LATVIJAS UNIVERSITĀTE
FIZIKAS UN MATEMĀTIKAS FAKULTĀTE
DATORIKAS NODAĻA

METAMODELIS ANIMĀCIJU APRAKSTĪŠANAI
BAKALAURA DARBS

Autors: Ronalds Zarīts
Stud. apl. DatZ040108
Darba vadītājs: profesors Dr. habil. sc. comp. Jānis Bārzdiņš

Rīga 2008

ANOTĀCIJA

Šajā darbā tiek piedāvāti principi animāciju metamodeļa būvei. Darbā izklāstītas MDA ietvara galvenās idejas un tā priekšrocības salīdzinājumā ar tradicionālu programmatūras izstrādi. Tiek apskatīti datoranimācijas principi un aplūkoti populāri programmatūras risinājumi animāciju izstrādei. Balsoties uz šo pieredzi, tiek piedāvāti būves principi metamodelim animāciju aprakstīšanai. Aprakstītie principi tiek demonstrēti, pielietojot tos orientēta grafa elementu animācijas aprakstīšanai. Darba ietvaros izstrādāts animāciju dziņa prototips un dziņa demonstrācijas programma, sniegts to realizācijas apraksts un identificēti iespējamie uzlabojumi.

ATSLĒGVĀRDI

Animācija, metamodelis, MDA, animācijas dzinis, programmatūras izstrāde.

ABSTRACT

This paper proposes principles of building metamodels for describing animation. This paper summarizes the main ideas behind the MDA framework and its' advantages compared to traditional software development. A brief exploration of principles of computer-based animation is performed. Based on this experience, this paper proposes principles for building a metamodel for describing animation. The described principles are demonstrated by applying them to the problem domain of directed graph element animation. A prototype animation engine and a demonstration program have been developed, a description of their implementation is given, and possible improvements are identified.

KEYWORDS

Animation, metamodel, MDA, animation engine, software development.

SATURS

IEVADS **2**

1. MODEL DRIVEN ARCHITECTURE IETVARS **3**

1.1. TRADICIONĀLAS PROGRAMMATŪRAS IZSTRĀDES PROBLĒMAS 3
1.1.1. Produktivitātes problēma *4*
1.1.2. Pārnesamības problēma *4*
1.1.3. Sadarbības problēma *5*
1.1.4. Uzturēšanas un dokumentācijas problēma *5*
1.2. MDA IZSTRĀDES PROCESS 5
1.3. VALODAS LĪDZEKĻI METAMODEĻU DEFINĒŠANAI 7
1.4. MDA IZSTRĀDE LU MII 9
1.4.1. GrTP grafisko rīku būves platforma *9*
1.5. KOPSAVILKUMS 11

2. DATORANIMĀCIJA **12**

3. EKSISTĒJOŠIE RISINĀJUMI ANIMĀCIJAS IZSTRĀDEI **13**

3.1. WINDOWS FORMS 13
3.1.1. Windows Forms grafikas sistēma *14*
3.1.2. Animācijas ar Windows Forms *15*
3.2. WINDOWS PRESENTATION FOUNDATION 16
3.2.1. WPF divdimensiju grafikas sistēma *17*
3.2.2. Animācijas ar WPF *21*
3.3. SILVERLIGHT 28
3.4. EXPRESSION BLEND 28
3.5. ADOBE FLASH 30
3.6. KOPSAVILKUMS 31

4. ANIMĀCIJAS METAMODELIS **33**

4.1. ANIMĀCIJU METAMODEĻA VEIDOŠANAS PAMATIDEJAS 33
4.1.1. Animējami atribūti *34*
4.1.2. Animācijas klases *34*
4.1.3. Notikumu klases *36*
4.1.4. Darbību klases *37*
4.1.5. Animāciju grupu klases *38*
4.1.6. Dienesta klases *38*
4.1.7. Animāciju metamodeļa instances piemērs *39*
4.2. GRAFA ANIMĀCIJAS METAMODELIS 40
4.2.1. Problēmapgabala apraksts *40*
4.2.2. Animējama orientēta grafa metamodelis *41*
4.2.3. Animēta orientēta grafa metamodeļa instances piemērs *44*
4.3. KOPSAVILKUMS 45

5. PROTOTIPA REALIZĀCIJAS APRAKSTS **46**

5.1. APRAKSTS 46
5.2. KLAŠU BIBLIOTĒKA 46
5.2.1. Animējami atribūti *47*
5.2.2. Grafa klases *48*
5.2.3. Animāciju klases *50*
5.2.4. Darbību klases *53*
5.2.5. Notikumu klase *54*
5.3. DEMONSTRĀCIJAS PROGRAMMA 54
5.3.1. Programmas apraksts *55*
5.3.2. Animētu grafu demonstrācijas konfigurācijas *55*
5.4. IESPĒJAMIE UZLABOJUMI 59
5.4.1. Integrācija ar LU MII repozitoriju *59*
5.4.2. Klašu struktūras uzlabojumi *60*
5.5. KOPSAVILKUMS 62

6. SECINĀJUMI **63**

7. IZMANTOTĀ LITERATŪRA UN AVOTI ... 65

PIELIKUMI ... 67

1. PIELIKUMS – MOF KLAŠU MODELĒŠANAS CENTRĀLO LĪDZEKĻU METAMODELIS ... 67
2. PIELIKUMS – WINDOWS FORMS KONTROĻU KLAŠU HIERARHIJAS FRAGMENTS ... 68
3. PIELIKUMS – ANIMĀCIJAS PROGRAMMA, IZMANTOJOT WINDOWS FORMS ... 69
4. PIELIKUMS – WPF INTERAKTĪVO LIETOTĀJA SASKARNES KLAŠU HIERARHIJA ... 71
5. PIELIKUMS – WPF IEKĻAUTĀS ANIMĀCIJU KLASES ... 72
6. PIELIKUMS – WPF DOUBLEANIMATION DEMONSTRĀCIJAS PROGRAMMA ... 73
7. PIELIKUMS – WPF EVENTTRIGGER DEMONSTRĀCIJA ... 74
8. PIELIKUMS – PILNS ANIMĀCIJAS METAMODEĻA BŪVES PIEMĒRA METAMODELIS ... 76
9. PIELIKUMS – ANIMĒJAMA ORIENTĒTA GRAFA METAMODELIS ... 77
10. PIELIKUMS – PROGRAMMATŪRAS CD ... 78

APZĪMĒJUMU SARAKSTS

Bakalaura darbā izmantoti šādi apzīmējumi:

Apzīmējums	**Paskaidrojums**
API	Saīsinājums no **Application Program Interface.** Lietojumprogrammu saskarne. Lietojumprocesos izmantojama pilna programmatūras funkciju kopas specifikācija, kā arī šo funkciju izmantošanas procedūru apraksts. Ar objektorientētās programmēšanas popularitātes plašo izplatību, standarta prakse ir API veidot objektorientētas, t.i. kā bibliotēkas, kuras sastāv no klasēm, kurām piemīt funkcijas jeb metodes. Šajā darbā ar API tiek apzīmēts plašāks jēdziens - programmatūras klašu kopa ar to funkcijām, kas paredzēta kādas problēmas risināšanai, piemēram, API grafikas zīmēšanai.
Atslēgkadrs	Animācijas kadrs, kurš apraksta objekta stāvokli fiksētā laika momentā.
Deklaratīva programma	Programma, kas apraksta (deklarē) struktūras, kuras tiek apstrādātas ar standarta algoritmu
Imperatīva programma	Programma, kura sastāv no komandām, kas manipulē programmas stāvokli.
Dzinis	Šī darba kontekstā, speciāla programmatūra, kas pēc vienota algoritma ģenerē izvadu no ievada datu struktūrām. Piemēram, trīsdimensiju grafikas renderēšanas dzinis ģenerē attēlu no trīsdimensiju objektu apraksta.
Mantošana	Viena no objektorientētās programmēšanas tehnikām. Mantošana ir veids kā definēt jaunas klases, izmantojot esošās. Mantotā klase satur jeb manto visas metodes un atribūtus, kuri ir klasei, no kuras notiek mantošanās. Klase, no kuras notiek mantošanās, tiek saukta par bāzes klasi vai virsklasi. Klase, kura manto bāzes klases īpašības, tiek saukta par mantoto klasi vai apakšklasi.
MDA	Saīsinājums no Model Driven Architecture. Pieeja programmatūras izstrādei, kas centrālajā vietā programmatūras izstrādes procesā nostāda modeļus un modeļu transformācijas
Repozitorijs	Centrāla vieta, kurā tiek organizēti veidots un uzturēts datu sakopojums.

IEVADS

Lietojumprogrammās arvien vairāk parādās multimediju elementi – papildus standarta lietotāja saskarnes kontrolēm bieži sastopami tādi elementi kā attēli, vektorgrafika, skaņa, video, kuri bagātina lietotāja saskarni un veido kopējo lietotāja pieredzi (*user experience* jeb *UX*). Viena no labas lietotāja pieredzes sastāvdaļām ir elementu animācija. Pateicoties mūsdienu personālo datoru jaudas pieaugumam, kļūst iespējams lietotājiem piedāvāt arvien bagātīgāku un krāšņāku pieredzi.

Animācijas noderīgums ir acīmredzams. Animācija var kalpot dažādiem mērķiem – gan praktiskiem, gan estētiskiem:

- uzskatāmības uzlabošanai un izpratnes padziļināšanai – ar animāciju var uzlabot lietotāja izpratni par attēlojamo kompozīciju, papildinot to ar elementiem, kas vizualizē procesus šajā kompozīcijā, vai arī vienkārši pievērst uzmanību atsevišķām kompozīcijas detaļām. Piemēram, biznesa procesa diagrammu varētu papildināt ar dekorācijām, kas ilustrē procesa gaitu vai paskaidro procesu dažādos tā punktos.
- lietotāja saskarnes dekorācijām – ar animāciju var padarīt pievilcīgāku lietotāja saskarni, padarot darbu ar to dabiskāku un patīkamāku. Piemēram, būtu iespējams animēt nākošās bildes pārslēgšanu pārlūkojot foto kolekciju, vai arī plūstoši atvērt vai aizvērt programmas logus.

Programmatūras izstrādes joma nepārtraukti attīstās. 2001. gadā Object Management Group iepazīstināja ar Model-driven architecture – pieeju programmatūras izstrādei, kas centrālajā vietā nostāda modeļus un modeļu transformācijas. Model-driven architecture (MDA) sola radikāli mainīt programmatūras izstrādes procesu un pamazām šo pieeju sāk izmantot produkcijas līmeņa programmatūras izstrādē.

Lai gan animācija datorā nav nekas jauns un ir izveidoti dažādi sarežģīti risinājumi animācijas veidošanai, tomēr, lai pilnvērtīgi integrētu animāciju MDA ideoloģijā, ir nepieciešami papildus pētījumi.

Darba mērķis – izstrādāt principus animāciju modelēšanai pēc MDA principiem.

Darba uzdevumi:

1) Izpētīt esošos pieejamos risinājumus, kas nodrošina animāciju veidošanu,
2) Formulēt principus animācijas metamodeļu būvei
3) Izveidot prototipu, kas demonstrē formulētos principus darbībā.

1. MODEL DRIVEN ARCHITECTURE IETVARS

Model Driven Architecture (turpmāk MDA) ir programmatūras izstrādes ietvars, kurš tiek izstrādāts Object Management Group (OMG) pārraudzībā. MDA pamazām kļūst svarīgu par programmatūras izstrādes nozares sastāvdaļu. MDA joprojām agrīnā stadijā, taču nākotnē sola radikāli mainīt programmatūras izstrādi (1).

MDA definē pieeju programmatūras izstrādei, kur centrālā loma ir modeļiem un modeļu transformācijām.

Modelis, MDA kontekstā, ir sistēmas apraksts, kas uzrakstīts labi definētā valodā. Labi definēta valoda ir valoda ar formālu sintaksi, semantiku, un kuru var interpretēt dators. Modelis var būt pierakstīts dažādi, piemēram, grafiskā modelēšanas valodā tādā kā UML, vai kādā programmēšanas valodā, piemēram, Java vai C#. (2)

Modeļi paši par sevi arī ir sistēmas, kuras var aprakstīt ar modeli. Šādu modeli, kurš apraksta iespējamo modeļu kopu, sauc par metamodeli. Citiem vārdiem, metamodelis uzdod iespējamo modeļu kopu.

Transformācijas likums ir apraksts kā vienu vai vairākas konstrukcijas avota valodā var transformēt par vienu vai vairākām konstrukcijām mērķa valodā. Transformācijas definīcija ir kopa ar transformācijas likumiem, kas kopā apraksta kā modeli avota valodā var transformēt par modeli mērķa valodā. Transformācija ir process, kurā notiek automātiska mērķa modeļa ģenerēšana no avota modeļa, vadoties pēc transformācijas definīcijas. (2)

Labs transformācijas piemērs ir datubāzes struktūras izveides programmas ģenerēšana no UML modeļa – šajā gadījumā UML ir avota valoda un SQL ir mērķa valoda. Pirmkods arī atbilst modeļa definīcijai, jo tas ir uzrakstīts labi definētā valodā un to spēj interpretēt dators.

Turpmākajās sadaļās ir īsi izklāstītas problēmas tradicionālā programmatūras izstrādē un sniegtas MDA pamatidejas.

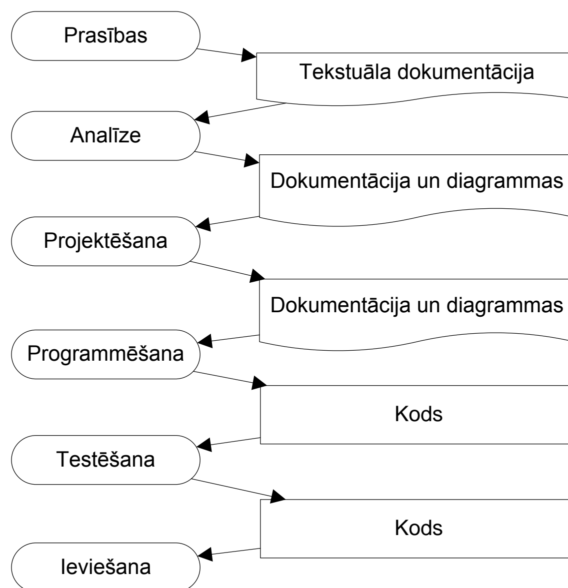
1.1. Tradicionālas programmatūras izstrādes problēmas

Programmatūras izstrādes sfēra joprojām cīnās ar vairākām lielām problēmām. Programmatūras izstrāde ir darbietilpīga. Strauji attīstās jaunas tehnoloģijas - parādoties jaunām tehnoloģijām, jāiegulda darbs, lai izmantotu šos jaunievedumus. Sistēmas bieži tiek izstrādātas izmantojot vairāk nekā vienu tehnoloģiju un tām ir nepieciešams sadarboties ar citām sistēmām. Prasības pret programmatūru pastāvīgi mainās. Dēļ uzskaitītajām programmizstrādes īpašībām tradicionālai programmatūras izstrādei ir vairākas problēmas. Turpmākajās sadaļās ir uzskaitītas dažas no būtiskākajām programmizstrādes problēmām.

1.1.1. Produktivitātes problēma

Mūsdienās programmatūras izstrāde var notikt pēc dažādiem izstrādes modeļiem. Lai arī kāds būtu programmizstrādes process, tajā parasti tiek iekļautas sekojošas fāzes (skat arī. *1.1. att. Programmizstrādes procesa fāzes*):

1. Prasību uzkrāšana
2. Analīze
3. Projektēšana
4. Programmēšana
5. Testēšana
6. Ieviešana



1.1. att. **Programmizstrādes procesa fāzes**

Lai arī kāds būtu izstrādes process - inkrementāls, iteratīvs vai tradicionāls ūdenskrituma process - izstrādes fāzēs tiek veidoti dokumenti un diagrammas. Taču šie izstrādes vienumi ir „tikai papīrs" – tā nav darbināma programmatūra. Šie dokumenti strauji zaudē aktualitāti sākoties programmēšanai.

Lai uzturētu šos vienumus aktuālus, nepieciešams papildus darbs. Šis darbs nav produktīvs, jo nenes taustāmu rezultātu programmatūras formā. Taču bez šiem dokumentiem programmatūras uzturēšana kļūst ļoti sarežģīta, īpaši, ja komanda, kas izstrādājusi sākotnējo produktu, vairs nav pieejama. Tādēļ, nobriedušā programmizstrādes projektā ir nepieciešams veikt šos papildus darbus.

1.1.2. Pārnesamības problēma

Katru gadu tiek izstrādātas aizvien jaunas tehnoloģijas, kuras kļūst populāras. Programmizstrādes uzņēmumiem nepieciešams sekot šīm tendencēm vairāku iemeslu dēļ:

- Jaunās tehnoloģijas pieprasa klienti. Piemēram, labākas lietotāja pieredzes nodrošināšanai, klienti var pieprasīt jaunu Web saskarni esošai biļešu pasūtīšanas sistēmai, kura izmantotu AJAX,.
- Tehnoloģijas ļauj risināt problēmas, kuras iepriekš nebija atrisināmas
- Izstrādes rīku piegādātāji pārtrauc uzturēt vecās izstrādes tehnoloģijas par labu jaunajām.

Iepriekš uzskaitīto iemeslu sekas ir tādas, ka esošā programmatūras realizācija ir jāpārnes uz jaunajām tehnoloģijām vai arī esošā programmatūra paliek nemainīga, taču

nepieciešams nodrošināt sadarbību starp programmatūru, kas izmanto vecās tehnoloģijas un jauno programmatūru, kas realizēta izmantojot jaunās tehnoloģijas.

1.1.3. Sadarbības problēma

Mūsdienās reti kura sistēma ir izolēta. Lielākajai daļai sistēmu nepieciešams nodrošināt sadarbību ar citām sistēmām. Labs piemērs ir Web programmas, kuras iegūst informāciju no citām sistēmām, piemēram, SQL datubāzes sistēmām vai Web servisiem. Katrs no šiem komponentiem var būt izstrādāts ar tehnoloģiju, kura vislabāk atbilst dotajam uzdevumam, taču tām ir nepieciešams savstarpēji sadarboties. Tas rada vajadzību pēc programmu sadarbības.

1.1.4. Uzturēšanas un dokumentācijas problēma

Sadaļā *1.1.1. Produktivitātes problēma* jau tika pieminēta uzturēšanas problēma. Programmatūras izstrādes dokumentācijas veidošana parasti ieņem pēdējo vietu. Lielākā daļa programmētāju uzskata, ka viņu galvenais uzdevums ir radīt kodu, jo dokumentācijas rakstīšana aizņem laiku un palēnina izstrādes procesu. Šis ir viens no iemesliem kādēļ dokumentācija ir neaktuāla – pēc katras izmaiņas kodā ir nepieciešams atjaunināt dokumentāciju.

Taču dokumentācija ir nepieciešama vēlākā programmatūras dzīves cikla posmā – uzturēšanā. Bez augsta līmeņa dokumentācijas, komandai, kurai jāuztur programmatūra, ir ļoti grūti orientēties programmatūras pirmkodā.

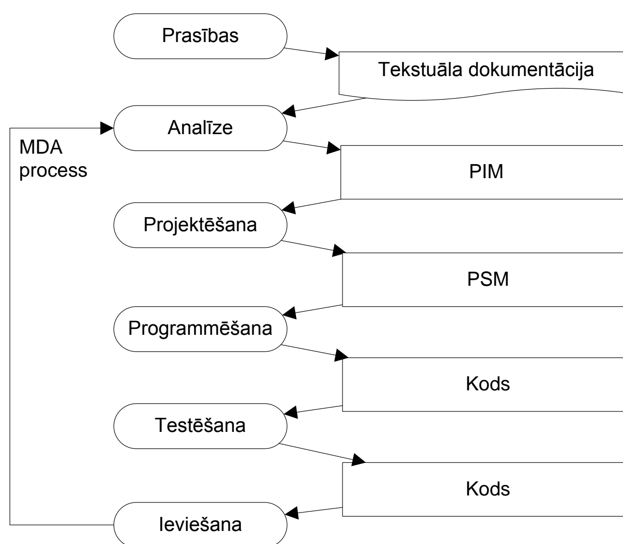
Ņemot vērā mūsdienu programmatūras sarežģītību, augstāka līmeņa pavadošā dokumentācija ir absolūti nepieciešama.

1.2. MDA izstrādes process

Šajā sadaļā izklāstīts MDA izstrādes dzīves cikls un mēģināts izskaidrot kā tas risina problēmas kas izklāstītas iepriekšējā sadaļā.

MDA izstrādes cikls, kas redzams attēlā *1.2. att. MDA izstrādes process*, neizskatās atšķirīgs no tradicionālā dzīves cikla – tiek identificētas tās pašas izstrādes fāzes. Galvenā atšķirība ir izstrādes vienumos, kas rodas fāžu rezultātā – tie ir formāli vienumi, t.i. modeļi, kurus var interpretēt ar datora palīdzību. Būtiskākie ir sekojošie trīs modeļi - no platformas neatkarīgais modelis (platform independent model – turpmāk „PIM”), specifiskās platformas modelis (platform specific model – turpmāk „PSM”) un pirmkods (2).

No platformas neatkarīgais modelis (PIM) ir pirmais modelis, kurš tiek izstrādāts MDA izstrādes procesa ietvaros. Tas ir neatkarīgs no realizācijas platformas un apraksta programmatūru, kas atbalsta kādu procesu, ar konkrētā problēmapgabala jēdzieniem.

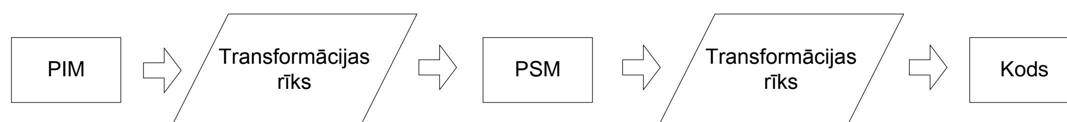


1.2. att. **MDA izstrādes process**

Specifiskās platformas modelis ir modelis, kas apraksta sistēmu konkrētās realizācijas tehnoloģijas jēdzienos. Piemēram, relāciju datubāzes modelis iekļauj tabulas, kolonnas, ārējās atslēgas, u.c. jēdzienus. Pēc PIM izstrādes tas tiek transformēts uz vienu vai vairākiem PSM.

Pēdējais solis ir transformāciju izstrāde, kas pārveido katru PSM par izpildāmu kodu. Tā kā PSM tuvu atbilst tā modelētajai tehnoloģijai, šīs transformācijas ir relatīvi vienkāršas.

Tradicionālā izstrādes pieejā, transformācijas no viena modeļa uz otru tiek veiktas manuāli – iesaistoties cilvēkam. MDA pieejā transformācijas starp modeļiem vienmēr tiek veiktas ar rīku palīdzību. Transformācijas process attēlots *1.3. att. Modeļu transformācijas process*.



1.3. att. **Modeļu transformācijas process**

Sadaļā *1.1. Tradicionālas programmatūras izstrādes problēmas* tika izklāstītas vairākas tradicionālās programmatūras izstrādes procesa problēmas. MDA pieeja sniedz šādas priekšrocības (3):

- Pārnesamība – tā kā PIM ir neatkarīgs no platformas, tas var tikt izmantots, lai ģenerētu dažādus PSM, tādējādi atbalstot pārnesamību uz dažādām platformām
- Produktivitāte – ļauj izstrādātājiem, biznesa analītiķiem lietot valodas un jēdzienus, kuri ir ērti, tajā pat laikā uzturot vienkāršu integrāciju. Agrīnajās fāzēs izstrādātie modeļi nav "tikai papīrs". Produktivitāte tiek uzlabota, ja tiek izmantoti rīki, kas pilnībā automatizē koda ģenerēšanu no PSM. Produktivitāte tiek uzlabota vēl jo vairāk, ja PSM ģenerēšana no PIM arī ir automatizēta.
- Sadarbība – izmantojot rīkus, kas ne tikai ģenerē dažādus PSM, bet arī spēj izveidot tiltu starp tiem, var tikt sasniegta sadarbība starp dažādām tehnoloģijām.

- Uzturēšana un dokumentācija – MDA izstrādes ietvaros ir jāizstrādā PIM, kas ir augsta līmeņa sistēmas apraksts un vienlaicīgi arī kalpo par sistēmas dokumentāciju. No PIM tiek ģenerēti nākamie vienumi, tādējādi tiek garantēts, ka PIM, kā arī PSM, ir aktuāls un saskan ar programmatūras kodu.

1.3. Valodas līdzekļi metamodeļu definēšanai

Kā jau minēts iepriekš nodaļas ievadā, metamodelis ir modelis, kas apraksta iespējamo modeļu kopu. Tā ir precīza definīcija konstrukcijām, kas var tikt lietotas, lai aprakstītu modeli. Tā kā metamodelis arī ir modelis, tad tas var būs specificēts jebkurā labi definētā valodā, kuru spēj interpretēt dators. Akceptēta valoda, kurā aprakstīt metamodeļus ir OMG izstrādātā Meta-Object Facility (turpmāk MOF). MOF ir noslēgta – MOF valodu var aprakstīt ar MOF līdzekļiem, tādējādi noslēdzot potenciāli bezgalīgo metamodeļu ķēdi. (2)

MOF modeļi izmanto UML 2.0 infrastruktūras apakškopu, kura definē pamatjēdzienus un to grafisko notāciju (skat. *1. Pielikums – MOF klašu modelēšanas centrālo līdzekļu metamodelis*). MOF ļauj aprakstīt modeļus, kas sastāv no klasēm, kurām var būt atribūti un operācijas, un asociācijām starp klasēm. MOF satur citas papildus iespējas, taču tās šajā darbā netiks apskatītas sīkāk. (4)

Atlikušajā nodaļā apskatīti MOF klases un asociācijas jēdzieni un to grafiskā notācija.

Klase apraksta sistēmas jēdzienu, piemēram, procesu aktivitātes diagrammā, vai operācijas, atribūta un klases jēdzienus UML. Klasei var būt atribūti un operācijas.

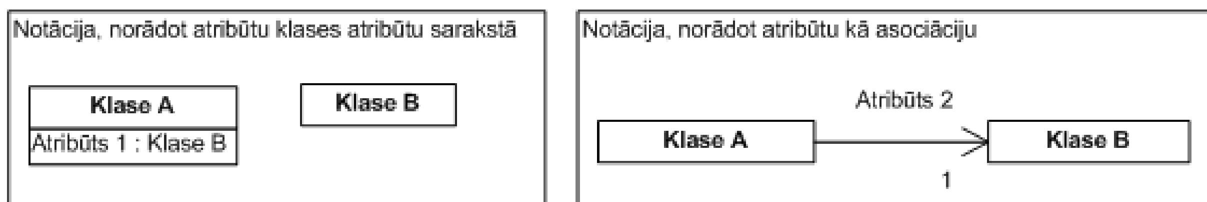
Atribūti apraksta kādu jēdziena īpašību, piemēram, riņķa līnijas klase var saturēt atribūtu rādiusa aprakstīšanai. Atribūtam ir fiksēts datu tips, kas var būt primitīvs datu tips vai citas klases tips.

Operācijas apraksta jēdziena funkcionalitāti, piemēram, loga klase var saturēt operāciju loga atvēršanai. Operācijai var būt nulle vai vairāki parametri ar fiksētu datu tipu un tā var atgriezt rezultātu ar fiksētu datu tipu.

Grafiski klase tiek attēlota ar taisnstūri, kuram var būt trīs nodalījumi. Augšējais satur klases nosaukumu, vidējais satur klases atribūtus, un apakšējais satur klases operācijas (skat. *1.4. att. Klases grafiskā notācija*). Ja klase nesatur operācijas, pēdējais nodalījums var tikt izlaists. Ja klase nesatur ne atribūtus, ne operācijas, tukšie nodalījumi var būt izlaisti, atstājot tikai klases nosaukumu. Atribūta vai operācijas parametra datu tipu var norādīt ar kolu un datu tipa nosaukumu aiz atribūta vai parametra nosaukuma.

1.4. att. **Klases grafiskā notācija**

Atribūtu var norādīt arī ar asociācijas palīdzību, velkot šķautni no klases, kas satur atribūtu uz klasi, kas apraksta atribūta datu tipu, liekot bultiņu un pierakstot atribūta nosaukumu jeb lomas vārdu datu tipa klases asociācijas galā. Piemēram, *1.5. att. Atribūtu notācijas* redzami ekvivalenti pieraksti. Šajā darbā tiek izmantotas abas notācijas, dodot priekšroku asociācijas notācijai, gadījumos, kad atribūta tips ir cita klase, bet tomēr izmantojot pirmo notāciju gadījumā, ja asociācija pārāk sarežģī diagrammas izskatu.



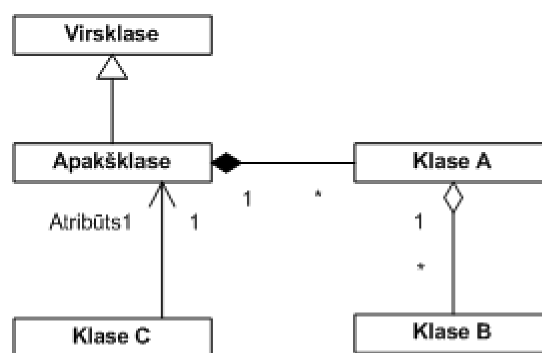
1.5. att. **Atribūtu notācijas**

Asociācijas starp klasēm apraksta attiecības starp sistēmas jēdzieniem. Papildus jau apskatītajai asociācijai, kas apraksta atribūtu, iespējamas cita veida asociācijas, piemēram, mantošanas relācija, kompozīcijas un agregācijas asociācijas.

Mantošanas relācija apraksta virsklases un apakšklases attiecību. Apakšklase manto visus virsklases un apakšklases atribūtus un operācijas un var papildināt to klāstu ar jauniem atribūtiem un jaunām operācijām. Mantošanas relāciju attēlo ar baltu bultiņu virsklases galā.

