekta datu modeļu kopu un ģenerēt tiem atbilstošās TypeScript datu modeļu klases. Pirmais aktuālais uzdevums ir identificēt visus datu modeļus, kas risinājumā tiek izmantoti datu apmaiņai ar klienta lietotni. Līdzīgi risinājumi parasti ļauj izstrādātājam pašam identificēt visus nepieciešamos datu modeļus ar klases atribūtu, tādējādi analizējamajai lietotnei radot atkarību no modeļu ģenerēšanas risinājuma. Datu modeļu strukturēšana risinājumā ir pilnībā ir atkarīga no izstrādātāja, taču parasti izstrādātāji visus modeļus loģiski apvieno zem kāda noteikta projekta, C# vārdu telpas vai mapes datņu sistēmā, tāpēc iespējams būtu saprātīgi ļaut izstrādātājam veikt augstāka līmeņa konfigurāciju un norādīt attiecīgo konteineru. Arī šim risinājumam ir savi trūkumi – servera API var atgriezt datu tipus, kas dažādu iemeslu dēl nav definēti kopā ar visiem datu modeļiem (un tiem arī tur nevajag tikt definētiem). Kā piemēru var minēt vienkāršu metodi, kas atgriež sarakstu ar atslēgas un vērtības pāriem (.NET klašu bibliotēkas datu tips `KeyValuePair<TKey, TValue>`) priekš kāda klasifikatora. Tipam ir iespējams un ir nepieciešams definēt TypeScript modeli, taču nepieciešamības faktu nav iespējams zināt, neapskatot analizējamās lietotnes tīmekļa saskarnes. Tas mūs noved pie nākamā un potenciāli pilnīgākā risinājuma – analizēt lietotnes tīmekļa saskarņu ieejas argumentus un atgriežamos tipus (tie arī būs visi tie datu modeļi, kas potenciāli var tik sūtīti

vai saņemti no klienta lietotnes), un izejot no šīs analīzes rezultātiem identificēt interesējošos datu modeļus.

Vienkāršiem datu modeļiem, uzdevums ir pietiekoši triviāls, ja tas tiek uzdots izstrādātājam ar labām Roslyn API zināšanām. Tomēr pilnvērtīgam risinājumam, kas spētu analizēt kompleksus datu modeļus, nepieciešams apstrādāt vairākus, netriviālus gadījumus. Detalizētāk šādi gadījumi apskatīti sekojošajā apakšpunktā.

Atbalstītās C# struktūras, koncepti, datu tipi.

- Klases (*reference types*) un struct konstrukcijas (*value types*).
- Klašu atribūti un lauki.
- CLR iebūvētie datu tipi (`int`, `bool`, `decimal`, `double`, `string`).
- Kompleksie datu tipi un atkarības starp datu modeļiem.
- Enumerācijas.
- Masīvi un kolekcijas – .NET realizē daudz dažādus kolekciju tipus un saskarnes, kas tos implementē (arī `string` datu tips teorētiski ir kolekcija ar simboliem).
- `Nullable` tipi.
- Mantošana.
- *Generics*.
- Saskarnes.

Katram no minētajiem C# konceptiem ir savs, atbilstošs TypeScript analogs (vai vismaz semantiski līdzīgs).

5.1.2. Tīmekļa API klienta ģenerēšana

Tīmekļa API klienta ģenerēšanas lietojuma uzdevums ir identificēt visas lietotnes publiskās metodes, kas sasniedzamas caur tīmekļa protokoliem un ģenerēt tām atbilstošas TypeScript servisu klases. Līdzīgi kā modeļu ģenerēšanas uzdevumā, primāri ir nepieciešams identificēt tās klases, kuru metodes ASP.NET kontekstā tiek izsauktas atbildot uz HTTP pieprasījumiem. ASP.NET būtiski atvieglo šo uzdevumu, jo visām klasēm, kas atbild par tīmekļa pieprasījumiem nepieciešams dot nosaukumu, kas beidzas ar vārdu `Controller` (piemēram, `PersonController`), kā arī tām ir jāmanto no ASP.NET definētās klases ar nosaukumu `Controller` (vai `ApiController`, atkarīgs no .NET versijas).

Tālāk izejot no identificēto kontrolieru kopas, katrai kontroliera metodei ir vairākas būtiskas īpašības, kuru analīze ir nepieciešama analoģiska TypeScript servisa ģenerēšanai. Uzskatāmībai dots vienkārša ASP.NET tīmekļa API kontroliera un analoģiska TypeScript servisa piemērs. C# kontroliera fragments redzams 5.1. att.

```
[RoutePrefix("api/students")]
public class StudentsController : ApiController
{
    // GET api/students/5
    [HttpGet]
    [Route("{id}")]
    public Student Get(long id)
    {
        return new Student()
        {
            Id = 1,
            FirstName = "John",
            LastName = "Wick",
            DateOfBirth = new DateTime(1980, 1, 1),
            YearOfGraduation = DateTime.Now
        };
    }

    // POST api/students
    [HttpPost]
    public int Post([FromBody]Student student) { }
}
```

5.1. att. **C# kontrolieris**

Analoģiska Angular servisa fragments TypeScript programmēšanas valodā redzams 5.2. att.

```
@Injectable()
export class StudentsService {
    private baseUrl = 'api/students';

    constructor(private http: Http) { }

    get(id: number): Promise<Student> {
        return this.http.get(this.baseUrl + ('/' + id))
          .toPromise()
          .then((response: Response) => response.json())
          .catch(this.handleError);
    }

    post(student: Student): Promise<void> {
        return this.http.post(this.baseUrl, student)
          .toPromise()
          .then((response: Response) => response.json())
          .catch(this.handleError);
    }

    handleError(error: any): Promise<any> {
        return Promise.reject(error.message || error);
    }
}
```

5.2. att. **Angular specifisks API klients**

Scenārija risinājums balstās uz pieņēmumu, ka katram kontrolierim atbilst viens serviss klienta pusē. Pēc REST specifikācijas tā tam arī vajadzētu būt, ka kontrolieris atbild par kādu noteiktu resursu vai kolekciju (šai gadījumā studentu kolekcija) un iekapsulē operācijas ar attiecīgo resursu, piemēram, GET, POST, PUT, DELETE, kas atbilst CRUD (*Create, Read, Update, Delete*) operācijām. No tā seko, ka klienta pusē ir kāda komponente vai modulis, kas atbild par operācijām ar studentu entītiju un serviss, kas izgūst vai sūta datus uz serveri. Kādas konkrētas implementācijas nolūkos, izstrādātājs var izvēlēties citu pieeju kontrolieru un servisu strukturēšanai, taču šī uzdevuma kontekstā apskatīt tādus gadījumus nav paredzēts.

Nepieciešamā funkcionalitāte, kas jāatbalsta risinājumam, lai spētu korekti analizēt ASP.NET kontrolierus un ģenerēt tiem atbilstošus Angular servisus.

- Kontrolieri un metodes – primāri interesē kontroliera un visu publisko metožu nosaukumi.
- Atribūti – kontroliera klasei un metodei var būt vairāki atribūti, kas sevī nes būtisku informāciju, piemēram, tīmekļa adreses fragmentu un HTTP metodes nosaukumu.
- ASP.NET maršrutēšana – maršrutēšanas informācija var tikt saturēta gan, C# atribūtos, gan lietotnes līmeņa konfigurācijā, nepieciešams izveidot nokodēt zināšanas par ASP.NET maršrutēšanu un attiecīgi analizēt servera puses risinājumu.
- Argumenti – tīmekļa metodes var saturēt dažāda tipa argumentus, gan primitīvos, gan kompleksos tipus. Arī metožu argumenti var tikt apzīmēti ar C# atribūtiem, kas var būt būtiski klienta puses lietotnei un veidam, kā tā veido pieprasījumu.
- Atgriežamie tipi – atgriežamie tipi var tikt definēti tieši metodes parakstā, vai netieši – metodes deklarācijas blokā.
- Drošība – atkarīgs no konkrētā lietojuma, bet parasti servera puses risinājumi tiek nodrošināti un sniedz atbildes tikai uz autorizētiem pieprasījumiem. Autorizācijas informāciju parasti iekļauj HTTP ziņojuma galvenē. Ģenerētajam TypeScript pirmkodam jāpieļauj iespēja atbalstīt klienta puses drošības implementācijas.

5.1.3. Validāciju nosacījumu ģenerēšana

Validāciju nosacījumu ģenerēšanas lietojuma uzdevums ir analizēt veidus kā servera puses lietotnē tiek validēti no klienta saņemtie datu modeļi un ģenerēt Angular komponentes, kas realizē datu modeļu validāciju klienta pusē ar ekvivalentiem validācijas nosacījumiem.

C# programmēšanas valodā datu modeļu validāciju var realizēt daudz dažādos patvaļīgos veidos, bet šī scenārija kontekstā autoram interesē tikai tās validācijas pieejas, kas kādā veidā ļauj izstrādātājam definēt unikālus nosacījumus vienreiz un atkāroti izmantot tos visur, kur nepieciešams. Sekojošajā apakšpunktā aprakstītas dažas modeļu validācijas realizācijas un kā tās plānots analizēt.

Atribūtu validācija ir validācijas pieeja, kas .NET ekosistēmā pastāv jau labu laiku un tika popularizēta reizē ar ASP.NET MVC tehnoloģiju. .NET realizē pieeju kā definēt savas atribūtu validācijas klases, kā arī piedāvā vairākus, iebūvētus atribūtus, kas iekapsulē bieži sastopamu validācijas loģiku – lauks ir obligāts, teksta laukam ir garuma ierobežojumi, skaitliskai vērtībai ir intervāla ierobežojumi un tamlīdzīgi. Izstrādātāji var definēt arī savus atribūtus un tādā veidā anotēt savus datu modeļus – validācijas nosacījumi ir tieši piesaistīti datu modelim un tie ir viegli lasāmi [35].

Ir arī populāri trešās puses risinājumi kā piemēram *FluentValidation*[20]. Tā ir neliela .NET bibliotēka, kas ļauj būvēt validāciju nosacījumus ar lambda izteiksmēm. 5.3. att. redzams piemērs datu modeļa validācijai ar FluentValidation bibliotēku.

```
public class CustomerValidator: AbstractValidator<Customer> {
  public CustomerValidator() {
    RuleFor(customer => customer.Surname).NotEmpty();
    RuleFor(customer => customer.Forename).NotEmpty().WithMessage("Please specify a first name");
    RuleFor(customer => customer.Discount).NotEqual(0).When(customer => customer.HasDiscount);
    RuleFor(customer => customer.Address).Length(20, 250);
    RuleFor(customer => customer.Postcode).Must(BeAValidPostcode).WithMessage("Please specify a valid postcode");
  }

  private bool BeAValidPostcode(string postcode) {}
}

Customer customer = new Customer();
CustomerValidator validator = new CustomerValidator();
ValidationResult results = validator.Validate(customer);

bool validationSucceeded = results.IsValid;
IList<ValidationFailure> failures = results.Errors;
```

5.3. att. **Datu modeļa validācija ar FluentValidation**

Angular iespējams definēt dinamiskas datu ievades formas un to validācijas nosacījumus iekš TypeScript pirmkoda. Tas ļauj izolēt validācijas komponentes atsevišķās klasēs un izmantot visur, kur nepieciešams.

5.1.4. Projekta šablona ģenerēšana

Projekta šablona ģenerēšanas lietojuma uzdevums ir apvienot visus iepriekšminētos lietojumus un ģenerēt veselas Angular projekta veidnes. Scenārijs ir noderīgs gadījumos, kad izstrādes modelis no sākuma paredz servera puses lietotnes izstrādi un klienta lietotne seko

[20] https://github.com/JeremySkinner/FluentValidation

tikai pēc tam. Iespējams vēl atbilstošāks ir scenārijs, kurā tīmekļa sistēmai tiek atjaunota klienta lietotne – AngularJS lietotne tiek aizstāta ar Angular 2+ lietotni. Abos gadījumos ir ērti, ja iespējams iegūt jau gatavu projekta šablonu ar definētu komponenšu hierarhiju, datu modeļiem un servisiem.

