STEP IT ACADEMY

{PROQRAMLAŞDIRMA}

LAYİHƏLƏNDİRMƏ PATTERNLƏRİ

LAYİHƏLƏNDİRMƏNİ DOĞURAN PATTERNLƏR

STRUKTUR PATTERNLƏR

DAVRANIŞ PATTERNLƏRİ

Dərs №2 Struktur patternlər

Mündəricat

1. Struktur pattern anlayışı

İnformasiya sistemlərinin tərifinə əsaslansaq aydın olur ki, hər bir proqram həlli, istər istəməz məlumatların məcmusuyla əməliyyat edir, bunların analizində biz, informasiya sisteminin özünü idarə edərkən bizə bisnes və ştat qərarlarını həyata keçirməyə imkan verən bəzi informasiyalar alırıq.

Məlumatların analizi nəzərdə tutur ki, onlar bir tərəfdən müxtəlif məlumatları identifikasiya etməyə (məntiqi olaraq onları bir-birindən fərqləndirmək), digər tərəfdən strukturlaşdırılmış obyektlərə və onların məcmusuna atomar kəmiyyətləri təşkil etməyə imkan yaradan bəzi strukturlarda təşkil edilməlidir. Məlumatların strukturlaşdırılması, analiz zamanı əməliyyat etdiyimiz anlayışların əks olunması, həmçinin obyektlərin yaranma prosesi zamanı onlar arasındakı münasibətlərin təsvir edilməsi üçün lazımdır.

Obyekt-yönümlü yanaşma bizi strukturlaşdırılmış anlayışların təsviri və onlar arasındakı münasibətlərin əks olunması mənasında zəngin instrumentari ilə təmin edir. Lakin, anlayışları işlək sistemə bağlayan strukturun təşkili, informasiya sisteminin effektli tətbiqi nöqteyi nəzərindən, həmçinin məshul tərtibatının prosesi zamanı alınan müşaiyət nöqteyi nəzərindən həlledici əhəmiyyətə malikdir.

Layihələndirmə patternləri ümumi mənada zamanla və tətbiq təcrübəsi ilə yoxlamadan keçən təsviri problemlərin ən yaxşı həllinin modellərini əks edir. Struktur patternləri, təsviri həllərimizdə istifadə etdiyimiz məlumat idarəsinin təşkili məsələsini ən yaxşı yolla həll edən, məlumatların strukturlara təşkil modellərini təklif edir. Anlamaq gərəklidir ki, məlumat kimi tək atomar kəmiyyətləri yox, həmçinin bizim tətbiqimizin çərçivəsində mövcud olan bütün obyektləri hesab etmək olar. Ardınca proqram təminatının tərtibatında beynəlxalq cəmiyyətdə ən yaxşı təcrübə, qəbul olunan əsas struktur patternlərinin sadalanması olacaq.

2. Pattern - Adapter

2.1.Patternin məqsədi

Təqdim olunan materialın daha yaxşı başa düşülməsi üçün, həmçinin, izahın asanlaşdırılması məqsədiylə, nəzərdən keçirilən suala spesifik olan növbəti terminologiyanı tətbiq edəcəyik:

(client) müştəri adı altında biz (adaptee) adlandırdığımız, bəzi class-ları istifadə edən bəzi class-ları başa düşəcəyik. (adapter) adı altında, müştərinin gözlədiyi interfeysə, adaptasiya edilən class interfeysinin gətirilməsini icra edən class-ı başa düşəcəyik.

Adapter layihələndirmə patterninin məqsədi, müştəriin gözlədiyi interfeysə, bəzi adaptasiya edilən class-ın interfeysini gətirməkdir (adaptasiya etmək).

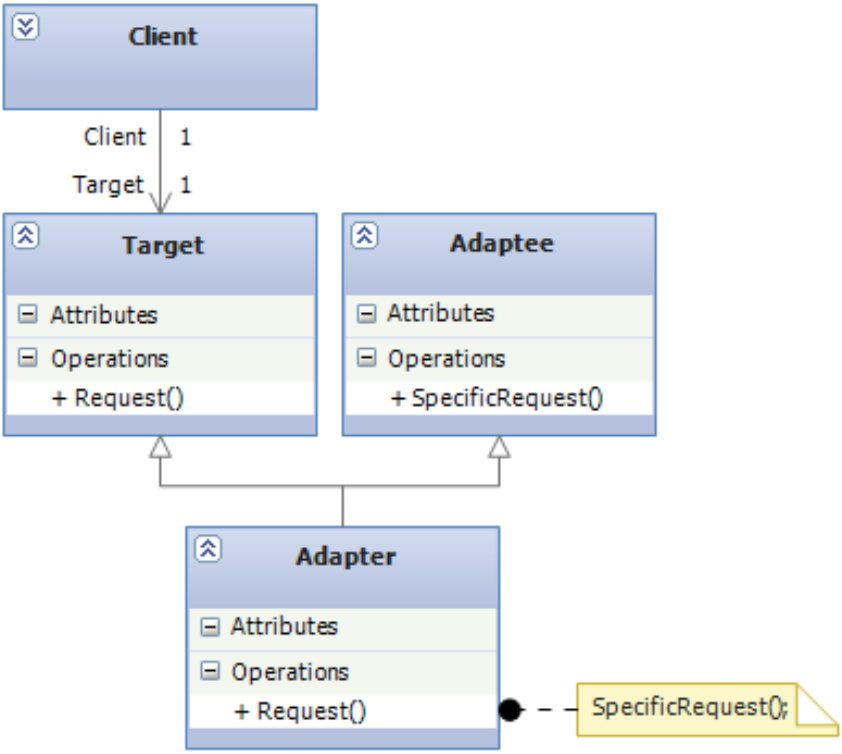
2.2 Patternin yaranma səbəbi

Çox tez-tez belə bir problemə rast gəlinir: Bizim alət dəstimizdə onun strukturuna qeyri-spesifik olan məsələdə istifadə etmək istədiyimiz hər hansı bir class var. Misal üçün, bizdə təsviri analiz məqsədiylə istifadə etdiyimiz (məsələn, paketlərin yerdəyişmə trassirovkası) IP – ünvan, mak-ünvan, hostun adı kimi xüsusiyyətlərlə təchiz edilmiş, şəbəkə qurğusunun anlayışını təsvir edən və IpendPoint adı verdiyimiz məlumat tipi elan edilib. Analiz, IPEndPoint tipli bir çox obyektləri aqreqasiya edən və NetView adlanan hər hansı bir class-la yerinə yetirilir. Lakin biz, proses və analizin nəticəsinin qrafik təsvirini, bizim əlavənin əsas pəncərəsinə çıxardaraq realizə etmək istəyirik. Problem bundadır ki, NetView class-ı qrafik alt-sistem obyektinə spesifik olan interfeysə malik deyil, və uyğun olaraq təsvir edilmə üçün pəncərə class-ı ilə istifadə oluna bilməz. Biz NetView class-ını tətbiqin tələblərinə uyğun "yenidən yazmaq" (ilkin mətni və uyğun olaraq strukturu dəyişmək) imkanına malik deyilik, belə ki o, kənar tərtibatçıların təqdim etdiyi dll-kitabxanasından istifadə etdiyimiz tip dəstinin bir hissəsidir, ya da, biz class-ın strukturunu "yenidən yükləmək" istəmirik, belə ki, o bizim tətbiqin hansısa bir məsələləri üçün kritikdir.