Kompozīcijas asociācija apraksta piederības attiecību, tādējādi norādot, ka īpašnieka instances sastāv no piederīgās klases instancēm. Iznīcinot īpašnieka klases instanci, tiek iznīcinātas piederīgās instances. Kompozīciju attēlo ar aizpildītu melnu rombu asociācijas īpašnieka klases galā.

Agregācijas asociācija, līdzīgi kompozīcijas asociācijai, apraksta objektu piederību, taču atšķirībā no kompozīcijas, klases instancei piederīgās instances ir neatkarīgas – īpašnieka instances iznīcināšana ne obligāti iznīcina piederīgās instances. Agregāciju attēlo ar neizpildītu rombu īpašnieka klases galā.



1.6. att. Asociāciju veidu notācija

Mantošanās relācijas, kompozīcijas un agregācijas asociācijas grafiskā notācija attēlotas *1.6. att. Asociāciju veidu notācija*.

Šajā darbā metamodeļi tiek aprakstīti ar MOF līdzekļu palīdzību.

1.4. MDA izstrāde LU MII

Latvijas Universitātes matemātikas un informātikas institūts (turpmāk LU MII) veic pētījumus MDA virzienā. LU MII izstrādā vairākus projektus, kas balstās uz MDA idejām, to skaitā ir rīki modeļu transformāciju definēšanai, repozitorijs modeļu glabāšanai un grafisko rīku izstrādes platforma. Šajā sadaļā aprakstīti iepriekš uzskaitītie LU MII izstrādātie komponenti un to pielietojums.

Viens no centrālajiem komponentiem MII izstrādē ir MII repozitorijs. MII repozitorijs ir efektīva datu glabātuve, kura satur datus modeļu formā. Repozitorijs var saturēt metamodeļu definīcijas un modeļu instances. Tas tiek izmantots kā datu glabātuve vairāku citu rīku izstrādē. (5)

Lx valodu saime, kas sastāv no valodām L0, L1, L2 un L3, ir tekstuālu valodu saime modeļu transformāciju definīciju izstrādei, kas satur konstrukcijas klašu, to atribūtu un attiecību izveidei un manipulācijai. Lx programma sastāv no metamodeļu definīcijām un procedūrām, kas apraksta modeļu transformāciju no avota metamodeļa uz mērķa metamodeli (6), (7).

MOLA ir grafiska valoda transformāciju definēšanai – valodas mērķis ir sniegt vienkāršu un viegli lasāmu transformācijas definīciju. MOLA valodai pastāv grafiska redaktora realizācija, kas ļauj vizuāli specificēt modeļu transformācijas. MOLA transformācijas programma sastāv no avota un mērķa metamodeļiem un MOLA diagrammu kopas, kas definē transformācijas procedūru. MOLA programma transformē avota metamodeļa instanci par mērķa metamodeļa instanci. Transformācijā var tikt izmantoti tādi tradicionāli programmēšanas elementi kā komandu virkne, cikls, zarošanās. MOLA transformācijas tiek kompilētas par Lx valodas procedūrām (5).

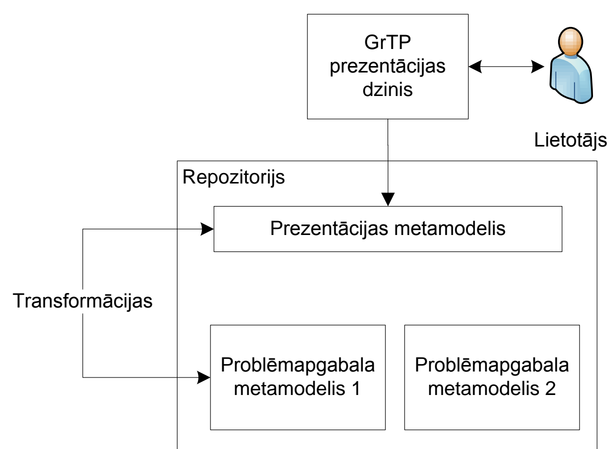
Viens no LU MII projektiem ir grafisko rīku būves platformas Graphical Tool Building Platform (GrTP) izstrāde, kura tuvāk apskatīta nākamajā sadaļā.

1.4.1. GrTP grafisko rīku būves platforma

Grafu diagrammas ir populārs informācijas vizualizācijas veids, kas ļauj uzskatāmi demonstrēt dažādu profilu informāciju. Šādām diagrammām ir plašs pielietojums daudzās nozarēs – piemēram, programmizstrādē, kur diagrammas tiek lietotas programmatūras projektēšanai un dokumentēšanai, pārvaldībā, kur aktivitāšu diagrammas spēj uzskatāmi attēlot dažādus procesus, u.c. lietojumos. Dažādām diagrammām ir dažādi lietotāji ar atšķirīgām prasībām.

GrTP piedāvā ietvaru, kas ļauj ātri izstrādāt atbalstu dažāda tipa diagrammu vizualizācijai. GrTP sastāv no grafiskā redaktora, prezentācijas dziņa, prezentācijas metamodeļa un dažādajiem problēmapgabala metamodeļiem (skat. *1.7. att. GrTP ietvara arhitektūra).*

Katrs no problēmapgabalu metamodeļiem apraksta kādu no diagrammu tipiem, kurus nepieciešams vizualizēt. Piemēram, tiek specificēts metamodelis UML diagrammu aprakstīšanai. Līdzīgi var tikt definēts metamodelis E-R diagrammu aprakstīšanai. MDA terminoloģijā šie metamodeļi ir PIM.



1.7. att. **GrTP ietvara arhitektūra**

GrTP definē prezentācijas metamodeli, kurš ļauj aprakstīt grafus, definēt to izskatu. Šis modelis MDA terminoloģijā ir PSM. Gan prezentācijas metamodeļi, gan dažādie problēmapgabalu metamodeļi, kā arī šiem metamodeļiem atbilstošie modeļi tiek glabāti MII repozitorijā.

Prezentācijas dzinis vizualizē prezentācijas metamodelim atbilstošus modeļus. Lai prezentācijas dzinis spētu attēlot dažādos diagrammu tipus, kuriem GrTP ir definēts metamodelis, tiek definētas transformācijas no problēmapgabala PIM uz prezentācijas metamodeļa PSM, izmantojot iepriekš aprakstītos rīkus. Tādā veidā tiek nodrošināta prezentācijas dziņa atkalizmantojamība – nav nepieciešams veidot jaunus prezentācijas dziņus katram jaunam diagrammas tipam. Sākotnējā GrTP platformas izstrāde ir parādījusi, ka pieeja ir efektīva no praktiskā viedokļa (7).

Jāatzīmē, ka šajā lietojumā ir interesanta atšķirība no klasiskās MDA filozofijas – šajā lietojumā nenotiek sistēmas koda ģenerēšana no PSM. Tā vietā tiek veikta transformācija (kuras definīcija tiek rakstīta manuāli) no PIM uz PSM, kuru spēj attēlot prezentācijas dzinis.

Lai arī animācijas var būt noderīgas dažādu diagrammu uzskatāmības palielināšanā, šobrīd GrTP prezentācijas metamodelī nav līdzekļu animāciju definēšanai. Lai integrētu animāciju GrTP platformā, nepieciešams metamodelis, kas spētu aprakstīt animācijas. Kā jau minēts ievadā, šī darba mērķis ir izstrādāt principus animācijas metamodeļa būvei. Šos principus varētu pielietot GrTP prezentācijas metamodelim, lai pievienotu platformai spēju attēlot animācijas, tādējādi ievērojami paplašinot platformas iespējas.

1.5. Kopsavilkums

Šajā sadaļā tika apskatīti MDA ietvara pamatjēdzieni, pamatidejas un izstrādes process, kā arī problēmas tradicionālā programmatūras izstrādē un kā tās tiek risinātas MDA ietvarā. Tika izklāstīti LU MII pētījumi MDA virzienā un izstrāde, kas balstās uz šī ietvara idejām. Detalizētāk apskatīta LU MII grafisko rīku izstrādes platformas GrTP arhitektūra un izklāstīta motivācija šī darba veikšanai.

Lai iegūtu tās priekšrocības, ko sniedz MDA ietvara izmantošana programmatūras izstrādes procesā, nepieciešams nostādīt modeli centrālajā lomā programmatūras izstrādē. Lai aprakstītu un operētu ar animācijām izmantojot MDA rīkus, nepieciešams definēt metamodeli animāciju aprakstīšanai.

2. DATORANIMĀCIJA

Animācija ir secīgu attēlu producēšana, kas, secīgi rādīti pietiekoši lielā ātrumā, rada kustības ilūziju. Lai radītu plūstošu kustību, katram attēlam jeb kadram animācijas virknē plūstoši jāsaskan ar pārējiem kadriem. (9)

Tradicionāli, animācija tika radīta zīmējot attēlus katram darbības kadram. Parasti animācijas process norit sekojoši – animators sākotnēji izveido zīmējumus svarīgākajiem animācijas kadriem jeb atslēgkadriem. Pēc tam animators vai viņa asistents, vadoties pēc atslēgkadriem, izveido zīmējumus kadriem starp atslēgkadriem. Animācijas vēstures gaitā ir izgudrotas vairākas pieejas un tehnikas animāciju veidošanā. Taču animēšana, izmantojot atslēgkadrus, un animēšana pa kadriem ir pamatveids kā tiek veidotas animācijas. (9)

Datoranimācija ir animāciju veidošana izmantojot datorus. Ir izstrādāts plašs klāsts ar rīkiem, kas atbalsta animācijas procesu un padara animāciju veidošanu vieglāku. Eksistē rīki gan divdimensionālas grafikas animēšanai, gan trīsdimensionālu objektu animācijai.

Lai arī animācijas process, izmantojot datorus, ir daudz ērtāks nekā tradicionālā animācija, tas ir līdzīgs tradicionālajam animācijas procesam.

Datoranimācijas rīki palīdz animatoram, automātiski ģenerējot daļu no animācijas kadriem. Viena no pamatmetodēm kadru ģenerēšanai ir vērtību interpolācija. Animators izveido sarakstu ar animējamā parametra vērtībām animācijas atslēgkadros, bet datorprogramma interpolē vērtības starp atslēgkadriem. Viens no vienkāršākajiem piemēriem ir punkta pozīcijas vērtību interpolācija – ja animators vēlas, lai objekts atrastos punktā (-5,0) 22. kadrā, bet 72. kadrā tas atrastos punktā (5, 10), animācijas rīkam nepieciešams ģenerēt kadrus sākot no 23. kadra līdz 72. kadram. Interpolācijai var izmanot vairākas metodes, vienkāršākā no tām ir lineāra interpolācija. Jebkura maināma objekta atribūta vērtība var tikt animēta ar aprakstīto metodi. Šādā veidā animējama punkta pozīcija telpā, objekta caurspīdīguma atribūts, objekta krāsa vai jebkurš cits atribūts, kurš tiek izmantots, lai manipulētu vai raksturotu grafisku elementu. (10)

Specializēti rīki, kuri nav paredzēti pilnīgi brīvu animāciju realizācijai, parasti iekļauj kopu ar gatavām animācijām, kuras pievienot kompozīcijas elementiem. Piemēram, Microsoft PowerPoint ļauj pievienot dažādas gatavas animācijas slaidu elementiem, piemēram, teksta „ielidošanu" no ekrāna malas. Šādas animācijas ir realizējamas arī ar tradicionālu atslēgkadru pieeju.

Šajā darbā centrālā uzmanība pievērsta animācijām, kuras var realizēt aprakstot objekta stāvokli atslēgkadros un interpolējot vērtības starp tiem.

3. EKSISTĒJOŠIE RISINĀJUMI ANIMĀCIJAS IZSTRĀDEI

Šajā nodaļā aplūkoti programmizstrādes nozarē plaši izmantoti risinājumi, kuri ietver animāciju veidošanas funkcionalitāti. Apskatītas divas populāras klašu bibliotēkas, kas izmantojamas programmējot lietojumprogrammu lietotāju saskarni, un divi programmatūras produkti, kas piedāvā animāciju veidošanas funkcionalitāti – multimediju satura izstrādes produkti Adobe Flash un Microsoft Expression Blend. Apskatot klašu bibliotēku animāciju veidošanas funkcionalitāti, mēģināts izvilkt to animāciju aprakstošo metamodeļa fragmentus, izmantojot pieejamo tehnisko dokumentāciju.

3.1. Windows Forms

Windows Forms (turpmāk WinForms) ir Microsoft izstrādāta tehnoloģija, kas ļauj veidot interaktīvas lietotāja saskarnes Windows saimes operētājsistēmām. WinForms ir .NET Framework sastāvdaļa kopš versijas 1.0, līdz ar to tā ir viena no nobriedušākajām un populārākajām tehnoloģijām klienta programmu veidošanai (11).

Ar WinForms iespējams veidot programmas, kas attēlo informāciju, pieprasa lietotāja ievadu un reaģē uz lietotāja darbībām.

Lietotāja saskarnes logs WinForms kontekstā tiek saukts par formu. Forma ir vizuāla virsma, kura var saturēt lietotāja saskarnes elementus vai citas formas.

Lietotāja saskarnes elements, kas attēlo datus un saņem lietotāja ievadu, tiek saukts par kontroli. WinForms bibliotēka satur plašu klāstu ar kontrolēm, kuras iespējams novietot uz formas – teksta ievades laukus, spiedpogas, izkrītošās izvēlnes, paneļus, rīkjoslas, izvēlnes, u.c. Tāpat, WinForms bibliotēkas satur virkni ar nevizuāliem komponentiem, piemēram, taimera klasi, klasi skaņu atskaņošanai, klasēm komunikācijai ar datora pieslēgvietām, u.c.

Lietotāja saskarnes elementi un citi komponenti var definēt notikumus, kurus programmētājs var apstrādāt, lai veiktu nepieciešamās darbības. Piemēram, spiedpoga definē notikumu peles klikšķim - programmētājs var apstrādāt notikumu, lai lietotājam noklikšķinot uz spiedpogas tiktu izvadīts paziņojums.

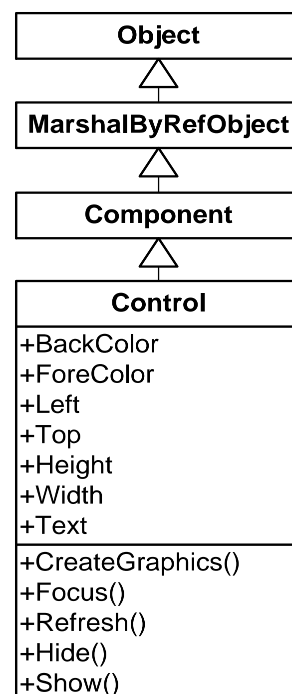
Visbeidzot, WinForms bibliotēka piedāvā klases grafisko elementu zīmēšanai uz izvadierīcēm, ja kontroļu piedāvātās iespējas ir nepietiekamas.

WinForms piedāvā daudz citas iespējas, tai skaitā datu piesaisti starp kontrolēm un datu avotiem, vieglu programmu instalēšanu atjaunināšanu u.c. iespējām. Pilns WinForms funkcionalitātes apskats ir ārpus šī darba aptvēruma. Turpmākajās sadaļās tuvāk tiek apskatīta WinForms grafikas sistēma un animāciju veidošanas iespējas.

3.1.1. Windows Forms grafikas sistēma

Kā jau minēts iepriekš, WinForms piedāvā plašu bibliotēku ar klasēm lietotāja saskarnes veidošanai. WinForms vizuālais pamatelements ir kontrole, kuru definē bāzes klase `Control`. Klases hierarhija līdz `Control` klasei redzama attēlā *3.1. att. Control klase*. Attēlā redzamas sekojošas klases:

- `Object` – bāzes klase visiem .NET Framework objektiem, realizē pamatfunkcionalitāti.
- `MarshalByRefObject`, `Component` – realizē WinForms dienesta funkcionalitāti sadarbībai ar operētājsistēmu.
- `Control` klase definē atribūtus kontroles elementa izmēra definēšanai, novietojuma definēšanai, priekšplāna un fona krāsas definēšanai.

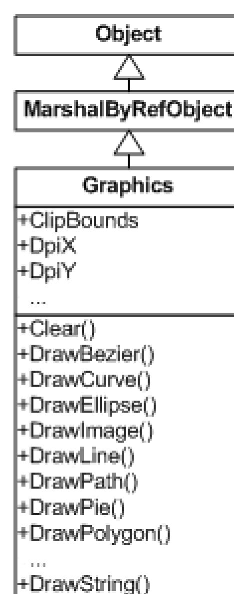


3.1. att. **Control klase**

`Control` klases apakšklases realizē kādu lietotāja saskarnes elementu, piemēram, `PictureBox` realizē virsmu, kura spēj attēlot rastra grafiku, `TextBox` klase realizē teksta ievades lauku, `ListBox` realizē izvēles sarakstu. `Control` klašu hierarhijas fragments pievienots pielikumā (skat. *2. Pielikums – Windows Forms kontroļu klašu hierarhijas fragments*).

WinForms logu apraksta `Form` klase, kura arī ir `Control` klases apakšklase. Klase definē `Children` atribūtu – kolekciju ar `Control` objektiem, tādējādi, forma var saturēt lietotāja saskarnes elementus. (12)

WinForms nepiedāvā iebūvētas klases ģeometrisku figūru zīmēšanai. Tā vietā tiek piedāvāta `Graphics` klase, kas satur metodes grafisko elementu kā līniju, līkņu renderēšanai uz izvadierīcēm (skat. *3.2. att. Graphics klase*). `Graphics` klases objektu var iegūt, izsaucot `Control` klases objekta `CreateGraphics()` metodi, iegūstot objektu, kurš izmantojams renderēšanai uz formu un kontroļu virsmām. (12)



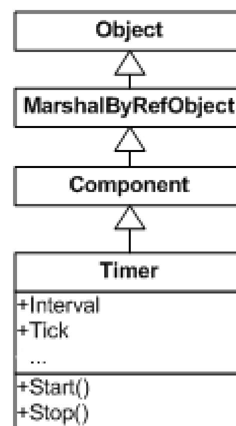
3.2. att. **Graphics klase**

Renderēšanas rezultāts ir rastra grafikas pikseļi, kuri programmatiski nav manipulējami savādāk kā pārzīmējot tiem pāri jaunus pikseļus. Šī iemesla dēļ šāda pieeja nav piemērota animācijas realizēšanai deklaratīvā stilā.

3.1.2. Animācijas ar Windows Forms

WinForms bibliotēkas nepiedāvā specializētas iebūvētas klases, kas ļautu definēt animāciju. Tīmeklī eksistē vairāki tās lietotāju izstrādāti risinājumi, taču tie visi paļaujas uz sekojošo ideju – izmantot taimeru objektus un pēc fiksēta laika intervāla manuāli pārvietot formas kontroles vai pārzīmēt formas virsmu (13), (14).

Taimera funkcionalitāti realizē klase `Timer` (skat. *3.3. att. Timer klase*). Startējot `Timer` klases objektu, tas pēc noteikta intervāla izraisa „tikšķa" notikumu. Klase definē atribūtu `Interval`, kas nosaka intervālu starp „tikšķa" notikumiem, un atribūtu `Tick` – notikuma objektu, kas satur kolekciju ar metodēm, kuras izsaukt katru reizi, kad taimeris „tikšķ". (5)



3.3. att. **Timer klase**

Viens no vienkāršākajiem veidiem kā realizēt kāda elementa animāciju ir izveidot `Timer` objektu, uzstādīt vēlamo intervālu, apstrādāt `Tick` notikumu, kuram notiekot, mainīt animējamā objekta atribūtu atkarībā no pagājušā laika.

Sekojošais programmas fragments demonstrē iepriekš aprakstīto pieeju:

```
...
public Program() {
   ...
   // Izveido melnu kvadrātu ar 100 vienību garu malu.
   r = new PictureBox();
   r.Height = r.Width = 100;
   r.BackColor = Color.Black;

   animationTimer = new Timer( );                        // Izveido Timer objektu
   animationTimer.Tick += new EventHandler( t_Tick ); // notikuma apstrādes funkcija
   animationTimer.Interval = 10;

   animationStartTime = DateTime.Now;
   animationTimer.Start(); // signalizē notikumu pēc 10 milisekundēm.
}

private void t_Tick( object sender, EventArgs e ) {
   DateTime now = DateTime.Now;

   // Aprēķina pagājušo laiku koopš animācijas sākuma
   TimeSpan elapsed = now - animationStartTime;

   // Ja laiks kopš animācijas sākuma lielāks par animācijas ilgumu - apstāties
   if ( animationDuration < elapsed )
      animationTimer.Stop();

   // Aprēķina animācijas progresu kā attiecību starp intervālu kopš animācija sākuma
   // un animācijai atvēlēto intervālu
   double progress = elapsed.Ticks / (double)animationDuration.Ticks;

   // Uzstāda animējamā objekta atribūtam jauno vērtību
   r.Left = animationStartValue +
       (int)(progress * ( animationEndValue - animationStartValue ));
}
...
```

Programma izveido `PictureBox` kontroli un `Timer` objektu. `Timer` objekta intervāls tiek uzstādīts uz 10 milisekundēm. Uz katra `Timer` objekta `Tick` notikuma tiek aprēķināta `PictureBox` kontroles jaunā atrašanās vieta. Pilns programmas teksts pievienots darba pielikumā (skat. *3. Pielikums – Animācijas programma, izmantojot Windows Forms*).

Šāda pieeja, speciālas apstrādes definēšana katram animējamam objektam, ir darbietilpīga. Vienkāršas lineāras animācijas definēšanai vajadzēja rakstīt daudz koda, tādējādi viegli pieļaut kļūdas. Kods kļūst sarežģītāks, ja nepieciešama sarežģītāka animācija, piemēram, nepieciešama kustība ar dažādu pārvietošanās ātrumu dažādos kustības posmos.

Šāda animēšanas metode slikti izmantojama MDA ietvaros, jo paļaujas uz imperatīvu realizāciju.

3.2. Windows Presentation Foundation

Windows Presentation Foundation (turpmāk WPF) ir Microsoft izstrādāta prezentācijas sistēma, kas vienotā ietvarā apvieno virkni datora izvada servisu: interaktīvu lietotāja saskarni, divdimensiju un trīsdimensiju zīmēšanu un attēlu attēlošanu un manipulāciju, dokumentu renderēšanu un drukāšanu, runas, audio un video servisus (16). Papildus tam, WPF piedāvā arī servisus animāciju veidošanai, kuriem arī šajā darbā tiek veltīta lielākā uzmanība.

WPF ir Microsoft .NET Framework izstrādes ietvara sastāvdaļa. Microsoft .NET Framework ir populārs programmēšanas ietvars programmatūras izstrādei programmatūrai, kas paredzēta darbināšanai Microsoft produktu platformā. .NET Framework ir iespējams izmantot, lai realizētu dažādu profila programmatūru – tīmekļa programmas, servera programmas, Windows klienta programmas, un citu profilu programmas (17). WPF pievienota .NET Framework 3.0 versijā, kas pieejama kopš 2006. gada rudens. WPF tiek uzskatīta par galveno API, kas paredzēta Windows lietotāja saskarņu programmēšanai (18), taču tās lietošana līdz 2008. gadam vēl nav sasniegusi plašu popularitāti.

WPF projektējumu īpaši interesantu tuvākai apskatei padara viens no galvenajiem WPF arhitektūras filozofijas principiem – priekšrokas došana klašu atribūtiem, nevis metodēm vai notikumiem. Atribūti pēc dabas ir deklaratīvi, pretēji metožu un notikumu imperatīvajam raksturam.

WPF iespējams programmēt ar divām pieejām – ar tradicionālu imperatīvu .NET Framework saimes valodu, piemēram, C# vai Visual Basic .NET, un ar jaunu iezīmēšanas valodu XAML (eXtensible Application Markup Language), kas ir XML stila valoda. XAML valoda ir ļoti piemērota, lai aprakstīt vizuālo elementu hierarhiju. Lai arī ir iespējamas

programmas, kas pilnībā rakstītas XAML, tomēr tipiska WPF programma saturēs gan XAML kodu, gan imperatīvu kodu. XAML, tāpat kā XML, ir deklaratīva valoda, tādēļ arī WPF klases ir projektētas tā, lai programmas pēc iespējas būtu iespējams rakstīt deklaratīvā stilā. Tomēr, ir iespējams rakstīt programmas, kas izmanto tikai imperatīvu kodu – vienkāršības labad tāda pieeja lietota šajā darbā – visi piemēri rakstīti valodā C#.

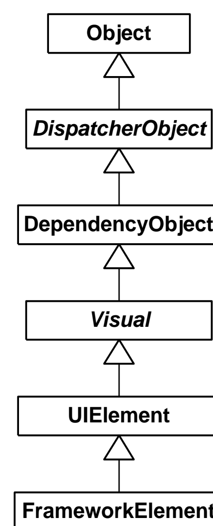
Ar WPF risināto problēmu un visu tā piedāvāto iespēju pilns apskats ir ārpus šī darba aptvēruma. Turpmākajās sadaļās uzmanība koncentrēta uz WPF servisiem, kas nodrošina animāciju veidošanu kā arī servisiem, kas nodrošina divdimensiju grafikas zīmēšanu.

3.2.1. WPF divdimensiju grafikas sistēma

Lai būtu iespējams demonstrēt WPF animāciju principus, vispirms nepieciešami objekti kurus animēt. Šajā nodaļā tiek apskatīta WPF piedāvāto servisu apakškopa, kura nodrošina divdimensiju grafikas zīmēšanu un izvietošanu uz ekrāna.

WPF prezentācijas servisus programmētājam pasniedz ar objektorientētas API palīdzību – klašu kopas bibliotēku, kuras klases kopā realizē WPF servisu funkcionalitāti.

Pirms apskatīt konkrētas klases, kas nodrošina konkrētu elementu zīmēšanu uz izvadierīces, ir derīgi apskatīt bāzes klases, lai saprastu WPF arhitektūras nianses, kas ļauj realizēt konkrēto klašu funkcionalitāti. Zemāk ir uzskaitītas WPF bāzes klases, no kurām mantojas konkrēto grafisko elementu klases.



3.4. att. **WPF dienesta funkcionalitātes bāzes klases**

Klašu attiecības ir attēlotas UML diagrammā *3.4. att. WPF dienesta funkcionalitātes bāzes klases*. Būtiskākās WPF bāzes klases ir:

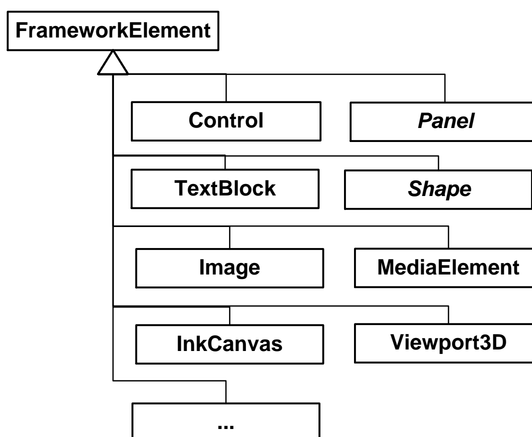
- `Object` – pamata klase, no kuras mantojas visas .NET Framework bibliotēku klases. `Object` klase nodrošina tādu pamatfunkcionalitāti, kā objektu vienādības noteikšanu, simbolu virknes reprezentācijas iegūšanu, objektu hešošanu, un objekta klases tipa iegūšanu. Šī klase nav specifiska WPF, taču kā jau minēts, visas .NET Framework klases, tai skaitā WPF klases, mantojas no `Object`.
- `DispatcherObject` – abstrakta WPF dienesta klase, kas nodarbojas ar paralēlās izpildes iespēju nodrošināšanu.
- `DependencyObject` – WPF dienesta klase, kas realizē paplašinātu atribūtu sistēmu, kas nodrošina atribūtu vērtību izmaiņu signalizēšanu, atribūtu sasaisti ar citiem atribūtiem. `DependencyObject` klase ievieš jaunu ievērības cienīgu jēdzienu - pievienotie atribūti (attached properties), kas ļauj objektiem jebkurai klasei, pievienot/definēt klases

atribūtu vērtības atribūtiem, kuri nav statiski specificēti. Piemēram, WPF teksta ievades lauka klasei `TextBox` nav atribūta, kas aprakstītu teksta lauka koordinātes uz ekrāna, jo WPF atbalsta vairākas izvietošanas paradigmas (izvietošanu orientētu uz absolūtām koordinātām, automātisku izvietošanu, kas orientēta uz tabulām vai plūsmām u.c.). Tamdēļ, lai aprakstītu `TextBox` objektu koordinātas, ja tiek izmantota izvietošana ar koordinātām, tiek lietoti pievienotie atribūti, lai definētu teksta lauka klasei X un Y koordinātas.

- `Visual` – abstrakta WPF dienesta klase, kas definē ietvara metodes vizuālo elementu zīmēšanai uz izvadierīcēm.
- `UIElement` – WPF dienesta klase, kas pievieno funkcionalitāti elementu izvietošanai uz izvada, kā arī nodrošina notikumu (*event*) paziņošanas un apstrādes sistēmu. Šī klase definē nepieciešamo struktūru lietotāja saskarnes notikumiem un renderēšanai uz izvadiedrīcēm, kas var noderēt dažādiem prezentācijas ietvariem. WPF ir viens no šādiem ietvariem, taču citi ietvari varētu izmantot `UIElement` klasi, lai realizētu citu elementu kompozīcijas filozofiju.
- `FrameworkElement` – dienesta klase, nodrošina vairākas būtiskas WPF infrastruktūras funkcijas, piemēram, datu piesaisti (*data binding*) WPF objektu atribūtiem, elementa trigeru kolekciju, kas satur trigerus, kas deklaratīvā stilā ļauj aprakstīt darbības, kuras izpildīt iestājoties kādam notikumam, u.c. funkcijas.