Nepieciešams atbalstīt Angular CLI izmantošana, Angular komponentu ģenerēšanu. un konfigurācijas ģenerēšanu no servera.

5.1.5. Sinhronizācija

Sinhronizācijas lietojuma scenārijs apvieno datu modeļu, tīmekļa API klientu un validāciju nosacījumu scenārijus un "sinhronizē" tīmekļa un klienta risinājumus. Šāds lietojums tiktu izmantots visa izstrādes cikla laikā, bez nepieciešamības atsevišķi darbināt katru no iepriekšminētajiem lietojumiem.

5.2. Lietošana

Risinājuma lietošanai tiks izstrādāta komandrindas saskarne ar konfigurācijas iespējām, kas ļautu to izmantot Windows vidē neatkarīgi no izmantotās izstrādes vides. Iespējams izveidot Visual Studio arī paplašinājumu, kas ļautu rīku izmantot tieši no Visual Studio izstrādes vides.

5.3. Ieguvumi

Pirmais un lielākais ieguvums, protams, ir izstrādātāju produktivitātes palielināšana. Izmantojot piedāvāto risinājumu, izstrādātājiem nebūtu jāpavada laiks rakstot klienta puses pirmkodu, kas atkarīgs no servera puses risinājuma, tā vietā koncentrējoties uz lietotnes specifisko biznesa loģikas implementāciju. Darba autors ikdienā nodarbojas ar ASP.NET un Angular projektu izstrādi, kur darbā apskatītā problēma ir aktuāla un piedāvātā risinājuma izmantošana būtu ļoti noderīga projektos iesaistītajiem izstrādātājiem.

6. REALIZĀCIJAS PIEEJA UN APRAKSTS

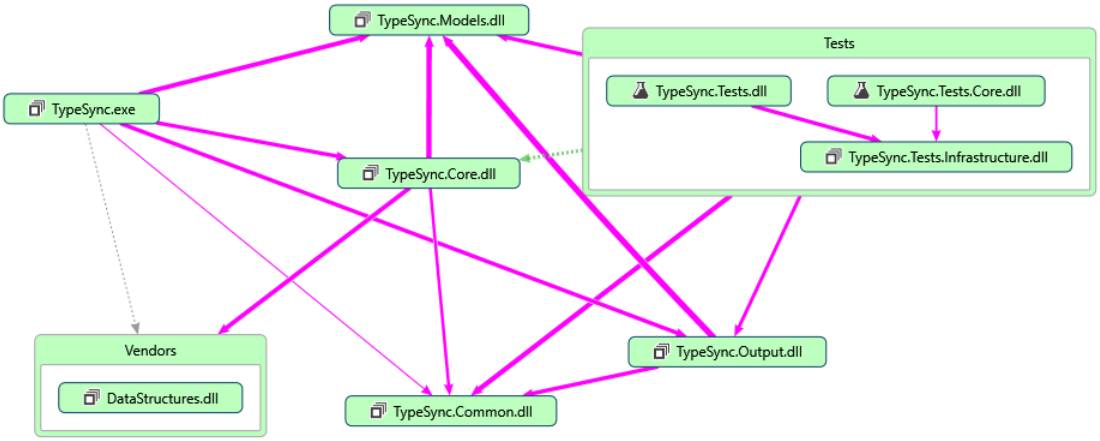
Nodaļā aprakstīts risinājums kopumā, risinājuma dažādās komponentes, to izstrādes principi un problēmas, ko tās risina. Izstrādātā risinājuma nosaukums ir *TypeSync* (no angļu valodas vārdiem *type synchronization*). Nosaukums tiek pieminēts darba tekstā un parādās dažādos attēlos, piemēram, klašu diagrammās. Termini komponente un modulis šajā nodaļā ir savstarpēji apmaināmi.

6.1. Arhitektūras pārskats

Risinājums sastāv no četrām loģiski nodalītām komponentēm, kur katra no tām atbild par noteiktu funkciju apgabalu. Risinājuma komponentes ir sekojošas (iekavās dots komponentes projekta nosaukums risinājumā):

- Komandrindas saskarne (*TypeSync.exe*) – rīka ieejas punkts, apstrādā un validē ieejas parametrus, organizē pārējo risinājuma komponenšu izmantošanu un veic paziņojumu izvadīšanu lietotājam (konsolē un teksta datnē).
- Analīzes komponente (*TypeSync.Core*) – veic .NET risinājumu analīzi ar Roslyn konkrēta lietojuma scenārija kontekstā un modelē nepieciešamos C# konceptus.
- Modeļu komponente (*TypeSync.Models*) – satur nepieciešamos C# un TypeScript modeļus un veic konvertāciju starp tiem.
- Pirmkoda ģenerēšanas un izvades komponente (*TypeSync.Output*) – ģenerē TypeScript komponentes un izvada pirmkoda datnes uz diska.

Papildus četrām pamata komponentēm, risinājumā iekļauti arī izmantoto datu struktūru (*Vendors/DataStructures*), testu (*Tests/TypeSync.Tests.**) un kopīgo funkciju/datu (*TypeSync.Common*) bibliotēkas. 6.1. att. redzamas attiecības starp risinājuma komponentēm.

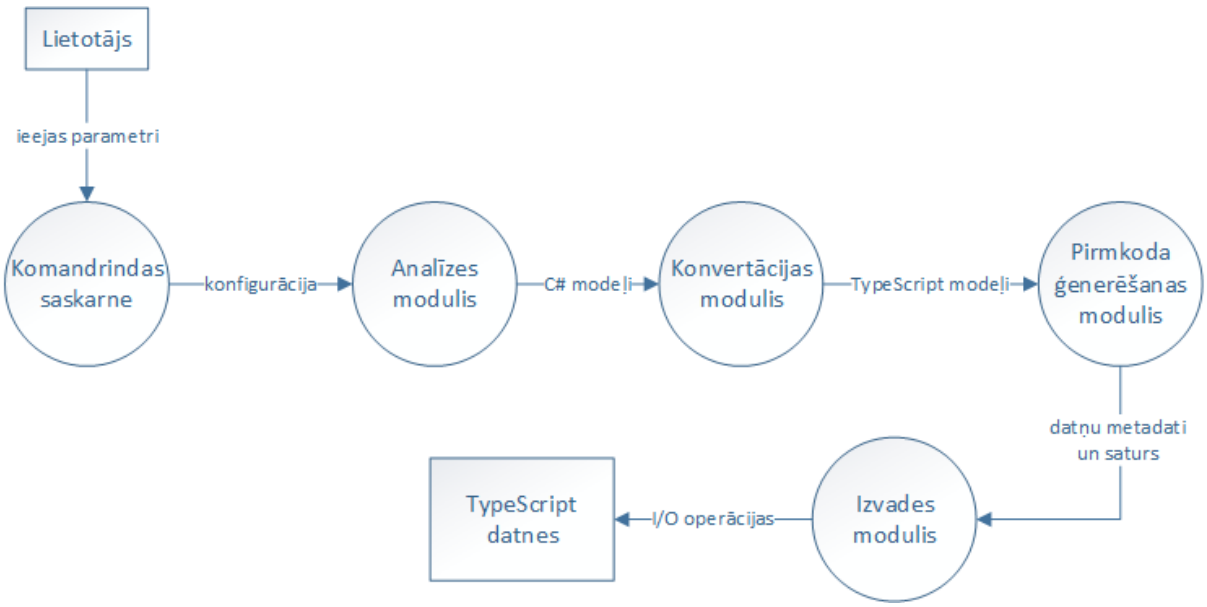


6.1. att. **Risinājuma komponentes un to atkarības**

Piemērs programmatūras darbībai:

1. Lietotājs iniciē programmas darbību, parametros nododot lietojuma scenāriju, ceļus uz analīzes vienumu (risinājums, projekts vai datne) un izvadīto rezultātu, papildus konfigurāciju.
2. Ieejas parametri tiek validēti, atkarībā no izvēlētā lietojuma scenārija, tiek izveidotas nepieciešamās komponenšu instances.
3. Ar Roslyn API palīdzību, tiek kompilēts un analizēts padotais analīzes vienums. Analīzes rezultāts ir modeļu kopa, kas modelē nepieciešamos C# konceptus priekš konkrētā scenārija.
4. Izveidotie modeļi tiek kartēti uz analoģiskiem TypeScript modeļiem. Daļa C# un TypeScript konstrukciju ir ekvivalentas, daļai eksistē semantiski līdzīgi koncepti abās programmēšanas valodās.
5. No TypeScript modeļiem tiek ģenerēts TypeScript pirmkods.
6. Ģenerētais pirmkods tiek izvadīts uz diska.

6.2. att. redzama risinājuma datu plūsmas diagramma.



6.2. att. **Risinājuma datu plūsmas diagramma**

Risinājums izstrādāts, izmantojot Visual Studio 2017 un Visual Studio Code izstrādes vides. Visi moduļi izstrādāti, izmantojot C# programmēšanas valodu un .NET pilnā ietvara 4.6.2 versiju. Alternatīvs TypeScript pirmkoda ģenerēšanas risinājums ar TypeScript kompilatora API izstrādāts, izmantojot TypeScript valodas 2.3.2 versiju un Node.js ietvara 7.8.0 versiju.

Darba 1. nodaļā pieminēti vairāki .NET izpildlaiki ar dažādām funkcionalitātes un atbalstīto platformu iespējām. Risinājuma izstrādei tika izvēlēts pilnais .NET ietvars, jo Roslyn darba izstrādes laikā vēl neatbalsta `MSBuildWorkspace` klases izmantošanu .NET Core projektos un .NET pilnā ietvara projektu kompilācija, izmantojot .NET Core, nav iespējama. Risinājuma pirmkoda bāzi gan ir iespējams pārnest uz .NET Core risinājumu pēc tam, kad nepieciešamā funkcionalitāte tiks atbalstīta. Šāds solis ļautu risinājumu izmantot arī Linux un macOS platformu izstrādātājiem.

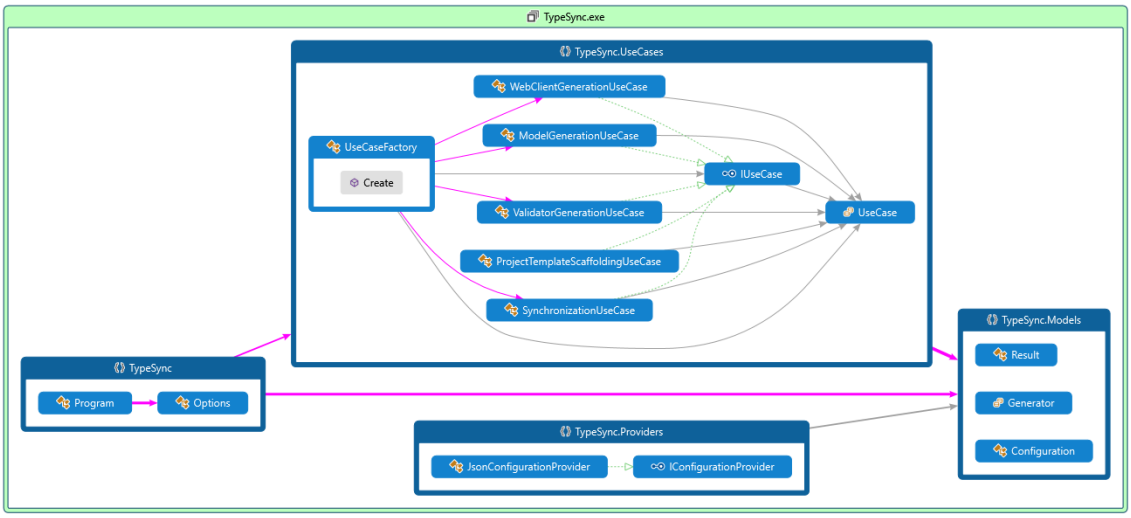
Risinājumā izmantota orientēta grafa struktūras implementācija. Grafs tiek izmantots, lai izveidotu klašu atkarību grafu datu modeļu lietojuma scenārijam. Tiek analizēti abstraktie sintakses koki.

