SALIKUMA UN MANTOŠANAS MODEĻU SALĪDZINĀJUMS
COMPOSITION AND INHERITANCE MODEL COMPARISON

Autors: **Aleksejs SERGEJEVS,** e-pasts: alexserg@inbox.lv
Zinātniskā darba vadītājs**:** Dr.sc.ing. **Sergejs KODORS**, e-pasts: Sergejs.Kodors@rta.lv

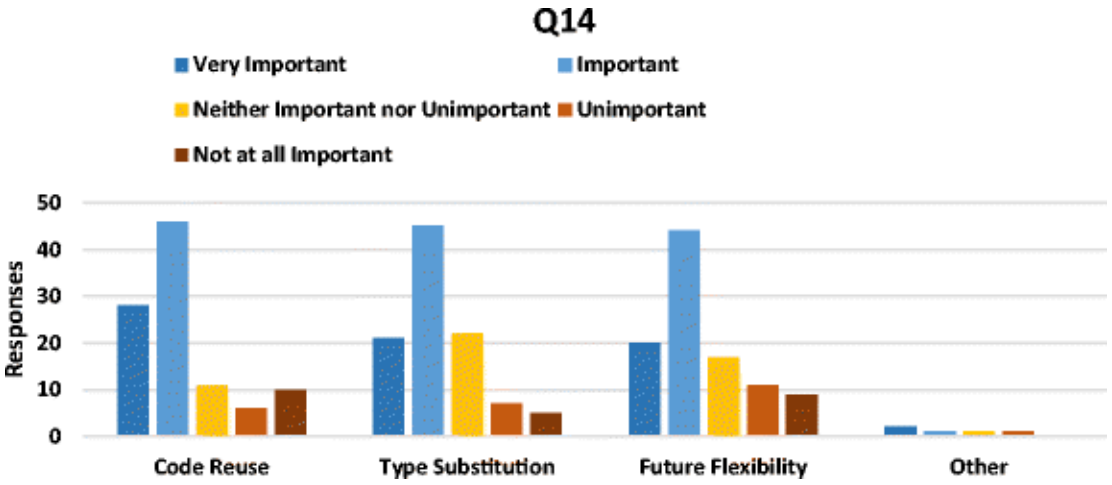
Abstract. *The inheritance seems to be the natural and the default solution of structuring the logic of software nowadays. But is it always the best option? Considering the increasing need for programming and the speed at which the projects are made, it's inevitable that the requirements of a project will be changing many times and a lot of fundamental building blocks in the software will have to be redeveloped. The problem with inheritance is that with a change in functionality it may become necessary to rewrite huge amounts of old code or even end up duplicating existing functionality which only makes things worse in a long run. An excellent solution that can be used to avoid essentially getting stuck in situations like these is composition. The goal of the study is to analyze the pros and cons of composition and inheritance and make a conclusion about their correct usage.*

Keywords: *composition, csharp, inheritance, programming.*

Ievads

Programmēšanas aizsākumos programmatūras kods bieži atkārtojās, bija grūti pārskatāms un nebija paredzēts otrreizējai lietošanai, procedūras un funkcijas tika reti lietotas un procedūru izsaukumi patērēja daudz resursu. Tas nebija pats labākais laiks. Taču vēlāk tika ieviesta objektorientētā programmēšana (*OOP*). Tās popularitāte sāka ievērojami pieaugt, kad to sāka izmantot lietotņu saskarņu izstrādē, jo tur tā patiešām noderēja, jo tā palīdzēja nodalīt elementu loģiku dažādās klasēs. Laikam ejot, *OOP* "mantošanas" modelis kļuva gandrīz vai par standartu risinājumu, jo tā bija pirmā lieta, ko programmētājs parasti izmantoja. Mūsdienās "mantošanas" modelis tiek izmantots neaizdomājoties par potenciālajām problēmām, kas var rasties projekta prasību izmaiņu rezultātā. [1]

Pamatojoties uz *S. Jamie* un *W. Murray* veidotās aptaujas rezultātiem (skatīt 1. attēlu), [2] var spriest, ka viens no svarīgiem aspektiem programmatūrā ir iespēja atkārtoti lietot vienu un to pašu kodu, nevis veidot kaut kādas jaunas kopijas ar modifikācijām.