Kopsavelkot būtiskāko no iepriekšējām rindkopām, visas .NET Framework klases, tajā skaitā WPF klases, mantojas no `Object`. WPF papildina atribūtu sistēmu ar klasi `DependancyObject`, ieviešot atribūtu izmaiņu signalizēšanu ar notikumu mehānismu, kā arī ieviešot *pievienotā atribūta* jēdzienu. (19)

Konkrētie vizuālie elementi, kuri apskatīti turpmāk, mantojas no kādas no iepriekš uzskaitītajām klasēm. No `FrameworkElement` klases mantojošo klašu skaits ir pārāk liels, lai šajā darbā uzskaitītu visas; tiek aplūkotas tikai tipiski izmantojamās klases. Tipiski izmantojamās klases, kas tiešā veidā mantojas no `FrameworkElement` apkopotas zemāk (skat. arī *3.5. att. WPF vizuālo elementu klases*):



3.5. att. **WPF vizuālo elementu klases**

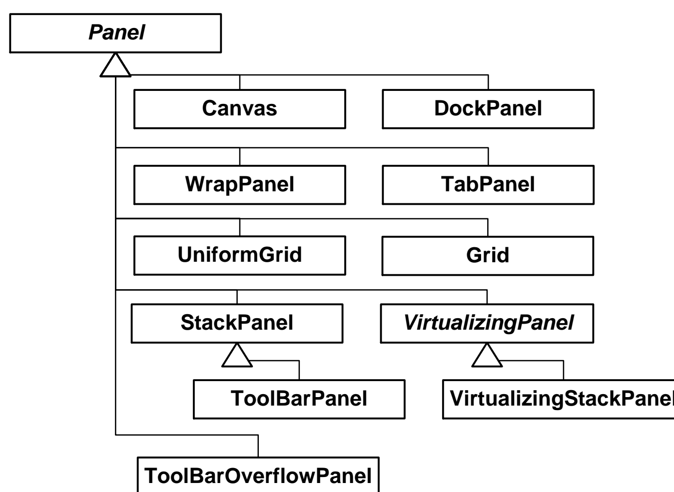
- `Panel` – abstrakta klase, kas definē saskarnes metodes klasēm, kas nodarbojas ar divdimensiju vizuālo elementu izvietošanu

- `Shape` – abstrakta klase, kas definē API saskarni divdimensiju figūru attēlošanai
- `Control` – bāzes klase interaktīvajām kontrolēm, tādām kā teksta ievades lauki, izkrītošās izvēlnes, izvēles saraksti, spiedpogas, u.c. Elementi, kas mantojas no šīs klases tipiski atšķiras no citiem elementiem ar to, ka tie ir spējīgi saņemt lietotāja ievadu – teksta ievadu, peles klikšķus, ievadu no planšetes un citus ievades veidus. Pilns WPF pieejamo kontroļu apskats ir ārpus šī darba aptvēruma, taču pilna WPF kontroļu hierarhija pievienota darba pielikumā *4. Pielikums – WPF interaktīvo lietotāja saskarnes klašu hierarhija*.
- `TextBlock` – elementu klase, kuras objekti satur noformētu tekstu
- `Image` – klase dažādu formātu rastra attēlu attēlošanai
- `MediaElement` – klase audio un video elementu aprakstīšanai
- `InkCanvas` – klase, kas apraksta elementu, kas nodrošina virsmu ievada saņemšanai no planšetes.
- `Viewport3D` – elements, kas nodrošina virsmu trīsdimensiju elementu attēlošanai

Tuvāk tiek aplūkotas divas klašu apakšhierarhijas – `Panel` un `Shape`, un to apakšklases.

`Panel` klase, kā jau minēts iepriekš, apzīmē elementu, kurš nodrošina virsmu, kura var saturēt citus elementus un realizē kādu izvietošanas algoritmu. Pilna `Panel` apakšklašu diagramma klasēm, kas iekļautas WPF, redzama attēlā *3.6. att. Panel klašu hierarhija*.

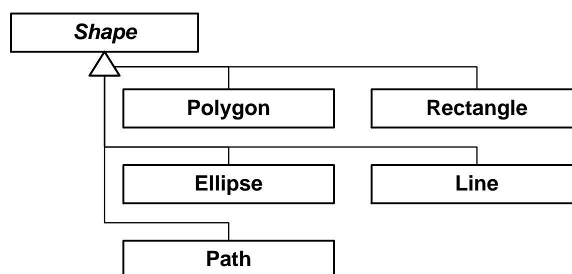
`Canvas` klase realizē tradicionālu manuālās izvietošanas algoritmu, kur elementi uz virsmas tiek izvietoti balstoties uz uzstādītajām elementu koordinātām. Citas `Panel` apakšklases realizē automātiskas izvietošanas algoritmus, kur koordinātas tiek izrēķinātas automātiski. Piemēram, `StackPanel` elementi tiek novietoti viens aiz otra, `Grid` elementi tiek novietoti tabulā, `DockPanel` elementi tiek novietoti gar virsmas malām. Interesanti atzīmēt, ka tā kā elements var atrasties uz dažādām virsmām, kurās izvietojums nosakās pēc dažādiem parametriem (piemēram, `Canvas` gadījumā tās ir koordinātas, `StackPanel` gadījumā tā ir elementu secība, utml.), atribūti, kas satur izvietošanas algoritmam nepieciešamās vērtības, realizēti kā pievienotie atribūti, kas pieminēti iepriekš, tādējādi jebkuru vizuālo elementu var novietot uz jebkura `Panel`



3.6. att. **Panel klašu hierarhija**

apakšklases elementa, tam statiski nedefinējot liekus atribūtus, kuri lielākoties paliktu neizmantoti. Šī darba piemēros tiks izmantots `Canvas` klases piedāvātais manuālās izvietošanas algoritms.

`Shape` klase ir abstrakta bāzes klase ģeometrisku figūru attēlošanai. Klase definē atribūtus elementu krāsai un izmēram. Zemāk uzskaitītas visas `Shape` apakšhierarhijas klases (skat. arī *3.7. att. Shape klašu hierarhija*):



3.7. att. **Shape klašu hierarhija**

- `Shape` – abstrakta bāzes klase ģeometriskām figūrām
- `Ellipse` – klase, kas apraksta elipses elementu
- `Line` – klase, kas apraksta nogriežņa elementu
- `Path` – klase, kas apraksta vairāku posmu elementu, kas var sastāvēt no nogriežņiem un līknēm
- `Polygon` – klase, kas apraksta daudzstūra elementu

WPF logu apraksta `Window` klase (`Window` klases novietojumu WPF klašu hierarhijā skat. *4. Pielikums – WPF interaktīvo lietotāja saskarnes klašu hierarhija*). `Window` klase definē `Content` atribūtu, kura vērtība nosaka loga saturu. Logs parasti satur kādu no izkārtojuma paneļiem nepieciešamo lietotāja saskarnes elementu izvietošanai.

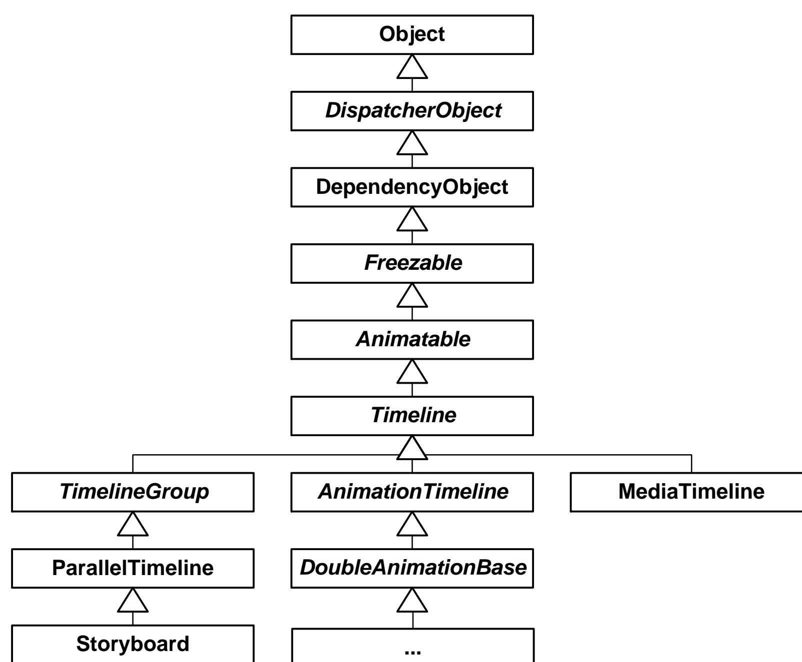
Šajā sadaļā pārskata veidā tika aplūkotas WPF divdimensiju grafikas iespējas, kas ir tikai daļa no WPF piedāvātajām iespējām, taču ar apskatīto ir pietiekami, lai būtu iespējams aplūkot animācijas sistēmu, kas apskatīta nākamajā nodaļā.

3.2.2. Animācijas ar WPF

WPF piedāvā plašu klāstu ar iebūvētām klasēm, kas ļauj deklaratīvā stilā definēt animācijas.

Animācijas pamatideja – laikā mainīt animējamā objekta atribūtu vērtības. Animācijas procesā piedalās divas klases – animējamais objekts un ārējs animācijas objekts.

WPF klašu hierarhijas fragments klasēm, kas attiecas uz objektu animēšanu redzams *3.8. att. WPF animāciju bāzes klašu hierarhija*. Klases `Object`, `DispatcherObject`, `DependencyObject` realizē dienesta funkcionalitāti un tika apskatītas iepriekšējā sadaļā (skat. sadaļu *3.2.1. WPF divdimensiju grafikas sistēma*) .



3.8. att. **WPF animāciju bāzes klašu hierarhija**

Zemāk ir uzskaitītas un īsi apskatītas bāzes klases, kuras atbild par objektu animāciju:

- `Freezable` – dienesta klase, kas definē atribūtus un metodes, kas nosaka vai objekta ir modificējams vai tikai lasāms. Šī klase spēlē lielu lomu WPF ātrdarbības uzlabošanai, bet netiks apskatīta sīkāk.
- `Animatable` – Abstrakta klase, kas nodrošina dienesta funkcionalitāti specifisku animāciju atbalstam.
- `Timeline` – klases objekti attēlo laika segmentu. Klases definē atribūtus, kas ļauj specificēt segmenta garumu, laiku, kad tas sākas, cik daudz reizes tas atkārtojas, u.c. `Timeline` klase definē šādus atribūtus (skat. arī *3.9. att. Timeline klases atribūti un operācijas*):
    - `Name` – atribūts var saturēt unikālu nosaukumu, lai programmatiski piekļūtu `Timeline` objektam

- `Duration` – aprakstītā laika segmenta garums
- `BeginTime` – laika nobīde; laika segments no momenta, kad saņemts stimuls `Timeline` sākšanai līdz brīdim, kad sākas `Timeline` aprakstītais laika segments
- `AutoReverse` – atribūts, kas nosaka vai beidzoties laika segmentam tas tiek atkārtots pretējā virzienā.
- `FillBehaviour` – atribūts, kas nosaka uzvedību beidzoties laika segmentam. Iespējamie varianti – tiek atiestatīts sākotnējais stāvoklis vai tiek uzturēts stāvoklis kāds sasniegts beidzoties animācijai.
- `RepeatBehaviour` – apraksta, cik reizes atkārtot laika segmentu.

Timeline
+Name +Duration +BeginTime +AutoReverse +FillBehaviour +RepeatBehaviour +SpeedRatio +AccelerationRatio +DecelerationRatio
+Clone() +CloneCurrentValue() +CreateClock()

3.9. att. **Timeline klases atribūti un operācijas**

- `SpeedRatio` – nosaka laika segmenta laika ritējuma ātruma attiecību pret patieso laika ritējuma ātrumu. Piemēram, uzstādot vērtību 1,5, `Timeline` objektam, kura segmenta ilgums ir 30 sekundes, segments attiecībā pret objektīvo laiku ilgs 20 sekundes.
- `AccelerationRatio` un `DecelerationRatio` – nosaka cik lielu daļu no kopējā laika segmenta ilguma laiks attiecīgi paātrinās un palēninās..

No `Timeline` klases mantojas trīs apakšklases:

- `AnimationTimeline` – abstrakta klase, kuras apakšklases nodrošina objektu animāciju. Šī klase un tās apakšklases animē vizuālos objektus un sīkāk tiks apskatīta vēlāk.
- `TimelineGroup` – abstrakta klase, kas definē saskarni klasei, kura var saturēt citus `Timeline` klases objektus. Šīs klases apakšklašu objekti tiek izmantoti, lai grupētu saistītas `Timeline` klases. WPF šai klasei ir tikai viena apakšklase:
    - `ParallelTimeline` – klase, kas var saturēt citus `Timeline` klases objektus. Startējot `ParallelTimeline` objektu, tiek startētas visas tā saturētie `Timeline` objekti. WPF klašu kopā šai klasei ir viena apakšklase:
        - `Storyboard` – klase, kas nodrošina atribūtus animācijas mērķa objekta un mērķa atribūta norādīšanai, kas attieksies uz tā saturētajiem `Timeline` objektiem.
- `MediaTimeline` – klase, kas paredzēta multimediju failu atskaņošanas kontrolēšanai un ļauj tos kontrolēt ar tādu pašu saskarni kā animācijas. Šajā darbā netiks apskatīta sīkāk.

Kā jau minēts iepriekš, animācija tiek panākta laikā mainot objekta atribūtu vērtības. Animācijas klases objektam norāda mērķa objektu, kuru animēt un tā atribūtu, kura vērtības mainīt. Animācijas raksturu nosaka izmantotā animācijas klases tips. WPF ir trīs veidu animācijas – lineāras pārejas, līknes funkcijas pārejas un animācijas ar atslēgkadriem.

Lineāras pārejas – šāda tipa animācijai tiek norādītas animējamā atribūta vērtības animācijas sākumā un animācijas beigās. Atribūta vērtības laika momentā tiek izrēķinātas kā funkcija no vērtībām animācijas sākumā un beigās, un pagājušā laika kopš animācijas sākuma.

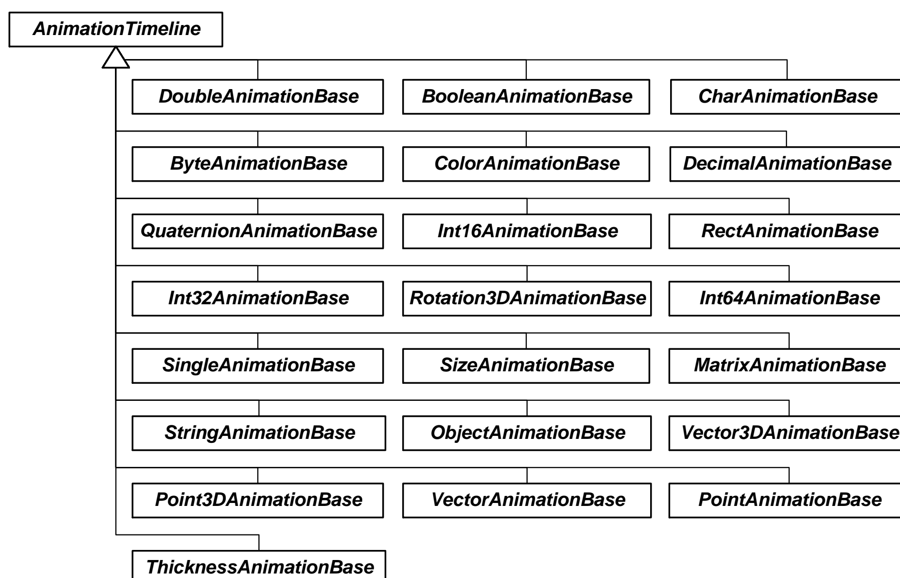
Līknes pārejas – atribūta vērtības laika momentā tiek rēķinātas no uzdotas līknes – vērtība laika momentā tiek iegūta no punkta uz līknes, kas atbilst pagājušajam laikam kopš animācijas sākuma. Piemēram, ja līknes garums ir 100 vienības, un animācijas kopējais garums ir 30 sekundes, un ir pagājušas 15 sekundes no animācijas sākuma, vērtība tiek noteikta no līknes punkta, kas atrodas 50 vienības no līknes sākumpunkta.

Atslēgkadru animācijas – atribūta vērtības laika momentā tiek rēķinātas izmantojot „momentuzņēmumus" jeb atslēgkadrus dažādos laika momentos. Atslēgkadrs satur atribūtus laika momentam un vērtību šajā laika momentā. Animējamo atribūtu vērtības laika momentos starp atslēgkadriem tiek izrēķinātas atkarībā no vērtības iepriekšējā atslēgkadrā, vērtības nākošajā atslēgkadrā un nākošā atslēgkadra tipa. Zemāk uzskaitīti atslēgkadru tipi:

- Diskrēti – iestājoties atslēgkadram atribūtam tiek piešķirta atslēgkadra vērtība. Laika posmā no iepriekšējā atslēgkadra līdz aplūkojamam atslēgkadram nenotiek vērtības interpolācija, t.i. iestājoties atslēgkadram, vērtība „pārlec" uz atslēgkadra vērtību.
- Lineāri – laika posmā no iepriekšējā atslēgkadra līdz aplūkojamajam atslēgkadram notiek vērtības interpolācija līdzīgi kā lineāras pārejas animācijas.
- Līknes atslēgkadri – laika posmā no iepriekšējā atslēgkadra līdz aplūkojamajam atslēgkadram notiek vērtības interpolācija, izmantojot uzdotu līkni.

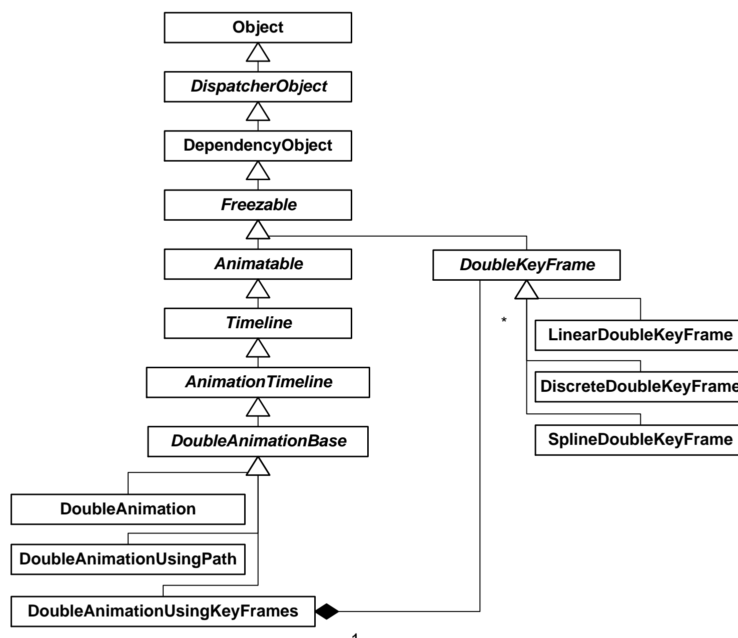
.NET Framework valodu saime ir stingri tipizēta, t.i. klasēm ir fiksēti atribūti ar fiksētiem datu tipiem, līdz ar to, katram atribūtu datu tipam, kuru ir nepieciešams animēt ir vajadzīga sava animācijas klase. WPF ir iekļautas animāciju klases 22 datu tipiem - klases grupētas ar abstraktu animācijas bāzes klasi, kuras nosaukums tiek veidots pēc šablona `<DatuTips>AnimationBase` (skat. *3.10. att. WPF datu tipu animāciju klašu hierarhija*). No bāzes klases mantojas animāciju klases konkrētajam datu tipam. Ne visiem datu tipiem ir pieejamas klases visiem iepriekš uzskaitītajiem animāciju tipiem. Piemēram, ideja par simbolu virkņu datu tipa lineāru interpolāciju nešķiet dabīga. *5. Pielikums – WPF iekļautās*

animāciju klases satur visu WPF iekļauto animācijas klašu apkopojumu, kas sadalīts pa datu tipiem un animāciju tipiem.



3.10. att. **WPF datu tipu animāciju klašu hierarhija**

Visbiežāk sastopamais atribūta datu tips WPF ir `Double`, kas kodē reālus skaitļus. Šim datu tipam arī ir pieejamas klases visu animāciju tipu aprakstīšanai. Klašu hierarhija `Double` ir redzama attēlā *3.11. att. DoubleAnimation animāciju klašu hierarhija* – attēlā redzamas visas klases, kas izmantojamas `Double` datu tipa atribūtu animēšanai.



3.11. att. **DoubleAnimation animāciju klašu hierarhija**

Animācijas klašu īss apraksts dots zemāk:

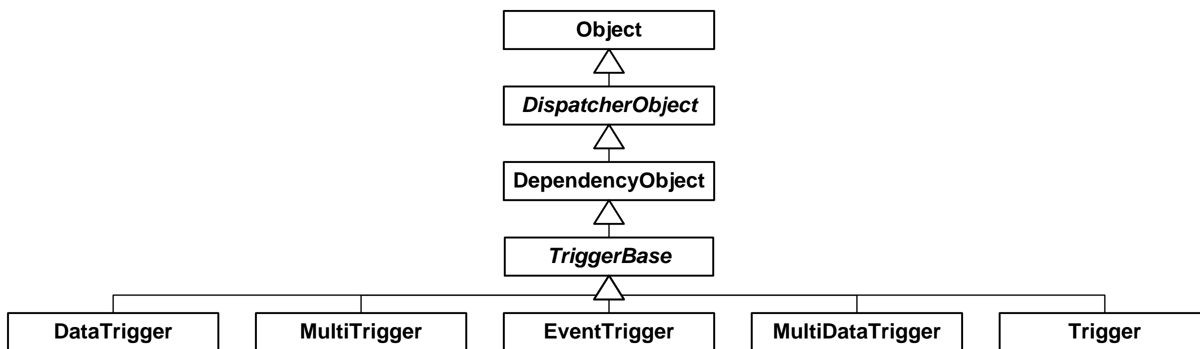
- `DoubleAnimation` – `Double` vērtību animācija ar lineāru interpolāciju
- `DoubleAnimationUsingPath` – animācija izmantojot uzdotu līkni,

- `DoubleAnimationUsingKeyFrames` – atslēgkadru animācijas.
- `DoubleKeyFrame` - abstraktās klase atslēgkadru klašu grupēšanai. Klasei ir trīs apakšklases:
    - `DiscreteDoubleKeyFrame` – diskrētas vērtības atslēgkadrs
    - `LinearDoubleKeyFrame` – lineāras interpolācijas atslēgkadrs
    - `SplineDoubleKeyFrame` – līknes interpolācijas atslēgkadrs.

Ir apskatīts pietiekami daudz, lai varētu izveidot nelielu izmēģinājuma programmu (pilnu programmas tekstu skatīt sadaļā *6. Pielikums – WPF DoubleAnimation demonstrācijas programma*). Zemāk dotais programmas fragments izveido četrstūri un startē animāciju, kas maina četrstūra novietojumu attiecībā no tā virsmas kreisās malas.

```
...
// Izveido melnu kvadrātu ar 100 vienību garu malu.
// Novieto kvadrātu uz loga virsmas punktā (50, 50)
Rectangle r = new Rectangle();
r.Height = r.Width = 100;
r.Fill = new SolidColorBrush( Colors.Black );
virsma.Children.Add( r );
Canvas.SetLeft( r, 50 );
Canvas.SetTop( r, 50 );

// Izveido lineāras animācijas objektu, kurš animē Double tipa vērtības
DoubleAnimation kustiba = new DoubleAnimation();
kustiba.Duration = new Duration( new TimeSpan( 0, 0, 0, 2 ) );
kustiba.BeginTime = new TimeSpan( 0, 0, 0, 5 );
kustiba.From = 50;
kustiba.To = 350;
kustiba.AutoReverse = true;

// Sāk animāciju kvadrātam, animējot kvadrāta atribūtu, kas apraksta
// kvadrāta  kreisās puses nobīdi no virsmas koordināta sākumpunkta
r.BeginAnimation( Canvas.LeftProperty, kustiba );
...
```

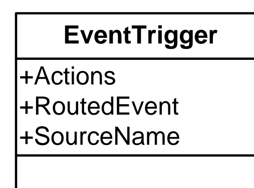
Animācijas var tikt uzsāktas programmatiski kā iepriekšējā piemērā, kur animācija uzsākta vai arī var tikt uzsāktas ar trigeriem. Trigeru klases instances definē darbības, kuras jāveic iestājoties noteiktam nosacījumam. Piemēram, peles klikšķis uz spiedpogas var uzsākt animāciju. Trigeru mehānismam WPF ir plašāks pielietojums par animāciju kontrolēšanu, taču šajā darbā tie tiks apskatīti tikai šī pielietojuma kontekstā. Trigeru klašu hierarhija redzama *3.12. att. Trigeru klašu hierarhija*.



3.12. att. **Trigeru klašu hierarhija**

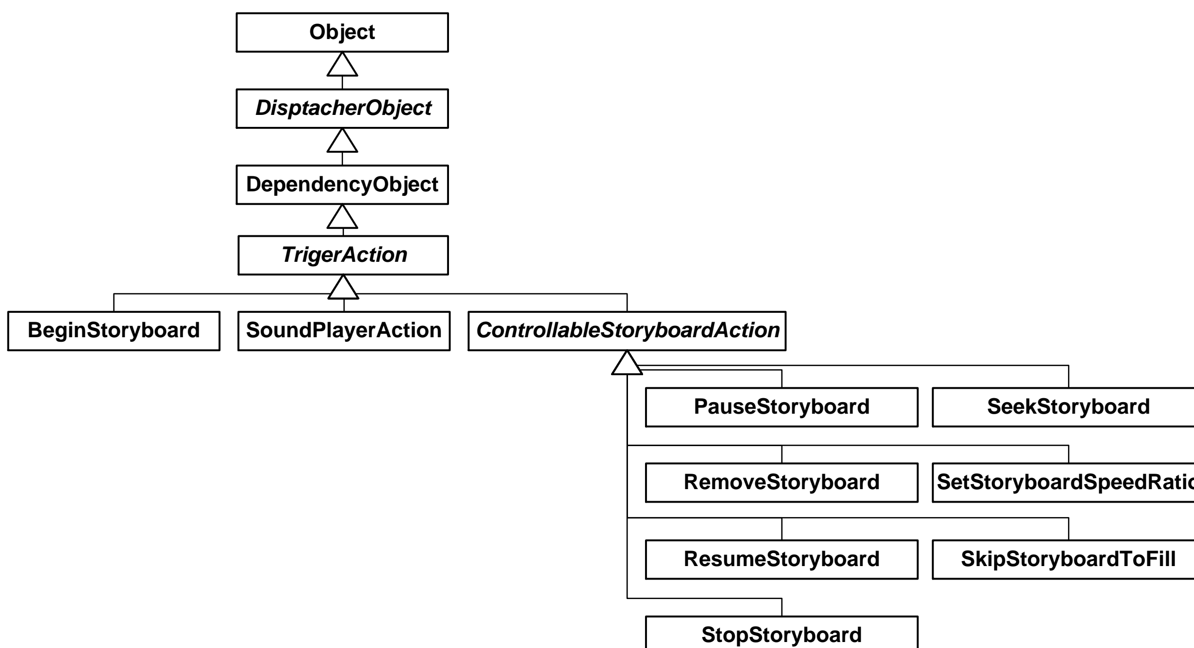
`FrameworkElement` definē `Triggers` kolekciju, kas var saturēt vairākus `TriggerBase` elementus. Animāciju kontrolēšanai izmanto `EventTrigger` klasi. Klase definē trīs būtiskus atribūtus (skat. arī *3.13. att. Event-Trigger klases atribūti*):

- `Actions` – kolekcija ar `TriggerAction` objektiem (apskatīti zemāk), kas apraksta darbības, kuras veikt
- `RoutedEvent` – notikums, uz kura iestāšanos jāizpilda definētās darbības
- `SourceName` – objekts, kura notikumus pārtvert (ja atšķirīgs no objekta, kuram pievienots `EventTrigger` objekts).



3.13. att. **Event-Trigger klases atribūti**

Kā jau minēts iepriekš `EventTrigger` klase definē `Actions` atribūtu, kurš var saturēt `TriggerAction` objektus. Pilna `TrigerAction` klašu hierarhija attēlota *3.14. att. TrigerAction klašu hierarhija*:



3.14. att. **TrigerAction klašu hierarhija**

`SoundPlayerAction` klases instance var tikt lietota, lai startētu `SoundPlayer` klases objektus, kas apraksta nevizuālu elementu, kas atskaņo skaņu. `BeginStoryboard` klase startē

norādīto `Storyboard` klases objekta aprakstīto laika segmentu, kamēr `ControllableStoryboardAction` klases apakšklases ļauj kontrolēt `Storyboard` objektus. Kā minēts iepriekš, `Storyboard` klases instances grupē vairākus animāciju klases objektus. Trigeru mehānisms ļauj deklaratīvā stilā aprakstīt veicamās darbības, kurām jānotiek iestājoties notikumiem.

Ir iespējams pārrakstīt iepriekšējo programmas fragmentu, lai atbrīvotos no imperatīvā stila animācijas palaišanas metodes izsaukuma. Sekojošais programmas fragments ir pārveidots, lai izmantotu `EventTrigger` klasi (pilns programmas teksts pievienots *7. Pielikums – WPF EventTrigger demonstrācija*):

```
...
// Izveido melnu kvadrātu ar 100 vienību garu malu.
// Novieto kvadrātu uz loga virsmas punktā (50, 50)
Rectangle r = new Rectangle();
r.Name = "Kvadrāts";
r.Height = r.Width = 100;
r.Fill = new SolidColorBrush( Colors.Black );
virsma.Children.Add( r );
Canvas.SetLeft( r, 50 );
Canvas.SetTop( r, 50 );

// Storyboard objektam nepieciešams, lai animējamā objekta nosaukums
// būtu reģistrēts vārdnīcā
this.RegisterName( r.Name, r );

// Izveido lineāras animācijas objektu, kurš animē Double tipa vērtības
DoubleAnimation kustiba = new DoubleAnimation();
kustiba.Duration = new Duration( new TimeSpan( 0, 0, 0, 2 ) );
kustiba.From = 50;
kustiba.To = 350;
kustiba.AutoReverse = true;

// Izveido Storyboard objektu un uzstāda tā mērķa objektu un atribūtu
Storyboard s = new Storyboard();
Storyboard.SetTargetName( s, r.Name );
Storyboard.SetTargetProperty( s, new PropertyPath( Canvas.LeftProperty ) );
s.Children.Add( kustiba );

// Izveido BeginStoryboard darbības klases objektu un konfigurē to, lai
// startētu iepriekš izveidoto Storyboard objektu
BeginStoryboard beginAction = new BeginStoryboard();
beginAction.Storyboard = s;

EventTrigger trig = new EventTrigger( Window.LoadedEvent );
trig.Actions.Add( beginAction );
this.Triggers.Add( trig );
...
```

Modificētā programma darbojas līdzīgi oriģinālajai. Tā ir garāka dēļ nepieciešamības veidot Storyboard objektu, taču garākās programmās atšķirība varētu nebūt tik jūtama, savukārt programma ir kļuvusi kvalitatīvāka – programma pilnībā paļaujas uz klašu objektu struktūru.