Bu halda, hər hansı bir varis class-ı istifadə etmək daha məntiqlidir, bu varisliyi istifadə edərək, NetView class-ını lazım olan məlumat tiplərinə gətirməyə imkan verəcək, həmçinin qrafik altsisteminin komponentinə spesifik olan metodların təkrar təyini vasitəsilə, NetView məlumat tiplərinin təsvir interfeysini təyin edəcək.

Bu cür vasitəçi-class-ı adətən adapter adlandırırlar.

2.3.Patternin strukturu



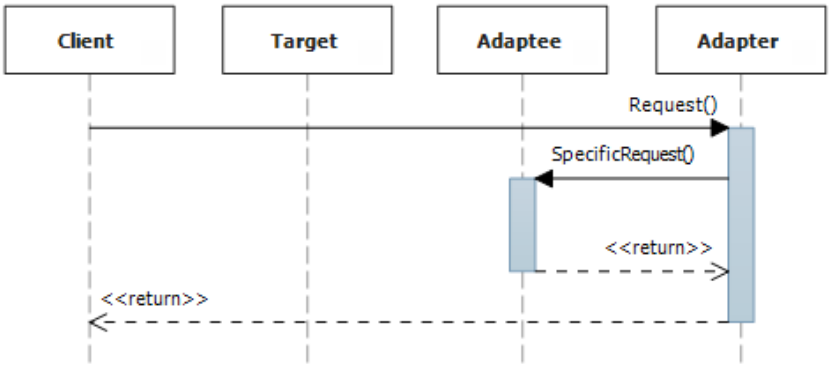
Adapter patterninin strukturu yuxarıda göstərilən diaqramda təsvir edilmişdir.

İştirak edən elementlər:

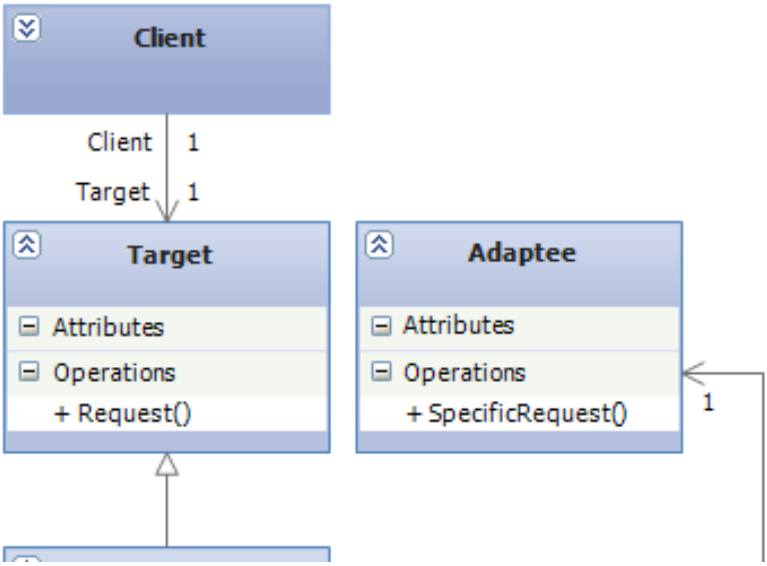
- *Client* – bəzi köməkçi məlumat tiplərini istifadə edən və onların Target class-ı ilə təsvir olunan standart qarşılıqlı fəaliyyət (istifadə) interfeysinin olduğunu gözləyən class.
- *Target* – müştərinin gözlədiyi interfeysə malik clsass-dır.
- *Adaptee* – müştərinin işinə lazım olan class-dır, lakin, müştərinin gözlədiyi interfeysdən fərqli interfeysə malikdir.
- *Adapter* – Adaptee class interfeysinin Target class interfeysinə gətirilməsini yerinə yetirən class.

Yuxarıda təklif olunan strukturda, interfeysin gətirilməsi, Adapter class-ı hər iki Adaptee və Target class-ının varisi olması və deməli hər iki clsass-ın interfeysinə malik olması hesabına olur. Daha sonra, Adapter class-ı Target class-ının interfeysinə spesifik olan metodların çağırılmasını, Adaptee class-ının müvafiq interfeys metodlarının çağırılmasına gətirib çıxarır.

Aşağıda göstərilən ardıcıllıq diaqramında göstərilir ki, Adapter class-ı obyektinin metodlarının çağırılması, onun baza class-ının obyektinin metodlarının çağırılmasına gətirib çıxarır.

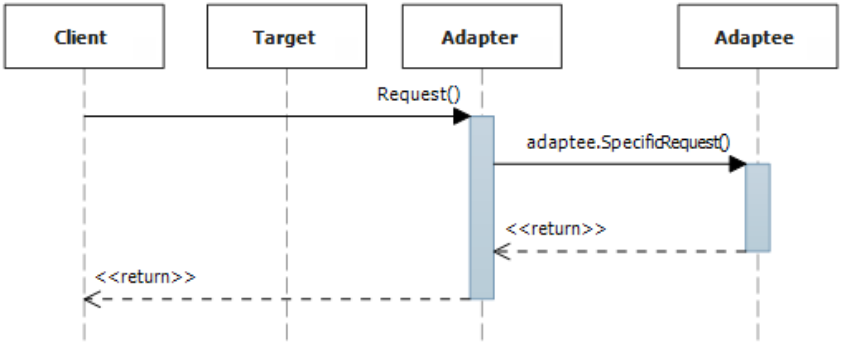


Adapter patterni də həmçinin alternativ üsulla realizə edilə bilər, aşağıdakı diaqramda onun strukturu təsvir edilib.



Əvvəlki strukturla fərq ondadır ki, cari modeldə Adapter və Adaptee class-ları qohum münasibətlərində yox, assosiasiya münasibətlərindədirlər, yəni Adapter, Adaptee class-ını aqreqasiya edir.

Beləliklə, Adaptee class-ı interfeysinin Target class-ı interfeysinə gətirilməsi onun hesabına olur ki, Target class-ı interfeysinə spesifik olan Adapter class-ı obyekt metodlarının çağırılması, Adapter class-ında inkapsulyasiya edilmiş Adaptee class-ının müvafiq obyekt metodlarının çağırılmasına gətirilir. Aşağıda göstərilən ardıcıllıq diaqramı baş verənləri illustrasiya edir.



2.3.Patternin istifadə nəticələri

Adapter layihələndirmə patterninin istifadəsinin əsas müsbət nəticəsi ondadır ki, biz, hər hansı bir class-ın interfeysini tətbiq tərəfindən gözlənilən class-ın öz strukturunu dəyişmədən interfeysə çevik gətirmə imkanı əldə edirik. Bu həm tiplərin strukturunun artıqlığının aradan qaldırılması nöqteyi nəzərindən, həm də yaradılan tətbiqlərin modulluq nöqteyi nəzərindən zəruridir.