1. attēls. ***S. Jamie* un *W. Murray* veidotās aptaujas rezultāti (jautājums "Kādus faktorus vai rādītājus jūs uzskatāt par svarīgākajiem pieņemot lēmumu izmantot "mantošanas" modeli"** [2]

Būtībā projektam virzoties uz priekšu bieži rodas neparedzētas prasības no klientiem, kas var veidot nepieciešamību veikt izmaiņas projekta arhitektūrā. Un strādājot ar "mantošanas" modeli, tās bieži vien nav tās labākās, jo tur arī gadās tā problēma, kas tālāk darbā tiks ilustrēta

http://dx.doi.org/10.17770/het2019.23.4394

detalizētāk, ka vai nu programmētājam ir jāveic apjomīgas izmaiņas projekta struktūrā, vai arī tam nākas radīt situāciju, kad kādas no klasēm satur funkcijas, kas nemaz netiek lietotas. Tas arī ir viens no iemesliem, kāpēc šī tēma ir tik aktuāla, jo "salikuma" modelis var risināt šo problēmu.

Materiāli un metodes

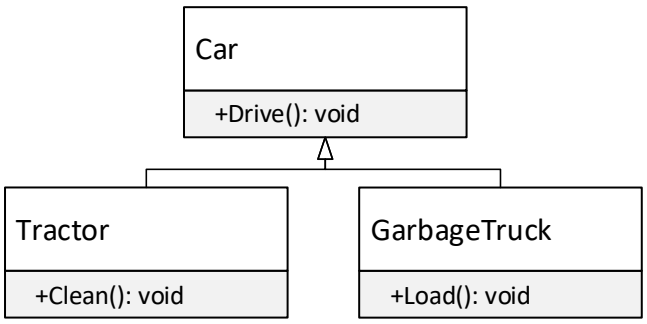
Pētījums tika veikts izmantojot *UML* (*Unified Modeling Language*) diagrammas, kas tika izveidotas izmantojot *Visio* rīku, salīdzināšanas un monografisko metodi, *C#* programmēšanas valodu, un kā arī tika veikta analīze.

Mantošanas modelis

"Mantošanas" modelis ir objektorientētās programmēšanas jēdziens. Tas dod iespēju klasei (virs klasei) mantot funkcijas un vērtības no kādas citas klases (apakšklases). Parasti ir pieņemts, ka virs klasi var nomantot jebkāds skaits apakšklašu, taču virs klase nevar nomantot vairākas apakšklases. Protams, eksistē programmēšanas valodas, kur ir iespēja nomantot vairākas apakšklases, taču šeit tiks vairāk pievērsta uzmanība tam, kā tās funkcionē programmēšanas valodā *C#*. [3]

Būtībā ar "mantošanas" modeļa palīdzību, var iegūt kādas klases saturu, to papildināt un virzīt uz priekšu. Piemēram, varētu būt kāda vispārīga klase dzīvnieks, kurai būtu funkcija elpot, tālāk jau to varētu nomantot klase suns, kas varētu gan elpot, gan vēl papildus riet un tālāk šo klasi varētu nomantot vēl kāda, kas saturētu vēl specifiskāku funkcionalitāti. Šādā veidā var nokļūt no kaut kā ļoti vispārīga uz kaut ko specifisku bez vajadzības atkārtot funkcionalitāti.

Zemāk aplūkojamos attēlos (skatīt 2. un 3. attēlu) tiek demonstrēts "mantošanas" modeļa lietošanas piemērs izmantojot *UML* diagrammu un *C#* programmēšanas valodas kodu. Būtībā šajā situācijā ir klase *Car* (Mašīna), kurai ir funkcija *Drive()*, kas dod tai iespēju braukt. Tālāk ir divas klases, kas manto šo funkciju no mašīnas – *Tractor* (Traktors) un *GarbageTruck* (atkritumu mašīna). Katrai no šīm klasēm ir sava unikāla funkcija – *Clean()* (tīrīt sniegu) un *Load()* (iekraut atkritumus). Rezultātā sanāk, ka abas šīs klases, kas nomanto *Car*, ir mašīnas, bet tajā pat laikā tās arī ir kas cits ar savu funkcionalitāti papildus.