3.3. Silverlight

Silverlight ir Microsoft izstrādāta tehnoloģija, kas paredzēta ar multimediju elementiem bagātu interaktīvu programmu izstrādei tīmekļa videi, un tiek pozicionēta kā konkurents Adobe Flash tehnoloģijai. Silverlight programmas tiek darbinātas pārlūkprogrammā un pieejamas vairākām platformām. Silverlight piedāvā tādus servisus kā interaktīva lietotāja saskarne, lietotāja datu ievades apstrāde, grafisko elementu renderēšana, iebūvētas saskarnes kontroles, automātiska vizuālo elementu izvietošana, multimediju failu atskaņošana, u.c. servisus. (20)

Silverlight būtībā ir sašaurināta WPF versija, kura atbalsta vairākas platformas. Kamēr WPF piedāvā pilnu klāstu lietotāja saskarnes servisiem, Silverlight padara daļu no WPF servisiem pieejamu vairākām pārlūkprogrammām. (21)

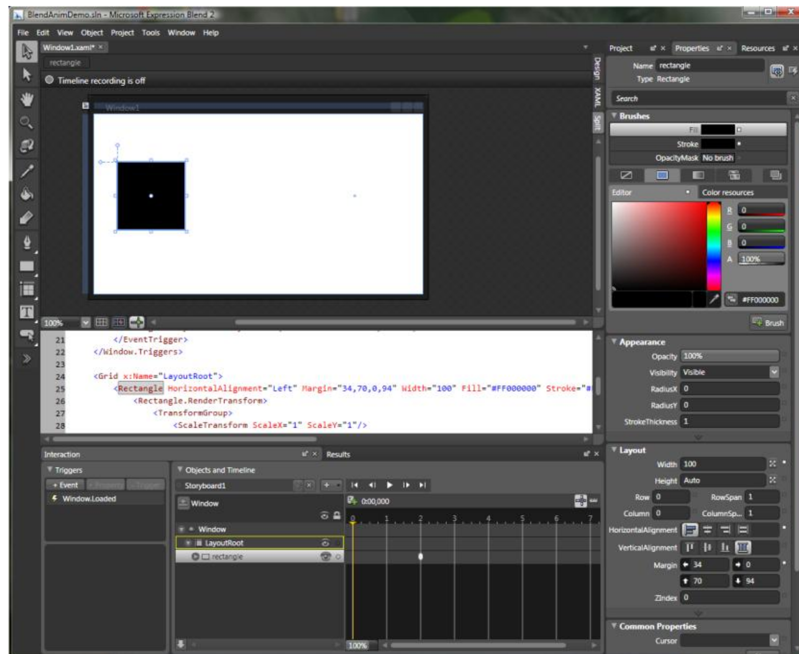
Silverlight realizē WPF definētā API apakškopu, realizējot izvēlētu API apakšsistēmu tādu pašu klašu kopu kā WPF. Lai arī WPF un Silverlight realizācijas būtiski atšķiras, liela daļa koda, kas darbojas Silverlight, darbojas arī WPF. Tāpat kā WPF, Silverlight piedāvā deklaratīvu programmēšanas pieeju izmantojot valodu XAML, un imperatīvu programmēšanas modeli izmantojot JavaScript un .NET Framework saimes valodas, piemēram, C# un Visual Basic.

Kā jau minēts, Silverlight realizē to pašu klašu kopu, kas definētas WPF, tajā skaitā arī animācijas klases (skat. *3.2.2. Animācijas ar WPF*) (22). Tā kā animāciju aprakstīšanas ideja ir tāda pati, kā WPF, Silverlight tehnoloģija netiks apskatīta sīkāk.

3.4. Expression Blend

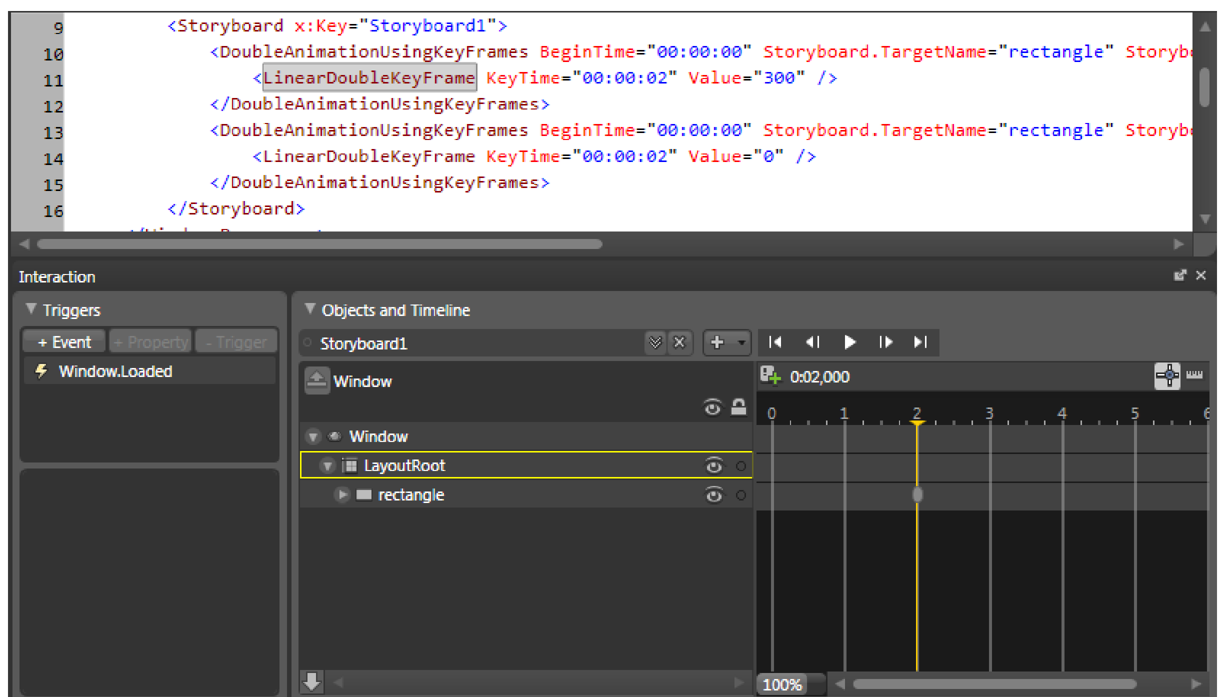
Microsoft Expression Blend ir dizaineru rīks lietotāja saskarņu un multimediju satura izstrādei Windows saimes operētājsistēmu programmām un tīmekļa programmām. Programmas tiek darbinātas izmantojot WPF tehnoloģiju Windows programmu gadījumā, un Silverlight tehnoloģiju tīmekļa programmu gadījumā. (23)

Būtībā Expression Blend ir valodas XAML, kas ir viena no iespējamajām WPF un Silverlight programmēšanas valodām, vizuāls redaktors.



3.16. att. **Expression Blend programmas logs**

Programmas logs sastāv no vizuālas virsmas, XAML koda redaktora un paneļiem, kas ļauj rediģēt izveidoto objektu atribūtu vērtības (skat. *3.16. att. Expression Blend programmas logs*). Uz vizuālās virsmas iespējams rediģēt grafiskos elementus, piemēram, ģeometriskas figūras, vizuāli mainīt to parametrus, piemēram, izmērus, novietojumu utml. Rediģēšanas paneļos iespējams kontrolēt izvēlētā objekta īpašības, piemēram, krāsu, pielietotās grafiskās transformācijas utml.



3.15. att. Objects and Timeline panelis

Animācijas iespējams kontrolēt izmantojot Objects and Timeline paneli, kas satur visu vizuālo elementu sarakstu kreisajā pusē un laika līniju, uz kuras iespējams atlikt atslēgkadrus un uzstādīt objektu atribūtu vērtības atslēgkadra laika momentā. Expression Blend ģenerē objekta atribūtu vērtības starp definētajiem atslēgkadriem.

Lai realizētu animāciju Expression Blend ģenerē XAML kodu, kas apraksta animāciju izmantojot animāciju klases, kuras tika apskatītas iepriekš (skat. *3.2.* Windows Presentation Foundation).

Attēlā *3.15. att. Objects and Timeline panelis* redzams, ka elementam `rectangle` ir definēts viens atslēgkadrs divas sekundes pēc animācijas sākuma. Koda rediģēšanas logā redzams, ka Expression Blend katram animējamam atribūtam ir uzģenerējis XAML kodu animācijas definēšanai, kas apraksta animāciju izmantojot `DoubleAnimationUsingKeyFrames` klases instanci un atslēgkadru, izmantojot `LinearDoubleKeyFrame` klases instanci.

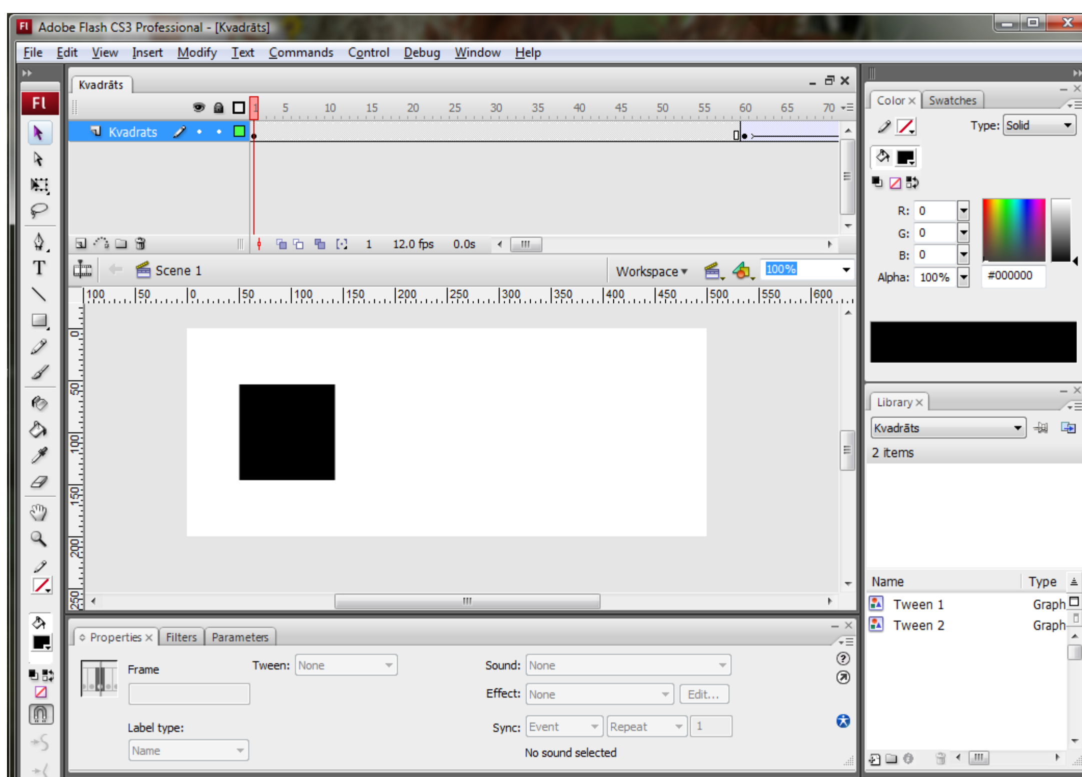
3.5. Adobe Flash

Adobe Flash ir multimediju tehnoloģiju kopa, kas atbalsta interaktīvu multimediju elementu veidošanu tīmeklim. Flash tehnoloģija ir pieejama uz vairākām platformām, tai skaitā Windows, Mac OS X, un Linux saimes operētājsistēmām.

Flash tehnoloģiju bieži izmanto, lai veidotu animācijas, reklāmas, dažādus tīmekļa lapu komponentus, lai integrētu multimediju failus tīmekļa lapās, un, lai izstrādātu tīmekļa programmas. Flash spēj manipulēt rastra grafiku un vektorgrafiku, kā arī animāciju veidošanu. Flash veidotos failus spēj attēlot Flash Player, kura realizācija ir pieejama gan kā neatkarīga programma, gan kā pārlūkprogrammu spraudnis. (24)

Flash failus iespējams veidot, izmantojot Adobe Flash izstrādes vidi.

Adobe Flash izstrādes vides logs (skat. *3.17. att.*) sastāv no vizuālas virsmas animācijas kompozīcijas veidošanai, rīku paletēm, atribūtu rediģēšanas paneļiem un laika joslas. Animācijas iespējams veidot izmantojot laika joslu. Zīmēšanas virsma tiek dalīta vairākos slāņos. Katrs slānis var saturēt grafiskos elementus.



3.17. att. **Adobe Flash programmas logs**

Lai veidotu animācijas, animators izvieto grafiskos elementus uz vizuālās redaktora virsmas un maina to novietojumu dažādos laika momentos. Animāciju realizēšanai pielieto divas tehnikas – kadra animāciju, kurā animators animē katru kadru atsevišķi, un *tween* animācijas, kurās animators norāda slāņa objektu novietojumu dažādos būtiskos atslēgkadros un izmanto Flash, lai ģenerētu vērtības starp šiem atslēgkadriem.

3.6. Kopsavilkums

Šajā sadaļā tika īsi apskatītas Windows Forms, Windows Presentation Foundation klašu bibliotēka un iespējas veidot programmas, kurās tiek animēti vizuālie elementi. Īpaša uzmanība pievērsta animācijas idejai katrā no bibliotēkām.

Apskatīti divi populāri risinājumi, kuri tiek lietoti animāciju izstrādei – Adobe Flash un Microsoft Expression Blend. Lai arī šo produktu realizācijas un izmantotās tehnoloģijas ir pilnīgi dažādas, abi rīki piedāvā līdzīgus rīkus animāciju veidošanai. Tiek aprakstīts animējamo objektu stāvoklis noteiktos laika momentos un tiek ģenerētas kadri starp šiem laika momentiem. Šī pieeja līdzīga tradicionālai animācijai, kur animators veido zīmējumus svarīgos animācijas momentos, kas tiek saukti par atslēgkadriem, un pēc tam veido zīmējumus kadriem starp šiem atslēgkadriem, tādējādi izveidojot plūstošu animāciju.

Lai arī WinForms piedāvā plašu klāstu ar klasēm lietotāja saskarnes elementu definēšanai, bibliotēkā nav iebūvētas klases animāciju aprakstīšanai. Tā vietā nepieciešams rakstīt speciālu kodu katrai animācijai.

WPF piedāvā plašu klāstu ar prezentācijas servisiem, tai skaitā sistēmu animāciju aprakstīšanai. Tika Aprakstīta WPF animācijas ideja, aprakstītas klases, kas apraksta animāciju un nodemonstrēti divi mehānismi animāciju startēšanai. WPF projektējumu īpaši interesantu padara viens no tā principiem – dot priekšroku objektiem un to atribūtiem, nevis metodēm un metožu izsaukumiem, kas ļauj deklarēt animācijas deklaratīvā stilā. Lai arī šis WPF princips izvirzīts no MDA filozofijas atšķirīgu mērķu dēļ, daudzas WPF projektējuma idejas izmantojamas animāciju metamodelī.

4. ANIMĀCIJAS METAMODELIS

MDA ietvars un uz modeļiem balstīta izstrāde ir praktiski pierādījusi savu efektivitāti programmatūras izstrādē (7). Metamodeļi labi var aprakstīt statisku struktūru, taču līdz šim mazāk pētīta problēma ir kā būvēt metamodeļus, kuri attēlo ne tikai statisko struktūru, bet arī dinamiku. Lai gan ir zināmi mēģinājumi ar metamodeļiem aprakstīt pielietojumus, kas ietver arī animāciju veidošanu (19), autoram nav zināms gadījums, kad tas būtu pētīts detalizēti.

Šajā nodaļā tiek piedāvāta animācijas metamodeļa būves ideja un konkrēta animācijas metamodeļa pielietojums. Ideja izkristalizējusies, apvienojot MDA idejas, LU MII izstrādes idejas un eksistējošo animācijas izstrādes produktu darbības principus. Metamodelis ir aizguvis vairākas idejas no Windows Presentation Foundation API (skat. sadaļu *3.2. Windows Presentation Foundation*), dēļ tā pārdomātā projektējuma un deklaratīvā animāciju definēšanas stila, tamdēļ klašu nosaukumos lietoti līdzīgi nosaukumi kā WPF klasēs ar līdzīgu funkcionalitāti.

Turpmākajās sadaļās, vienkāršības labad, par primitīviem datu tipiem tiek uzskatīts arī tips `Color`, kurš apraksta krāsu, un tips `Time`, kurš apraksta laika ilgumu.

Sekojošās sadaļās inkrementāli pasniegtas animācijas metamodeļa sastāvdaļas - sniegts to klašu, atribūtu un metožu apraksts.

4.1. Animāciju metamodeļa veidošanas pamatidejas

Šajā sadaļā tiek piedāvāta metamodeļa, kurš spējīgs aprakstīt animācijas, būves pamatidejas. Idejas tiek izklāstīta uz piemēra bāzes - aplūkots vienkāršs metamodelis ar vienu klasi `Rectangle` (skat. *4.1. att. Rectangle klase*), kura apraksta taisnstūri. Klasei ir atribūti ar datu tipu `double` - `X`, `Y`, `Width`, `Height`, kuri attiecīgi apraksta instanču X un Y koordinātas plaknē, platumu un augstumu, kā arī atribūts ar datu tipu `Color` – `FillColor`, kurš apraksta taisnstūra krāsu. Šī metamodeļa instances ir taisnstūri plaknē ar fiksētu novietojumu, izmēriem un krāsu. Pieņemsim, ka ir nepieciešamība aprakstīt taisnstūra instanču pārvietošanu plaknē – laikā mainīt X un Y atribūtu vērtības.

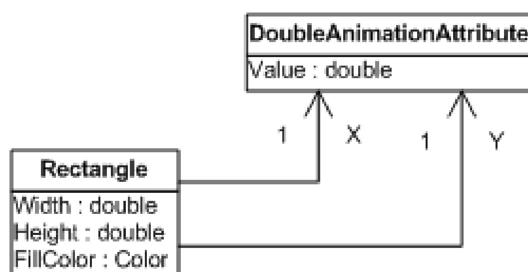
Rectangle
X : double Y : double Width : double Height : double FillColor : Color

4.1. att. **Rectangle klase**

Turpmākajās sadaļās seko apraksts metamodeļa papildināšanai ar klasēm, kas ļauj aprakstīt animāciju. Metamodeļa gala variants pievienots pielikumā *8. Pielikums – Pilns animācijas metamodeļa būves piemēra metamodelis*.

4.1.1. Animējami atribūti

Lai ieviestu iespēju laikā mainīt atribūta vērtību, tiek ieviesta jauna klase `AnimationAtribute` – klases instances apzīmē atribūtus, kuru vērtības var tikt mainītas laikā. Tā kā atribūtam ir fiksēts datu tips, tad katram atribūtu datu tipam nepieciešama `AnimationAttribute` apakšklase, kura satur atribūtu `Value` ar attiecīgo datu tipu, kurš savukārt satur atribūta vērtību. Šajā gadījumā nepieciešama `DoubleAnimationAttribute` klase, kas satur `Value` atribūtu ar datu tipu `double`. Pārveidojot sākotnējo metamodeli, lai klases `Rectangle` atribūti X un Y būtu animējami, iegūst modificētu metamodeli, kas attēlots *4.2. att. Animējama Rectangle klase.*

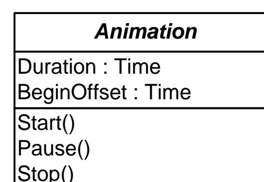


4.2. att. **Animējama Rectangle klase**

Semantiski iegūtais metamodelis ir līdzvērtīgs sākotnējam – tas aprakstīta taisnstūrus plaknē, taču tā atribūti X un Y ir animējami. Šādā veidā iespējams definēt brīvi izvēlētu kopu ar animējamiem atribūtiem. Piemēram, lai pārveidotu `Rectangle` klases `FillColor` atribūtu par animējamu atribūtu, nepieciešams definēt klasi, kas aprakstītu `Color` datu tipa atribūtus – `ColorAnimationAttribute` ar atribūtu `Value`, kuram būtu `Color` datu tips.

4.1.2. Animācijas klases

Lai aprakstītu animācijas tiek ieviesta `Animation` klase (skat. *4.3. att. Animation klase*). Klase apzīmē laika segmentu, kurā notiek animācija – klase definē `Time` datu tipa atribūtus `Duration` un `BeginOffset`, kuri attiecīgi apraksta animācijas ilgumu un laiku kopš animācijas sākšanas signāla saņemšanas līdz animācijas uzsākšanai. Animāciju var sākt, pārtraukt un pilnībā apturēt – šīs darbības tiek aprakstītas attiecīgi ar klases operācijām `Start()`, `Pause()` un `Stop()`. `Start()` operācija sāk animāciju, `Pause()` operācija to pārtrauc, saglabājot animējamā atribūta iegūto vērtību, un `Stop()` operācija pārtrauc animāciju, atgriežot animējamā atribūta vērtību sākotnējā stāvoklī.



4.3. att. **Animation klase**

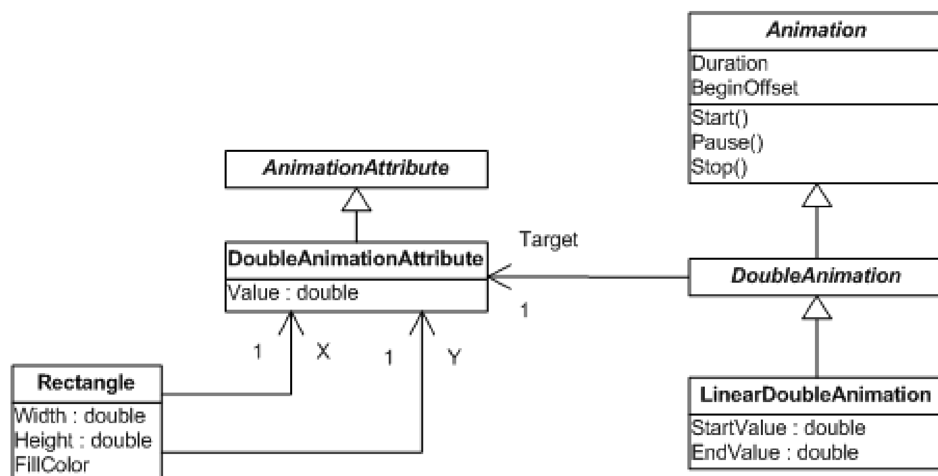
`Animation` klase ir abstrakta, jo animējami var būt vairāku datu tipu atribūti un animācijas raksturs var būt dažāds.

Tā kā `Animation` klase ir abstrakta un nesatur informāciju par animējamā atribūta vērtībām, katrai animējama atribūta klasei nepieciešama `Animation`

apakšklase. Piemēram, `double` datu tipa animācijas atribūtu animēšanai tiek definēta klase `DoubleAnimationAttribute`.

Animācijas klase konkrētam datu tipam nesatur informāciju par animācijas raksturu. Dažādiem datu tipiem var būt definētas dažāda rakstura animācijas. Viena no vienkāršākajām animācijām ir animācija ar lineāru vērtību interpolāciju – tiek uzdota atribūta vērtība animācijas sākuma un beigu momentos, savukārt vērtības laikā starp animācijas sākuma un beigu momentiem tiek izrēķinātas izmantojot lineāro interpolāciju. Šāda rakstura animācija labi saprotama `double` datu tipa gadījumā, taču nav intuitīvi saprotama `string` datu tipa gadījumā, līdz ar to dažādiem datu tipiem var tikt definēts atšķirīgs komplekts ar animāciju raksturiem.

Lai realizētu apskatītā piemēra `Rectangle` animējamo atribūtu `X` un `Y` klases animāciju ar lineāru vērtību interpolāciju, tiek definēta klase `LinearDoubleAnimation`, kura ir `Animation` apakšklase. Klase satur atribūtus `StartValue` un `EndValue`, kuri attiecīgi apraksta atribūta vērtību animācijas sākuma un beigu momentos. Iegūtais metamodelis dots *4.4. att. Animāciju metamodelis*.

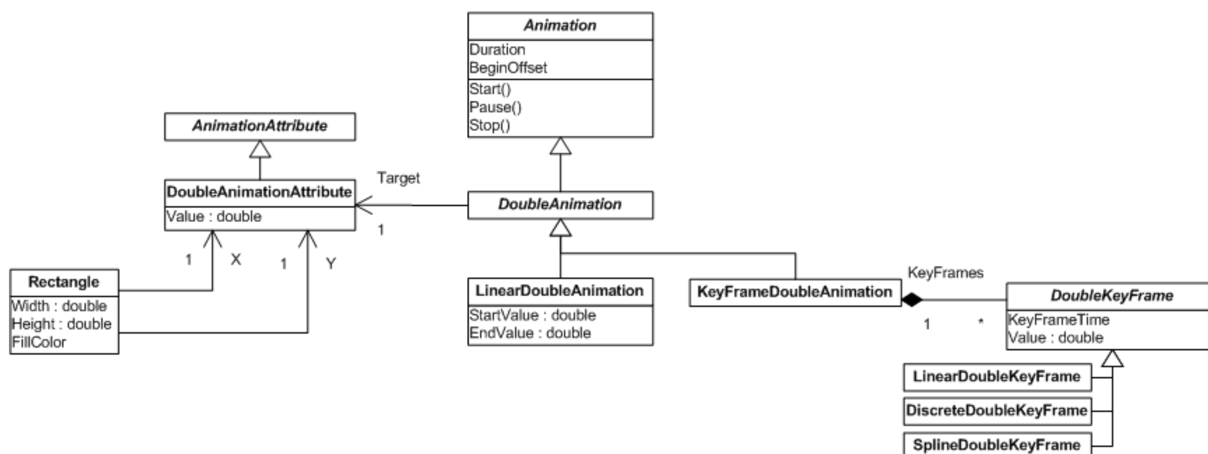


4.4. att. **Animāciju metamodelis**

Papildinātais metamodelis ļauj modelēt taisnstūra figūras plaknē, kuru X un Y atribūtus iespējams laikā mainīt, tādējādi iegūts vienkāršs animācijas metamodelis.

Iegūto metamodeli iespējams papildināt ar klasēm, kas definē cita rakstura animācijas. Piemēram, šajā darbā apskatītajos risinājumos animāciju izstrādei plaši tika izmantotas atslēgas ar atslēgkadriem. Šāda tipa animācija satur kolekciju ar atslēgkadriem, kas apraksta animējamā atribūta vērtību fiksētos laika momentos. Lai aprakstītu šādu animāciju, tiek ieviesta `KeyFrameDoubleAnimation` klase. Klases instances var saturēt kolekciju ar atslēgkadriem, kurus apraksta `DoubleKeyFrame klase`, kura satur atribūtus `KeyFrameTime` ar datu tipu `Time`, un `Value` ar datu tipu `Double`. Līdzīgi kā animācijas raksturs, arī atslēgkadra raksturs var būt dažāds – laika momentos starp atslēgkadriem var pielietot dažāda rakstura

interpolāciju vai interpolāciju nelietot vispār. Atslēgkadra raksturu apraksta dažādās `DoubleKeyFrame` apakšklases, kuras `double` datu tipa gadījumā varētu būt `LinearDoubleKeyFrame`, kura apraksta atslēgkadru, kura vērtības no iepriekšējā atslēgkadra tiek iegūtas interpolētas lineāri, `DiscreteDoubleKeyFrame`, kura vērtības no iepriekšējā atslēgkadra vērtības netiek interpolētas, un `SplineDoubleKeyFrame`, kura vērtības no iepriekšējā atslēgkadra tiek interpolētas izmantojot uzdotu parametrizētu līknes funkciju. Papildinātais metamodeļa fragments, kas satur augstāk aprakstīto atslēgkadru animāciju klases, redzams *4.5. att. Papildināta animāciju klašu hierarhija*.



4.5. att. **Papildināta animāciju klašu hierarhija**

Pēc līdzīgiem principiem būtu iespējams konstruēt klases `Color` datu tipa atribūtu animēšanai. Šie principi arī veido pamatu animāciju aprakstīšanai. Turpmākajās sadaļās apskatīti metamodeļa papildinājumi animāciju vieglākai pārvaldīšanai un kontrolēšanai.

4.1.3. Notikumu klases

Lai būtu iespējams definēt sarežģītākas animācijas, kas reaģē uz notikumiem un maina savu uzvedību atkarībā no dažādiem iepriekš nenosakāmiem apstākļiem, nepieciešami līdzekļi notikumu definēšanai.

Lai aprakstītu notikumus un darbības, kas jāveic iestājoties notikumam, metamodelis tiek papildināts ar `Event` un `Action` klasēm.

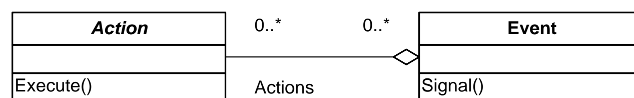
Klases var saturēt atribūtus ar tipu `Event`, tādējādi definējot notikumus, kas var notikt ar klases instancēm. Par notikuma iestāšanos tā īpašnieks (klases instance, kas definē konkrēto notikumu) var signalizēt – `Event` klase definē `Signal()` operāciju, kas signalizē notikuma iestāšanos.