6.2. Komandrindas saskarne un konfigurācija

Komandrindas saskarnes galvenais uzdevums ir saziņa ar lietotāju. Konsoles lietotne ir pietiekoši dinamisks risinājums, tas nav atkarīgs no specifiskas izstrādes vides (un potenciāli arī no platformas). Rīka izmantošanu var automatizēt pēc ieskatiem (piemēram, uzraugot projekta datnes un iniciējot rīka darbību, kad kāda no tām mainās), izmantojot *PowerShell* vai *Command Prompt* tehnoloģijas. Kā alternatīvs risinājums komandrindas saskarnei tika

izskatīts Visual Studio paplašinājuma izstrāde. Tādā veidā būtu iespējams iniciēt rīka darbību tieši no izstrādes vides, taču tas izveidotu ciešu atkarību no Visual Studio izstrādes vides un priekš izstrādātājiem, kas izmanto Visual Studio Code (vai vēl ko citu), nāktos izstrādāt papildus paplašinājumus.

Komandrindas saskarnes moduļa galvenais uzdevums ir nodrošināt vadības moduļa funkciju un orķestrēt visu pārējo sistēmas moduļu darbību. Komandrindas saskarne ir vienīgais projekts sistēmā, kam ir zināšanas par lietojuma scenārijiem un, kas ir atkarīgs no citiem sistēmas moduļiem. 6.3. att. redzamā diagramma ilustrē komandrindas saskarnes projektā iekļautās klases un to atkarības.



6.3. att. **Komandrindas saskarnes moduļa klases un to atkarības**

Komandrindas saskarnei ir vairāki ieejas parametri:

- lietojuma scenārijs, [*-u*] [*--usecase*], iespējamās vērtības:
  - *ModelGeneration,*
  - *WebClientGeneration,*
  - *ValidatatorGeneration,*
  - *ProjectTemplateScaffolding,*
  - *Synchronization;*
- analīzes risinājums, [*-i*] [*--input*], iespējamās vērtības:
  - absolūts vai relatīvs ceļš uz analīzes vienumu (C# datne, projekts vai risinājums) datņu sistēmā;
- izvades direktorija, [*-o*] [*--output*], iespējamās vērtības:
  - absolūts vai relatīvs ceļš uz direktoriju datņu sistēmā (tā tiks izveidota, ja neeksistē);

- TypeScript pirmkoda ģenerēšanas risinājums (tādi ir vairāki, detalizētāk par to stāstīts 6.5. sadaļā), [*-g*] [*--generator*], iespējamās vērtības:
  - *Template*,
  - *Compiler*.

Izmantošanas piemēri:

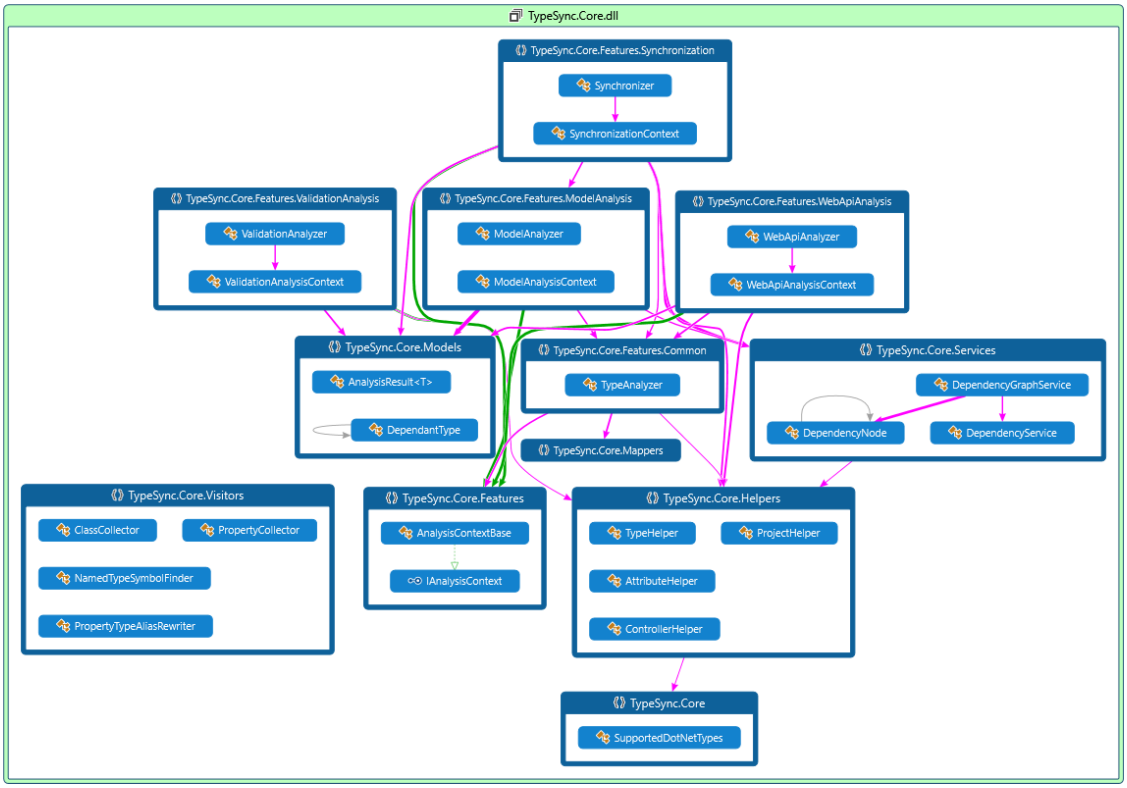
```
TypeSync.exe -usecase ModelGeneration -input
Samples.DotNetFull.ViewModels.csproj -output C:\Dev\Generated -generator
Template

TypeSync.exe -u ModelGeneration -i Samples.DotNetFull.ViewModels.csproj -o
C:\Dev\Generated -g Template
```

Dažādiem projektiem un izstrādātājiem atbilst dažādas pirmkoda strukturēšanas un formatēšanas prakses. Katru lietojuma scenāriju iespējams individuāli konfigurēt ar JSON datnes palīdzību. Iespējamie konfigurācijas vienumi aprakstīti zemāk.

- Nosaukumu formāts – datņu, klašu, funkciju un mainīgo nosaukumu formāti parasti izraisa plašas debates. TypeScript un Angular ir izveidotas vadlīnijas, kas sniedz subjektīvu viedokli šajā jautājumā, taču vislabāk ir atstāt šo jautājumu izstrādātāja ziņā [36].

6.3. .NET risinājumu analīze

Analīzes moduļa galvenais uzdevums ir izmantot Roslyn API, iegūt nepieciešamo informāciju priekš pirmkoda ģenerēšanas un atgriezt C# modeļus. Roslyn pielietojums sastāda pārliecinoši lielāko darba praktiskās daļas apjomu. Katram lietojuma scenārijam ir virkne analizatoru, kas analizē risinājuma sintaktiskās un semantiskās šķautnes, lai iegūtu TypeScript pirmkoda ģenerēšanai nepieciešamo informāciju. 6.4. att. redzamā diagramma ilustrē analīzes projektā iekļautās klases un to atkarības.

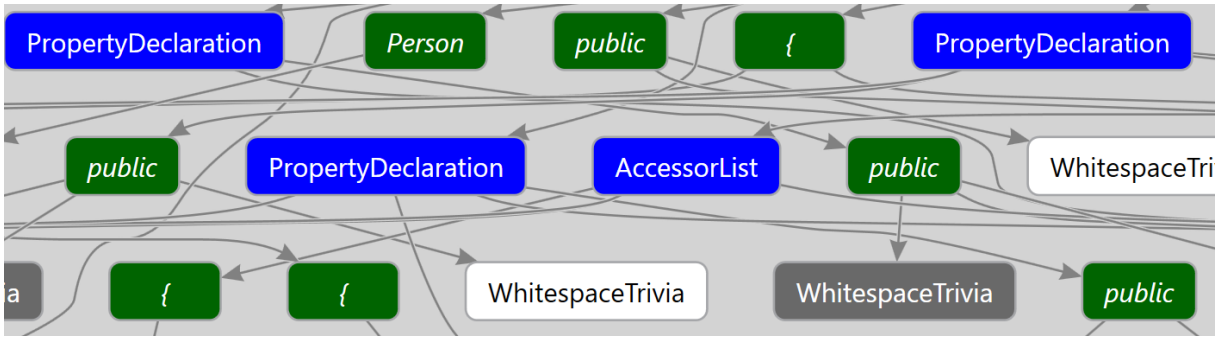


6.4. att. **Analīzes moduļa klases un to atkarības**

Risinājums atbalsta .NET pilnā ietvara un .NET Core projektu analīzi – tiem ir savas atšķirības ASP.NET iespēju realizācijā.

6.3.1. Sintaktiskā analīze

6.5. att. redzama ar Roslyn iegūta C# datu modeļa deklarācijas sintakses koka attēlojums (orientēts grafs). Diagrammu iespējams iegūt ar Visual Studio rīku *Syntax Visualizer*.



6.5. att. **C# sintakses koka fragmenta vizualizācija**

Katrs Roslyn sintakses koks visā pilnībā attēlo to pirmkoda tekstu, no kura tas tika izveidots. Attiecīgi izveidojot sintakses koku un pēc tam to apstaigājot ar *postorder* algoritmu un izdrukājot katru tekstvienību (zaļos mezglus), iespējams iegūt identisku pirmkoda tekstu

(ieskaitot visus tukšuma simbolus un komentārus). Tas nepieciešams, lai būtu iespējams tieši rediģēt sintakses kokus un iegūt atbilstoši rediģētu pirmkodu, nezaudējot komentārus un teksta formatējumu.

Zilie mezgli reprezentē sintaktiskas konstrukcijas, piemēram, klases deklarāciju, atribūta deklarāciju vai piešķiršanas izteiksmi. Zaļie mezgli reprezentē visas teksta vienības, kuras tieši sastopamas pirmkodā. Pelēkie mezgli reprezentē komentārus un tukšuma simbolus. Ir vairāki veidi, kā iespējams iegūt informāciju no sintakses koka:

1. manuāli apstaigāt visus koka mezglus un izdarīt secinājumus balstoties uz mezgla `SyntaxKind` atribūta vērtību;
2. izmantot LINQ, lai atlasītu noteikta tipa mezglus, piemērs klases deklarāciju izgūšanai: `tree.GetRoot().DescendantNodes().OfType<ClassDeclarationSyntax>().ToList();`
3. izmantot *Visitor* pieeju un implementēt metodi, kas rekursīvi apmeklē interesējošo sintakses elementu, piemēram klases deklarāciju, kā redzams 6.6. att.:

```
public class ClassCollector : CSharpSyntaxWalker
{
    public readonly List<ClassDeclarationSyntax> Classes = new List<ClassDeclarationSyntax>();

    public override void VisitClassDeclaration(ClassDeclarationSyntax node)
    {
        Classes.Add(node);

        base.VisitClassDeclaration(node);
    }
}
```

6.6. att. ***CSharpSyntaxWalker* implementācija**

Strādājot ar pirmkodu sintaktiskā līmenī, ir ērti iegūt metožu vai atribūtu implementācijas detaļas strukturētā veidā – viennozīmīgi daudz ērtāk kā parsēt CIL pirmkodu vai dekompilēt klašu bibliotēkas un mēģināt izgūt funkciju blokus no tām.