Artılıq, bizim yazdığımız kodu təkrar istifadə etmək (məsələn digər tətbiqdə) lazım gəldiyində mənfi rol oynayır, bu cür kod - reusable kod adlanır. Reusable kodun yaradılması gözəl praktika hesab edilir, belə ki, tərtibatın sürətini artırır və qiymətini azaldır. Həmçinin yaradılan tətbiqlərin miqyaslılığını və çevikliyini hazır tətbiqin modul strukturu hesabına artırır.

Tətbiqləri yeni tipləri əlavə edərək genişləndirmək daha asandır, nəinki hər hansı bir modulları tam yenidən hazırlamaq.

2.4.Patterndən istifadənin praktiki nümunəsi

Biz Adapter patterninin istifadəsini hər hansı bir müəssisənin mal dövriyyəsinin idarəsini yerinə yetirən tətbiq nümunəsində nəzərdən keçirək. Mallar üzrə idarəetməni həyata keçirmək üçün biz, ProductsCollection class-ının köməyi ilə mal kolleksiyasında təşkil edilən məlumat tiplərini Product (məhsul, mal) kimi təsvir edəcəyik.

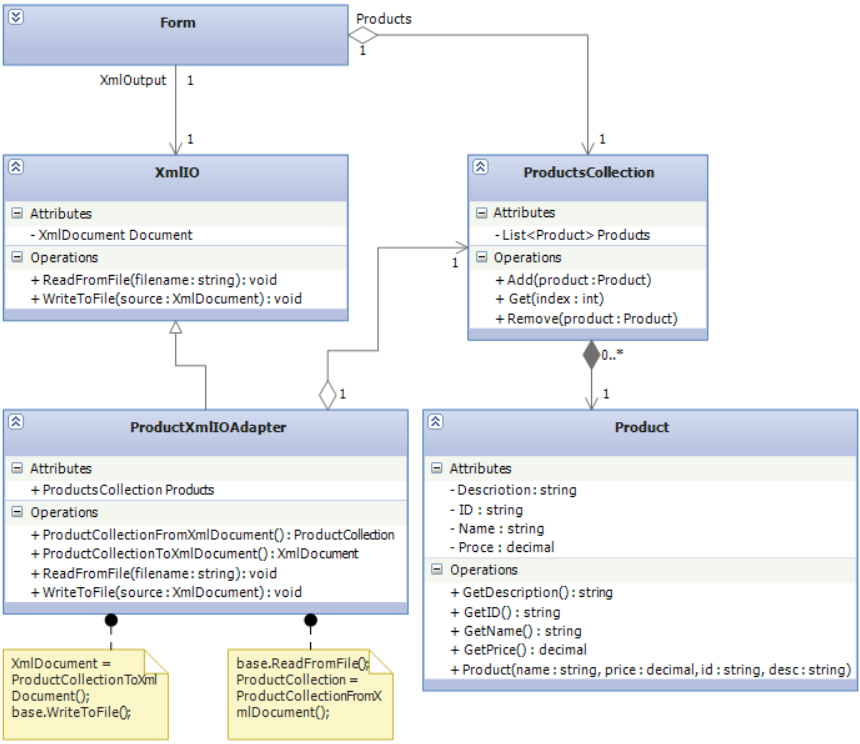
Bizim tətbiqimiz həmçinin ticarət müəssisəsinin işinin digər aspektlərini də yerinə yetirir, biz onları nümunənin şəffaflığı üçün "alçaldacayıq".

Praktiki olaraq bütün məsələ həllərində istifadə olunan bilən məlumatların aparıla bilən rezerv nüsxəsinin yaradılma imkanını realizə etmək üçün, biz belə qərara gəldik ki, bütün məlumatları ayrıca xml-fayllara eksport etmək imkanı olsun. Buna görə biz Xml-sənədin yazılma və oxunmasını həyata keçirən və obyekti bizim tətbiqin

pəncərə class-ı ilə inkapsulyasiya edən XmlIO class-ını yaratdıq. Yazının məhsul kolleksiyası fayllarına realizəsi üçün, ProductXmlIOAdapter class-ını yaratdıq, hansı ki, məhsul kolleksiyasını inkapsulyasiya edir, bu kolleksiyanın xml – sənədə gətirilməsini həçminin xml - sənədin kolleksiyaya gətirilməsini realizə edir.

Beləcə, xml-formatında olan fayla, kolleksiya yazısı funksiyasını realizə edirik, eyni vaxtda ilkin məlumat tiplərinin strukturunu dəyişmirik və Product class-ının əlavə funksionalla doldurulma zamanı yarana bilən artıqlığı aradan qaldırırıq. Artıqlığın aradan qaldırılması zəruridir, çünki Product məlumatlar tipi, növbəti analizin yerinə yetirilməsi və s. üçün bəzi məlumat bazalarından alınan məhsullar barəsində informasiyanın obyekt təsvirinə gətirilməsində istifadə oluna bilər. Bütün bu əməllərin yerinə yetirilməsi zamanı tip strukturunun artıqlığı əlavə çətinliklər yarada bilər. Həmçinin modullu struktur, layihənin müşaiyəti zamanı və resusable-kodunun istifadəsi zamanı çeviklik əlavə edir.

Yuxarıda təsvir olunan tətbiqi illustrasiya edən aşağıdakı diaqram göstərlimişdir.


Form
Products
1
XmlOutput
1
1
1
XmlIO
Attributes
- XmlDocument Document
Operations
+ ReadFromFile(filename: string): void
+ WriteToFile(source : XmlDocument) : void
ProductsCollection
Attributes
- List<Product> Products
Operations
+ Add(product : Product)
+ Get(index : int)
+ Remove(product : Product)
1
0..*
1
1
ProductXmlIOAdapter
Attributes
+ ProductsCollection Products
Operations
+ ProductCollectionFromXmlDocument() : ProductCollection
+ ProductCollectionToXmlDocument() : XmlDocument
+ ReadFromFile(filename: string): void
+ WriteToFile(source : XmlDocument) : void
Product
Attributes
- Descriotion : string
- ID : string
- Name : string
- Proce : decimal
Operations
+ GetDescription() : string
+ GetID() : string
+ GetName() : string
+ GetPrice() : decimal
+ Product(name : string, price : decimal, id : string, desc : string)
XmlDocument = ProductCollectionToXmlDocument();
base.WriteToFile();
base.ReadFromFile();
ProductCollection = ProductCollectionFromXmlDocument();

3. Pattern Bridge

3.1. Patternin məqsədi

Bridge (ingiliscə "körpü") patterninin məqsədi abstraksiyanı və onun realizəsini bir-birindən asılı olmayaraq dəyişilməsi üçün ayırmaqdır.

3.2. Patternin yaranma səbəbləri

Adətən, hər hansı bir abstraksiyanın (abstrakt class) bir neçə konkret realizəyə malik olduğu hallarda, oxşar interfeysə (ümumi hallarda buna eyni və ya uyğun gələn deyilir) malik bir çox class-ların müəyyənləşdirilməsi üçün varislikdən istifadə edirlər. Abstrakt class öz xələfləri üçün "müxtəlif" üsullarla realizə edən interfeysi müəyyən edir.