Iestājoties notikumam, iespējams izpildīt nevienu vai vairākas darbības. Darbības klases apraksta veicamu darbību. Darbību klases sīkāk apskatītas nākamajā sadaļā – šobrīd tikai tiek

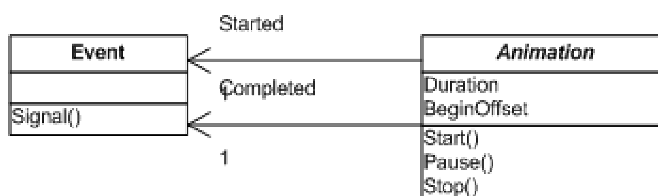
ieviests abstrakts darbības jēdziens ieviešot abstraktu `Action` klasi. Darbību var izpildīt – klase `Action` definē `Execute()` operāciju, kura izpilda darbību.

`Event` un `Action` klašu attiecība attēlota attēlā *4.6.* att.

Ar šiem jaunajiem līdzekļiem iespējams papildināt līdz šim iegūto animāciju metamodeli, lai definētu animācijas notikumus, kas ir nepieciešami, lai nodrošinātu iespēju reaģēšanai uz animāciju stāvokļa izmaiņām. Šim nolūkam iepriekš definētā `Animation` klase tiek papildināta ar `Started` un `Completed` notikumiem - notikumi attiecīgi notiek izpildot `Start()` un beidzoties animācijai. Papildinātā `Animation` klase un to attiecība ar `Event` klasi sniegta attēlā *4.7.*



4.6. att. **Event un Action klašu attiecība**



4.7. att. **Ar notikumiem papildināta Animation klase**

4.1.4. Darbību klases

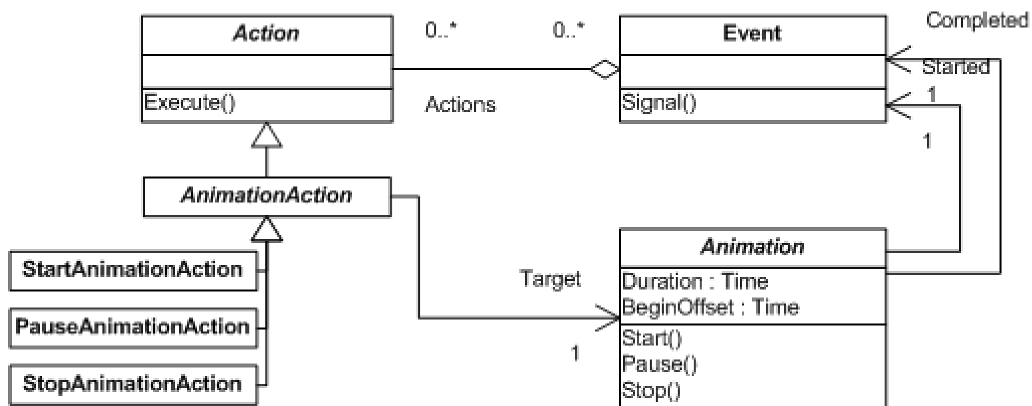
Lai arī animāciju klase definē operācijas animāciju kontrolēšanai, līdz šim nav definēti līdzekļi, lai operācijas aprakstītu ar metamodeļa līdzekļiem.

Iepriekšējā sadaļā saistībā ar notikumu klasēm tika pieminētas darbību klases – klases, kuru instances, izpildot `Execute()` operāciju, izpilda noteiktu darbību. Šādā veidā iespējams ar metamodeļa palīdzību aprakstīt arī klašu operācijas.

Animāciju operāciju aprakstīšanai metamodelī tiek ieviesta `AnimationAction` klase – abstrakta klase, kura definē atribūtu `Target` ar datu tipu `Animation`, kas nosaka, kura `Animation` klases instance būs darbības mērķis. Katrai `Animation` klases operācijai tiek definēta `AnimationAction` apakšklase, kas apraksta izpildāmo `Animation` klases operāciju - tiek definētas trīs `AnimationAction` apakšklases – `StartAnimationAction`, `PauseAnimationAction` un `StopAnimationAction`.

Papildinātās animāciju klašu hierarhijas fragments redzams attēlā *4.8.*

Iegūtais metamodelis ļauj aprakstīt animācijas, kuras kontrolē dažādu notikumu iestāšanās. Piemēram, beidzoties vienai animācijai, var tikt startēta cita animācija.



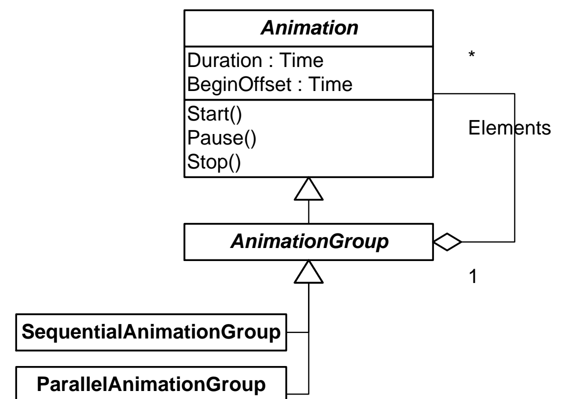
4.8. att. **Animāciju darbību klases**

Līdzīgā veidā var ieviest arī citas `Action` apakšklases, kas apraksta citas darbības, piemēram transformācijas programmas izsaukumu

4.1.5. Animāciju grupu klases

Animāciju ērtākai pārvaldīšanai iespējams papildināt animāciju metamodeli ar klasēm animāciju grupēšanai. Metamodelis tiek papildināts ar klasi abstraktu `AnimationGroup`, kura definē atribūtu `Elements`, kurš apraksta kolekciju ar animācijām, kuras pārvaldīt kā grupu. Animāciju grupu arī var uztvert kā animāciju – elementi tiek pārvaldīti kā viena animācija, tamdēļ `AnimationGroup` ir `Animation` apakšklase. Šāda struktūra ir klasisks kompozīta (*composite*) projektējuma šablona piemērs (26).

Kā jau minēts, pārvaldības shēma var būt dažāda. Tamdēļ pati `AnimationGroup` klase ir abstrakta, bet dažādā uzvedība aprakstīta ar `AnimationGroup` apakšklasēm. Piemēram, dabiskas ir paralēlā un virknes pārvaldības shēmas. Paralēlā animāciju grupā visi tās elementi tiek startēti vienlaicīgi. Virknes animāciju grupā elementi tiek startēti virknē – vispirms startēta pirmā animācija, tai beidzoties tiek startēta otrā, utt.



4.9. att. **Animāciju grupu klases**

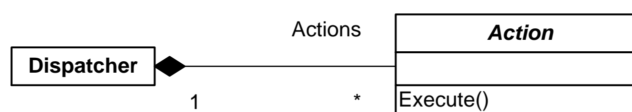
Animāciju metamodeļa fragments, kas satur animāciju grupu klases redzams *4.9. att. Animāciju grupu klases*.

4.1.6. Dienesta klases

Ar līdz šim aprakstīta metamodelis līdzekļiem iespējams aprakstīt animējamu atribūtu dažāda rakstura animācijas, kuras var reaģēt uz dažādiem notikumiem.

Pamatlīdzeklis modeļu manipulācijai ir modeļu transformācijas (2), kuras iespējams uzrakstīt transformāciju valodās, piemēram, MOLA vai Lx saimes valodās (7). Ar transformācijas programmu iespējams modificēt jau izveidotu animācijas modeli, tādējādi mainot animāciju. Atbalstu transformācijas programmu startēšanai var pievienot ieviešot jaunu `Action` apakšklasi `TransformationStartAction`, kura satur atribūtu `TransformationProgram`, kura norāda startējamo transformāciju. Līdz ar to pastāv līdzekļi modificēt animācijas modeli arī pēc tā izveides.

Tomēr ar metamodeļa līdzekļiem nav iespējams izpildīt darbības pašreizējā laika momentā – iespējams aprakstīt darbību izpildi tikai reaģējot uz notikumiem. Šāda līmeņa kontrole ir nepieciešama, piemēram, ja nepieciešams startēt animāciju atkarībā no transformācijas programmas rezultāta.



4.10. att. **Dispatcher klase**

Šo nepilnību var novērst ieviešot metamodelī jaunu „dispečera" klasi `Dispatcher`, kura definē saiti uz darbībām (skat. *4.10. att. Dispatcher klase*), kas izpildāmas nekavējoties. Līdz ar to transformācijas programma var izveidot `Action` apakšklašu instances un izveidot saiti ar `Dispatcher` klases instanci, lai aprakstītu situāciju, kad darbības jāizpilda uzreiz pēc tam, kad transformācijas programma ir beigusi darbu.

Lai arī šāds līdzeklis varētu būt noderīgs reālos pielietojumos, šajā darbā tas netiks aplūkots sīkāk.

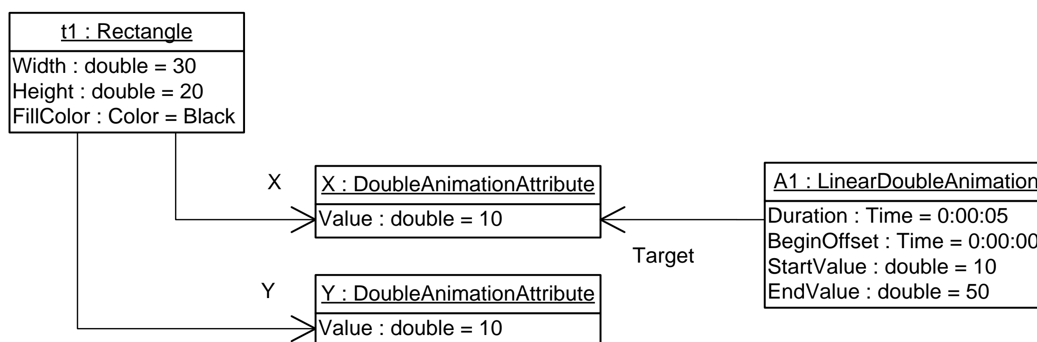
4.1.7. Animāciju metamodeļa instances piemērs

Metamodeļa definīcija uzdod kopu ar modeļiem, kurus iespējams izveidot ar metamodeļa līdzekļiem. Metamodeļa instance uzdod konkrētu klašu instanču konfigurāciju. Animāciju metamodeļa gadījumā šī konfigurācija veido animācijas „filmiņu" – tā modelē animējamos objektus un to atribūtu vērtību maiņu laikā.

Iepriekšējās sadaļās aprakstīto piemēra metamodeli var izmantot, lai definētu modeļus, kuri apraksta taisnstūri un tā animāciju. Šajā sadaļā tiek sniegts vienkāršs piemērs vienai metamodeļa instancei. Modelis apraksta taisnstūra pārvietošanos horizontālā virzienā.

Modeļa klašu instanču diagramma redzama *4.11. att. Animācijas metamodeļa instance*. Modelī ir viena taisnstūra klases `Rectangle` instance `t1`, kura apraksta melnas krāsas taisnstūri, kurš ir 30 vienības plats, 20 augsts un novietots plaknes punktā ar koordinātām

X = 10 un Y = 10. Modelis satur animācijas klases instanci `A1`, kura apraksta `t1` instances atribūta `X` animāciju, kas ilgst 5 sekundes, un maina atribūta vērtību no 10 līdz 50.



4.11. att. **Animācijas metamodeļa instance**

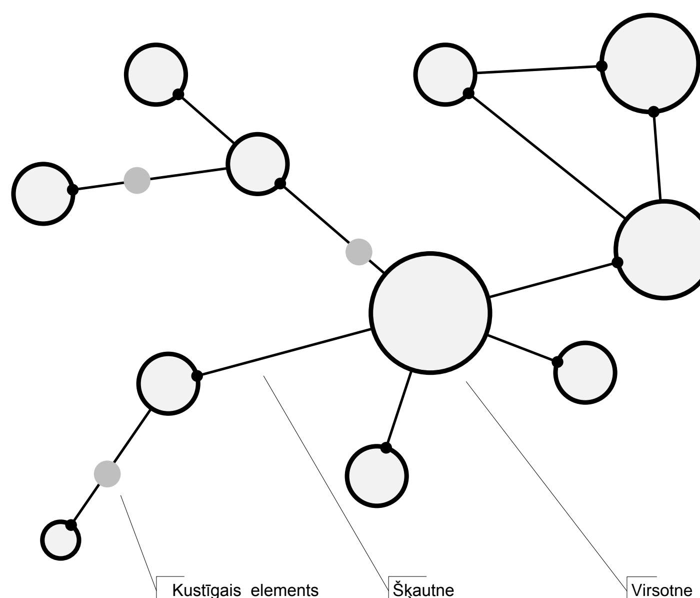
Modelis uzdod vienu animācijas „filmiņu”. Līdzīgā veidā iespējams veidot citas metamodeļa instances.

4.2. Grafa animācijas metamodelis

Šajā sadaļā izklāstīts animācijas metamodeļa ideju pielietojums konkrētā problēmapgabalā – aprakstīt animāciju orientētā grafā. Sekojošās apakšsadaļās aprakstīti modelējamie objekti, dots metamodelis šādu objektu modelēšanai un šis modelis papildināts ar klasēm animāciju aprakstīšanai.

4.2.1. Problēmapgabala apraksts

Šajā sadaļā neformāli aprakstīts apskatāmais problēmapgabals - orientēts grafs plaknē, kurš papildināts ar kustīgajiem elementiem. Apskatāmais grafs sastāv no virsotnēm, šķautnēm un kustīgajiem elementiem (skat. *4.12. att. Orientēta grafa piemērs*).



4.12. att. **Orientēta grafa piemērs**

Virsotnes tiek izkārtotas plaknē. Tās tiek attēlotas kā iekrāsoti apļi, kuru kontūras krāsa atšķiras no iekrāsojuma krāsas. Virsotnes raksturo X un Y koordinātas plaknē un virsotnes rādiuss.

Šķautnes tiek attēlotas kā līnijas ar nelielu iekrāsotu apli tās beigu galā. Šķautne savieno divas virsotnes no vienas virsotnes malas līdz otrai, šķautnes iedomātam turpinājumam savienojot virsotņu centrus. Vienkāršības labad, netiek pieļauts gadījums, ka šķautne savieno virsotni pašu ar sevi (veidojot „cilpu").

Virsotnes un šķautnes kopā veido grafa statiskos elementus.

Grafs tiek papildināts ar jaunu elementu – kustīgo elementu, kas var atrasties uz statiskajiem elementiem. Kustīgais elements tiek attēlots kā iekrāsots riņķis bez kontūras ar fiksētu rādiusu. Elementa atrašanos raksturo pozīcija, kas ir reāls skaitlis intervālā [0; 1]. Pozīcija šajā gadījumā tiek definēta sekojoši:

- Ja elements atrodas uz šķautnes, pozīcija raksturo elementa atrašanās vietu uz šķautnes – ja pozīcija ir 0, tad elements atrodas uz šķautnes sākuma virsotnes centra, ja pozīcija ir 1, tad elements atrodas uz šķautnes beigu virsotnes centra.
- Ja elements atrodas uz virsotnes, kustīgais elements novietots uz virsotnes centra un pozīcija tiek definēta kā elementa caurspīdīguma maiņa – ja pozīcija ir 0, elements ir pilnīgi caurspīdīgs, ja pozīcija ir 1, elements ir pilnīgi necaurspīdīgs.

Kustīgais elements var pārvietoties pa savstarpēji savienotajiem statiskajiem elementiem vai arī speciālos gadījumos „pārlekt" no viena statiskā elementa uz citu. Animācijas uzdevums ir plūstoši realizēt kustīgo elementu pārvietošanos, kā arī realizēt kustīgā elementa „lēcienus".

4.2.2. Animējama orientēta grafa metamodelis

Iepriekšējā sadaļā *4.2.1. Problēmapgabala apraksts* tika aprakstīts modelējamais problēmapgabals. Iespējami vairāki metamodeļi, kas spēj aprakstīt doto statisko grafu. Šajā sadaļā piedāvāts viens no iespējamajiem grafa metamodeļiem un sadaļas gaitā tas pakāpeniski tiek papildināts ar klasēm nepieciešamo animāciju atbalstam.

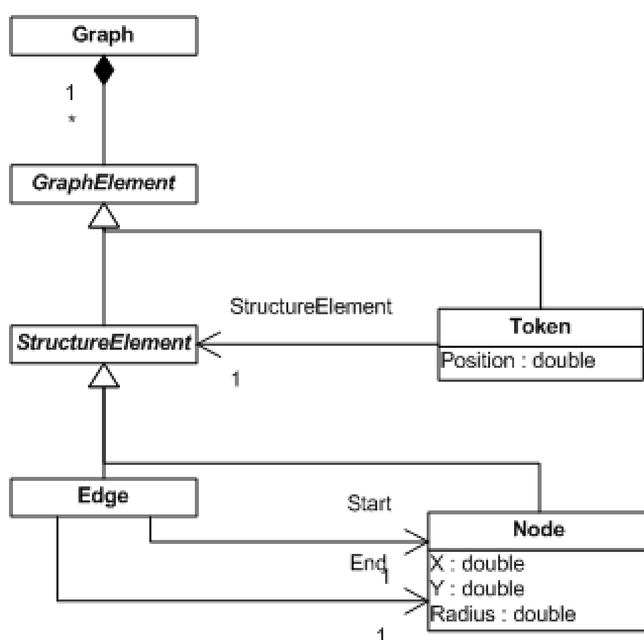
Metamodelis grafa aprakstīšanai attēlots *4.13. att. Orientēta grafa metamodelis*.

Virsotnes apraksta klase `Node`. `Node` klase satur atribūtus `X`, `Y` un `Radius` attiecīgi `X`, `Y` koordinātu aprakstīšanai un rādiusa aprakstīšanai.

Šķautnes apraksta klase `Edge`. `Edge` klase satur atribūtus `Start` un `End`, kas norāda attiecīgi uz šķautnes sākuma un beigu virsotnēm.

Virsotnes un šķautnes veido grafa struktūru, tādēļ tie ir loģiskā līmenī grupēti ar `StructureElement` virsklasi.

Kustīgos elementus apraksta klase `Token`. Kustīgais elements atrodas uz kāda no struktūras elementiem. `Token` klase satur atribūtu `Position`, kas apraksta kustīgā elementa atrašanās vietu uz struktūras elementa (pozīcijas definīcija tika dota sadaļā *4.2.1. Problēmapgabala apraksts*).

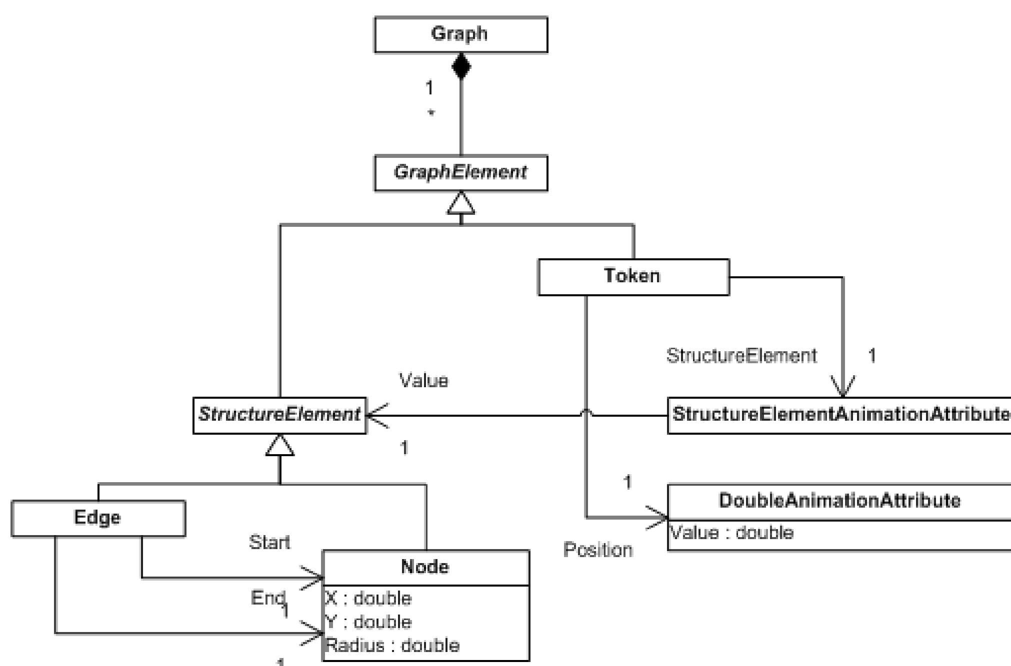


4.13. att. **Orientēta grafa metamodelis**

Struktūras elementi un kustīgie elementi kopā veido visus grafa elementus, tādēļ tie loģiskā līmenī grupēti ar abstraktu `GraphElement` virsklasi.

Grafu apraksta klase `Graph`. Klase satur kolekciju ar `GraphElement` elementiem (diagrammā attēlots ar kompozīcijas saiti).

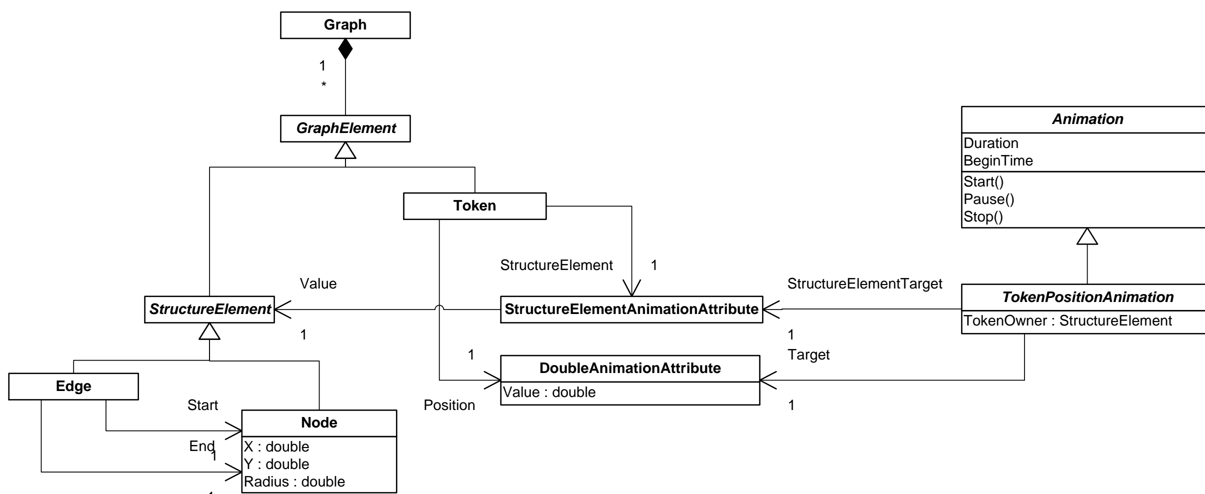
Līdz šim aprakstītais metamodelis spēj aprakstīt grafu statiskā stāvoklī. Lai būtu iespējams animēt kustīgā elementus atribūtus, nepieciešams pārveidot `Token` klases atribūtus par animējamiem atribūtiem. Šajā gadījumā tie ir divi – laikā var mainīties struktūras elements, uz kura atrodas kustīgais elements, un var mainīties kustīgā elementa pozīcija. Modificētais metamodelis redzams *4.14. att. Animējama orientēta grafa metamodelis*.



4.14. att. **Animējama orientēta grafa metamodelis**

Lai aprakstītu kustīgā elementa animāciju, nepieciešams ieviest jaunas klases animējamo atribūtu animēšanai. Tiek ieviesta abstrakta animācijas klase `Animation`, kura apraksta animācijas ilgumu un iespējamās animācijas operācijas (šī klase tika apskatīta sadaļā *4.1. Animāciju metamodeļa veidošanas pamatidejas*). Nepieciešams ieviest papildus klases, kas aprakstītu konkrētās animācijas.

Šajā piemērā nepieciešams animēt kustīgā elementa atrašanās vietu. Šis piemērs ir interesants, jo kustīgā elementa atrašanās vietu apraksta divi atribūti – struktūras elements uz kura atrodas kustīgais elements un kustīgā elementa pozīcija. Lai arī būtu iespējams definēt divas animāciju klases katra animējamā atribūta atsevišķai animācijai, dabiskāk šķiet veidot vienu animācijas klasi, kura apstrādā abus `Token` atribūtus vienlaicīgi. Šim nolūkam ieviesta `TokenAnimation` klase, kura aprakstīs kustīgā elementa pozīcijas animāciju tam atrodoties uz viena struktūras elementa. Klase satur atribūtus `Target` un `StructureElementTarget`, kuri norāda attiecīgi uz `Token` klases `Position` un `StructureElement` animējamiem atribūtiem. Papildus tam bāzes klasē iespējams definēt atribūtu `TokenOwner`, kurš definē uz kura struktūras elementa atrodas kustīgais elements animācijas laikā. Tomēr `TokenPositionAnimation` klase vēl neapraksta animācijas raksturu. Lai aprakstītu animāciju ar lineāru interpolāciju tiek ieviesta klase `TokenPositionLinearAnimation`, kura satur atribūtus `StartValue` un `EndValue`, kuri attiecīgi apraksta animējamā `Target` atribūta vērtību animācijas sākuma momentā un beigu momentā. Papildinātais metamodelis redzams *4.15.att. Animēta orientēta grafa* metamodelis.



4.15.att. **Animēta orientēta grafa metamodelis**

Iegūtais metamodelis sniedz līdzekļus orientētu grafu un kustīgo elementu animāciju aprakstīšanai. Lai sniegtu bagātīgākus līdzekļus animāciju aprakstīšanai, iespējams metamodeli papildināt ar jaunām animāciju klasēm, piemēram, animāciju ar atslēgkadriem.

Lai aprakstītu animāciju ar atslēgkadriem, metamodelī tiek ieviesta klase `TokenPositionKeyFrameAnimation`. Klase definē kolekciju ar atslēgkadriem, kurus apraksta

abstrakta `TokenPositionKeyFrame`, kuri, savukārt, apraksta kustīgā elementa atrašanās vietu fiksētos laika momentos. Animācijas pāreju no iepriekšējā pozīcijas stāvokļa uz pozīciju, ko apraksta nākamais kolekcijas atslēgkadrs, definē atslēgkadra klase, piemēram, lineāru pāreju apraksta klase `TokenPositionLinearKeyFrame`, bet „lēcienveidīgu" pāreju apraksta klase `TokenPositionDiscreteKeyFrame`.

Lai aprakstītu notikumus, metamodelī tiek ieviesta `Event` klase. `Animation` klase tiek papildināta ar `Event` tipa atribūtiem, kas apraksta animācijas instanču notikumus.

Lai ar metamodeli aprakstītu darbības, tiek ieviesta abstrakta darbību klase un konkrētas tās apakšklases, kas apraksta animācijas operāciju izsaukumus.

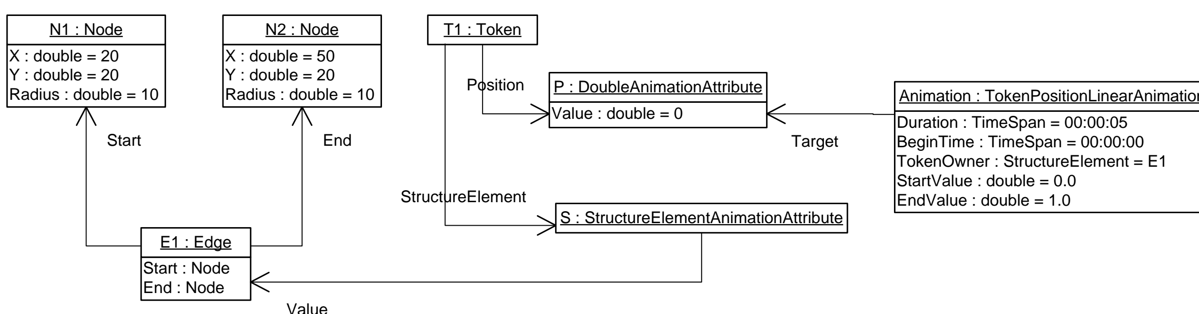
Animāciju vieglākai pārvaldībai tiek ieviesta abstrakta `AnimationGroup` animāciju grupēšanas klase un `AnimationGroup` apakšklases `ParallelAnimationGroup` un `SequentialAnimationGroup` klases, kas apraksta animāciju pārvaldības shēmu.

Animāciju metamodeļi papildinošās notikumu, darbību un animāciju grupēšanas klases ir identiskas ar klasēm, kuras aprakstītas sadaļā *4.1. Animāciju metamodeļa veidošanas pamatidejas* – šīs klases iespējams pārņemt bez izmaiņām.

Pilna animējama orientēta grafa metamodeļa diagramma pievienota pielikumā – skat. *9. Pielikums – Animējama orientēta grafa metamodelis*.

4.2.3. Animēta orientēta grafa metamodeļa instances piemērs

Šajā sadaļā sniegts piemērs animēta orientēta grafa metamodeļa instancei. Metamodeļa instance attēlota diagrammā *4.16. att. Animēta grafa metamodeļa instance*.



4.16. att. **Animēta grafa metamodeļa instance**

Grafs sastāv no divām virsotnēm – N1 un N2, kuras savieno šķautne E1. Grafam pieder arī kustīgais elements T1. Kustīgais elements atrodas uz šķautnes E1 pozīcijā 0. Modelis satur animāciju Animation, kura animē kustīgā elementa pozīciju uz šķautnes E1 no pozīcijas 0.0 līdz pozīcijai 1.0. Animācijas ilgums ir 5 sekundes.

4.3. Kopsavilkums

Šajā nodaļā tika piedāvāta vispārīgas animāciju aprakstoša metamodeļa būves idejas. Šāds metamodelis spēj aprakstīt objektu atribūtu izmaiņu laikā, tādējādi aprakstot animāciju. Izmantojot šajā nodaļā aprakstītos līdzekļus, iespējams definēt animācijas, kontrolēt animācijas izmantojot darbību klases, un definēt darbību kopu, kas veicamas iestājoties notikumiem, izmantojot notikumu klases.

Lai aprakstītu atribūta animāciju, nepieciešams ieviest animējama atribūta jēdzienu un definēt, kuri klases atribūti ir animējami. Pilnīgai elastībai, iespējams definēt visus klašu atribūtus kā animējamus atribūtus.

Šajā pieejā katram animējamā atribūta datu tipam nepieciešama animējama atribūta klase. Savukārt, katram animējama atribūtam nepieciešama klašu kopa, kura apraksta atribūta instanču animāciju. Šajā kopā ietilpst klases, kuras apraksta dažāda rakstura animācijas, piemēram, lineāras animācijas vai animācijas ar atslēgkadriem. Tā kā ne visiem datu tipiem animācijas raksturs var atšķirties, dažādiem animējamiem atribūtiem arī šī klašu kopa var atšķirties.