Autora risinājumā sintaktiskā informācija tiek izmantota manuāli implementētu validācijas funkciju apstrādei.

6.3.2. Semantiskā analīze

Sintakses kokiem ir informācija tikai par sevi pašu, tie nezina neko par citiem sintakses kokiem un savas attiecības ar tiem. Sintaktiskā līmenī iespējams identificēt klašu un lauku deklarācijas, taču nav iespējams viennozīmīgi noteikt, kur kāds pirmkoda fragments atsaucas

uz tām. Iespējams identificēt izsaukumu izteiksmes, bet nevar noteikt, kas tiek izsaukts. Piemēram, `System.Console.WriteLine()`, nav iespējams noteikt, vai `System` ir vārdu telpa vai klase, jo no sintaktiski tas varētu būt jebkurš no abiem. Darba kontekstā ir svarīgas atbildes uz jautājumiem, kas sintakses kokam ir sveši – kā minimums nepieciešams identificēt klašu atkarības un piekļūt .NET iebūvēto tipu publiskajiem laukiem, piemēram, `KeyValuePair<TKey, TValue>`.

Meklētā informācija atrodas lietotnes semantiskajā līmenī un Roslyn piedāvā funkcionalitāti, kas ļauj strādāt ar lietotnes semantisko modeli. Semantiskā līmeņa API izmantošana izmanto vairāk resursus, jo nepieciešams veikt kompilāciju analizējamajam risinājumam. Semantiskā līmeņa API ļauj piekļūt lietotnes simbolu informācijai. Kā tika minēts 4. nodaļā, simboli reprezentē visu, ko kompilators zina par unikāliem C# elementiem – tipiem, metodēm, laukiem u.t.t. Lai piekļūtu simboliem, no sākuma ir nepieciešams izveidot kompilācijas objektu. Parasti kompilācija asociējas ar procesu, kura rezultātā uz diska tiek izvadīta izpildāma datne (*.exe*) vai klašu bibliotēka (*.dll*). Šajā gadījumā kompilācijas objekts vienkārši reprezentē visu, ko Roslyn zina par kompilējamo vienību – iekļautie sintakses koki, references uz nepieciešamajām bibliotēkām u.c. Kompilāciju iespējams izveidot manuāli, iekļaujot nepieciešamos sintakses kokus un references, sk. 6.7. att.

```
var text = File.ReadAllText(path);
var tree = CSharpSyntaxTree.ParseText(text).WithFilePath(path);
var mscorlib = MetadataReference.CreateFromFile(typeof(object).Assembly.Location);

_compilation = CSharpCompilation.Create(
    "MyCompilation",
    syntaxTrees: new[] { tree },
    references: new[] { mscorlib }
);
```

6.7. att. **Roslyn manuālas kompilācijas izveide**

Parasti gan nepieciešams analizēt gatavu C# projektu un tādā gadījumā Roslyn piedāvā `MSBuildWorkspace` klasi, kas ļauj "atvērt" projektus/risinājumus un iegūt kompilācijas objektus no tiem, izmantojot MSBuild, sk. 6.8. att.

```
var workspace = MSBuildWorkspace.Create();
var project = _workspace.OpenProjectAsync(path).Result;
var compilation = project.GetCompilationAsync().Result;
```

6.8. att. **Roslyn kompilācijas iegūšana no .NET risinājuma**

Tālāk no kompilācijas iespējams iegūt semantisko modeli specifiskam sintakses kokam, `compilation.GetSemanticModel(syntaxTree)`. Semantiskais modelis kalpo par

tiltu starp simbolu un sintakses līmeņiem. Šim nolūkam semantiskais modelis piedāvā divas metodes, `GetDeclaredSymbol` un `GetSymbolInfo`. Abas metodes pieņem sintakses mezglu no padotā sintakses koka un atgriež attiecīgo simbolu, sk. 6.9. att.

```
var classNode = root.DescendantNodes().OfType<ClassDeclarationSyntax>().First();
var classSymbol = semanticModel.GetDeclaredSymbol(classNode) as INamedTypeSymbol;
```

6.9. att. **Klases simbola iegūšana**

Alternatīvi, sintakses kokus var neizmantot nemaz un manuāli izgūt informāciju par visiem kompilācijas vienībā esošajiem simboliem. To nepieciešams darīt, ja sintakses koki nav pieejami (ārēja bibliotēka). Līdzīgi kā ar sintakses mezgliem, arī simboliem ir specifiskas implementācijas, kas manto no `ISymbol` klases kā piemēram `IMethodSymbol` vai `INamedTypeSymbol`. Specifiskus simbolus iespējams apmeklēt ar *Visitor* pieeju. 6.10. att. redzams pirmkoda fragments klasei, kas izgūst visus tipus (klases) no kompilācijas.

```
private class NamedTypeSymbolVisitor : SymbolVisitor
{
    public List<INamedTypeSymbol> NamedTypeSymbols { get; } = new List<INamedTypeSymbol>();

    public override void VisitNamespace(INamespaceSymbol symbol)
    {
        Parallel.ForEach(symbol.GetMembers(), s => s.Accept(this));
    }

    public override void VisitNamedType(INamedTypeSymbol symbol)
    {
        NamedTypeSymbols.Add(symbol);

        foreach (var childSymbol in symbol.GetTypeMembers())
        {
            base.Visit(childSymbol);
        }
    }
}
```

6.10. att. ***SimbolVisitor*** **implementācija**

Autora risinājumā semantiskā informācija tiek ekstensīvi izmantota dažādu uzdevumu veikšanai:

- klašu atkarību grafa izveidei;
- dažādu faktu izgūšanai no klašu un to lauku simboliem – publisks, privāts, abstrakts, noslēgts, bāzes klases, implementētās saskarnes;
- tipu klasifikācijai – CLR tipiem ir sintaktiski aizstājvārdi, kas kompilācijas laikā reprezentē vienu tipu, simboli ievērojami atvieglo šādu tipu identifikāciju. .NET ir vairāki paņēmieni, kā implementēt kolekcijas, taču TypeScript tie tiek

reprezentēti, kā masīvi, tāpēc interesē tikai fakts, vai tips implementē `IEnumerable<T>` saskarni un semantiskais modelis ļauj viegli to noskaidrot.

- tipu parametru apstrādei;
- klašu un metožu atribūtu apstrādei;

6.3.3. Projektu analīze

Roslyn dod iespēju strādāt arī risinājumu līmenī (projektu kopa). Risinājumiem iespējams piekļūt izmantojot dažādas darba vides. Viena no tām ir jau pieminētā `MSBuildWorkspace`, kas saprot *.csproj* datnes un māk kompilēt projektus, izmantojot tās. Papildus tam ir pieejamas:

- `VisualStudioWorkspace` – ja risinājumu izplata, ka Visual Studio paplašinājumu.
- `AdHocWorkspace` – iespējams manuāli pievienot projektus un dokumentus.

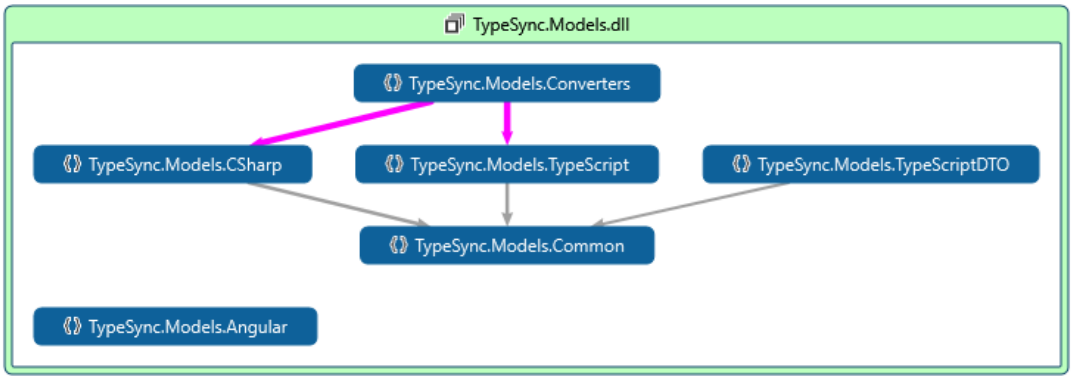
Iepriekš jau apskatīta .NET projektu struktūra – Roslyn ļauj pārlūkot to ērtā veidā, reizē piekļūstot katra vienuma metadatiem.

Autora risinājumā analīze projektu līmenī tiek izmantota, lai noteiktu atkarības starp projektiem, identificētu projektu .NET ietvara versiju un atlasītu dokumentus ar noteiktu nosaukumu vai tipu.

6.4. Modeļi un to konvertācija

Katra lietojuma scenārija vajadzībām tiek modelēti nepieciešamie C# un TypeScript objekti, kā arī izveidota kartēšana starp tiem. Modelēti tiek gan datu modeļu struktūra, lauki, to tipi, parametri, atkarības starp klasēm, kontrolieri un citi nepieciešamie koncepti.

Modelēšana tiek veikta dažādos līmeņos, sākot ar augsta līmeņa modeļiem, kas reprezentē tādus konceptus kā HTTP serviss un validācijas vienums, beidzot ar zema līmeņa modeļiem, kas ir tuvu TypeScript sintakses koka elementiem. 6.11. att. redzamā diagramma ilustrē modeļu projektā iekļautās klases un to atkarības.



6.11. att. **Modeļu klases**

Modelēšana ļauj stingri nodalīt risinājuma komponentes atsevišķos moduļos, kur tās savstarpēji sazinās tikai izmantojot tām paredzētos modeļus. Tādā veidā katram projektam ir stingri nodalītas atkarības un atbildības – *TypeSync.Core* ir vienīgais projekts, kas zina par Roslyn un *TypeSync.Output* ir vienīgais projekts, kas strādā ar pirmkoda ģenerēšanas risinājumiem.

6.5. Pirmkoda ģenerēšana un izvade

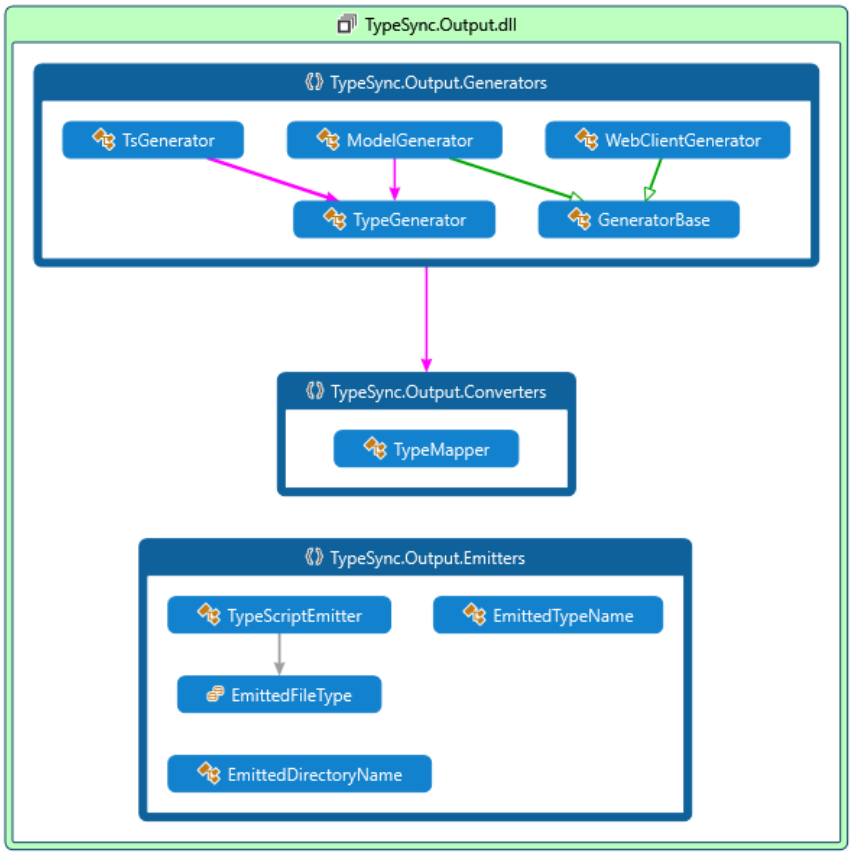
Pirmkoda ģeneratori izmanto TypeScript modeļus un TypeScript zināšanas, lai ģenerētu simbolu virknes ar TypeScript pirmkodu.