Lakin, bü cür yanaşma hər zaman kifayət qədər çevik olmur və kodun çoxluğuna gətirib çıxara bilən, həmçinin layihənin müşaiyətində onun qiymətini əsaslı dərəcədə dəyişən, əlavə çətinliklər yarada bilən bəzi zəif tərəfləri vardır. Hər hansı bir konkret class tərəfindən abstraksiyanın interfeysinin birbaşa varisliyi abstraksiya ilə realizəni birbaşa bağlayır, bu modifikasiyanın növbəti realizəsində (onun genişləndirilməsində) çətinliklər yaradır, həmçinin abstraksiya və onun realizəsinin bir-birindən ayrı, təkrarən istifadəsinə icazə vermir. Realizə demək olar ki, abstraksiya ilə "sərt bağlı" olur.

Körpü layihələndrimə patterni interfeysin yerləşdrilməsi və onun müxtəlif ierarxiyalarda realizəsini nəzərdə tutur, bu interfeysi realizədən ayırmağa və bir-birindən ayrı

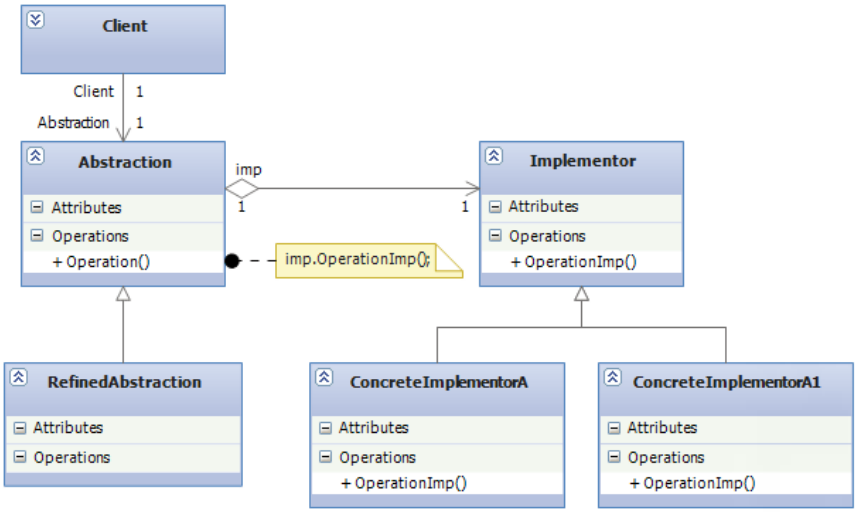
müstəqil istifadə etməyə, həmçinin istənilən realizə variantlarını abstraksiyanın müxtəlif dəqiqləşdirilmiş variantları ilə kombinasiya etməyə imkan verir.

3.3. Patternin strukturu

Körpü layihələndirmə patterni aşağıdakı struktur elementləri ilə təqdim olunub:

- Abstraction (Abstraksiya) – abstraksiyanın interfeysini müəyyənləşdirir, həmçinin realizənin interfeysini müəyyən edən icraçı obyekti vardır.
- Implementor (icraçı) – realizə class-ları üçün interfeysi müəyyən edir. İcraçının interfeysinin, abstraksiyanın interfeysinə uyğun gəlməyi mütləq deyildir. Prinsipcə, abstraksiyayla və icraçıyla müəyyən edilmiş interfeyslər tam fərqli ola bilər, bu da kifayət qədər çox çevikdir. İcraçı tam olaraq, nəticədə abstraksiyanın yüksəkmərhələli məntiqinə əsaslanan baza əməliyyatlarını müəyyən etməlidir.
- RefinedAbstraction (dəqiqləşdirilmiş abstraksiya) – abstraksiya ilə müəyyən edilmiş abstraksiyanı genişləndirir.
- ConcreteImplementor (konkretləşdirilmiş icraçı) – icraçının interfeysini realizə edən və onun xüsusi realizəsini müəyyən edən class.

Abstraksiya və icraçı birgə olaraq dəqiqləşdirilmiş abstraksiyanı konkret realizə ilə bağlayan "körpü" yaradır. Körpü layihələndirmə patterninin strukturu aşağıda class diaqramı kimi təsvir edilmişdir.



3.4. Patternin istifadə nəticələri

Körpü layihələndirmə patterninin istifadəsinin əsas üstünlüyü bundadır ki, abstraksiyanın onun realizəsindən məntiqi və struktur ayrılması həyata keçirilir, bu da kodu daha çevik edir. Həmçinin körpü patterninin tətbiqi, kodun genişləndirilmə kimi keyfiyyətini yaxşılaşdırır, belə ki, abstraksiya və icraçı müxtəlif ierarxik strukturlarda yerləşirlər, deməli realizənin genişləndirilməsi abstraksiyadan müstəqil şəkildə və tərsinə mümkündür.

Körpü layihələndirmə patterninin istifadəsinin xeyrinə daha bir səbəb ola bilər ki, o realizənin detallarını müştəridən (client) gizlətməyə imkan verir, yəni abstraksiyanı istifadə edən tətbiqdən, bu da müştəriyə bu və ya

digər halda məhz hansı realizənin seçilməsindən müstəqil olmağa imkan verir.

Patterndən prkatiki istifadə nümunəsi.

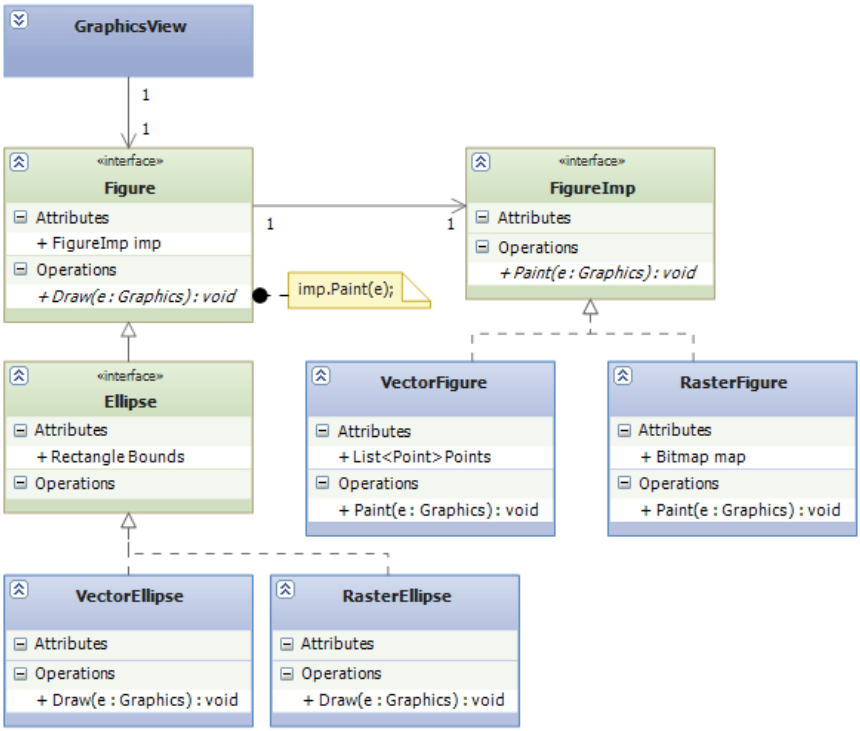
Biz körpü layihələndirmə patterninin tətbiqinə "qarışıq" qrafik redaktor (bir təsvir çərçivəsində həm rastr həm də vektor qrafikasını redaktə etməyə imkan verən redaktor) nümunəsində baxış keçirək.