Pārdomas izraisa animāciju aprakstīšanai nepieciešamo klašu skaits. Taču, tā kā tipiski izmantojamo datu tipu kopa sastāv no pārskatāma skaita klašu, maz ticams, ka nepieciešamo klašu skaits kļūs nepārskatāms.

Aprakstītie animāciju metamodeļa būves principi tika demonstrēti orientētu grafu problēmapgabalā. Orientēts grafa definīcija tika papildināta ar kustīgajiem elementiem.

Kā tika parādīts šajā piemēra situācijā, iespējams veidot animācijas klases, kas apraksta vairāku atribūtu animāciju – šādas animācijas piemērs ir kustīgā elementa animācija, kura kontrolē gan `Token` klases `Position` atribūtu, gan `StructureElement` atribūtu.

Aprakstītā metamodeļa instances uzdod konkrētu animētu orientētu grafu. Animēta orientēta grafa izveidei nepieciešams izveidot modeļa instanci, kas atbilst metamodelim. Šāda metamodeļa esamības priekšrocība ir tāda, ka pastāvot realizācijai, kas interpretē un vizualizē metamodelim atbilstošās modeļu instances, iespējams veikt modeļu transformācijas no citiem metamodeļiem uz šādu metamodeli. Piemēram, būtu iespējams definēt transformācijas programmu, kas aktivitāšu diagrammas modeļu instances spēj transformēt par animēta orientēta grafa instancēm.

MDA izstrādes kontekstā (skat. *1. Model Driven Architecture ietvars*), šie metamodeļi no rīku būves veidotāju viedokļa apraksta animāciju PIM – no realizācijas platformas neatkarīgu modeli. Pastāvot realizācijai, kura interpretē un vizualizē šādu metamodeļu instances tas kļūst par PSM no transformāciju rakstītāja viedokļa.

5. PROTOTIPA REALIZĀCIJAS APRAKSTS

Darba ietvaros izstrādāts animāciju dziņa prototips, kas realizē objektu animācijas funkcionalitāti. Apskatīts pielietojums, kas aprakstīts sadaļā *4.2. Grafa animācijas metamodelis* - realizēta klašu kopa animējamu orientētu grafu aprakstīšanai un tā kustīgo elementu animācijas iespēju nodrošināšanai (skat. sadaļu *4.2.1. Problēmapgabala apraksts*).

Turpmākajās sadaļās seko realizācijas apraksts un iespējamo uzlabojumu diskusija.

5.1. Apraksts

Prototipa programma sastāv no diviem moduļiem – klašu bibliotēkas AE un klienta programmas AEClient.

Klašu bibliotēka realizē klases grafa elementu aprakstīšanai un animāciju aprakstīšanai.

Programma AEClient demonstrē klašu bibliotēkas iespējas ar vairākiem grafiem dažādos scenārijos, izmantojot klašu bibliotēkas klases.

Realizācijas pirmkods atrodams pielikumā (skat. *10. Pielikums – Programmatūras CD*).

Prototipa programma izstrādāta izmantojot .NET Framework 3.5 ietvaru, izmantojot Microsoft Visual Studio 2008. Tās darbināšanai uz datora nepieciešama .NET Framework 3.5 (pieejama bezmaksas lejupielādei no Microsoft tīmekļa vietnes un pievienota pielikumā *10. Pielikums – Programmatūras CD*).

5.2. Klašu bibliotēka

Klašu bibliotēkā realizētas sadaļā *4.2.2. Animējama orientēta grafa metamodelis* aprakstītā metamodeļa klases – realizēts specifiskās platformas modelis (platform specific model – PSM). Dažas klases ir papildinātas ar jauniem atribūtiem, kas ļauj smalkāk kontrolēt animācijas raksturu.

Veidojot bibliotēku pēc objektorientētās programmēšanas principiem, dabiski bija izmantot tās pašas klases, kuras aprakstītas grafa animācijas metamodelī. Tamdēļ bibliotēkas klases sakrīt ar problēmapgabala metamodeļa klasēm (skat. *9. Pielikums – Animējama orientēta grafa metamodelis*), modelējot realizējamo mērķa problēmapgabalu ar tiem pašiem jēdzieniem.

Klašu bibliotēka realizēta kā .NET Framework DLL bibliotēka. Bibliotēkas realizācijā izmantotas WPF grafiskās bibliotēkas klases elementu zīmēšanai, taču neizmanto iebūvētās WPF animāciju sistēmas klases. Bibliotēka realizē savu animācijas sistēmu, lai būtu iespējams pārnest tās realizāciju uz citām vidēm.

Pirmkods organizēts, ievietojot katru klasi savā failā, faila nosaukumam sakrītot ar klases nosaukumu. Faili organizēti mapēs, kas izdalītas pēc klašu funkcionālā apgabala:

- Elements - mape satur klases grafa un tā elementu definēšanai
- Animations – satur klases lineāru animāciju un atslēgkadru animāciju definēšanai. Mape satur apakšmapi KeyFrames, kas satur klases atslēgkadru definēšanai
- Actions – satur klases darbību definēšanai
- Events – satur klases notikumu sistēmas realizēšanai

Turpmākajās apakšsadaļās sīkāk aplūkotas prototipa klases, to atribūti un dažas realizācijas nianses. Visi koda piemēri rakstīti valodā C#, kas ir viena no .NET Framework valodu saimes populārākajām valodām.

5.2.1. Animējami atribūti

Sadaļā *4.1.1. Animējami atribūti* tika aprakstīta ideja kā aprakstīt animējamos atribūtus – atribūtus, kuru vērtību var mainīt animācijas klases. Šajā realizācijā par animējamiem atribūtiem tiek uztverti visi .NET Framework klašu atribūti.

.NET Framework realizācijā, klašu atribūti un lauki ir atšķirīgi jēdzieni. Atribūti īpaša veida metodes, kuras uzvedās kā lauki. Atribūts .NET Framework klases definīcijā tiek identificēts ar datu tipu un nosaukumu, līdzīgi kā klases lauks. Patiesībā var tikt izveidotas divas metodes – metode vērtības uzstādīšanai un metode vērtības nolasīšanai. Viena no abām metodēm var nebūt definēta vispār, tādējādi izveidojot tikai lasāmu vai, retāk, tikai rakstāmu atribūtu. Metode, kura uzstāda atribūta vērtību saņem vienu argumentu ar datu tipu, kādu definē atribūts. Savukārt metode, kura nolasa atribūta vērtību atgriež vērtību ar datu tipu, kādu definē atribūts. Parasti atribūti glabā vērtības klases laukos, taču atribūta rakstīšanas/lasīšanas metodē var tikt sauktas citas metodes, vērtība var tikt ģenerēta, utml (14). Tādējādi daudzviet AE bibliotēkā panākts efekts, ka uzstādot klases atribūta vērtību automātiski notiek izmaiņas, piemēram, piešķirot kustīgā elementa pozīcijas atribūtam jaunu vērtību tas uzreiz tiek pārzīmēts jaunajā vietā.

Tomēr šādu uzvedību var emulēt arī citās vidēs vairākos veidos – piemēram, atklāti izmantojot uzstādīšanas/nolasīšanas metodes un funkciju rādītājus vai izmantojot klases laukus un rādītājus uz laukiem.

5.2.2. Grafa klases

Šajā sadaļā sniegts apraksts klasēm, kas definē grafu un tā elementus. Klases atrodamas AE bibliotēkas pirmkoda bibliotēkas „\Elements”.

`Graph` klase definē grafu. Klase nodrošina virsmu grafa elementiem un uztur sarakstus ar grafa elementiem. Virsmas realizācijai izmantota WPF bibliotēkas klase `Canvas`, kas ļauj manuāli izvietot elementus balsoties uz to koordinātām plaknē. Klase definē sekojošos atribūtus:

- `Nodes` – grafam pievienoto virsotņu kolekcija
- `Edges` – grafam pievienoto šķautņu kolekcija
- `Tokens` – grafam pievienoto kustīgo elementu kolekcija
- `Animations` – grupa ar animācijām, kuras startēt, sākoties grafa attēlošanai. Realizēts kā `ParallelAnimationGroup` atribūts.
- `Host` – grafa „īpašnieks”. WPF bibliotēkas panelis, kas satur grafa virsmu. Grafs var atrasties uz jebkura `ContentControl` elementa vai tā apakšklases, tai skaitā `Window` klases loga, `Frame` klases paneļa u.c. vizuālajiem elementiem.

`Graph` klase definē sekojošas metodes:

- `Node AddNode(...)` – pievieno virsotni grafam un atgriež norādi uz izveidoto virsotnes objektu
- `void RemoveNode(...)` – izņem virsotni no grafa
- `Edge AddEdge(...)` – pievieno šķautni grafam un atgriež norādi uz izveidoto objektu
- `void RemoveEdge` – izņem šķautni no grafa
- `Token AddToken(...)` – pievieno kustīgo elementu grafam un atgriež norādi uz izveidoto objektu
- `void RemoveToken(...)` – izņem kustīgo elementu no grafa

`GraphElement` abstraktā klase ir bāzes klase grafa elementiem. Klase definē sekojošos atribūtus:

- `Name` – satur elementa nosaukumu tā identifikācijai. Ja tas nav norādīts, grafs piešķir ģenerētu nosaukumu.

`GraphElement` klase definē sekojošas metodes:

- `UIElement[] GetUIElements()` - abstrakta infrastruktūras metode, kuru jārealizē visām apakšklasēm, lai atgrieztu vizuālos elementus, kurus jārāda uz grafa. Tādā veidā grafs viegli var iegūt vizuālos elementus, kas vizualizē attiecīgo elementu

`Token` ir `GraphElement` apakšklase, kas apraksta grafa kustīgo elementu. `Token` klase definē sekojošus atribūtus:

- `StructureElement` – atribūts, kas norāda struktūras elementu uz kura atrodas kustīgais elements.
- `Position` – atribūts, kas norāda kustīgā elementa pozīciju uz struktūras elementa

`Token` klase nedefinē publiskas metodes, kuras ir izsaucamas no AE bibliotēkas lietotāja koda – tā tikai satur infrastruktūras metodes.

`StructureElement` ir abstrakta klase, kas apraksta statisku grafa struktūras elementu. Tā ir `GraphElement` apakšklase un nedefinē jaunas atribūtus vai metodes – tā eksistē tikai lai grupētu virsotņu un šķautņu klases.

`Node` klase apraksta grafa virsotni. Klase mantojas no `StructureElement`. Tiek definēti šādi atribūti:

- `X` – virsotnes X koordināta grafa plaknē
- `Y` – virsotnes Y koordināta grafa plaknē
- `Radius` – virsotnes rādiuss

`Node` klase nedefinē publiskas metodes – tā satur tikai infrastruktūras metodes.

`Edge` klase apraksta grafa šķautni, kas savieno divas virsotnes. Klase mantojas no `StructureElement`, tātad uz tās var būt atrasties kustīgais elements. Klase definē šādus atribūtus:

- `Start` – definē šķautnes sākuma virsotni
- `End` – definē šķautnes beigu virsotni

`Edge` klase nedefinē publiskas metodes – tā satur tikai metodes, kuras paredzētas izsaukšanai no animāciju dziņa.

Izmantojot aprakstītās klases iespējams izveidot statisku grafu. Zemāk dotais programmas fragments izveido grafu

```
public Graph CreateGraph(){
   Graph g = new Graph();

   Node n1 = g.AddNode( 100, 300, 40 ); // virsotne punktā (100, 300), rādiuss=40
   Node n2 = g.AddNode( 300, 100, 25 );
   Node n3 = g.AddNode( 450, 250, 40 );
   Node n4 = g.AddNode( 250, 450, 25 );

   Edge e1 = g.AddEdge( n1, n2 ); // šķautne no virsotnes n1 uz n2
   g.AddEdge( n2, n3 );
   Edge e2 = g.AddEdge( n3, n4 );
   g.AddEdge( n4, n1 );

   Token t1 = g.AddToken( e1, 0.0 ); // kustīgais elements uz šķautnes e1, poz=0
   Token t2 = g.AddToken( e2, 1.0 );
}
```

Programma apraksta grafu, kas sastāv no četrām virsotnēm n1, n2, n3 un n4. Šķautnes savieno virsotnes n1 un n2, n2 un n3, n3 un n4, un n4 un n1. Uz šķautnes e1 tiek novietots kustīgais elements pozīcijā 0, un uz šķautnes e2 tiek novietots kustīgais elements t2 pozīcijā 1.

5.2.3. Animāciju klases

Šajā sadaļā sniegts apraksts klasēm, kas apraksta kustīgo elementu animāciju. Klases atrodamas AE bibliotēkas pirmkoda apakšmapē '.\Animations'

`Animation` klase ir bāzes klase visām animāciju klasēm un satur dienesta funkcionalitāti laika skaitīšanai. Klase iekšēji izmanto WPF klasi `DispatcherTimer`, lai izpildītu iekšēju algoritmu, kas pārrēķina animējamā atribūta vērtību. Klase ir papildināta ar vairākiem atribūtiem, piemēram atribūtu, kas apraksta animācijas uzvedību beidzoties animācijai, un atribūtu, kas apraksta animācijas atkārtojumu skaitu. Klase definē šādus atribūtus:

- `Duration` – atribūts definē laika intervālu cik ilgā laikā jānotiek vienam animācijas atkārtojumam
- `BeginTime` – atribūts definē laika intervālu kopš animācijas sākšanas saņemšanas impulsa, līdz brīdim, kad AE dzinis sāk izpildīt animāciju
- `AutoReverse` – atribūts definē vai beidzoties animācijai, tā būtu automātiski jāatkārto pretējā secībā. Ja šis atribūta vērtība ir 'patiess', animācija aizņem divas reizes ilgāk nekā definēts `Duration` atribūtā
- `CompletedBehaviour` – uzskaitījuma datu tipa (definēts failā `AnimationCompletedBehaviour`) vērtība, kas definē animācijas uzvedību, beidzoties animācijai. Iespējamie varianti ir:
    - `Rewind` – atribūta vērtība tiek atgriezta sākotnējā stāvoklī pirms animācijas sākšanas

- Hold – atribūta vērtība saglabā vērtību kāda tā ir pēc animācijas beigšanas
- RepeatCount – definē atkārtojumu skaitu

Animation klase definē vairākas metodes animācijas kontrolēšanai. Animācijas iespējams kontrolēt arī ar darbību klasēm (skat. *5.2.4. Darbību klases*). Klase definē šādas metodes:

- Start() – sāk animāciju
- Stop() – pārtrauc animāciju, izpildot CompletedBehaviour definēto darbību.
- Pause() – aptur animāciju. Animāciju iespējams atsākt izsaucot Start metodi
- Seek(...) – uzstāda animācijas vērtību uz vērtību kāda tā ir norādītajā laika momentā kopš animācijas sākuma.

Animation klase definē vairākus notikumus, kas ļauj veidot sarežģītāku animācijas struktūru (notikumu realizācija tuvāk apskatīta *5.2.5. Notikumu*). Klase definē sekojošus notikumus:

- AnimationStarted – notikums tiek signalizēts sākoties animācijai
- AnimationCompleted – notikums tiek signalizēts animācijai pilnībā beidzoties
- AnimationRepeatCompleted – notikums tiek signalizēts beidzoties vienam animācijas atkārtojumam
- AnimationPaused – notikums tiek signalizēts pārtraucot animāciju
- AnimationStopped – notikums tiek signalizēts apturot animāciju

Animation klase definē vairākas infrastruktūras metodes, interesantākā no tām ir UpdateCore(), kura apstrādā iekšējā taimera „tikšķu" notikumu un aprēķina animācijas progresu procentos un izsauc citu infrastruktūras metodi Update(double progress), kura pieejama tikai klasēm kuras mantojas no Animation klases. Animation klases apakšklases var izmantot Update(double progress) metodi, lai viegli izrēķinātu animējamā atribūta vērtību.

TokenPositionAnimation ir bāzes klase grafa kustīgo elementu animācijas klasēm. Klase ir Animation apakšklase un definē kopējos atribūtus kustīgo elementu animācijai:

- Target – atribūts definē grafa kustīgo elementu, kurš ir animācijas mērķis
- StructureElement – neobligāts atribūts, kas definē kuram grafa struktūras elementam pieder grafa kustīgais elements. Ja šis atribūts ir norādīts, animācija automātiski pārvietos kustīgo elementu uz norādīto struktūras elementu, ja tas uz tā neatradīsies.

Klase jaunas publiskas metodes nedefinē. Klase satur metodes, kas paredzētas izsaukšanas no animācijas dziņa.

`TokenPositionLinearAnimation` ir `TokenPositionAnimation` apakšklase, kas definē animāciju ar lineāru interpolāciju. Klase definē sekojošus atribūtus:

- `StartPosition` – definē kustīgā elementa sākuma pozīciju sākoties animācijai
- `EndPosition` – definē kustīgā elementa pozīciju animācijas beigās.

Klase jaunas publiskas metodes nedefinē, taču pārdefinē vairākas `Animation` klases infrastruktūras metodes, piemēram, iepriekš pieminēto `Update(..)` metodi. Pārdefinējot `Update()` metodi, lineārās animācijas realizācija kļūst triviāla:

```
protected override void Update( double progress ) {
   TargetToken.Position = StartPosition + ( EndPosition - StartPosition ) * progress;
}
```

`TokenPositionKeyFrameAnimation` ir `TokenPositionAnimation` apakšklase, kas definē animāciju ar atslēgkadriem. Klase pārdefinē vairākas dziņa infrastruktūras metodes, taču jaunas metodes nedefinē.

Klase definē sekojošus atribūtus:

- `Keyframes` – definē kolekciju ar `TokenPositionKeyFrame` klases apakšklases objektiem, kas apraksta animācijas atslēgkadrus.

Atslēgkadru klases atrodas `Animations` mapes apakšmapē `KeyFrames`. Mape satur failus ar trim klasēm – `TokenPositionKeyFrame`, `TokenPositionLinearKeyFrame` un `TokenPositionDiscreteKeyFrame`.

`TokenPositionKeyFrame` klase ir abstrakta bāzes klase, kas apraksta atslēgkadru. Klase publiskas metodes nedefinē. Klase definē sekojošus atribūtus:

- `Time` – apraksta laika momentu kopš animācijas sākuma, uz kuru attiecas atslēgkadrs
- `Position` – apraksta animējamā objekta vērtību atslēgkadrā.

`TokenPositionLinearKeyFrame` klase apraksta atslēgkadru ar lineāru interpolāciju no iepriekšējā atslēgkadra. Klase jaunus atribūtu un metodes nedefinē. Klase pārdefinē infrastruktūras metodes, definējot atslēgkadra lineāro uzvedību.

`TokenPositionDiscreteKeyFrame` klase apraksta atslēgkadru bez interpolācijas no iepriekšējā atslēgkadra. Klase mantojas no `TokenPositionKeyFrame` un jaunus atribūtus

nedefinē. Klase pārdefinē infrastruktūras metodes, definējot atslēgkadra diskrētās animācijas uzvedību.

Visbeidzot, dzinis satur klases animāciju grupēšanai.

`AnimationGroup` ir abstrakta bāzes klase animāciju grupas klasēm, kas mantojas no `Animation` klases. Klase definē sekojošu atribūtus:

`Duration` – klase pārdefinē `Animation` klases definēto `Duration` atribūtu, lai uzstādot tā vērtību tā proporcionāli tiktu sadalīta grupas elementiem. Nolasot vērtību atribūts atgriež grupas elementu intervālu summu.

`Elements` – definē animāciju grupas elementus.

Klase definē `Add(..)`, `AddRange(..)` un `Remove(..)` metodes grupas elementu pievienošanai un izņemšanai no grupas.

`ParallelAnimationGroup` klase mantojas no `AnimationGroup` klases un definē grupu, kuras elementi tiek palaisti paralēli. Klase nedefinē jaunus atribūtus, taču pārdefinē `Duration` atribūtu, lai atribūts atgrieztu garākās animācijas ilgumu.

`SequentialAnimationGroup` klase mantojas no `AnimationGroup` klases un definē grupu, kur elementi tiek palaisti virknē – beidzoties pirmajai virknes animācijai tiek palaista nākamā utt. Klase jaunus atribūtus nedefinē, taču pārdefinē `Duration` atribūtu, lai atribūts atgrieztu grupas animāciju ilgumu summu.

5.2.4. Darbību klases

Šajā sadaļā sniegts apraksts klasēm, kas realizē darbību klases. Klases atrodamas AE bibliotēkas pirmkoda apakšmapē '.\Actions'.

Action ir abstrakta klase, kas definē saskarni ar klases apakšklasēm.

Klase nedefinē atribūtus – tā definē tikai vienu metodi:

- `Execute()` – abstrakta metode, kuru jāpārdefinē visām apakšklasēm. Piemēram, animācijas darbību apakšklasēs šī metode izsauc atbilstošo darbību mērķa animācijas objektam

`AnimationAction` ir abstrakta bāzes klase visām animāciju darbību klasēm. Klase definē tikai vienu atribūtu:

- `Target` – definē animāciju, uz kuru iedarbosies darbība.

No `AnimationAction` klases mantojas `StartAnimationAction`, `PauseAnimationAction` un `StopAnimationAction`. Klases nedefinē jaunus atribūtus un jaunas metodes, taču pārdefinē `AnimationAction` definēto `Execute()` metodi, lai izsauktu attiecīgi `Start()`, `Pause()`, `Stop()` un `Seek(...)` metodes `Target` atribūta definētajai mērķa animācijai.

Līdzīgi, no `AnimationAction` mantojas `SeekAnimationAction` klase, kura pārdefinē `Execute()` metodi un definē vienu atribūtu `Time`, `kurš` tiek nodots kā parametrs `Seek(..)` metodes izsaukumā.

`ProgramStartAction` ir `Action` apakšklase, kas ļauj izpildīt norādītu funkciju. Šī darbība var pildīt līdzīgu lomu kā transformācijas programmas – brīvi mainīt animētā grafa struktūru, veidot jaunus elementus, izņemt elementus no grafa utml.

Klase satur norādi uz izpildāmu funkciju, kura jāizpilda izpildot darbību.

Darbību klases ir noderīgas kombinācijā notikumiem, jo ļauj aprakstīt darbību, kas notiks nenoteiktā nākotnē, nerakstot notikuma apstrādes funkciju.

5.2.5. Notikumu klase

Šajā sadaļā sniegts apraksts klasei, kas realizē notikumus. Klase atrodama AE bibliotēkas pirmkoda apakšmapē '.\Events'

Event klase definē notikumu apstrādes mehānismu. Klases instances uztur sarakstu ar darbībām, kuras jāveic reaģējot uz notikumu. Notiekot notikumam, klases instance secīgi izpilda reģistrētās darbības, izsaucot `Action` klases definēto `Execute()` metodi.

Klase nedefinē publiskas metodes. Klase definē vienu atribūtu:

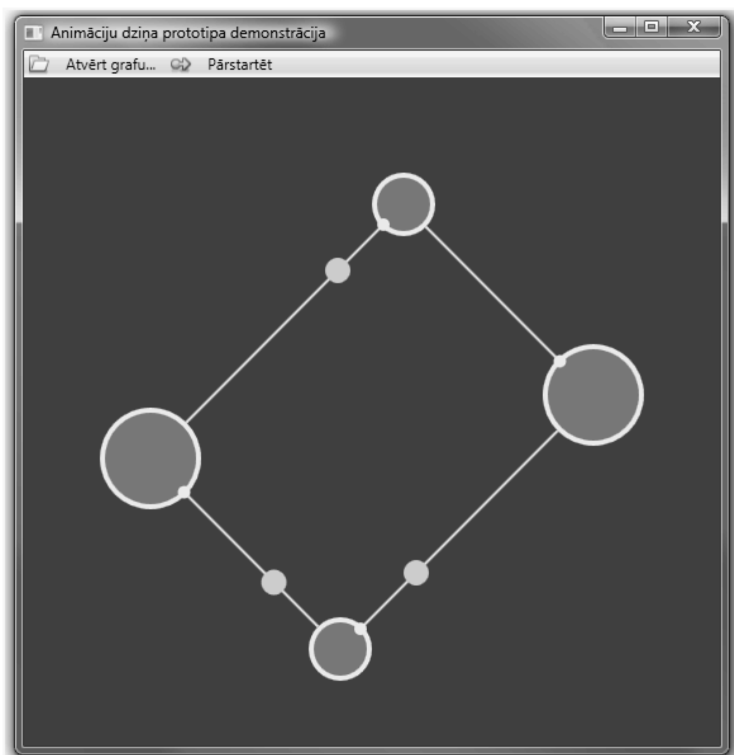
- `Actions` – saraksts ar `Action` klases apakšklasēm, kuras jāizpilda iestājoties notikumam.

5.3. Demonstrācijas programma

Demonstrācijas programma AEClient demonstrē iepriekš aprakstītās AE bibliotēkas iespējas. Aprakstītās bibliotēkas klašu instances uzdod grafa animētu orientētu grafu modeļus jeb konfigurācijas. Lai izveidotu vienu animēta grafa konfigurāciju, nepieciešams izveidot nepieciešamās klašu instances un uzstādīt to atribūtu vērtības. Programma, izmantojot AE bibliotēkas dzini, vizualizē animētu orientētu grafu konfigurācijas.

5.3.1. Programmas apraksts

Programmas logs (redzams *5.1. att. AEClient programmas saskarne*) sastāv no programmas izvēlnes loga augšējā daļā un grafa zīmēšanas virsmas, kas aizņem lielāko loga daļu.



5.1. att. **AEClient programmas saskarne**

Izvēlnē pieejamas sekojošas komandas:

- Atvērt grafu... – parāda lietotājam izkrītošu izvēlni ar vairākiem vienumiem. Klikšķinot uz kāda no vienumiem tiek atvērts attiecīgā grafa konfigurācija, kas demonstrē kādu no dziņa īpašībām.
- Pārstartēt – sāk animāciju no sākuma

5.3.2. Animētu grafu demonstrācijas konfigurācijas

Šajā sadaļā aprakstīti vairākas animētu grafu piemēra konfigurācijas, kas demonstrē AE bibliotēkas funkcionalitāti.

Aprakstītās konfigurācijas pirmkods atrodams pielikuma AEClient pirmkoda apakšmapē failā MainWnd.Graphs.cs.

Piemēra konfigurācijas apskatāmas palaižot programmas AEClient izpildfailu, programmas logā klikšķinot uz izvēlnes punkta ‘Atvērt grafu...’ un izvēloties attiecīgo piemēru no izkrītošās izvēlnes. Sekojošo piemēru aprakstu virsraksti sakrīt ar izvēles elementiem izkrītošajā izvēlnē.

5.3.2.1. Lineāras animācijas demonstrācija

Šajā piemērā tiek demonstrētas lineāras animācijas. Tiek definēts grafs ar četrām virsotnēm – n1, n2, n3 un n4. Šķautnes e1, e2, e3 un e4 attiecīgi savieno virsotnes (n1, n2), (n2, n3), (n3, n4) un (n1, n4). Kustīgie elementi t1, t2, t3 un t4 tiek novietoti katrs uz savas šķautnes. Sākotnēji kustīgie elementi atrodas uz šķautnes sākuma gala - pozīcijā 0. Katram kustīgajam elementam tiek definēta lineāra animācija no šķautnes sākuma līdz šķautnes beigām.

Kustīgā elementa t1 animācija ilgst 5 sekundes. Animācija tiek atkārtota divas reizes, beidzoties atkārtojumam animācija atgriež kustīgo elementu sākuma stāvoklī. Kopējais animācijas laiks ir 10 sekundes.

Kustīgā elementa t2 animācija ilgst 2 sekundes un animācijai beidzoties elements saglabā savu pozīciju šķautnes beigu galā.

Kustīgā elementa t3 animācija ilgst 1 sekundi un tā tiek atkārtota 5 reizes. Beidzoties atkārtojumam animācija tiek automātiski atspēlēta pretējā virzienā – tādējādi kustīgais elements pārvietojas starp virsotnēm n3 un n4. Kopējais animācijas ilgums ir 10 sekundes.

Kustīgā elementa t4 animācija ilgst 2 sekundes un animācijai beidzoties elements saglabā savu pozīciju šķautnes beigu galā. Animācija sākas ar 3 sekunžu nobīdi, līdz ar to tā sākas tikai pēc tam, kad t2 elementa animācija ir beigusies.

Visbeidzot, visas definētās animācijas tiek ievietotas grafa animāciju grupā, kas tās startē, notiekot grafa konfigurācijas ielādei.

Programma, kas konfigurē grafu pēc augstāk dotā apraksta dota zemāk:

```
...
Graph g = new Graph();

// Grafa virsotņu izveide, norādot X un Y koordinātas un rādiusu
Node n1 = g.AddNode( 100, 300, 40 );
Node n2 = g.AddNode( 300, 100, 25 );
Node n3 = g.AddNode( 450, 250, 40 );
Node n4 = g.AddNode( 250, 450, 25 );

// Grafa šķautņu izveide, norādot sākuma un beigu virsotnes
Edge e1 = g.AddEdge( n1, n2 );
Edge e2 = g.AddEdge( n2, n3 );
Edge e3 = g.AddEdge( n3, n4 );
Edge e4 = g.AddEdge( n4, n1 );

// Grafa kustīgo elementu izveide, norādot struktūras elementu un pozīciju
Token t1 = g.AddToken( e1, 0.0 );
Token t2 = g.AddToken( e2, 0.0 );
Token t3 = g.AddToken( e3, 0.0 );
Token t4 = g.AddToken( e4, 0.0 );

// Animāciju izveide un konfigurācija
TokenPositionLinearAnimation a1 =
   new TokenPositionLinearAnimation( t1, null, 0.0, 1.0, new TimeSpan( 0, 0, 0, 5, 0 ) );
a1.RepeatCount = 2;
```

```
TokenPositionLinearAnimation a2 =
   new TokenPositionLinearAnimation( t2, null, 0.0, 1.0, new TimeSpan( 0, 0, 0, 2 ) );

TokenPositionLinearAnimation a3 =
   new TokenPositionLinearAnimation( t3, null, 0.0, 1.0, new TimeSpan( 0, 0, 0, 1, 0 ) );
a3.AutoReverse = true;
a3.RepeatCount = 5;

TokenPositionLinearAnimation a4 =
   new TokenPositionLinearAnimation( t4, null, 0.0, 1.0, new TimeSpan( 0, 0, 0, 2 ) );
a4.BeginTime = new TimeSpan( 0, 0, 0, 3 );

g.Animations.AddRange( a1, a2, a3, a4 );

return g;
```

Piemērs demonstrē, ka ar relatīvi īsu programmu var aprakstīt sarežģītu animētu orientētu grafu.