Pirmkoda ģenerēšanai izveidoti divi risinājumi:

1. autora izstrādāts pirmkoda ģenerēšanas risinājums, kas balstās uz simbolu virkņu šabloniem un to konkatenāciju;
2. autora izstrādāts risinājums ar iepriekš pieminēto TypeScript kompilatora API.

6.5.1. Veidnes

Veidņu bāzēta pirmkoda ģenerēšana jau vairākkārt pieminēta darba gaitā. Risinājums izmanto TypeScript modeļus un izdod TypeScript pirmkodu attiecīgi definētajai veidnei. 6.12. att. redzamā diagramma ilustrē pirmkoda ģenerēšanas un izvades projektā iekļautās klases un to atkarības.



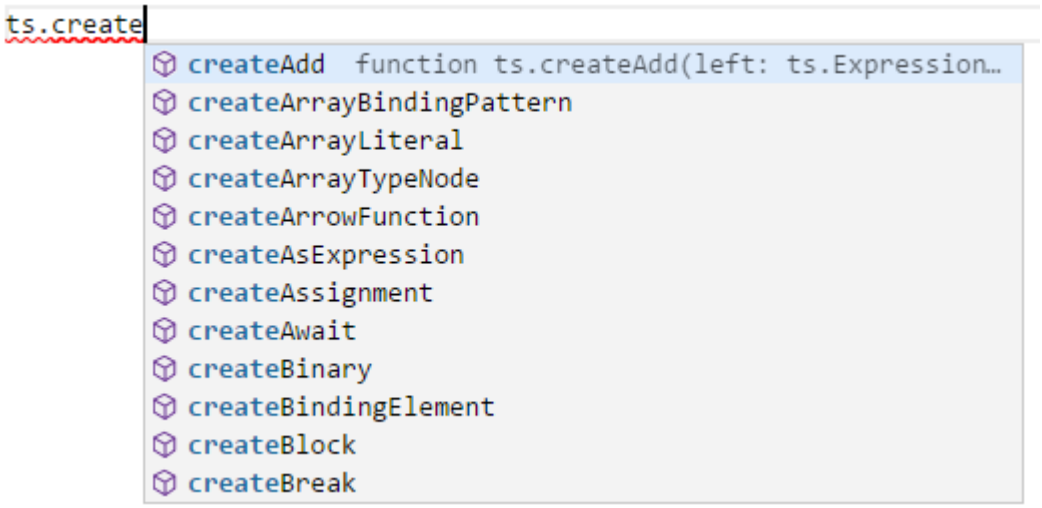
6.12. att. **Pirmkoda ģenerēšanas un izvades modulis**

6.5.2. TypeScript kompilatora API

Veidņu bāzēta pirmkoda ģenerēšana ir itin labi piemērota viegli paredzamu pirmkoda konstrukciju izveidei (datu modelis, HTTP klienta komponente), bet to ir grūti adaptēt kompleksas loģikas ģenerēšanai. Labāks risinājums būtu CodeDOM piedāvātā ideja – reprezentēt TypeScript pirmkodu strukturētā manierē un izstrādāt parseri, kas ļauj šo datu struktūru transformēt pirmkodā, nodrošinot visas nepieciešamās formatēšanas iespējas. Šādas pieejas galvenais izaicinājums ir korektas datu struktūras izveidošana, kas spētu modelēt visas nepieciešamās (un iespējamās) TypeScript pirmkoda konstrukcijas.

TypeScript kompilatora pirmkods jau sākotnēji tika piedāvāts kā atvērtā pirmkoda projekts, kas izstrādāts iekš TypeScript, taču ilgu laiku tam nebija publiski pieejama API pirmkoda ģenerēšanai. Bija iespējams apskatīt kā kompilators ir izveidots, kā darbojas parseria un skenera komponentes, kādas datu struktūras reprezentē dažādos TypeScript pirmkoda elementus, bet ne vairāk. Reaģējot uz izstrādātāju lūgumiem, Microsoft nolēma "atvērt" šīs zināšanas un pielāgot līdz šim tikai iekšēji izmantoto API publiskai lietošanai.

Rezultātā TypeScript kompilatora API ir pieejamas metodes dažādu sintaktisko konstrukciju izveidei. Līdzīgi kā Roslyn, datu struktūras, kuras iespējams izveidot ar šīm metodēm ir tās pašas datu struktūras, ko kompilators izmanto pirmkoda parsēšanai. Tāpēc izveidoti sintakses koki atbalsta visas tekošās TypeScript sintakses iespējas, un tām mainoties, iespējams viegli adaptēt risinājumu. Ja autors izvēlētos manuāli implementēt, ko līdzīgu, nāktos lieki dublēt visu to, kas jau implementēts TypeScript kompilatorā. 6.13. att. redzamas funkcijas TypeScript AST izveidei.



6.13. att. **TypeScript kompilatora API AST izveides funkcijas**

Rezultātā izstrādātas iestrādnes ietvaram, kas izmanto kompilatora API un ar kuru iespējams izveidot sintakses koku, kas atbilst nepieciešamās klases deklarācijai un izgūt pirmkoda tekstu, kas tam atbilst. 6. pielikumā redzams pirmkoda fragments klases deklarācijas izveidei ar TypeScript kompilatora API. TypeScript pirmkoda ģenerators noformēts kā Node.js lietotne, kas komunicē ar C# risinājumu caur HTTP REST saskarni. 4. pielikumā redzams, kā izskatās TypeScript sintakses koks datu modeļa klases deklarācijai.

Pēc katras datnes apstrādes, tā tiek izvadīta uz diska. Izvades process ir triviāls, jo visas nepieciešamās atkarības jau ir apstrādātas un moduļi importēti. Izvadītās datnes tiek grupētas pa atbilstošu datņu struktūru (ir būtiski, ka struktūra tiek saglabāta, jo atkarīgie TypeScript moduļi tiek importēti relatīvi pret datnes tekošo atrašanās vietu):

- *models;*
- *enums;*
- *services;*
- *validators.*

6.6. Sinhronizācijas lietojuma scenārijs

Detalizētāk aprakstīts sinhronizācijas lietojuma scenārijs, kas aptver lielāko daļu atbalstītās funkcionalitātes. Scenārija izpildei tiek veiktas sekojošas darbības:

1. atlasīt visas kontrolieru klases dotā risinājumā;
2. atlasīt visas publiskās metodes kontrolieros;
3. savākt unikālos atgriežamos un parametru tipus;
4. atlasīt ģenerējamos datu modeļus (apstrādājot ģenēriskos tipus), atmetot CLR tipus un neatbalstītos .NET tipus;
5. apstrādāt ģenerējamo tipu atkarības (5. pielikumā redzams pirmkoda fragments rekursīvi iegūta atkarību grafa izveidei), rezultātā apkopojot visus unikālos datu modeļus, kas tiek izmantoti komunikācijā starp serveri un klientu;
6. ģenerēt datu modeļus:
    a. apstrādāt bāzes tipus;
    b. apstrādāt ģenēriskus tipus;
    c. apstrādāt atkarības;
7. ģenerēt datu modeļu validācijas nosacījumus:
    a. apstrādāt iebūvētos validācijas atribūtus;
    b. apstrādāt definētos validācijas atribūtus;
8. ģenerēt tīmekļa API klientus:
    a. apstrādāt maršrutēšanas atribūtus;
    b. apstrādāt metožu parametrus un atgriežamos tipus.

7. REZULTĀTI

Veicot esošo risinājumu apskatu, autors izpētīja .NET rīkus pirmkoda analīzei un ģenerēšanai. `System.Reflection` API sniedz iespējas .NET pirmkoda metadatu analīzei CLR līmenī, taču Roslyn API piedāvā daudz plašākas iespējas C# specifiskas sintaktiskās un semantiskās informācijas izgūšanai. Pirmkoda ģenerēšanas kontekstā, .NET eksistē iespējas veidņu bāzētai pirmkoda teksta ģenerēšanai ar T4 rīku, CodeDOM iespējas pirmkoda struktūras definēšanai, kā arī citi .NET vai CLR specifiski risinājumi pirmkoda ģenerēšanai, kuri nav piemēroti TypeScript pirmkoda ģenerēšanai. Tika apskatīti populārākie publiski pieejamie risinājumi TypeScript pirmkoda ģenerēšanai no C# risinājumiem. Esošie risinājumi nespēja pilnībā piedāvāt nepieciešamo funkcionalitāti problēmas risināšanai un, izvērtējot katru no tiem, tika apstiprināta nepieciešamība pēc autora izstrādāta risinājuma.

Darba praktiskajā daļā definēts piedāvātais risinājums – nepieciešamie lietojuma scenāriji, kas veic noteiktas dublētas informācijas sinhronizāciju starp klienta un servera lietotnēm. Maģistra darba ietvaros realizēti (vai daļēji realizēti) četri no pieciem definētajiem lietojuma scenārijiem:

- datu modeļu ģenerēšana;
- tīmekļa API klienta ģenerēšana;
- validāciju ģenerēšana (veiktas iestrādnes šī lietojuma izstrādei);
- sinhronizācija.

Risinājums izstrādāts ar nodomu ērti paplašināt atbalstīto scenāriju un funkcionalitāšu kopu. Risinājuma arhitektūra ir stingri definēta – nepieciešams tikai implementēt attiecīgos pirmkoda analizatorus, modeļus un ģeneratorus. Pabeidzot implementēt pirmkoda ģenerēšanu ar TypeScript kompilatora API, būs iespējams veikt arī patvaļīgu C# pirmkoda bloku ģenerēšanu uz TypeScript.

Risinājuma pirmkods pieejams publiskā GitHub repozitorijā, kas atrodams tīmekļa vietnē[21]. Repozitorijā atrodamie vienumi:

- risinājuma pirmkods (.NET/C# un Node.js/TypeScript daļas);
- dokumentācija (instrukcijas, augsta līmeņa apraksts, diagrammu datnes, moduļu klašu atkarības);
- piemēri (servera un klienta puses risinājumu piemēri, lai nodemonstrētu dublētas informācijas vienumu sinhronizāciju starp tiem);
- skripti rīka darbināšanai (priekš piemēriem).

[21] https://github.com/kristersz/TypeSync

7.1. Salīdzinājums

Aprakstīti būtiskākie faktori, kas autora risinājumu padara labāku (uzturamāku, ērtāk realizējamu) par apskatītajiem, esošajiem risinājumiem.

Funkcionalitāte. Novērtējot eksistējošos risinājumus, rīki tika vērtēti vairākos kritērijos un neviens no tiem neatbilda pilnībā neapmierināja visus no tiem. Autora risinājums spēj ģenerēt datu modeļus, API klientus un Angular specifiskus validāciju nosacījumus (tuvākajā laikā plānots implementēt) no ASP.NET risinājumiem. Neviens no apskatītajiem risinājumiem neatbalsta Angular validācijas direktīvu ģenerēšanu. Tāpat apskatītie risinājumi pieprasa kaut kāda veidā identificēt ģenerējamos C# vienumus. Autora risinājums iziet no tīmekļa saskarnes perspektīvas un automātiski spēj identificēt HTTP metodes, kas jāizsauc, datu modeļus, kas tām jānodod un validācijas nosacījumus, kas validē datu modeļu atribūtus. Tādējādi nekāda papildus konfigurācija nav jāveic – norādot ieejas un izejas risinājumus, iespējams sinhronizēt visu dublēto informāciju.