2. attēls. "**Mantošanas" modeļa lietošanas piemērs**

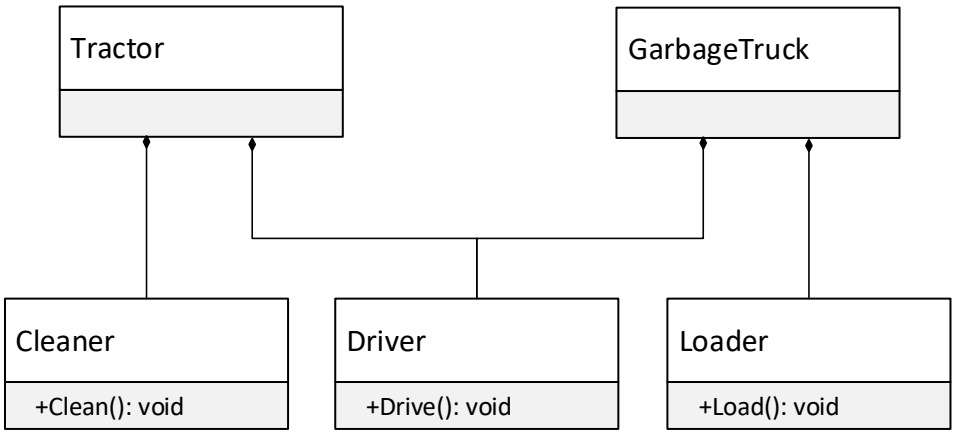
```
class Car
{
   public void Drive()
   {
      // Braukšanas kods
   }
}

class Tractor : Car
{
   public void Clean()
   {
      // Sniega tīrīšanas kods
   }
}

class GarbageTruck : Car
{
   public void Load()
   {
       // Atkritumu iekraušanas kods
   }
}
```

3. attēls. "**Mantošanas" modeļa lietošanas piemērs, izmantojot *C#* programmēšanas valodu**

Salikuma modelis

"Salikuma" modeli raksturo "ir" attiecības starp klasēm. Būtībā tā panāk atkārtotu koda izmantošanu veidojot citu klašu objektus iekš savas veidotās klases. Ja, piemēram, ir klase spuldzīte, kas satur funkciju spīdēt un ir klase lukturis, kam ir nepieciešama šī funkcija, "salikuma" gadījumā, lukturis saturētu mainīgo ar spuldzītes klases objektu. Būtībā spuldzīte būtu luktura sastāvdaļa.

Zemāk aplūkojamos attēlos (skatīt 4. un 5. attēlu) tiek demonstrēts "salikuma" modeļa lietošanas piemērs izmantojot *UML* diagrammu un *C#* programmēšanas valodas kodu. Būtībā, šajā piemērā, ir trīs klases, kas tiek lietotas kā sastāvdaļas klasēm *Tractor* un *GarbageTruck*. Būtībā, *Cleaner*, *Driver* un *Loader* klasi var uzskatīt par rīku, ko var izmantot kāda cita klase. Šādā veidā var veidot dažādas kombinācijas funkcijām. Ja rodas nepieciešamība, var ātri uztaisīt klasi, kas piemēram var darīt visas trīs funkcijas, vai arī var iekraut atkritumus, tīrīt ceļu, bet nevar braukt. Augot klašu hierarhijas izmēriem, šo kombināciju skaits aug tikai eksponenciāli un ļauj veidot aizvien vairāk kombināciju.