Grafa konfigurācijas pirmkods atrodams AEClient pirmkoda failā MainWnd.Graphs.cs, kur tas tiek definēts programmas klases atribūtā `Lineāra_animācija`.

5.3.2.2. Animāciju grupas demonstrācija

Šajā piemēra tiek demonstrēta animāciju grupēšanas iespējas. Tiek definēts grafs ar trim virsotnēm n1, n2 un n3. Šķautnes e1 un e2 attiecīgi savieno virsotnes (n1, n2) un (n2, n3). Tiek definēts kustīgais elements t1, kurš tiek novietots uz šķautnes e1 sākuma gala.

Tiek definētas divas lineāras animācijas a1 un a2, kuras katra ilgst 5 sekundes. Animācija a1 pārvieto kustīgo elementu t1 no šķautnes e1 sākuma gala uz beigu galu, a2 - šķautnes e2 sākuma gala uz beigu galu.

Tiek izveidota `SequentialAnimationGroup` animācijas grupa g, kurai tiek pievienotas animācijas a1 un a2. Beidzoties atkārtojumam animācija tiek izpildīta apgrieztā secībā. Animāciju grupa tiek izpildīta divas reizes, tādējādi animāciju kopējais ilgums ir 20 sekundes.

Animāciju grupa g tiek ievietota grafa animāciju kolekcijai, tādējādi tā tiek startēta notiekot grafa konfigurācijas ielādei.

Grafa konfigurācijas pirmkods atrodams AEClient pirmkoda failā MainWnd.Graphs.cs, kur tas tiek definēts programmas klases atribūtā `Lineāras_animācijas_grupa`.

5.3.2.3. Atslēgkadru demonstrācija

Šajā piemērā tiek demonstrētas animācijas, kas sastāv no atslēgkadriem. Tiek definēts grafs ar trim virsotnēm n1, n2 un n3. Šķautnes e1 un e2 attiecīgi savieno virsotnes (n1, n2) un (n2, n3). Tiek definēts kustīgais elements t1, kurš tiek novietots uz šķautnes e1 sākuma gala.

Tiek definētas divas animācijas, kuras sastāv no atslēgkadriem, kuras katra ilgst 4 sekundes. Animācija a1 pārvieto kustīgo elementu t1 no šķautnes e1 sākuma gala uz beigu galu, a2 - šķautnes e2 sākuma gala uz beigu galu.

Animācija a1 satur 6 atslēgkadrus ar lineāru interpolāciju, tādējādi tā plūstoši pārvieto kustīgo elementu t1 pa šķautni e1. Animācija a2 satur 6 atslēgkadru, kas uzstāda diskrētas vērtības (interpolācija nenotiek), tādējādi animācija a2 rada iespaidu, ka kustīgais elements pa šķautni e2 pārvietojas ar lēcieniem.

Tiek izveidota `SequentialAnimationGroup` animācijas grupa g, kurai tiek pievienotas animācijas a1 un a2. Beidzoties atkārtojumam animācija tiek izpildīta apgrieztā secībā. Animācija rada iespaidu, ka kustīgais elements plūstoši pārvietojas no virsotnes n1 uz virsotni n2, pēc tam ar lēcieniem pārvietojas no virsotnes n2 uz virsotni n3, pēc animāciju beigām.

Animāciju grupa g tiek pievienota grafa animāciju kolekcijai, kas tiek startēta automātiski notiekot grafa konfigurācijas ielādei.

Grafa konfigurācijas pirmkods atrodams AEClient pirmkoda failā MainWnd.Graphs.cs, kur tas tiek definēts programmas klases atribūtā `Atslēgkadru_animācija`.

5.3.2.4. Notikumu demonstrācija

Šajā piemērā tiek demonstrēta notikumu un darbību klašu.

Tiek definēts grafs ar četrām virsotnēm – a, b, c un t. Šķautnes ab, cb, un tb attiecīgi savieno virsotnes (a,b), (c,b) un (t,b). Uz šķautņu sākuma galiem tiek novietots kustīgais elements – tiek izveidoti kustīgie elementi tab, tcb un ttb.

Tiek izveidotas animācijas a1, a2 un a3, kas pārvieto attiecīgi elementus tab, tcb un ttb no šķautnes sākumu uz šķautnes beigām. Visas animācijas, sasniedzot beigu punktu, automātiski izpilda kustību pretējā virzienā – radot iespaidu, ka kustīgie elementi pārvietotos starp attiecīgās šķautnes virsotnēm. Animācija a1 ilgst 2 sekundes, a2 – 3 sekundes, bet a3 – 7 sekundes.

Animācijai a1 atkārtojuma beigšanās notikumam tiek pievienota darbība, kura izpildoties vienam animācijas atkārtojumam, izpilda animācijas pauzēšanas darbību uz animāciju a3. Savukārt, animācijai a2 atkārtojuma beigšanās notikumam tiek pieveinota darbība, kura izpildoties vienam animācijas a2 atkārtojumam, izpilda animācijas startēšanas darbību animācijai a3. Līdz ar to viens kustīgais elements aptur animāciju a3, kamēr otrs to atkal startē.

Visas animācijas tiek pievienotas grafa animāciju grupai, kura tiek startēta automātiski.

Grafa konfigurācijas pirmkods atrodams AEClient pirmkoda failā MainWnd.Graphs.cs, kur tas tiek definēts programmas klases atribūtā `Notikumu_demonstrācija`.

5.4. Iespējamie uzlabojumi

Šajā sadaļā apskatīti vairāki iespējamie uzlabojumi animācijas dziņa uzlabojumi, kas palielinātu tā pielietojuma iespējas, uzlabotu tā klašu struktūru un līdz ar to arī pārnesamību uz citām vidēm.

5.4.1. Integrācija ar LU MII repozitoriju

Pirmais solis izstrādātās bibliotēkas plašāku lietojumu nodrošināšanai ir integrācija ar esošo Latvijas Universitātes Matemātikas un informātikas institūtā (turpmāk LU MII) rīku platformu.

Viens no centrālajiem komponentiem LU MII platformā, kas jau pieminēta sadaļā *1.4. MDA izstrāde LU MII*, ir MII repozitorijs – ātrdarbīga un efektīva datu glabātuve, kas izmanto datora operatīvo atmiņu kā fizisko datu glabātuvi. LU MII rīki, kas risina MDA problēmas, izmanto repozitoriju kā datu glabātuvi – repozitorijā tiek glabāti apstrādājamo metamodeļu definīcijas un to modeļi (11).

MII repozitorijs ir realizēts kā DLL bibliotēka, kas piedāvā API funkcijas metamodeļa definēšanai un modeļu instanču definēšanai. Repozitorijā definētas funkcijas metamodeļa pārvaldībai - klašu izveidošanai, atribūtu pievienošanai, asociāciju saišu definēšanai. Tiek definētas funkcijas arī modeļu instanču pārvaldībai – klašu objektu veidošanai un dzēšanai, asociāciju saišu izveidei, utml. (12)

MII repozitorija lietošanai ir acīmredzamas priekšrocības. LU MII izstrādātas vairākas valodas, kas ļauj definēt transformācijas, kas transformē modeļa instanci no viena metamodeļa modeļa uz citu metamodeli. Tādējādi iespējams, piemēram, uzrakstīt transformāciju, kas pārveido aktivitāšu diagrammu par animētu grafu, tādējādi efektīvi izmantojot esošo realizēto funkcionalitāti, jo nav speciāli jāprogrammē programma, kas spētu attēlot aktivitāšu diagrammu – aktivitāšu diagramma tiek realizēta ar animēta grafa līdzekļiem.

Pašreizējā AE animācijas dziņa bibliotēkas realizācija glabā modeļu instances savās iekšējās atmiņas datu struktūrās.

Formulējamas divas stratēģijas kā integrēt AE animācijas dziņa prototipu ar MII repozitoriju.

Pirmā stratēģija ir paturēt AE objektu datus iekšējās datu struktūrās veidot sinhronizācijas slāni starp MII repozitoriju un AE, kas izmaiņas vienā modeļa instancē pārsūta uz otru. Šai pieejai ir priekšrocība, ka animācijas dzinim nepieciešamie dati ir viegli pieejami – nav nepieciešami funkciju izsaukumi uz ārēju komponentu. Šādai pieejai ir trūkums, ka dati

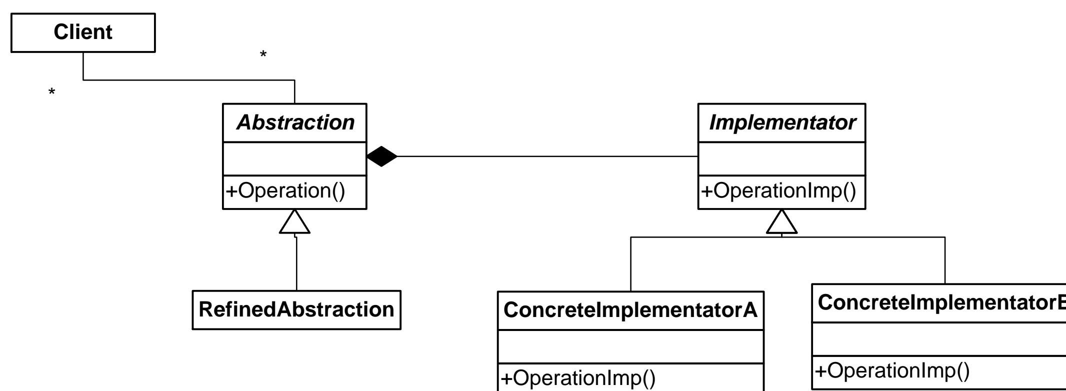
glabājas divās vietās – vienas klases objekta atribūta vērtība glabājas gan AE bibliotēkas atmiņā, gan repozitorijā. Turklāt sinhronizācijas divvirzienu sinhronizācijas algoritms nav triviāls – nepieciešams definēt rīcību gadījumos, kad atribūta vērtība veicot sinhronizāciju ir mainījusies abos modeļos.

Otrā stratēģija ir glabāt AE klašu atribūtus repozitorijā un piekļūt tiem izsaucot MII repozitorija piedāvātās funkcijas. Šai pieejai ir tāda priekšrocība, ka klašu instanču atribūtu vērtības tiek glabātas vienuviet – repozitorijā, un tie nav jāsinhronizē. Tomēr, dēļ tā, ka animāciju dzinim nepieciešama ātra piekļuve atribūtiem, jo to vērtības ir jānolasa un jāmaina vairākus simtus reizes sekundē, šai pieejai pastāv risks, ka risinājumam ir nepietiekama ātrdarbība.

5.4.2. Klašu struktūras uzlabojumi

Aprakstītajai klašu bibliotēkai ir vērā ņemama problēma – klases, kas apraksta abstrakciju vienlaikus satur arī abstrakcijas realizāciju. Piemēram, virsotne, šķautne, un kustīgais elements visi satur tiešas atsauces uz WPF klasēm, kas piesaista animāciju dziņa realizāciju WPF bibliotēkai. Piemēram, lai realizētu šādu klašu bibliotēku, izmantojot Windows Forms klašu bibliotēkas līdzekļus, nepieciešams pārrakstīt visas bibliotēkas klases, kuras satur atsauces uz WPF objektiem. Izstrādātā klašu bibliotēka ir tikai prototips, kura uzdevumos nav atbalstīt vairākas grafiskās sistēmas, taču realizējot izklāstītos principus produkcijas līmeņa kodā, ir iespējams, ka šāda īpašība ir nepieciešama.

Lai uzlabotu klašu struktūras pārnesamību iespējams veikt modifikācijas klašu struktūrā ieviešot tilta projektējuma šablonu (bridge design pattern) (26). Tilta projektējuma šablona vispārīgā forma redzama ***5.2. att.*** Tilta projektējuma šablona vispārīgā forma:

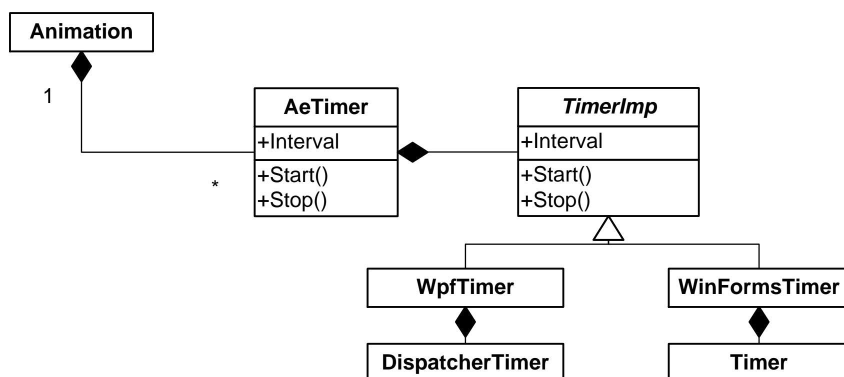


5.2. att. **Tilta projektējuma šablona vispārīgā forma**

Šajā gadījumā `Client` klase (AEClient programmas klases), izmanto `Abstraction` (AE bibliotēkas klases). `Abstraction` klases satur atsauci uz `Implementator` klases apakšklasi. `Implementator` abstraktā klase definē saskarni konkrētajām realizācijas klasēm – no tās

mantojas klases, kas realizē funkcionalitāti (izmantojot, piemēram, WPF un WinForms klases). Tādā veidā iespējams viegli nomainīt abstrakcijas realizāciju – nomainot atsauci uz realizācijas klasi.

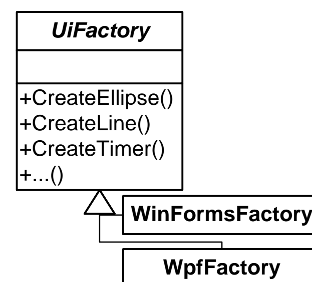
Šo šablonu var pielietot visām AE klasēm, kas satur atsauces uz WPF klasēm, tādējādi izslēdzot atkarību no WPF. Piemēram, pielietojot šablonu uz `Animation` klasi, lai izslēgtu atkarību no `DispatcherTimer` klases, iegūst struktūru, kas redzama *5.3. att. Tilta projektējuma šablona pielietojums*:



5.3. att. **Tilta projektējuma šablona pielietojums**

Protams, `AeTimer` abstrakcijas klasei nepieciešams zināt kuru realizācijas klasi izveidot – `WpfTimer` vai `WinFormsTimer`. Ja `AeTimer` abstrakcijas klase izveidotu kādu no šīm realizācijām tieši, tad tā atkal būtu piesaistīta kādai no realizācijām. Šo problēmu var atrisināt pielietojot abstraktās fabrikas (abstract factory) projektējuma šablonu.

Abstraktā fabrika sastāv no abstraktās klases, kas definē metodes elementu izveidošanai un konkrētām klasēm, kas izveido attiecīgās bibliotēkas elementus. Piemēram, var tikt definēta klase `UiFacotry`, kas definē metodes taimera elementa izveidei. No `UiFactory` mantojas divas klases – `WinFormsFacotry` un `WpfFactory`, kuras katra pārdefinē šo metodi, lai atgrieztu pareizā tipa elementu (skat. *5.4. att. Abstraktās fabrikas šablona pielietojums*). Programmā veicot elementu izveidi caur centrālu objektu, tiktu izslēgta atkarība no konkrētas bibliotēkas. Tādā veidā grafu iespējams konfigurēt ar jebkuru no šīm fabrikām, līdz ar to nodrošinot neatkarību no grafiskās bibliotēkas.



5.4. att. **Abstraktās fabrikas šablona pielietojums**

Diskutabls ir jautājums vai šādu konstrukciju ieviešana ir nepieciešama un vai tā labākā pieeja pārnesamības realizēšanai. Piemēram, tilta šablons ļauj izstrādāt vairākas realizācijas dažādām grafiskajām bibliotēkām. MDA pieeja būtu tā vietā definēt transformāciju no abstrakcijas modeļa uz realizācijas modeli. Diemžēl, nav aprakstīts WPF un WinForms

metamodelis, kuru varētu izmantot transformāciju definēšanai. Turklāt, ne vienmēr šāda transformācija ir iespējama, piemēram, WinForms gadījumā šī klašu bibliotēka nepiedāvā klases ģeometrisku figūru zīmēšanai – nepieciešams izmantot `Graphics` objekta metodes, kuras grūti izmantojamas no transformācijas programmām. Līdz ar to nepieciešama papildus klase, kas izsauc attiecīgo metodi. Gadījumos, kad transformācijas programmu ir iespējams izveidot, MDA pieeja varētu būt izdevīgāka.

5.5. Kopsavilkums

Šajā sadaļā tika aplūkota animācijas dziņa prototipa realizācija, kā arī programma, kas demonstrē tā darbību ar vairākām animētu grafu konfigurācijām. Veikta iespējamo prototipa uzlabojumu apskats un analīze.

6. SECINĀJUMI

Šajā darbā izstrādāti principi animāciju metamodeļa definēšanai, kas ļauj modelēt animācijas, kuras balstās uz objektu atribūtu vērtību mainīšanu laikā. Apskatīts MDA ietvars, eksistējošie risinājumi animāciju veidošanai, formulēti principi animāciju aprakstoša metamodeļa būvei un realizēts prototips animāciju dzinim.

Nodaļā *1. Model Driven Architecture ietvars* apskatītas Model Driven Architecture ietvara pamatidejas, tradicionālās programmatūras izstrādes problēmas un MDA piedāvātie šo problēmu risinājumi. Nodaļā *2. Datoranimācija* sniegts īss ieskats animācijas principos. Nodaļā *3. Eksistējošie risinājumi animācijas izstrādei* aprakstīti vairāki izpētītie eksistējoši risinājumi, kuru funkcionalitāte iekļauj animāciju veidošanu. Nodaļā *4. Animācijas metamodelis* formulēti principi animācijas metamodeļa veidošanā – izstrādāta vispārīgā metamodeļa forma. Kā piemērs apskatīts viens no iespējamiem pielietojumiem, kurš arī kalpo kā problēmapgabals animāciju dziņa prototipa bibliotēkas izstrādē, un definēts tā metamodelis. Darba ietvaros realizēta animāciju dziņa prototipa bibliotēka, kas ļauj definēt orientētus grafus un animēt tā kustīgos elementus atbilstoši aprakstītajām problēmapgabalam. Nodaļā *5. Prototipa realizācijas apraksts* aprakstīta prototipa bibliotēkas realizācija un demonstrēta tās darbība ar vienkāršu demonstrācijas programmu, izmantojot vairākas animētu grafu konfigurācijas kā piemērus.

MDA pieeja sola nākotnē būtiski mainīt programmatūras izstrādes procesu, automatizējot sistēmas apraksta transformēšanu par reālu izpildāmu programmatūras produktu.

Apskatītā LU MII izstrādātā GrTP arhitektūra, kas izmanto MDA idejas, ir interesanta ar to, ka tā vietā, lai ģenerētu izpildāmu kodu no PSM, tā to interpretē, tādējādi iegūstot lielāku rīcības elastību.

Apskatītajos eksistējošajos animācijas veidošanas risinājumos novērojami līdzīgi animāciju veidošanas principi – aprakstīt kompozīcijas stāvokli atsevišķos laika momentos, kuri tiek saukti par atslēgkadriem, un automātiski ģenerēt kompozīcijas stāvokli laika momentos starp atslēgkadriem. Šī ideja arī izmantota animācijas metamodeļa būvē – definēt animējamo atribūtu vērtības noteiktos laika momentos un pārējās vērtības ģenerēt automātiski. Aprakstītā animācijas metamodeļa pamatideja – aprakstīt objekta atribūtu vērtību maiņu laikā. Tādā veidā iespējams animēt jebkuru objekta raksturlielumu, kuru apraksta atribūts.

Aplūkotajā pielietojumā tika animēti grafa elementi divdimensiju plaknē. Taču aprakstītie principi ir pielietojami plašāk – šādi būtu iespējams aprakstīt animāciju arī citos scenārijos, piemēram, objektu kustību trīsdimensiju telpā.

Pieminēšanas vērts ir novērojums, ka apskatītā problēmapgabala animācijas dziņa realizācijas klases ļoti tuvu atbilst aprakstītā problēmapgabala metamodelim. Lai gan realizācijas klašu veidošanā tika izmantota tradicionāla pieeja programmatūras izstrādei, šis novērojums liek domāt par to, ka transformācija no metamodeļa uz kodu tiešām būtu iespējama.

Prototipa realizācijai būtu iespējams veidot zemākas abstrakcijas līmeņa metamodeli ar kuru būtu iespējams realizēt daudz plašāku funkcionalitāti, ne tikai orientētu grafu zīmēšanu. Šāds zemāka abstrakcijas līmeņa metamodeļa piemērs būtu WPF klašu bibliotēkas metamodelis, kurā programmētājam jāoperē ar tādiem objektiem kā ģeometriskām figūrām, nevis grafa virsotnēm un šķautnēm kā AE bibliotēkas gadījumā. Šādā gadījumā lielāks darbs būtu transformāciju programmas programmētājam, kuram jātransformē sava avota metamodeļa instances uz ģeometrisko figūru problēmapgabalu.

Izstrādājot prototipu radies iespaids, ka MDA pieeja atmaksājas lielu projektu izstrādē, kuriem paredzams ilgs mūžs. Liels darbs jāiegulda projekta sākumā, izstrādājot sistēmas modeli (PIM), PSM, nepieciešamos dziņus, kas realizē PSM un transformācijas starp PIM un PSM. Tas ir nepieciešams, jo pagaidām nav izstrādātu gatavu rīku, kas realizētu šo darbu. Taču nākotnē, kad jau būs pieejami izstrādāti gatavi komponenti, kuri pieejami MDA rīku platformai, kā arī definēti PSM dažādiem realizācijas tehnoloģijām, būs nepieciešams izstrādāt tikai sistēmas PIM un definēt transformāciju no PIM uz PSM. Tādējādi būtu iespējams ģenerēt veselas sistēmas no PIM, tādējādi šī MDA varētu radīt milzīgus produktivitātes ieguvumus pat mazos projektos.

7. IZMANTOTĀ LITERATŪRA UN AVOTI

1. **OMG.** *MDA Guide Version 1.0.1.* [Tiesšaiste] : OMG, 2003. [Atsauce 03.05.2008] http://www.omg.org/docs/omg/03-06-01.pdf.

2. **Kleppe, A., Warmer, J, Bast, W.** *MDA Explained: The Model Driven Arcitechture.* Boston : Addison Wesley, 2003. p. 1 - 45. ISBN 0-321-19442-X.

3. **Warmer, J, Kleppe, A.** *The Object Constraint Language: Getting Your Models Ready for MDA.* Boston : Addison Wesley Professional, 2003. ISBN 0-321-17936-6.

4. **OMG.** *Meta Object Facility (MOF) 2.0 Core Specification.* [Tiešsaiste]: OMG, 2003. [Atsauce 10.05.2008] http://www.omg.org/docs/omg/03-06-01.pdf.

5. **Latvijas Universitātes Matemātikas un informātikas institūts.** MOLA Tool Architecture. [Tiešsaiste] [Atsauce: 2008.01.20.] http://mola.mii.lu.lv/.

6. **Latvijas Universitātes Matemātikas un informātikas institūts.** *Model transformation languages L1, L2 and L3.* [Tiešsaiste] Rīga : Latvijas Universitātes matemātikas un informātikas institūts, 2007. [Atsauce 2008.01.20] http://lx.mii.lu.lv/L1L2L3.pdf.

7. **Latvijas Universitātes Matemātikas un informātikas institūts.** *The Base Transformation Language L0+.* [Tiešsaiste] Rīga : Latvijas Universitātes matemātikas un informātikas institūts, 2007. gada. [Atsauce 2008.01.20] http://lx.mii.lu.lv/L0_plus_CurrVers.pdf.

8. **Bārzdiņš, J., Zariņš, A., Čerāns, K., Kalniņš, A., Rencis, E., Lāce, L., Liepiņš, R., Sproģis, A..** *GrTP: Transformation Based Graphical Tool Building Platform.* Nashville, Tennessee, USA : MDDAUI, 2007. Proceedings of MODELS 2007.

9. **Hodgins, J., O'Brien, J., Bodenheimer, R.** *Computer Animation.* [Tiešsaiste] Atlanta : Georgia Institute of Technology, 1990.
[Atsauce 2008.03.30] http://gvu.cc.gatech.edu/animation/papers/ency.pdf.

10. **Parent, R.** *Computer Animation: algorithms and Techniques.* San Francisco : Morgan Kaufmann, 2001. p. 63-69. ISBN:1558605797.

11. **Microsoft Corporation.** Windows Forms Overview. *MSDN.* [Tiešsaiste] Microsoft Corporation. [Atsauce: 2008.04.23.]
http://msdn.microsoft.com/en-us/library/8bxxy49h(VS.80).aspx.

12. **Petzold, C.** *Programming Microsoft Windows Forms.* Redmond, Washington : Microsoft Press, 2005. ISBN 0735621535.

13. **Yakhnin, A.** Creating a Microsoft .NET Compact Framework-based Animation Control. *MSDN.* [Tiešsaiste] Microsoft Corporation, 2003. [Atsauce: 2008.04.20.] http://msdn.microsoft.com/en-us/library/aa446483.aspx.

14. **Powell, R.** Animating graphic objects in Windows Forms. [Tiešsaiste] [Atsauce: 2008.04.20.] http://www.bobpowell.net/animation.htm.

15. **Petzold, C.** *Programming Microsoft Windows with C#.* Redmond, Washington : Microsoft Press, 2002. ISBN 0-7356-1370-2.

16. **Sneath, T.** *Architectural Overview of the Windows Presentation Foundation Beta 1 Release.* [Tiešsaiste] Redmond, Washington : Microsoft Corporation, 2005. gada augusts. Windows Vista Technical Articles. [Atsauce: 2008.04.14.] http://msdn.microsoft.com/en-us/library/aa480177.aspx.

17. **Microsoft Corporation.** Overview of the .NET Framework. *MSDN.* [Tiešsaiste] [Atsauce: 2008.04.14.] http://msdn.microsoft.com/en-us/library/zw4w595w(VS.71).aspx.

18. **Petzold, C.** *Applications = Code + Markup: A Guide to the Microsoft Windows Presentation Foundation.* Redmond, Washington : Microsoft Press, 2006. lpp. p. 3 - 487, p. 859 - 935. 978-0-7356-1957-9.

19. **Microsoft Corporation.** WPF Architecture. *MSDN.* [Tiešsaiste] [Atsauce: 2008.03.30.] Pieejams: http://msdn.microsoft.com/en-us/library/ms750441.aspx.

20. **Cohen, B.** Silverlight Architecture Overview. [Tiešsaiste] Microsoft Corporation, 2007. [Atsauce: 2008.02.15.] http://msdn.microsoft.com/en-us/library/bb428859.aspx.

21. **Egger, M.** A Silverlight to Illuminate the Path Ahead. *CoDe Magazine.* [Tiešsaiste] 2007. gada 17. septembris. [Atsauce: 2008.02.15.] http://www.code-magazine.com/Article.aspx?quickid=070143.

22. **Microsoft Corporation.** Silverlight Animation Overview. *MSDN.* [Tiešsaiste] Microsoft Corporation, 2008. [Atsauce: 2008.02.16.] http://msdn.microsoft.com/en-us/library/cc189019(vs.95).aspx.

23. **Microsoft Corporation**. Microsoft Expression Blend. [Tiešsaiste] Microsoft Corporation, 2008. [Atsauce: 2008.05.10.] http://www.microsoft.com/expression/.

24. **Adobe, Inc.** Adobe Flash . [Tiešsaiste] Adobe Inc., 2008. [Atsauce: 2008.05.11.] http://www.adobe.com/products/flash/.