Əlavə modeli hər hansı bir abstrakt fiquralarla operasiya edən hər hansı bir qrafik təsvirin varlığını, bu nümunədə abstraksiya rolunu oynayan Figure interfeysilə təsvir olunan qarşılıqlı əlaqə interfeysini nəzərdə tutur.

İcraçının interfeysi buna uyğun olaraq, vektor və raster fiqurlarının təsvir realizəsini təsvir edən VectorFigure və RasterFigure class-larının varisliyini ondan aldğımız FigureImp interfeysi ilə müəyyən olunur.

Realizənin təyin olunmasından sonra hər hansı bir konkret fiqurun (məsələn ellips) interfeysini təsvir edərək abstraksiyanı dəqiqləşdirə bilərik. Bundan sonra dəqiqləşdirilmiş abstraksiyanı konkret realizə ilə, məsələn, uyğun olaraq vektor və rastr ellipslərini təsvir edən VectorEllipse və RasterEllipse class-larını təyin edərək əlaqələndirmək olar.

Yuxarıda göstərilmiş model aşağıda təsvir olunmuşdur.


GraphicsView
1
1
«interface»
Figure
Attributes
+ FigureImp imp
Operations
+ Draw(e : Graphics) : void
1
1
«interface»
FigureImp
Attributes
Operations
+ Paint(e : Graphics) : void
imp.Paint(e);
«interface»
Ellipse
Attributes
+ Rectangle Bounds
Operations
VectorFigure
Attributes
+ List<Point> Points
Operations
+ Paint(e : Graphics) : void
RasterFigure
Attributes
+ Bitmap map
Operations
+ Paint(e : Graphics) : void
VectorEllipse
Attributes
Operations
+ Draw(e : Graphics) : void
RasterEllipse
Attributes
Operations
+ Draw(e : Graphics) : void

3. Pattern Comosite

4.1. Patternin məqsədi

Pattern - Composite (düzüb-qoşan) obyektləri hissə-tam ierarxik əlaqədə ağac strukturu formasında təsvir etmək üçün nəzərdə tutulub.

4.2. Patternin yaranma səbəbləri

Kompozisiya edən pattern ona görə mövcuddur ki, "ağac" tipli struktur kifayət qədər genis yayılmışdır və müntəzəm struktura malik məlumatların təşkili üçün tez-tez istifadə olunur. Yəni, hissə-tam ierarxik bağlılığı, hər hansı bir məlumatların kompozisiya elementi tək elementar element yox, həm də belə bir kompozisiyanın da ola bildiyini təxmin edən bir strukturu realizə edir.

Bu cür daxiletmə rekursiv sistemli strukturun ən sadə realizəsi, tərkibində class, elementar komponentlər və elementar komponentlərin konteyneri kimi istifadə olunacaq class-lar olan hər hansı bir tip sistemində ifadə oluna bilər.

Lakin, bu cür yanaşmanın belə bir ciddi qüsuru vardır ki, göstərilmiş class-ları istifadə edən kod, elemenlər və onların konteynerlərini ayrıca emal etməlidir, baxmayaraq ki, çox hallarda onlar eyni cür emal edilir. Və bu, tətbiqin realizəsini kəskin çətinləşdirir.

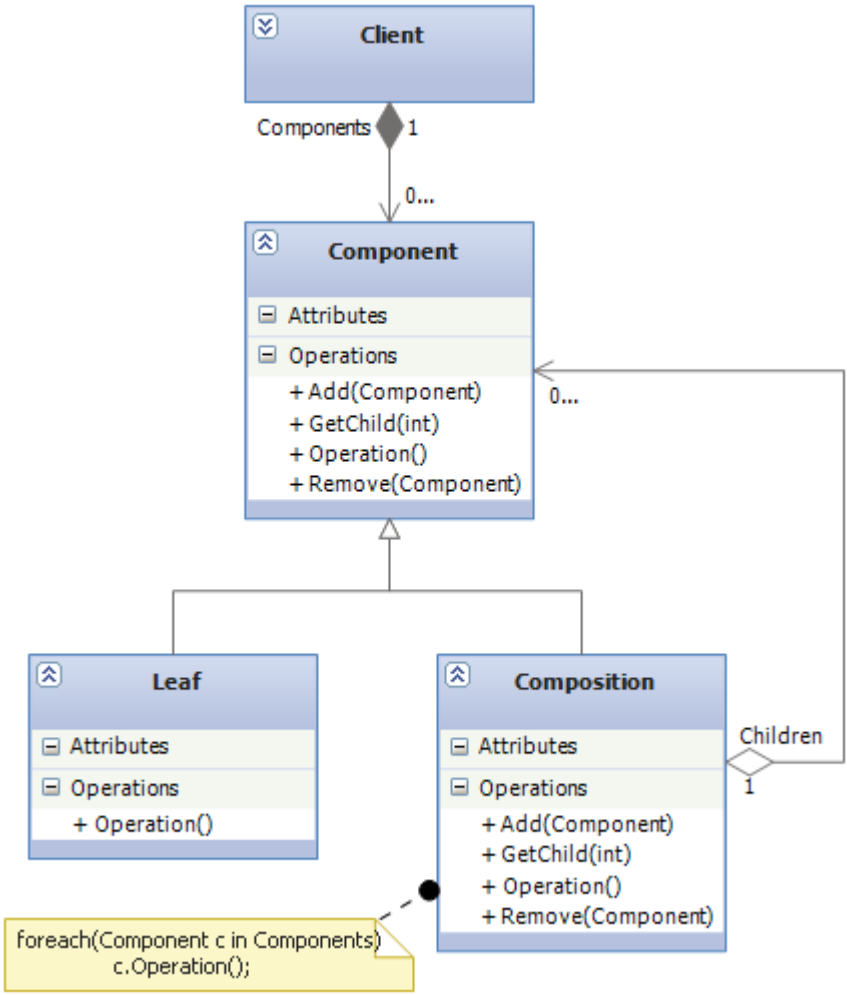
Kompozisiya edən pattern, məlumatların ümumi

məcmusunun ayrıca elementlərinin necə emal edilməsi barədə fikrin qəbul edilməsi lazım gəlməyən rekursiv strukturu təklif edir. Kompozisiya edən patternin təbiətini anlamaq üçun açar, elementar komponent və onların komponentlərinin abstrakt identik interfeysin təyin edilməsindən ibarətdir. Belə ki, konteynerlər və elementar komponentlər qohum əlaqələrindədirlər və ümumi istifadəçi interfeysinə malikdirlər, onda konteyner rahat müntəzəm (rekursiv) struktur yaradan başqa bu cür konteynerin elementi ola bilər.