4. attēls. "**Salikuma" modeļa lietošanas piemērs**

Zemāk aplūkojamā attēlā (skatīt 5. attēlu), kas demonstrē "salikuma" modeļa piemēru izmantojot *C#* programmēšanas valodu, var novērot to, ka tur tiek lietoti interfeisi (*interface*). Pamatojums šim lietojumam ir tāds, ka rakstot programmas kodu var rasties situācija, kad nav zināms tieši, kāda klase tiek saņemta kādā funkcijā un ir nepieciešams identificēt vai tas ir objekts, kam, piemēram, piemīt spēja braukt. "Salikuma" modeli nav obligāti nepieciešams realizēt šādā veidā, taču šis veids var rasties nepieciešams, ja rodas vajadzība aizstāt "mantošanas" modeli ar "salikuma" modeli.

```
interface IDriver
{
   void Drive();
}

class Driver : IDriver
{
   public void Drive()
   {
      // Braukšanas kods
   }
}

interface ICleaner
{
   void Clean();
}

class Cleaner : ICleaner
{
   public void Clean()
   {
      // Sniega tīrīšanas kods
   }
}

interface ILoader
{
   void Load();
```

```
}


class Loader : ILoader
{
    public void Load()
    {
        // Atkritumu izkraušanas kods
    }
}

class Truck : IDriver, ICleaner
{
    private Driver driver;
    private Cleaner cleaner;

    public void Drive()
    {
        driver.Drive();
    }

    public void Clean()
    {
        cleaner.Clean();
    }
}

class GarbageTruck : IDriver, ILoader
{
    private Driver driver;
    private Loader loader;

    public void Drive()
    {
        driver.Drive();
    }

    public void Load()
    {
        loader.Load();
    }
}
```

5. attēls. **“Salikuma” modeļa lietošanas piemērs izmantojot *C#* programmēšanas valodu**

Rezultāti un to izvērtējums

Pēc “salikuma” un “mantošanas” modeļa analīzes tika konstatēts, ka tiem abiem ir ne tikai priekšrocības, bet arī trūkumi. Tāpēc ir jāņem vērā projekta prasības pirms lemt, cik lielā mērā lietot kādu no tiem.

Lielākais trūkums strādājot ar "salikuma" modeli ir tas, ka to realizējot parasti nākas rakstīt lielāku koda apjomu vienas un tās pašas funkcionalitātes izstrādei. Sanāk, ka tā ir ilgtermiņa investīcija. Programmētājam ir nepieciešams ieguldīt lielāku darba apjomu, kas atmaksājas tikai vēlāk, kad rodas kādas izmaiņas projektā. Veidojot nelielus projektus, prototipus vai arī projektus ar 100% skaidrām prasībām, "salikuma" modelis var patērēt liekus darba resursus un palielināt darba izmaksas.

Savukārt "mantošanas" modelim ir cita problēma. Ja projekts ir liels un rodas daudz neparedzētu prasību no klienta, "mantošanas" modelis var radīt tādu kā strupceļa situāciju, kur ir jālauž kādi no koda principiem, lai implementētu nepieciešamo funkcionalitāti vai arī jāveic nopietnas izmaiņas vecajā kodā, kas var aizņemt pārāk daudz laika, ja projektam ir izpildes termiņš.

Zemāk atrodamais attēls (skatīt 6. attēlu) ilustrē piemēru, kad var rasties šāda veida situācija izmantojot "mantošanas" modeli. Šajā situācijā ir tās pašas iepriekš minētās klases, klase *Animal*, kurai piemīt funkcija *.eat(),* un klase *Dog* un *Cat*, kurai katrai ir sava individuāla funkcija. Problēma šajā situācijā ir tāda, ka ir vēl papildus izveidota *BarkingTractor* klase. Būtībā klients ir izdomājis pieprasīt izveidot mašīnu, kas māk tīrīt sniegu, braukt un vēl arī riet, bet riešanas funkcijas *(.bark()*) ir jau pieejama *Dog* klasē, kas nozīmē, ka ir ticis kopēts koda fragments, kas lauž koda principu *don't repeat yourself* (neatkārto sevi). To nav vēlams darīt, jo var potenciāli rasties situācija, kad nāksies mainīt *.bark()* funkcijas saturu. Ja tai eksistē kāda kopija, tad to būs jāveic divas reizes. Bet tā nav vienīgā problēma. Šis fakts var tikt palaists garām un var tikt mainīta tikai viena no *.bark()* funkcijām un otra tikt aizmirsta, kas var potenciāli radīt vēl nopietnākas problēmas. Alternatīvi, lai neatkārtotu *.bark()* funkciju, varētu ielikt kopīgu klasi, ko nomantotu gan *Car*, gan *Animal*, kas saturētu *.bark()* funkciju. Problēma šajā situācijā ir tāda, ka tajā gadījumā būs kaudze ar dažādām klasēm, kam piemitīs iespēja izmantot *.bark()* funkciju, lai arī tā nav nemaz tām nepieciešama.

```
Car
   .drive()

   Tractor
      .clean()

      BarkingTractor
         .bark()

   GarbageTruck
      .load()

Animal
   .eat()

   Dog
      .bark()

   Cat
      .meow()
```