Izmantotās tehnoloģijas. Programmēšanas valodas kompilatoram ir dziļas un pilnīgas zināšanas par attiecīgo programmēšanas valodu. Kompilatora transformācija par kompilatora servisu (platformu) ir milzīga iespēja izstrādātājiem, kas vēlas izstrādāt meta-programmēšanas rīkus ar stabilas, uzticamas un funkcionāli bagātas platformas palīdzību. Autoraprāt, .NET un TypeScript kompilatoru API izmantošana ir liela risinājuma priekšrocība vairāku iemeslu dēļ:

- platforma ar plašākajām iespējām pirmkoda analīzei un ģenerēšanai (vismaz no brīvi pieejamajiem/atvērtā pirmkoda risinājumiem);
- risinājumus attīsta Microsoft iekš GitHub (aktīvas diskusijas, visiem ir iespēja iesaistīties izstrāde, izstrādātāji klausās lietotājos, pieņem ierosinājumus);
- kompilatora API vienmēr ir saskaņā ar programmēšanas valodas attīstību – vairāki no apskatītajiem risinājumiem ir implementējuši savu TypeScript CodeDOM analogu, kurš nu jau novecojis.

Risinājuma arhitektūra. Esošos risinājumus ir grūti adaptēt vai papildināt to iespējas pēc saviem ieskatiem, jo tiem trūkst dokumentācijas un nav pārdomātas arhitektūras. Tie bieži vien izveido nesaprotamas klienta puses atkarības, kas realitātē nav nepieciešamas. Autors risinājuma izstrādei izvēlējās sistēmas arhitektūru, kurai ērti pievienot papildus iespējas/jaunus lietojuma scenārijus.

7.2. Testēšana

Pirmkoda ģenerēšana ir ļoti pateicīgs uzdevums testēšanas automatizācijai, jo iespējams skaidri definēt ieejas datus (C# klases, metodes, atribūtus) un sagaidāmo rezultātu (TypeScript klases, metodes, laukus). Šo iemeslu dēļ, risinājumā iesākta testēšanas automatizācija, kas testē atsevišķas klases un moduļu integrāciju kopumā. Testu automatizācijai izmantots NUnit vienībtestēšanas ietvars. Testi savā ziņā arī kalpo par risinājuma dokumentāciju – pēc tiem var objektīvi spriest par implementēto (un testēto) funkcionalitāti. Testu kopu iespējams atrast risinājuma mapes *Tests* iekļautajos projektos.

7.3. Pielietojums projektā un uzņēmumā

Izstrādātais risinājums tiek izmantots praksē, vairāku radniecīgu tīmekļa risinājumu izstrādei un uzturēšanai – mikro servisu arhitektūras sistēmai ir vairāki moduļi, kur katram no tiem ir savs lietotāja saskarnes projekts, kas izstrādāts ar ASP.NET Web API 2 un Angular tīmekļa tehnoloģijām.

Pēc rīka ieviešanas ikdienas izstrādes ciklā ir novērota produktivitātes un apmierinātības paaugstināšanās. Atsauksmes no projekta izstrādātājiem liecina, ka jaunu komponenšu pievienošana ir kļuvusi daudz ērtāka un esošo komponenšu uzturēšana vairs nav tik nogurdinošs process. Ja kādreiz modeļu sinhronizācija projekta izstrādes sākuma fāzē aizņēma būtisku darba dienas daļu, tad tagad šis process neatstāj nekādu iespaidu. 7.1. att. redzams rīka sinhronizācijas lietojuma izpildes rezultāts vienā no projektiem.

```
PS C:\Users\k.zimecs> cd C:\Dev\TypeSync\src\TypeSync\bin\Debug

PS C:\Dev\TypeSync\src\TypeSync\bin\Debug> .\TypeSync.exe -u Synchronization -i C:\work\byjuno_payout\ICP\ICP.Payout\ICP.Payout.Api\ICP.Payout.Api.csproj -o C:\Dev\Generated -g Template
2017-05-22 02:16:57,905 [1] INFO TypeSync.Program - Starting TypeSync
2017-05-22 02:17:07,721 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Source analyzed
2017-05-22 02:17:07,729 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Models converted
2017-05-22 02:17:07,734 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class TemplateVM emitted
2017-05-22 02:17:07,735 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ClientInfoDTO emitted
2017-05-22 02:17:07,736 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ClientBusinessProcessConfigurationDTO emitted
2017-05-22 02:17:07,737 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ClientInfoDTO emitted
2017-05-22 02:17:07,738 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class WorkerScheduleConfigurationDTO emitted
2017-05-22 02:17:07,739 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ClientBusinessProcessConfigurationDTO emitted
2017-05-22 02:17:07,740 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ClientInfoDTO emitted
2017-05-22 02:17:07,741 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ClientVM emitted
2017-05-22 02:17:07,742 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutHistoryVM emitted
2017-05-22 02:17:07,743 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class BankAccountVM emitted
2017-05-22 02:17:07,745 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridResponse emitted
2017-05-22 02:17:07,746 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutQueueItemVM emitted
2017-05-22 02:17:07,747 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class CommentVM emitted
2017-05-22 02:17:07,748 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ActivityLogItemVM emitted
2017-05-22 02:17:07,749 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutQueueItemVM emitted
2017-05-22 02:17:07,750 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class CommentVM emitted
2017-05-22 02:17:07,751 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ActivityLogItemVM emitted
2017-05-22 02:17:07,752 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutQueueFilterVM emitted
2017-05-22 02:17:07,753 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridRequest emitted
2017-05-22 02:17:07,754 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class CommentVM emitted
2017-05-22 02:17:07,755 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class ActivityLogItemVM emitted
2017-05-22 02:17:07,755 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridResponse emitted
2017-05-22 02:17:07,756 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutDetailItemVM emitted
2017-05-22 02:17:07,757 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutDetailItemVM emitted
2017-05-22 02:17:07,758 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutDetailFilterVM emitted
2017-05-22 02:17:07,759 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridRequest emitted
2017-05-22 02:17:07,760 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PayoutAdjustmentLineVM emitted
2017-05-22 02:17:07,761 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridResponse emitted
2017-05-22 02:17:07,762 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PendingPayoutItemVM emitted
2017-05-22 02:17:07,763 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PendingPayoutItemVM emitted
2017-05-22 02:17:07,764 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PendingPayoutFilterVM emitted
2017-05-22 02:17:07,765 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridRequest emitted
2017-05-22 02:17:07,765 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridResponse emitted
2017-05-22 02:17:07,766 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PaymentOrderSearchItemVM emitted
2017-05-22 02:17:07,767 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class IncludedPayoutItemVM emitted
2017-05-22 02:17:07,768 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PaymentOrderSearchItemVM emitted
2017-05-22 02:17:07,769 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class IncludedPayoutItemVM emitted
2017-05-22 02:17:07,770 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PaymentOrderSearchFilterVM emitted
2017-05-22 02:17:07,771 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridRequest emitted
2017-05-22 02:17:07,772 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class CreateNewPaymentOrderVM emitted
2017-05-22 02:17:07,773 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PaymentOrderVM emitted
2017-05-22 02:17:07,773 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class IncludedPayoutItemVM emitted
2017-05-22 02:17:07,774 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class IncludedPayoutItemVM emitted
2017-05-22 02:17:07,775 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridResponse emitted
2017-05-22 02:17:07,776 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class TransactionSearchItemVM emitted
2017-05-22 02:17:07,777 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class TransactionSearchItemVM emitted
2017-05-22 02:17:07,777 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class TransactionSearchFilterVM emitted
2017-05-22 02:17:07,778 [1] DEBUG TypeSync.UseCases.ModelGenerationUseCase - Class PagedDataGridRequest emitted
```

7.1. att. **TypeSync izpildes rezultāts**

Plānots rīku piedāvāt arī citām izstrādes komandām, kas izmanto attiecīgās tīmekļa tehnoloģijas un kurām tāda rīka klātbūtne projektā varētu dot labumu.

SECINĀJUMI

Darba teorētiskās un praktiskās daļas laikā gūtie secinājumi un atziņas:

- Roslyn un TypeScript kompilatoru API nav pietiekoši detalizēti dokumentēti (TypeScript gadījumā – gandrīz nemaz) un to apguve prasa pašrocīgu, praktisku API izpēti (jautājumu uzdošana tīmekļa vietnēs arī bieži vien nenes gaidīto palīdzību). Nākas pētīt GitHub publicēto pirmkodu un tā izmaiņas, lai saprastu, kas ticis mainīts. Informāciju visvieglāk iegūt no citiem izstrādātājiem, kas saskarušies ar līdzīgām problēmām, raduši risinājumu un publicējuši to tīmeklī.
- Roslyn piedāvā rīkus sintakses koku vizualizācijai, taču TypeScript tādu nav. Autors izveidoja savu sintakses koka izveidošanas un apstaigāšanas rīku, lai varētu vizualizēt TypeScript lietotnes struktūru un vieglāk saprast, kādi sintakses mezgli izmantojami konkrētu struktūru izveidei. Rīka darbības rezultāts apskatāms 4. pielikumā.
- Roslyn paļaujas uz MSBuild projektu kompilācijai – var rasties problēmas izpildot lietotni uz dažādām mašīnām, kurām atbilst dažādas MSBuild versijas. Var gadīties, ka netiek atrasta īstā MSBuild versija projekta kompilācijai. Risinājumā par to ir padomāts, lietotne ir attiecīgi konfigurēta un ir pieejama dokumentācija. Turpmāk, Roslyn komanda plāno ieviest atkarību no MSBuild, NuGet pakotnes veidā.
- Kompilatora API visticamāk kļūs par trendu dažādām programmēšanas valodām, kas vēlas attīstīt izstrādes rīku iespējas un ļaut izstrādātājiem radīt inovācijas meta-programmēšanas jomā. Autoraprāt, tam viennozīmīgi ir jābūt atvērtā pirmkoda projektam. Savādāk ir grūti attīstīt programmatūru pareizajā virzienā un paiet ilgs laika periods, līdz API izstrādātājs var novalidēt savu produktu un saprast, kas ir realizēts pareizi un pie kā vajag piestrādāt.

Turpmākā darba primārais fokuss ir aprakstītās bet trūkstošās funkcionalitātes implementācija, testu un dokumentācijas izstrāde. Tālāko darbu varētu saistīt ar citu klienta puses ietvaru atbalstu. Jebkurš ietvars, kas atbalsta TypeScript izmantošanu, var būt labs kandidāts. Darba autors uzskata, ka rīka tvērumu jāsaglabā .NET C# un TypeScript rāmjos, taču klienta puses pirmkods var tikt pielāgots jebkuram klienta puses ietvaram, kurš var izmantot TypeScript un kura izmantošanu pieprasa izstrādes scenārijs.