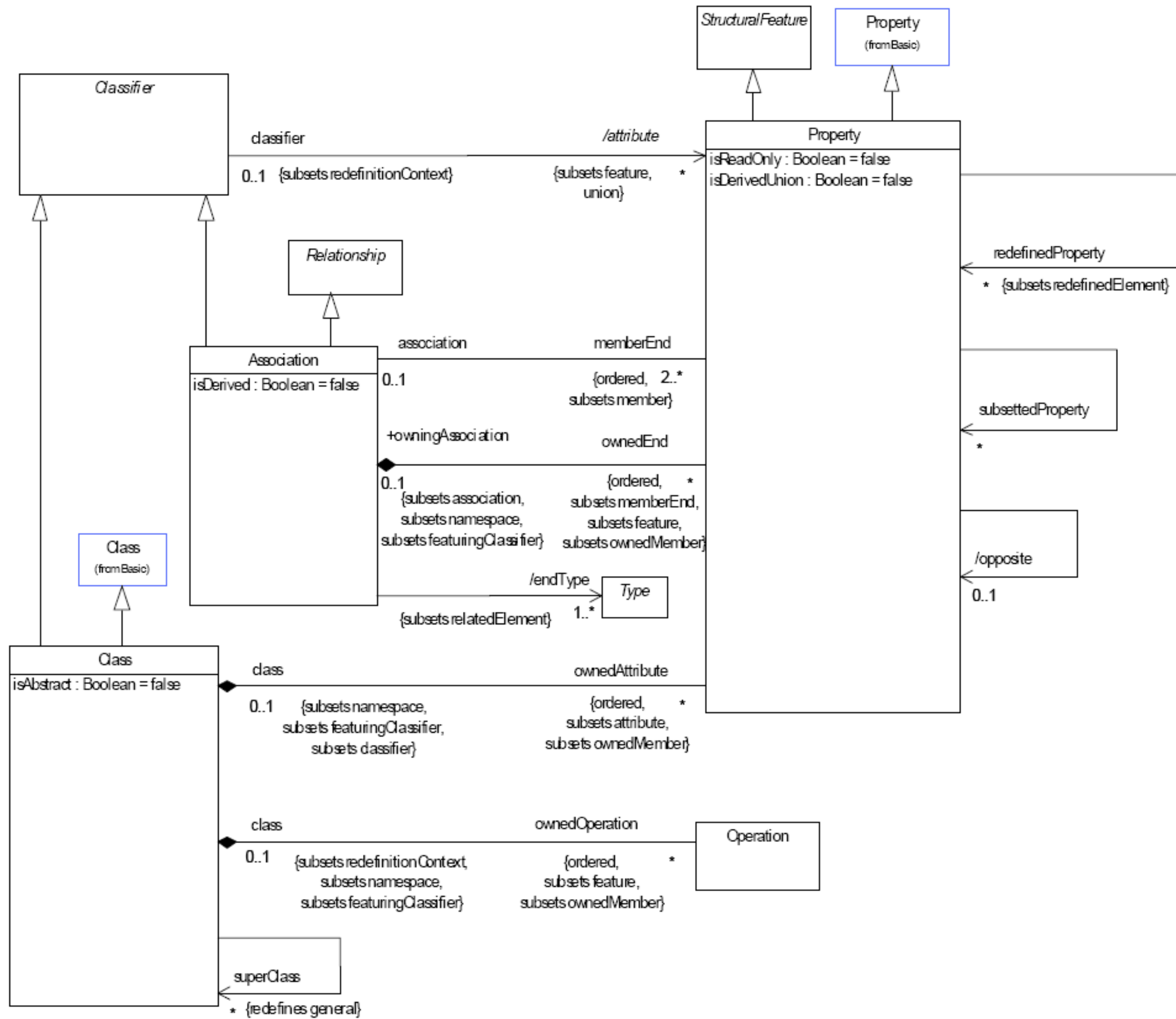
25. **Pleuss, A., Vitzthum, A., Hussmann, H.** *Integrating Heterogeneous Tools into Model-Centric Development of Interactive Application.* Munich : Springer-Verlag, 2007. MoDELS 2007. p. 241-255.

26. **Gamma, E., Helm, R., Johnson, R., Vlissides, J.** *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston : Addison-Wesley, 1995. 0-201-63361-2.

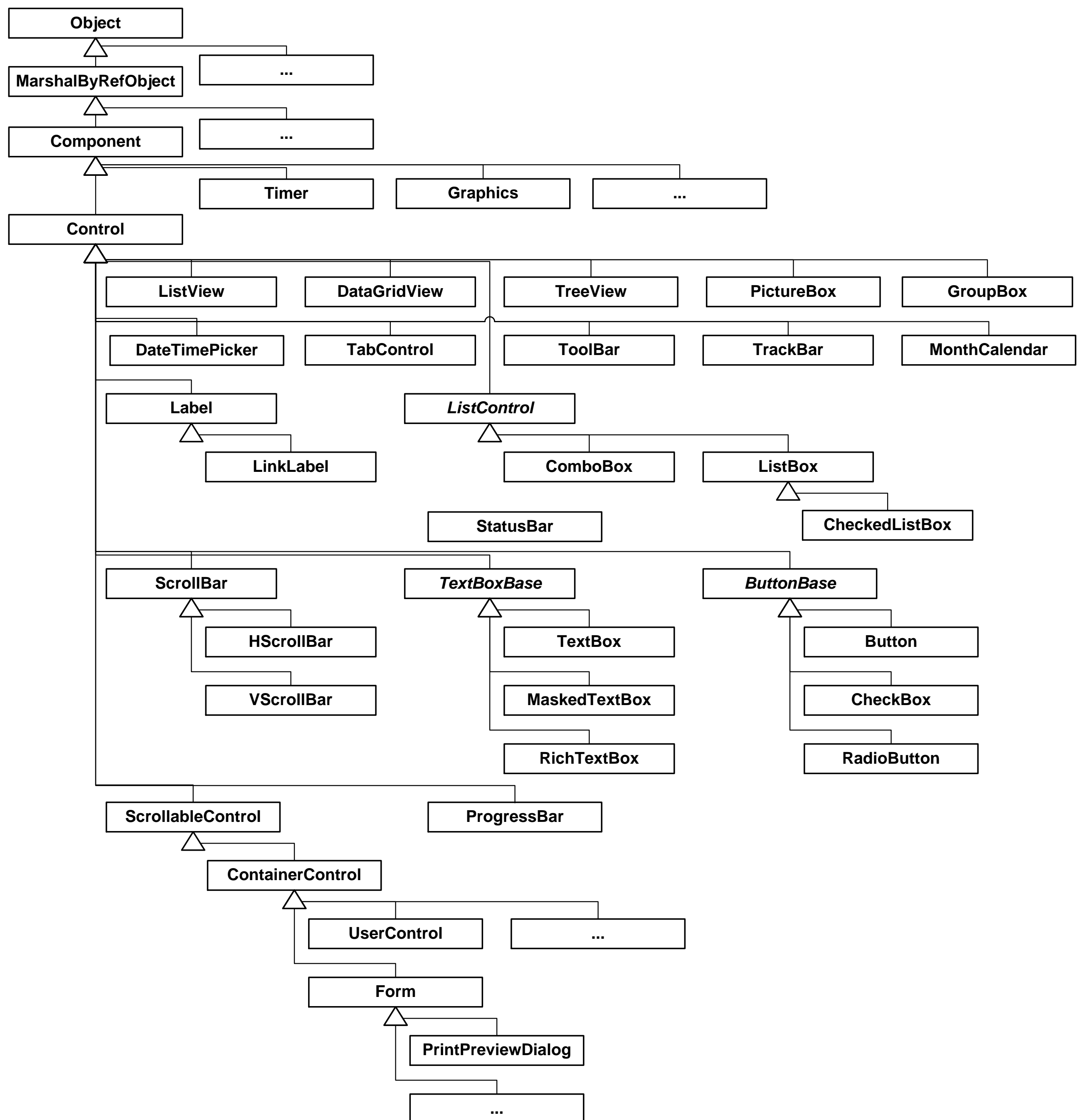
27. **Latvijas Universitātes matemātikas un informātikas institūts.** *Repozitorija mii_rep.dll saskarnes funkciju un konstanšu apraksts.* [Tiešsaiste] Rīga : LU MII, 2006. [Atsauce: 2008.01.20] http://mola.mii.lu.lv/.

PIELIKUMI

1. Pielikums – MOF klašu modelēšanas centrālo līdzekļu metamodelis



2. Pielikums – Windows Forms kontroļu klašu hierarhijas fragments

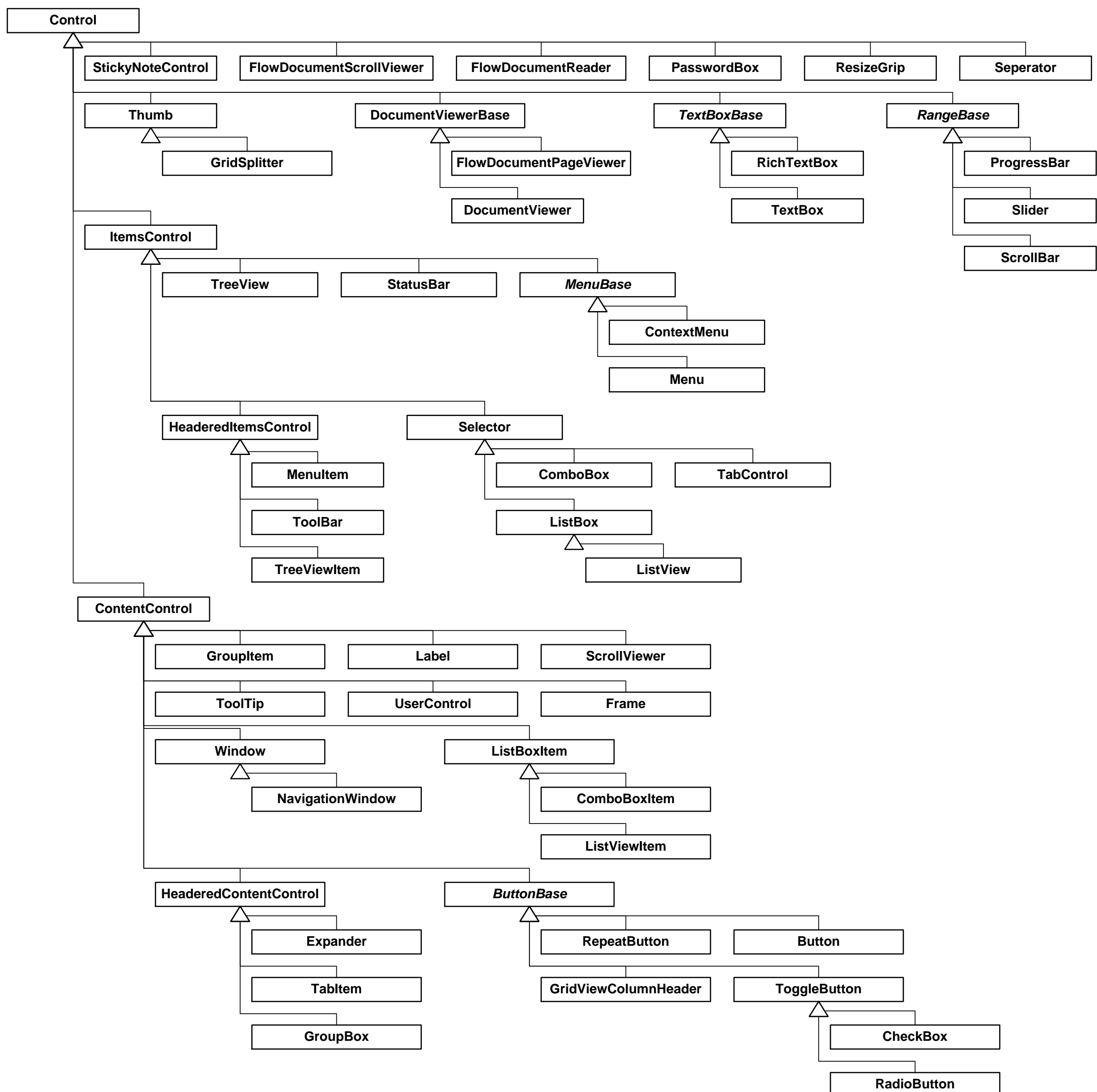


3. Pielikums – Animācijas programma, izmantojot Windows Forms

```
using System;
using System.Drawing;
using System.Windows.Forms;

namespace WinFormsAnimationDemo {
   // Programmas klase. Mantojas no Windows Forms Form klases.
   class Program : Form {

      [STAThread]
      static void Main() {
         // Main() metode. Startē programmu, parādot uz ekrāna programmas logu
         Application.Run( new Program() );
      }

      // Klases lauki: elements animēšanai un pulkstenis animācijas realizācijai
      PictureBox r;
      Timer animationTimer;

      // Klases mainīgie, animācijas paramtru vērtības
      int animationStartValue; // Atribūta sākuma vērtība
      int animationEndValue;   // Animējamā atribūta beigu vērtība
      TimeSpan animationDuration;      // Animācijas ilgums
      TimeSpan animationBeginOffset;   // Animācijas sākuma nobīde
      DateTime animationStartTime;     // Animācijas sākšanas laiks

      // Klases konstruktors. Izveido vizuālās kontroles un Timer objektu, uzstāda
      // animācijas parametru vērtības
      public Program() {
         // Loga parametru uzstādīšana
         Text = "Windows Forms animācijas demonstrācija";
         Width = 500;
         Height = 250;

         // Izveido melnu kvadrātu ar 100 vienību garu malu.
         // Novieto kvadrātu uz loga virsmas punktā (50, 50)
         r = new PictureBox();
         r.Height = r.Width = 100;
         r.BackColor = Color.Black;
         Controls.Add( r );
         r.Left = 50;
         r.Top = 50;

         // Animācijas parametru vērtības
         animationStartValue = r.Left;
         animationEndValue = 350;
         animationDuration = new TimeSpan( 0, 0, 0, 2 );
         animationBeginOffset = new TimeSpan( 0, 0, 0, 5 );

         // Izveido pulksteni, kurš signalizēs notikumu pēc 10 milisekundēm.
         // Pievieno notikuma apstrādātāju, kurš pārvieto kvadrātu
         animationTimer = new Timer( );
         animationTimer.Tick += new EventHandler( t_Tick );
         animationTimer.Interval = 10;

         animationStartTime = DateTime.Now;
         animationTimer.Start();
      }
```

```
        private void t_Tick( object sender, EventArgs e ) {
            DateTime now = DateTime.Now;
            if ( now < animationStartTime + animationBeginOffset ) {
                return;
            }

            // Aprēķina pagājušo laiku koopš animācijas sākuma
            TimeSpan elapsed = now - animationStartTime - animationBeginOffset;

            // Ja laiks kopš animācijas sākuma lielāks par animācijas ilgumu - apstāties
            if ( animationDuration < elapsed ) {
                animationTimer.Stop();
            }

            // Aprēķina animācijas progresu kā attiecību starp intervālu kopš animācija sākuma
            // un animācijai atvēlēto intervālu
            double progress = elapsed.Ticks / (double)animationDuration.Ticks;

            // Uzstāda animējamā objekta atribūtam jauno vērtību
            r.Left = animationStartValue +
                (int)(progress * ( animationEndValue - animationStartValue ));
        }
    }
}
```

4. Pielikums – WPF interaktīvo lietotāja saskarnes klašu hierarhija



5. Pielikums – WPF iekļautās animāciju klases

Datu tips	<Tips>Animation	<Tips>Animation-UsingPath	<Tips>AnimationUsingKeyFrames		
			Discrete	Linear	Spline
Boolean			x		
Byte	x		x	x	x
Char			x		
Color	x		x	x	x
Decimal	x		x	x	x
Double	x	x	x	x	x
Int16	x		x	x	x
Int32	x		x	x	x
Int64	x		x	x	x
Matrix		x	x		
Object			x		
Point	x	x	x	x	x
Point3D	x		x	x	x
Quaternion	x		x	x	x
Rect	x		x	x	x
Rotation3D	x		x	x	x
Single	x		x	x	x
Size	x		x	x	x
String			x		
Thickness	x		x	x	x
Vector	x		x	x	x
Vector3D	x		x	x	x

Tabulā apkopotas WPF pieejamās animācijas klases. Kolonnā „Datu tips” uzskaitīti animējamie datu tipi. WPF animāciju klašu nosaukumi ir veidoti pēc šabloniem <*Tips*>Animation – lineāras animācijas, <*Tips*>AnimationUsingPath – līknes animācijas, <*Tips*>AnimationUsingKeyFrames – atslēgkadru animācijas. Pieejamo atslēgkadru tipi sadalīti kolonnās zem kolonnas <*Tips*>AnimationUsingKeyFrames, kur Discrete – diskrētas vērtības atslēgkadrs, Linear – lineāras interpolācijas atslēgkadrs, Spline – līknes interpolācijas atslēgkadrs.

Tabulas šūnas satur atzīmi, ja atbilstošajam datu tipam ir pieejama konkrētais animācijas tips.

6. Pielikums – WPF DoubleAnimation demonstrācijas programma

Demonstrācijas programma WPF animācijas klasēm. Programma izveido logu ar melnu kvadrātu, kuram tiek animēta tā pozīcija no kreisās loga malas.

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Shapes;
using System.Windows.Media.Animation;
using System.Windows.Media;

namespace WPFAnimDemo {
   // Programmas klase. Klase mantojas no WPF loga klases Window,
   // līdz ar to programma ir logs, kuru var parādīt uz ekrāna
   class Program : Window {

      [STAThread]
      public static void Main() {
         Application programma = new Application();
         programma.Run( new Program() );
      }

      // Program klases konstruktors - izveido virsmu ar taisnstūri
      public Program() {
         // Izveido virsmu
         Canvas virsma = new Canvas();
         virsma.Width = 500;
         virsma.Height = 200;

         // Uzstāda loga virsrakstu un novieto virsmu logā
         this.Title = "WPF animācijas demonstrācija";
         this.SizeToContent = SizeToContent.WidthAndHeight;
         this.Content = virsma;

         // Izveido melnu kvadrātu ar 100 vienību garu malu.
         // Novieto kvadrātu uz loga virsmas punktā (50, 50)
         Rectangle r = new Rectangle();
         r.Height = r.Width = 100;
         r.Fill = new SolidColorBrush( Colors.Black );
         virsma.Children.Add( r );
         Canvas.SetLeft( r, 50 );
         Canvas.SetTop( r, 50 );

         // Izveido lineāras animācijas objektu, kurš animē Double tipa vērtības
         DoubleAnimation kustiba = new DoubleAnimation();
         // Animācija ilgst 2 sekundes
         kustiba.Duration = new Duration( new TimeSpan( 0, 0, 0, 2 ) );
         // Animācija sākas 5 sekundes pēc sākšanas stimula saņemšanas
         kustiba.BeginTime = new TimeSpan( 0, 0, 0, 5 );
         // Animācija maina vērtības no 50 līdz 350
         kustiba.From = 50;
         kustiba.To = 350;
         // Beidzoties animācijai, automātiski apgriezt
         kustiba.AutoReverse = true;

         // Sāk animāciju kvadrātam, animējot kvadrāta atribūtu, kas apraksta
         // kvadrāta  kreisās puses nobīdi no virsmas koordināta sākumpunkta
         r.BeginAnimation( Canvas.LeftProperty, kustiba );
      }
   }
}
```

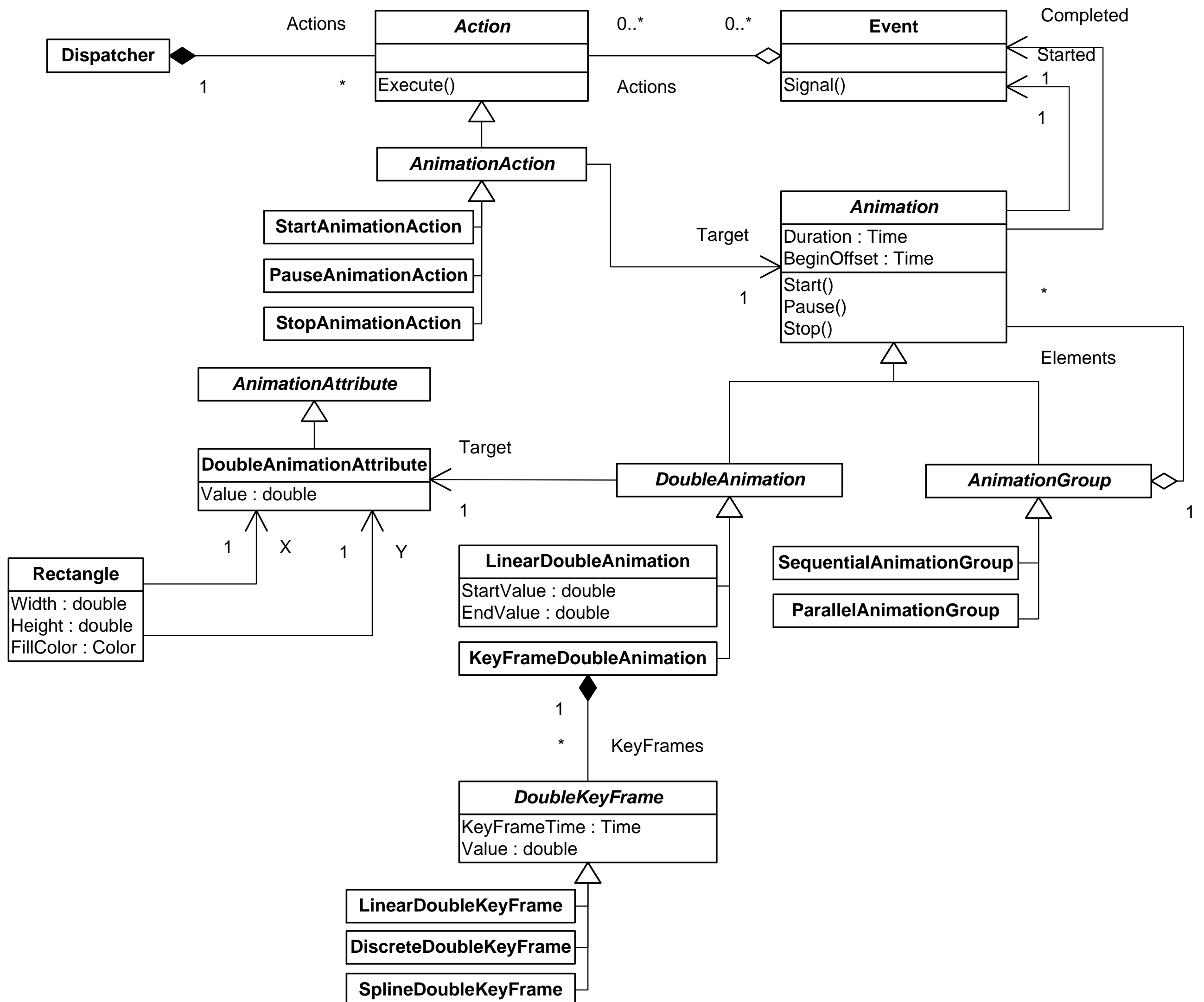
7. Pielikums – WPF EventTrigger demonstrācija

Programma ir *6. Pielikums – WPF DoubleAnimation demonstrācijas programma* programmas modifikācija, kas modificēta, lai programmatiskās animācijas palaišanas vietā, tiktu izmantots trigeru mehānisms. Animācija tiek palaista, kad programmas logs ir ielādēts atmiņa (`Window.LoadedEvent` notikums) un, kad lietotājs noklikšķina uz loga virsma (`Windows.MouseLeftButtonDownEvent` notikums).

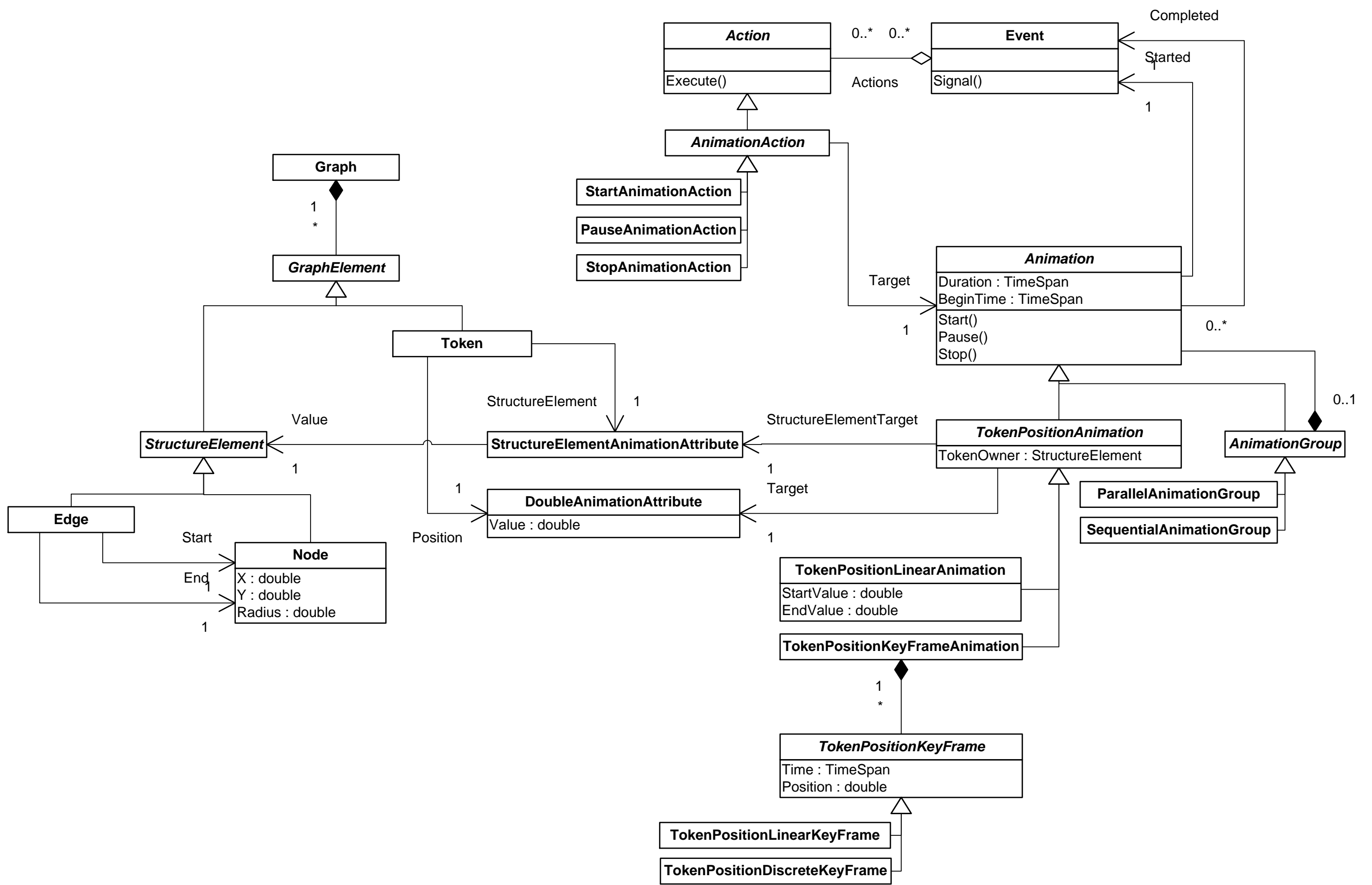
```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Shapes;
using System.Windows.Media.Animation;
using System.Windows.Media;

namespace WPFAnimDemo {
   class Triggers : Window {

      [STAThread]
      public static void Main() {
         Application programma = new Application();
         programma.Run( new Triggers() );
      }

      // Program klases konstruktors - izveido virsmu ar taisnstūri
      public Triggers() {
         // Izveido nosaukumu vārdnīcu. Nepieciešama Storyboard objektam
         NameScope.SetNameScope( this, new NameScope() );

         // Izveido virsmu
         Canvas virsma = new Canvas();
         virsma.Width = 500;
         virsma.Height = 200;

         // Uzstāda loga virsrakstu un novieto virsmu logā
         this.Title = "WPF EventTriger demonstrācija";
         this.SizeToContent = SizeToContent.WidthAndHeight;
         this.Content = virsma;

         // Izveido melnu kvadrātu ar 100 vienību garu malu.
         // Novieto kvadrātu uz loga virsmas punktā (50, 50)
         Rectangle r = new Rectangle();
         r.Name = "Kvadrāts";
         r.Height = r.Width = 100;
         r.Fill = new SolidColorBrush( Colors.Black );
         virsma.Children.Add( r );
         Canvas.SetLeft( r, 50 );
         Canvas.SetTop( r, 50 );

         // Storyboard objektam nepieciešams, lai animējamā objekta nosaukums
         // būtu reģistrēts vārdnīcā
         this.RegisterName( r.Name, r );

         // Izveido lineāras animācijas objektu, kurš animē Double tipa vērtības
         DoubleAnimation kustiba = new DoubleAnimation();
         kustiba.Duration = new Duration( new TimeSpan( 0, 0, 0, 2 ) );
         kustiba.From = 50;
         kustiba.To = 350;
         kustiba.AutoReverse = true;

         // Izveido Storyboard objektu un uzstāda tā mērķa objektu un atribūtu
         Storyboard s = new Storyboard();
         Storyboard.SetTargetName( s, r.Name );
```

```
            Storyboard.SetTargetProperty( s, new PropertyPath( Canvas.LeftProperty ) );
            s.Children.Add( kustiba );

            // Izveido BeginStoryboard darbības klases objektu un konfigurē to, lai
            // startētu iepriekš izveidoto Storyboard objektu
            BeginStoryboard beginAction = new BeginStoryboard();
            beginAction.Storyboard = s;

            // Izveido EventTrigger objektu, kas izpilda iepriekš izveidoto darbību,
            // kad ir inicializēts programmas logs
            EventTrigger trig = new EventTrigger( Window.LoadedEvent );
            trig.Actions.Add( beginAction );
            this.Triggers.Add( trig );

            // Izveido EventTrigger objektu, kas izpilda iepriekš izveidoto darbību,
            // kad lietotājs noklikšķinājis uz loga virsmas
            EventTrigger trigClick = new EventTrigger( Window.MouseLeftButtonDownEvent );
            trigClick.Actions.Add( beginAction );
            this.Triggers.Add( trigClick );
        }
    }
}
```

8. Pielikums – Pilns animācijas metamodeļa būves piemēra metamodelis



9. Pielikums – Animējama orientēta grafa metamodelis



10. Pielikums – Programmatūras CD

Pievienotā CD saturs:

- .\Microsoft .NET Framework – satur instalācijas programmu .NET Framework 3.5.
- .\AE\Pirmkods – satur animācijas dziņa prototipa un dziņa demonstrācijas programmas realizācijas pirmkodu. Šai mapei ir divas apakšmapes:
    - .\AE\Pirmkods\AE0 – satur prototipa animācijas dziņa pirmkodu
    - .\AE\Pirmkods\AEClient – satur animācijas dziņa demonstrācijas programmas pirmkodu.
- .\AE\Bin – satur animācijas dziņa demonstrācijas programmas kompilētus izpildfailus.

Bakalaura darbs

Metamodelis animāciju aprakstīšanai

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

Autors: ______________________________

(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augšminēto bakalaura darbu un atzīstu to par **piemērotu/nepiemērotu** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu bakalaura studiju programmas gala pārbaudījuma komisijas sēdē.

Darba vadītājs(-ja): ______________________________

(Vadītāja paraksts)

Darbs iesniegts Datorikas nodaļā ___________________________________.

(Iesniegšanas datums)

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

Metodiķe: ______________________________.

(Metodiķes paraksts)

Recenzents: ______________________________

(Recenzenta paraksts)

Darbs aizstāvēts bakalaura darbu gala pārbaudījuma komisijas sēdē

__________________________ prot. Nr. ______, vērtējums ______________________

(Darba aizstāvēšanas datums)

Komisijas sekretārs: ________________________________

(Sekretāra paraksts)