6. attēls. "**Mantošanas" modeļa pseidokoda piemērs ar tā saucamo strupceļa situāciju**

Savukārt, ja šis piemērs būtu veidots ar "salikuma" modeli, tad šādas problēmas nebūtu, jo šīs funkcijas būtu nodalītas pa klasēm un tās būtu iespējams kombinēt jebkādā veidā. Būtībā šādas situācijas ir viens no galvenajiem iemesliem, kāpēc ir vēlams biežāk izmantot "salikuma" modeli nevis "mantošanas" modeli.

Zemāk atrodamajās tabulās (skatīt 1. un 2. tabulu) var aplūkot "mantošanas" un "salikuma" modeļa priekšrocības un trūkumus.

1. tabula

"Mantošanas" modeļa priekšrocības un trūkumi

Priekšrocība	Trūkums
Atkārtots koda lietojums – viss kods, ko satur virs klase, var tikt atkārtoti lietots apakšklasē.	**Lieka funkcionalitāte** – mantojot klasi, pat, ja ir nepieciešama tikai viena vienīga funkcija no tās, tiks mantots pilnīgi viss, visas funkcijas un visi mainīgie.
Polimorfisms – virs klases tipa mainīgais var glabāt atsauci ne tikai uz virs klasēm, bet arī uz jebkuru tā apakšklasi.	**Cieši saistītas klases** – virs klase ir ļoti cieši saistīta ar tās apakš klasēm. Ja tiek veiktas izmaiņas virs klasē, tās var potenciāli sabojāt apakšklases. [5]
Apakšklases izveidošanas vienkāršība – ja ir nepieciešams veidot jaunu apakš klasi, kas mantos virs klasi, tā ne kādā veidā neiespaido virs klasi vai kādu citu apakš klasi, kas arī to manto.	**Iespējami metožu un mainīgo nosaukumu konflikti** – augot klašu hierarhijai, rodas palielināta iespēja konfliktam metožu un mainīgo nosaukumos.

2. tabula

"Salikuma" modeļa priekšrocības un trūkumi

Priekšrocība	Trūkums
Necieši saistītas klases – pati par sevi klase, kas tiek lietota neietekmē klasi, kas to izmanto, tāpēc ir mazākas izredzes lauzt kādu funkcionalitāti programmatūrā veicot izmaiņas.	**Lielāks programmas koda apjoms** – lai realizētu to pašu loģiku, "salikuma" modelis parasti aizņem ievērojami vairāk koda rindu nekā "mantošanas" modelis. Šādā veidā arī var rasties lielākas grūtības pievienot jaunu klasi, jo pa virsu vēl ir jāsaraksta funkcijas priekš klases, ko ir nepieciešams izmantot atkārtotai koda lietošanai.
Atkārtots koda lietojums – viss kods var tikt atkārtoti lietots tāpat kā "mantošanas" modelī, taču visa funkcionalitāte ir manuāli jālieto.	**Zemāka ātrdarbība** – bieži rodas situācijas, kad ir jātaisa pa virsu vēl funkcija, kas izsauc funkciju no klases, kas tiek dabūta ar "salikuma" modeļa palīdzību. Šādā veidā tiek patērēti papildus resursi.
Funkcionalitātes kombinēšanas vienkāršība – "salikuma" gadījumā ir mazāk ierobežojumu saistībā ar dažādu klašu funkcionalitātes kombinēšanu, kas palīdz izvairīties no problēmas, kas tika minēta iepriekš (skatīt 6. attēlu).	**Grūtāk lasāms kods** – ja salīdzināt tīru "mantošanas" un "salikuma" modeļa kodu, tad "salikuma" gadījumā, tas var būt grūtāk lasāms, jo daudzas darbības tiek veiktas izpildlaikā, kods ir apjomīgāks un tiek veidots lielāks skaits klašu. [4]
Mazāk mainīgo un metožu nosaukumu konfliktu – tā kā klase netiek mantota, mazinās iespēja sastapties ar situāciju, kad ir kādi konflikti saistībā ar metožu un mainīgo nosaukumiem. [4]	