IZMANTOTĀ LITERATŪRA UN AVOTI

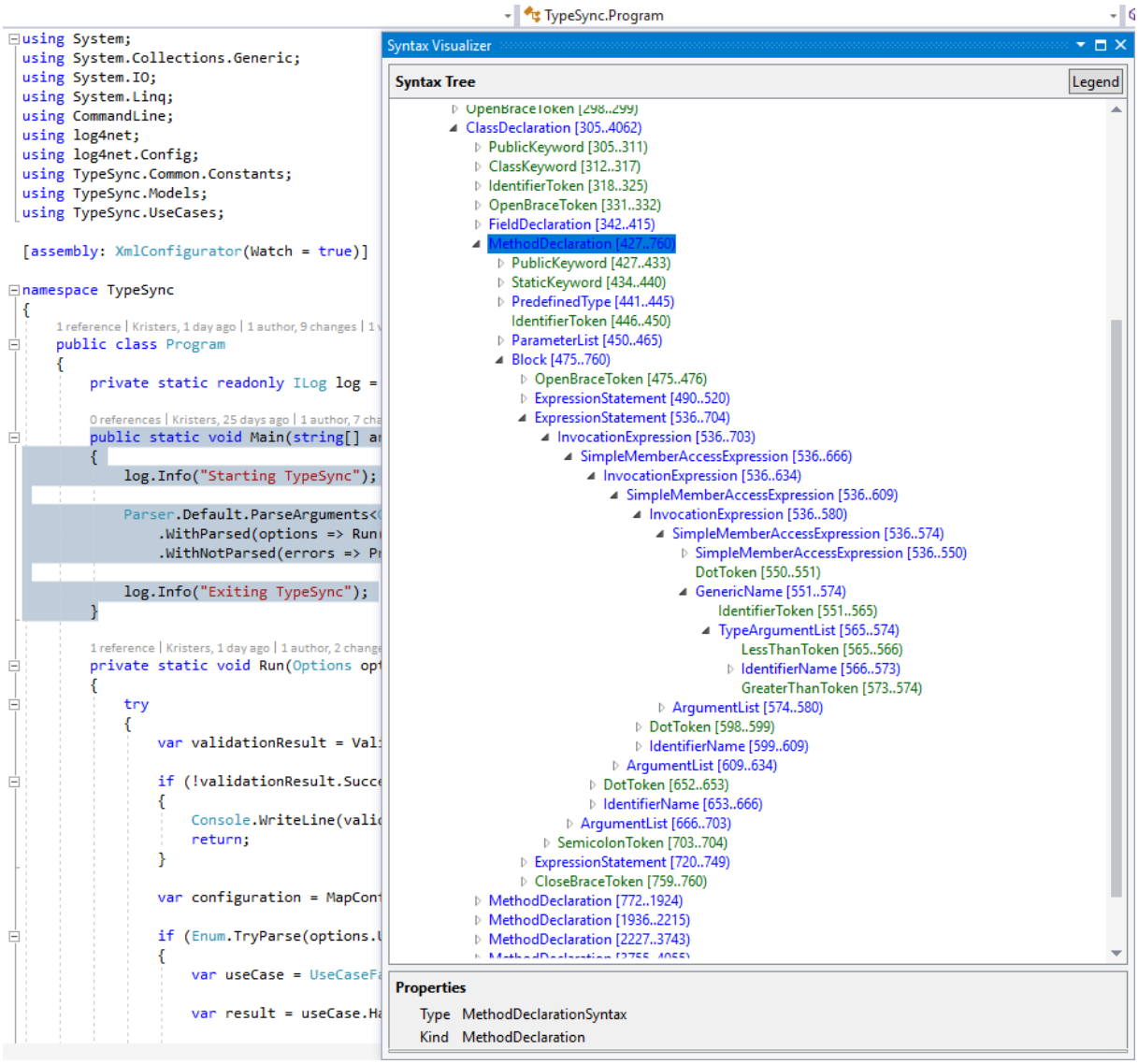
1. *HTML 5.1, W3C Recommendation,* World Wide Web Consortium, 1. Nov 2016. Pieejams: https://www.w3.org/TR/2016/REC-html51-20161101.
2. *CSS Snapshot 2017, W3C Working Group Note,* World Wide Web Consortium, 31. Jan 2017. Pieejams: https://www.w3.org/TR/css-2017.
3. Mozilla Developer Network, "JavaScript Reference". Pieejams: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference.
4. Microsoft, "TypeScript Language Specification", *GitHub*. Jan 2016. Pieejams: https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md.
5. DA-14, "5 Best JavaScript Frameworks in 2017", 18 Jan 2017. Pieejams: https://da-14.com/blog/5-best-javascript-frameworks-2017.
6. *ISO/IEC 23271:2012 Information technology -- Common Language Infrastructure (CLI),* International Organization for Standardization, Feb 2012.
7. *ISO/IEC 23270:2006 Information technology -- Programming languages -- C#,* International Organization for Standardization, Sep 2006.
8. ".NET Framework history", Wikipedia. Pieejams: https://en.wikipedia.org/wiki/.NET_Framework#History.
9. Microsoft, ".NET Architectual Components", 16 Nov 2016. Pieejams: https://docs.microsoft.com/en-us/dotnet/articles/standard/components.
10. Microsoft, ".NET Standard", 17 Mar 2017. Pieejams: https://docs.microsoft.com/en-us/dotnet/articles/standard/library.
11. Mono Project, "About Mono". Pieejams: http://www.mono-project.com/docs/about-mono.
12. Microsoft, "Get started with the .NET Framework", 30 Mar 2017. Pieejams: https://docs.microsoft.com/en-us/dotnet/articles/framework/get-started.
13. Microsoft, ".NET Core", 20 Jun 2016. Pieejams: https://docs.microsoft.com/lv-lv/dotnet/articles/core.
14. Microsoft Developer Network, "Solutions and Projects in Visual Studio". Pieejams: https://msdn.microsoft.com/en-us/library/b142f8e7.aspx.
15. Microsoft, "Introduction to ASP.NET Core", 14 Oct 2016. Pieejams: https://docs.microsoft.com/en-us/aspnet/core.
16. Microsoft. "A Tour of the C# Language", 10 Aug 2016. Pieejams: https://docs.microsoft.com/en-us/dotnet/articles/csharp/tour-of-csharp.

17. Microsoft. “C# Attributes”, 10 Aug 2016. Pieejams: https://docs.microsoft.com/en-us/dotnet/articles/csharp/tour-of-csharp/attributes.
18. *Standard ECMA-262 ECMAScript 2016 Language Specification*, Ecma International, Jun 2016. Pieejams: http://www.ecma-international.org/publications/standards/Ecma-262.htm.
19. *ISO/IEC 16262:2011 ECMAScript language specification*, International Organization for Standardization, Jun 2011. Pieejams: http://www.iso.org/iso/catalogue_detail.htm?csnumber=55755.
20. Microsoft, “TypeScript Docs”. Pieejams: https://www.typescriptlang.org/docs/home.html.
21. Microsoft, “TypeScript Handbook – Basic Types”. Pieejams: https://www.typescriptlang.org/docs/handbook/basic-types.html.
22. Microsoft, “TypeScript Handbook – Classes”. Pieejams: https://www.typescriptlang.org/docs/handbook/classes.html.
23. Microsoft, “TypeScript Handbook – Modules”. Pieejams: https://www.typescriptlang.org/docs/handbook/modules.html.
24. Basarat, “TypeScript Won”, 27 Apr 2016. Pieejams: https://medium.com/@basarat/typescript-won-a4e0dfde4b08.
25. Tom Preston-Werner, “Semantic Versioning”. Pieejams: http://semver.org.
26. Jon Preece, “A high level look on Angular”, 3 Sep 2016. Pieejams: http://www.developerhandbook.com/angular/high-level-look-angular-2.
27. Google, “Angular Docs”. Pieejams: https://angular.io/docs/ts/latest.
28. Kevin Hazzard, Jason Bock, *Metaprogramming in .NET*, NY, USA: Manning Publications Co., 2013.
29. “Using the Compiler API”, *GitHub*, 24 Mar 2017. Pieejams: https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API.
30. “Using the Language Service API”, *GitHub*, 9 Sep 2016. Pieejams: https://github.com/Microsoft/TypeScript/wiki/Using-the-Language-Service-API.
31. Nicholas Wolverson, “TypeScript Compiler APIs Revisited”, 2 Mai 2017. Pieejams: http://blog.scottlogic.com/2017/05/02/typescript-compiler-api-revisited.html.
32. “.NET Compiler Platform (Roslyn) Overview”, *GitHub*, 6 Jan 2016. Pieejams https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview.
33. Alessandro Del Sole, *Roslyn Succinctly*. 19 Feb 2016. Pieejams: https://www.syncfusion.com/resources/techportal/details/ebooks/roslyn
34. Nick Harrison, *Code Generation with Roslyn*, NY, USA: Apress, 2017.

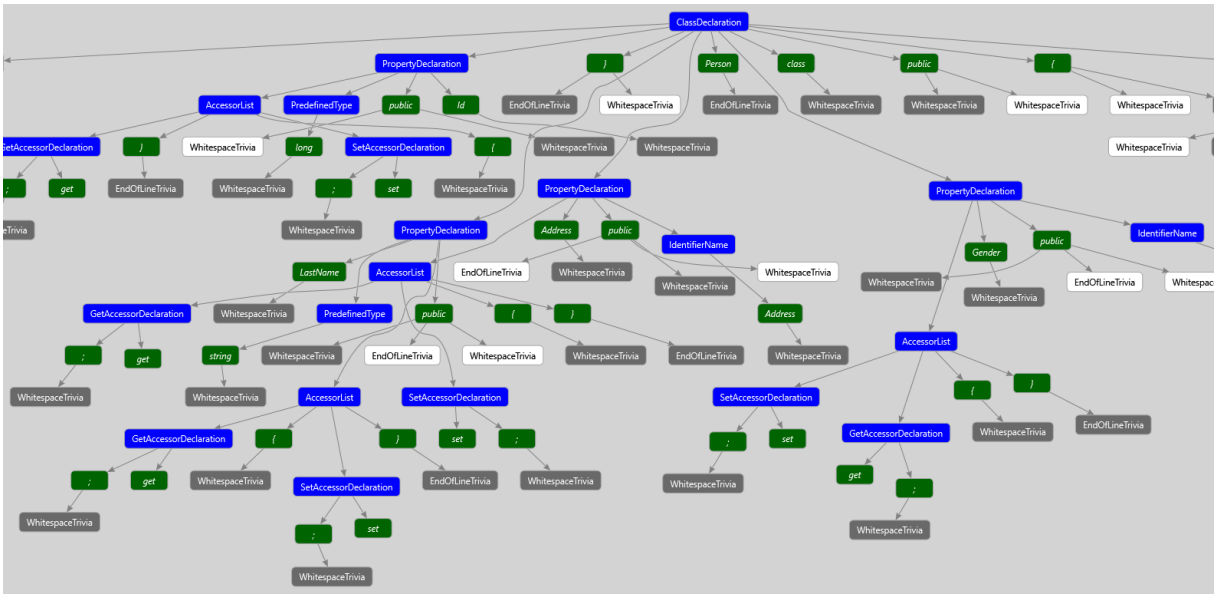
35. Micrsoft. “Introduction to model validation in ASP.NET Core MVC”, 14 Oct 2016. Pieejams: https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation.
36. Microsoft. “TypeScript Coding Guidelines”, *GitHub*, 11 Mai 2017. Pieejams: https://github.com/Microsoft/TypeScript/wiki/Coding-guidelines.