4.3. Patternin Strukturu

- Component (komponent) – obyekt və onların kompozisiyaları üçün interfeysi təsvir edir; ayrıca komponent və onların kompozisiyaları üçün spesifik olan baza davranışlarını realizə edir; kompozisiya elemntlərinə keçidin interfeysini, həmçinin, valideyn rekursiv struktur elementinə keçid interfeysini təyin edir.
- Leaf (yarpaq) – kompozisiyanın ayrıca elementini təyin edir və ümumi struktur "primitiv" (baza) elementlərinin davranışlarını təsvir edir.
- Composition (kompozisiya) – tərkibində törəmə elementləri olan komponentlərinin davranışını təyin edir, həmçinin komponentin interfeysiylə təyin olunmuş törəmə elementləri və onlara keçid əməliyyatlarını realizə edir.
- Client (müştəri) – kompozisiya elementlərini komponent interfeysi vasitəsi ilə idarə edir.

Kompozisiya edən patterninin strukturu aşağıdakı class diaqramı şəklində təsvir edilmişdir.



4.4.Patterndən istifadənin nəticələri

Kompozitçi patterninin istifadəsi, elementlərin məlumatların daxil edilmiş strukturu şəklində təşkil olunmasını sadələşdirməyə kömək edir və bu məsələnin realizə olunması zamanı kodun artıqlığını aradan qaldırır. Həmçinin, kompozitçi müştəri obyektini daha sadə edir və ona elementar komponentləri və onların kompozisiyalarını onların komponent – class-ı ilə təyin edilmiş unifikasiya edilmiş interfeysinin gücünə eyni cür emal etməyə imkan yaradır.

Kompozitçi yeni komponentlərin əlavə edilməsini asanlaşdırır. Yeni class-lar, elementar komponent və ya kompozisiya olduqlarından asılı olmayaraq onların yaranma zamanı mövcud olan strukturla işləyəcəklər. Və digər sözlərlə, yeni komponent yaradarkən müştərini dəyişməyə zəruriyyət yoxdur.

4.5. Patterndən istifadənin praktiki nümunəsi

Biz kompozitçi layihələndirmə patterninin tətbiqinə istifadəçinin müntəzəm struktura malik çox saylı qrafik interfeys elementinin yaradılması bazasında baxacayıq. Başqa sözlə, hər hansı bir pəncərə idarəetmə pəncərəsini, həmçinin digər pəncərələri özündə saxlaya bilər. Pəncərələri sadə yolla idarəetmə elementlərinin kompozisiyaları kimi təyin etmək olar.

Kompozitçini bizim təyin etdiyimiz məsələdə təşkil etmək üçün biz, istifadəçinin hər hansı bir qrafik inter-

feys elementini və kompozisiyanın törəmə elementlərinin idarəetmə interfeysini təsvir edən UIElement abstrakt class-ını elan edirik.

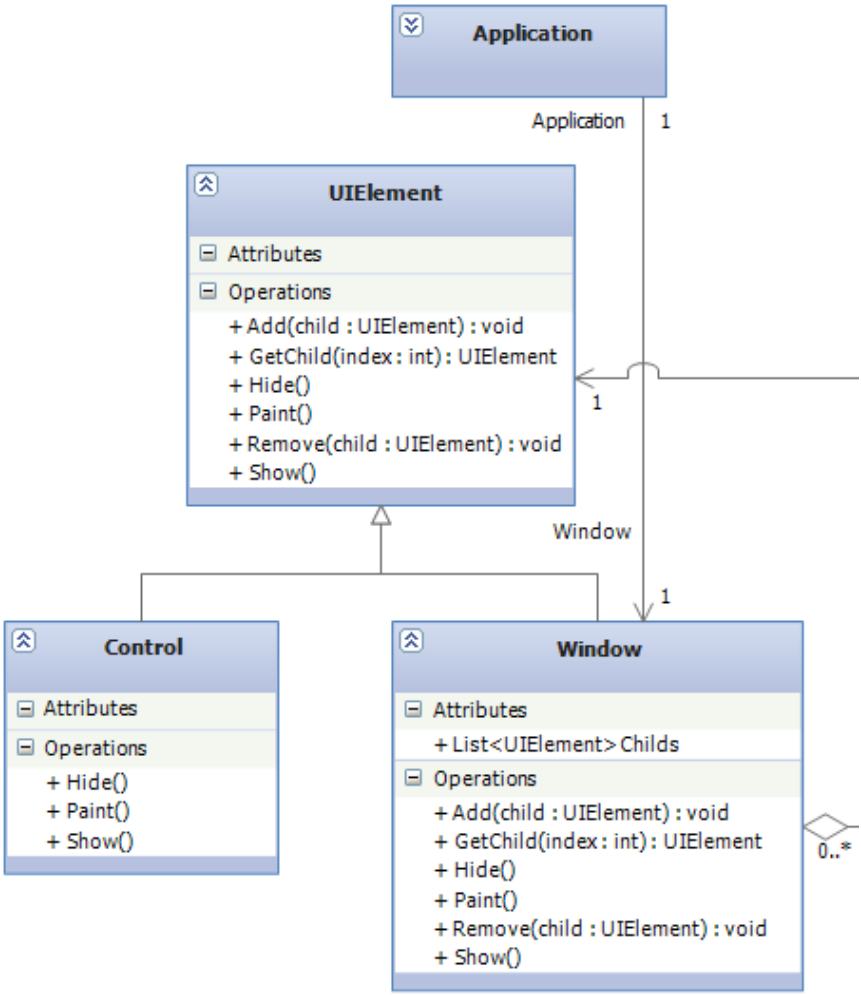
Qrafik interfeys elementinin baza davranışına Hide, Paint və Show metodları aiddir. Bu metodlar uyğun olaraq elementi göstərmək, onun qrafik təsvirini çəkmək və gizlətmək imkanlarını realizə edir.

Kopmozisiyanın törəmə elementlərinin idarəsi-keçidi üçün Add, GetChild və Remove metodları təyin edilir. Bu metodlar uyğun olaraq yeni törəmə elementini əlavə etmək, əldə etmək və kompozisiyadan silməyə imkan yaradır.

Control class-ı, bizim realizə etdiyimiz kompozitçinin "primitiv" elementi rolunu oynayan və baza davranışını realizə edən idarəetmə elementini təsvir edir. Window class-ı idarəetmə elementlərinin kompozisiyasıdır, həmçinin müəyyən bir qrafik təsvirə malik olduğu üçün baza davranışını da realizə edir.

Application class-ı hər hansı bir pəncərə ilə assosiasiya edən müştəri rolunu oynayır, yəni Window class-ından olan obyektdən və UIElement class-ı ilə təsvir olan interfeysdən istifadə edir.

Təsvir olunan tətbiqin modeli class diaqramlarında təsvir olunan aşağıdakı şəkildə göstərilir.