Secinājumi

1. Viena no situācijām, kad "mantošanas" modelis teorētiski var būt efektīvs risinājums ir, ja ir garantija, ka prasības netiks mainītas un projekta struktūra ir rūpīgi izplānota.
2. "Salikuma" modeļa realizācija izmantojot *C#* programmēšanas valodu ir ievērojami garāka nekā izmantojot "mantošanas" modeli, kas arī varētu būt viens no iemesliem, kāpēc tā netiek lietota tik bieži, cik "mantošanas" modelis.
3. Gandrīz jebkuru projektu, kas veidots izmantojot "mantošanas" modeli, var pārtaisīt, lai tas darbotos izmantojot "salikuma" modeli.
4. Vairumā gadījumu "mantošanas" modelis netiek realizēts pēc plāna, jo vienmēr rodas neparedzētas prasības vai nepieciešamība pēc izmaiņām.
5. "Mantošanas" un "salikuma" modeļi nav savstarpēji izslēdzoši. Projektā var lietot gan vienu, gan otru – atkarībā no situācijas.
6. "Salikuma" modelis palīdz sadalīt programmatūras loģiku ļoti smalki, ja to pareizi lietot. Šādā veidā var izvairīties no klasēm, kas dara pārāk daudz un ir grūti pārskatāmas.
7. Ar "salikuma" modeļa palīdzību var ieviest funkcionalitāti, kas ir līdzīga vairāku klašu mantošanai valodās, kur tā nav pieejama.
8. Programmēšanas valodā *C#* "salikuma" modelim ir iespējams iegūt to pašu polimorfisma priekšrocību, kas piemīt "mantošanas" modelim, izmantojot interfeisus.

Summary

Given the rapid development situation nowadays, it's important to have software, which is easily extendable and reconstructed. The use of inheritance is often misused and traps programmers into situations where they're forced to break some of the code principles in order to avoid rewriting extremely huge chunks of code. Composition is a way of avoiding this problem because it allows to combine functionality as if it consisted of some kind of components or tools. Using this solution, it is easier to introduce new functionality without copying code or rewriting huge chunks of the old code. There're however some problems with composition that might be one of the reasons why it isn't as used as it should be. It can substantially increase the code size and that requires additional work in advance which only pays off in a long run often times long after the initial developers of the project are gone. It should be noted that even though composition should be used more, inheritance still has its use, and it can actually be used together with composition.

Literatūra

1. L. Steven. *Composition vs. Inheritance: How to Choose?* Sk. Internetā (19.04.2019) https://www.thoughtworks.com/es/insights/blog/composition-vs-inheritance-how-choose
2. Stevenson J., Wood M. *Recognising object-oriented software design quality: a practitioner-based questionnaire survey*. Sk. Internetā (19.04.2019) https://link.springer.com/article/10.1007/s11219-017-9364-8
3. Rupak. *Inheritance vs composition*. Sk. Internetā (20.04.2019) https://www.techjini.com/blog/inheritance-vs-composition/
4. Dev.Interview. *What are advantages of composition and aggregation over inheritance?* Sk. Internetā (21.04.2019) http://developer-interview.com/p/oop-ood/what-are-advantages-of-composition-and-aggregation-over-inheritance-14
5. T. Jignesh. *Inheritance VS Composition*. Sk. Internetā (21.04.2019) https://www.c-sharpcorner.com/UploadFile/ff2f08/inheritance-vs-composition/