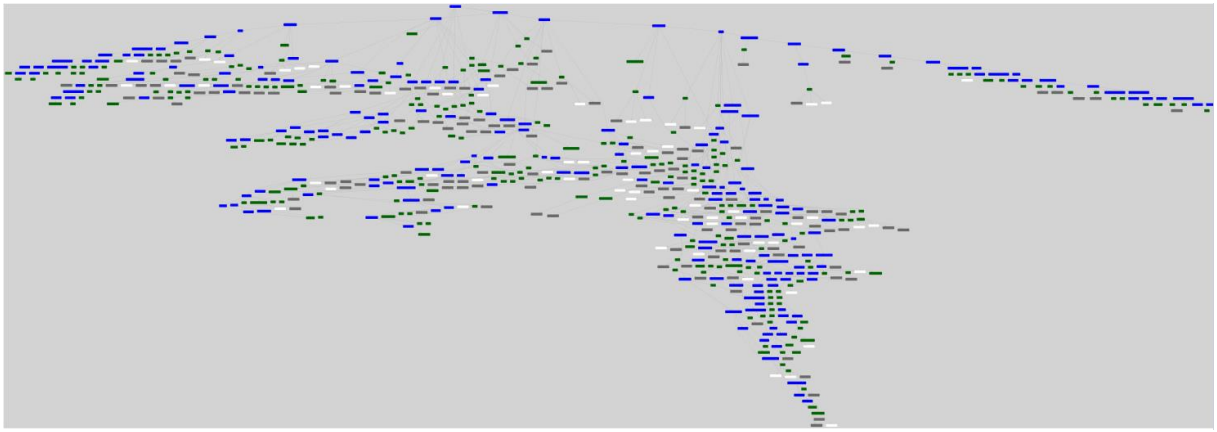
PIELIKUMI

1. PIELIKUMS. C# SINTAKSES KOKA VIZUALIZĀCIJAS RĪKS

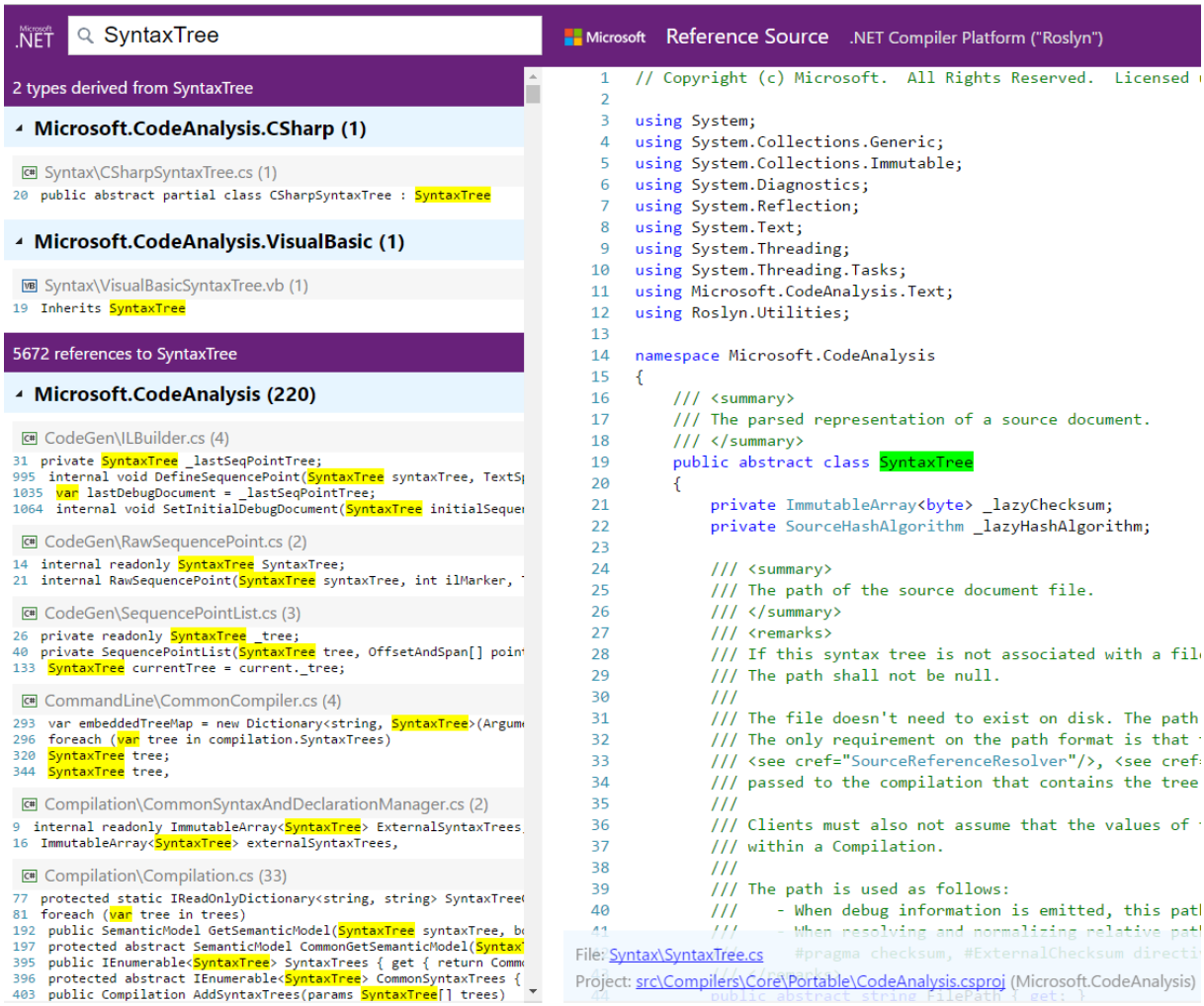


2. PIELIKUMS. C# SINTAKSES GRAFI





3. PIELIKUMS. ROSLYN PIRMKODA TĪMEKĻA VIETNE



4. PIELIKUMS. TYPESCRIPT SINTAKSES KOKA VIZUALIZĀCIJA

```
ClassDeclaration : export class Person {
        id: number;
        firstName: string;
        lastName: string;
        dateOfBirth: Date;
        address: Address;
        gender: Gender;
}
ExportKeyword : export
Identifier : Person
    PropertyDeclaration : id: number;
        Identifier : id
        NumberKeyword : number
    PropertyDeclaration : firstName: string;
        Identifier : firstName
        StringKeyword : string
    PropertyDeclaration : lastName: string;
        Identifier : lastName
        StringKeyword : string
    PropertyDeclaration : dateOfBirth: Date;
        Identifier : dateOfBirth
        TypeReference : Date
            Identifier : Date
    PropertyDeclaration : address: Address;
        Identifier : address
        TypeReference : Address
            Identifier : Address
    PropertyDeclaration : gender: Gender;
        Identifier : gender
        TypeReference : Gender
            Identifier : Gender
EndOfFileToken
```

5. PIELIKUMS. ATKARĪBU GRAFA IZVEIDES PIRMKODA FRAGMENTS

```
public class DependencyGraphService
{
    public DirectedSparseGraph<DependencyNode> Graph { get; private set; }

    public DependencyGraphService()
    {
        Graph = new DirectedSparseGraph<DependencyNode>();
    }

    public DirectedSparseGraph<DependencyNode>
BuildForNamedTypeSymbol(INamedTypeSymbol namedType)
    {
        GetDependenciesRecursive(namedType, null);

        return Graph;
    }

    private void GetDependenciesRecursive(INamedTypeSymbol classSymbol, DependencyNode
parent)
    {
        var node = new DependencyNode()
        {
            NamedTypeSymbol = classSymbol
        };

        var existingNode = Graph.Vertices
            .FirstOrDefault(v => v.NamedTypeSymbol.Name == classSymbol.Name);

        if (existingNode != null)
        {
            node = existingNode;
        }
        else
        {
            Graph.AddVertex(node);
        }

        if (parent != null)
        {
            Graph.AddEdge(parent, node);
        }

        var depService = new DependencyService();

        var dependencies = depService.GetTypeDependencies(classSymbol);

        foreach (var dep in dependencies)
        {
            GetDependenciesRecursive(dep, node);
        }
    }
}


public class DependencyService
{
    private List<INamedTypeSymbol> _dependencies = new List<INamedTypeSymbol>();

    public List<INamedTypeSymbol> GetTypeDependencies(INamedTypeSymbol classSymbol)
    {
```

```
            if (classSymbol.BaseType != null)
            {
                ProccessType(classSymbol.BaseType);
            }

            var members = classSymbol.GetMembers().ToList();

            var properties = members.Where(m => m.Kind == SymbolKind.Property).ToList();

            foreach (var property in properties)
            {
                var propertySymbol = property as IPropertySymbol;

                ProccessType(propertySymbol.Type);
            }

            return _dependencies;
        }

        private void ProccessType(ITypeSymbol typeSymbol)
        {
            if (typeSymbol is IArrayTypeSymbol)
            {
                var arrayTypeSymbol = typeSymbol as IArrayTypeSymbol;

                ProccessType(arrayTypeSymbol.ElementType);
            }
            else if (typeSymbol is ITypeParameterSymbol)
            {
                var typeParameterSymbol = typeSymbol as ITypeParameterSymbol;

                return;
            }
            else if (typeSymbol is INamedTypeSymbol)
            {
                var namedTypeSymbol = typeSymbol as INamedTypeSymbol;

                if (!namedTypeSymbol.TypeArguments.IsDefaultOrEmpty)
                {
                    foreach (var typeArgument in namedTypeSymbol.TypeArguments)
                    {
                        ProccessType(typeArgument);
                    }
                }
                else
                {
                    if (TypeHelper.IsSupportedType(namedTypeSymbol))
                    {
                        _dependencies.Add(namedTypeSymbol);
                    }
                }
            }
        }
    }
```

6. PIELIKUMS. TYPESCRIPT KLASES ĢENERĒŠANAS PIRMKODA FRAGMENTS

```
private createClass = (classModel: models.ClassModel): ts.ClassDeclaration =>
{
    // decorators
    const decoratorNodes = classModel.decorators.map(d =>
ts.createDecorator(ts.createIdentifier(d)));

    // class modifiers
    const modifiers = [
        ts.createToken(ts.SyntaxKind.ExportKeyword)
    ];

    // type parameters
    const typeParameters = classModel.typeParameters.map(p =>
ts.createTypeParameterDeclaration(p, undefined, undefined));

    // base class or interfaces
    const heritageClauses = [];
    if (classModel.baseClass) {
        heritageClauses.push(ts.createHeritageClause(
            ts.SyntaxKind.ExtendsKeyword,
            [ts.createExpressionWithTypeArguments(null,
ts.createIdentifier(classModel.baseClass))]
        ))
    }

    const classElements = [];

    // properties
    classElements.push(...classModel.properties.map(p =>
this.createProperty(p)));

    // constructor
    if (classModel.constructorDef) {
        classElements.push(this.createConstructor(classModel.constructorDef));
    }

    let hasHttpMethod = false;

    // methods
    if (classModel.methods) {
        for (const method of classModel.methods) {

            // http methods
            const httpMethod = method as models.HttpMethodModel
            if (httpMethod.httpMethod >= 0) {
                classElements.push(this.createHttpMethod(httpMethod));
                hasHttpMethod = true;
```

```
            }
        }
    }

    if (hasHttpMethod) {
        classElements.push(this.createHttpErrorHandler());
    }

    return ts.createClassDeclaration(decoratorNodes, modifiers,
classModel.name, typeParameters, heritageClauses, classElements);
}


generateClass = (classModel: models.ClassModel): string => {
    // imports
    const importNodes = classModel.imports.map(model =>
this.createImport(model));

    // class declaration
    const classNode = this.createClass(classModel);

    // transform
    const result: ts.TransformationResult<ts.Node> = ts.transform(
        classNode, [transformers.quotemarkTransformer]
    );

    const transformedClassNode: ts.Node = result.transformed[0];

    // print text
    const printed = Utilities.concatNodes([...importNodes,
transformedClassNode]);

    // format
    // const formatted = new Formatter().format(generated);
    const replaced = Utilities.replaceQuotemarks(printed);

    return replaced;
}
```

DOKUMENTĀRĀ LAPA

Maģistra darbs “Angular specifiska TypeScript pirmkoda ģenerēšana ar Roslyn .NET kompilatora platformu” izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 22.05.2017.

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

Autors: _____ (Kristers Zīmecs)

(Autora paraksts un datums)

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: _____ (Dr. dat. Sergejs Kozlovičs)

(Vadītāja paraksts un datums)

Darbs iesniegts **maģistratūras sekretariātā** _____.

(Iesniegšanas datums)

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

Studiju metodiķe: _____.

(Metodiķes paraksts)

Recenzents: docents, Dr. dat. Agris Šostaks

(Akad.amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____

(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____

(Sekretāra paraksts)