Application
Application
1
UIElement
Attributes
Operations
+ Add(child : UIElement) : void
+ GetChild(index : int) : UIElement
+ Hide()
+ Paint()
+ Remove(child : UIElement) : void
+ Show()
1
Window
1
Control
Attributes
Operations
+ Hide()
+ Paint()
+ Show()
Window
Attributes
+ List<UIElement> Childs
Operations
+ Add(child : UIElement) : void
+ GetChild(index : int) : UIElement
+ Hide()
+ Paint()
+ Remove(child : UIElement) : void
+ Show()
0..*

5. Decorator Patterni

5.1. Patternin məqsədi

Decorator (ing. dekorator) patterninin məqsədi obyektə dinamik funksionalın əlavə edilmə imkanını realizə etməkdir, həmçinin ayrıca class-lar arasında ayrıca funksiyaların icra məsuliyyətini bölüşdürməkdir.

Həmçinin dekorator özünü, obyektlərin funksionalının genişləndirmə mənasında varisliyə alternativ kimi təqdim edir.

5.2. Patternin yaranma səbəbləri

Kifayət qədər yayılan yanaşmaya görə bu patternin yaranma səbəbi ayrıca class-lar arasında ayrıca əməliyyatların icra məsuliyyətinin bölüşdürülməsidir. Bu pattern hər bir class-ın idarəetmə məntiqinin inkapsulyasiyanı onun üzərinə qoyulmuş vəzifələrlə icra edən struktur yaradır. Burada məna kəsb edən tək tərtibatı sadələşdirən modulluq deyil, həmçinin məna etibarı ilə idarə etməməli olduğu əməliyyatların keçidini class-lara yasaqlamaqdır.

Vəzifələri ayrıca class-lar arasında bəzi ümumi məsələlərin həlli üçün bölüşdürməyə imkan yaradan metodlardan biri də varislikdir. Lakin bu cür yanaşma çevik deyildir, belə ki, varislik zamanı əlavə olunan funksiya statik olaraq bu class-ın bütün nəsillərində bərkidilir. Və müxtəlif tipləri (müxtəlif funksioanllıq dəsti olan) yarat-

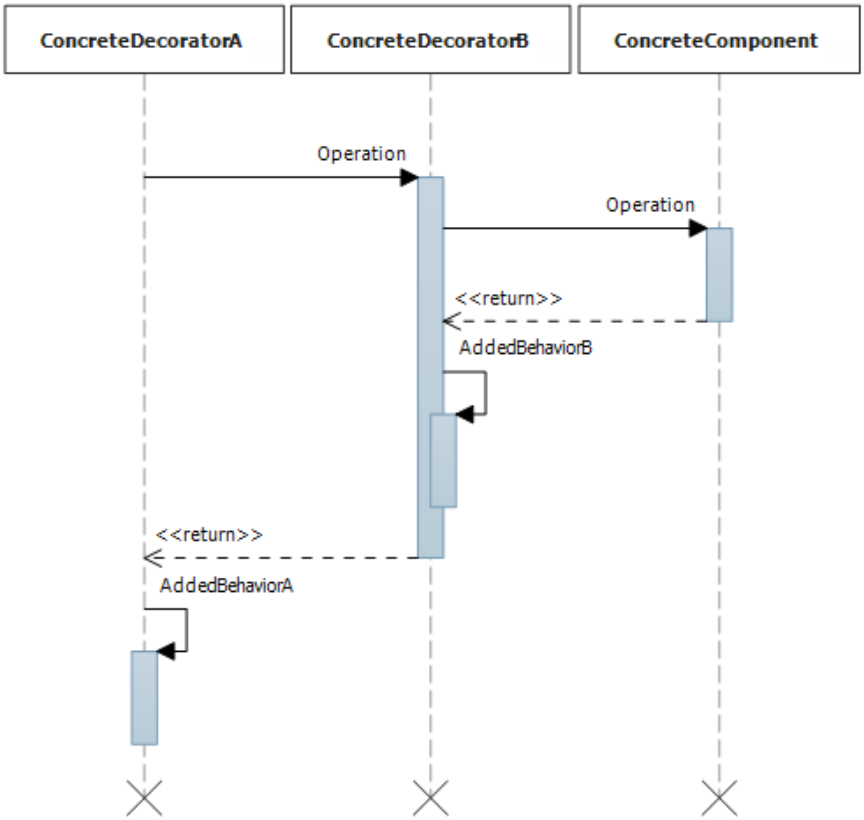
maq üçün varisliyi “bütün hər bir mümkün” kombinasiyalarda icra etmək lazımdır.

Dekorator həmçinin, nəinki obyektin funksionallığını daxili məntiq ilə çevik kombinasiya etmək, həm də icra zamanı obyektin funksiya dəstini dinamik dəyişmək imkanı verir, belə ki, funksiyanın əlavə edilməsi lazım olan class-ın komponent obyektinin yaradılmasına gətirilir.

5.3. Patternin strukturu

Decorator patterninin strukturu aşağıdakı elementlərlə təsvir edilmişdir:

- Component – hər hansı bir elementin “dekorasiyaların” (müəyyən şəklə salma) təyin olunması üçün onun abstraksiyasını ifadə edir. Əlbəttə ki, “dekorasiya” altında müəyyən vizual şəklə salma yox, spesifik funksiyaların əlavə edilməsi başa düşülür. Component – in abstrakt əməliyyatın konkret realizəyə elanı var, dekarator da bu realizəyə “yeni” funksionallıq əlavə edəcək.
- ConcreteComponent – komponentin realizəni təmsil edir.
- Decorator – komponentin varisi və onu aqreqsiya edən class. O realizəni yenidən elə müəyyən edir ki, komponentdə inkapsulyasiya olunmuş funksiyanı icra etsin, sonra yeni funksionallıq əlavə etsin.

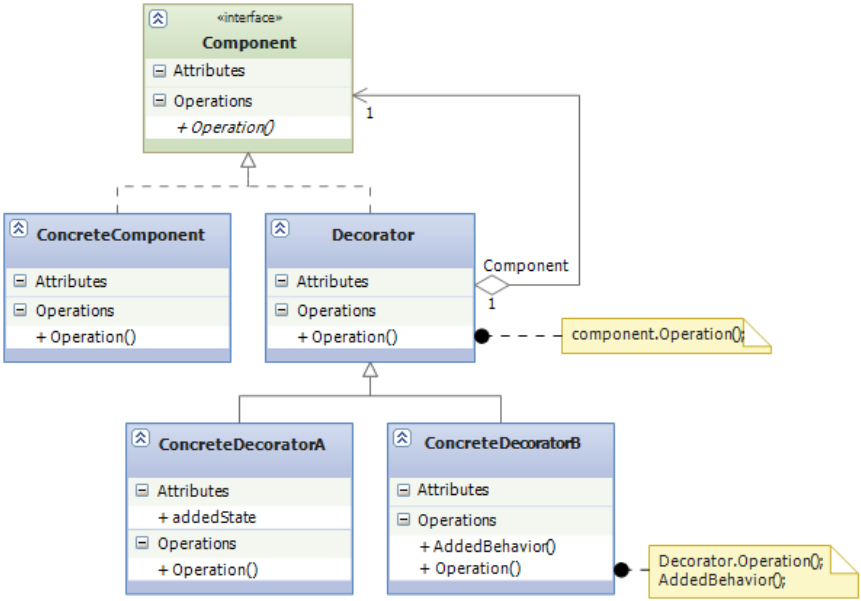


Belə ki dekorator, komponentin xüsusi halıdır (Componentin varisidir),

onda inkapsulyasiya edilmiş komponent kimi başqa dekorator istifadə edilə bilər, bu da yenidən təyin olunmuş əməliyyatın çağırılmasının rekursiv ardıcıllığını, funksionalın tədricən "toplanması" ilə icra etməyə imkan verir (yuxarıdakı diaqramda göstərildiyi kimi).

ConcreteDecoratorA və ConcreteDecoratorB — komponent üçün dekoratorun xüsusi realizəsidir.

Dekorator (Decorator) layihələndirmə patterninin strukturu aşağıda göstərilmiş class diaqramları şəklində illustrasiya olunur.



5.4. Patterndən istifadənin nəticələri

Yuxarıda deyildiyi kimi, patternin istifadəsi varislikdən fərqli olaraq, müxtəlif class-lar arasında bəzi mürəkkəb məsələlərin icrasındakı vəzifələrin bölüşdürülməsini realizədən daha çevik etməyə imkan verir.

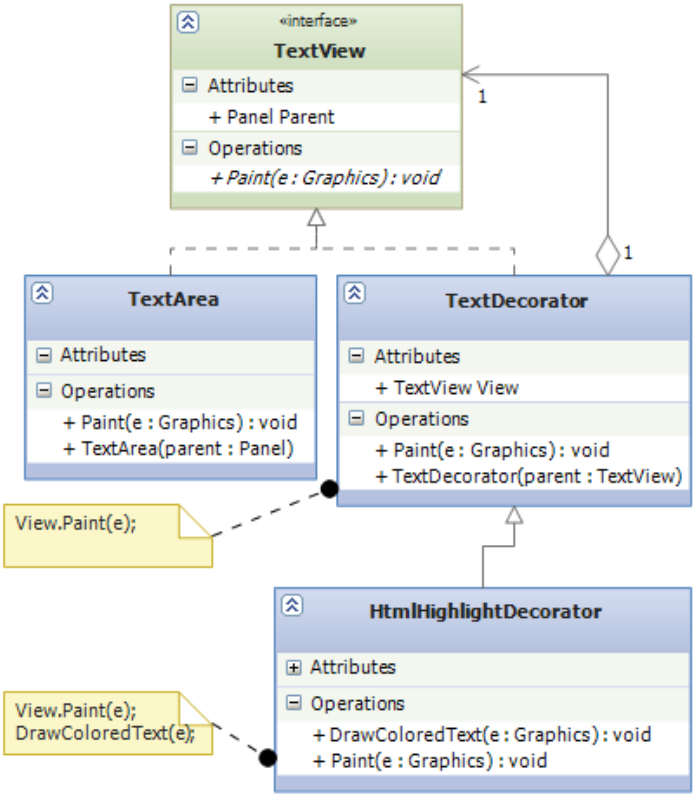
Digər tərəfdən dekorator layihələndirmə patterni class ierarxiyalarının həddən çox doldurulmasının qabağını alır,

belə ki, bütün kombinasiyalarda özündə funksionalı olan class-ların yaradılmasının qarşısını almağa imkan verir.

Patternin istifadəsindən praktiki nümunə.

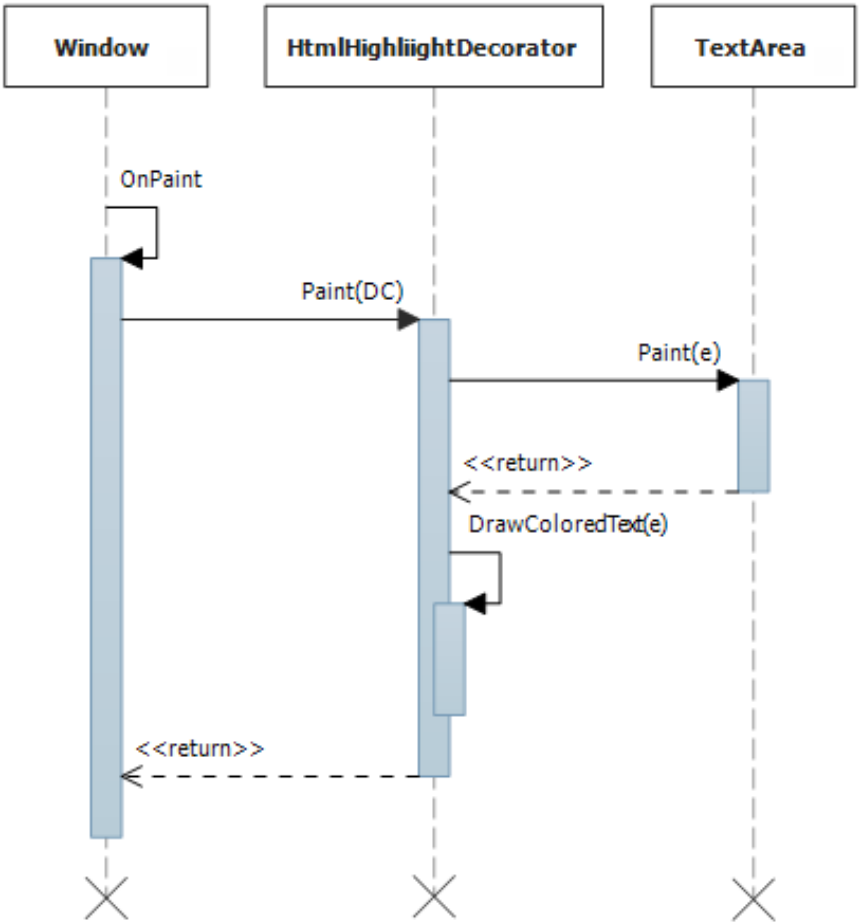
Nümunə kimi html-sintaksisin işıqlandırılması ilə mətn redaktorunun modelinə baxaq.

İstifadəçiyə mətn informasiyasının təsvir məntiqinin realizəsi üçün mətn təsvirinin class-ını istifadə etmək ehtimal edilir. Mətn təsvirinin modeli aşağıda təsvir edilmişdir.



(TextView) mətn təsvirinin class-ı, təsvirin konkret realizə class-ı ilə redaktəni dəstəkləyən mətn sahəsi kimi (TextArea class-ı), bir tərəfdən varisliyə malik olur. Mətn təsvirinin funksionalının genişləndirilməsi imkanının dəstəklənməsi üçün biz TextDecorator class-ını elan edirik, o da öz növbəsində Decorator patternini realizə edir. Html-sintaksisin işıqlandırılma imkanının realizəsi üçün biz konkret TextDecorator class-ının realizəsini yaradırıq, hansı ki, html-sintaksinin işıqlandırılmasının icrası məntiqini mətn təsvirində inkapsulyasiya edir.

Mətn təsvirinin istifadə olunduğu pəncərə təsvirinin yerinə yetirilməsində (hadisənin emalçısı Paint) müştərinin əlavə sahəsinin olçülərinə uyğun təsvir yaradılır və qrafik kontent inisializasiya edilir.Yaradılmış təsvirin qrafik kontekst obyekti, HtmlHighlightDecorator class obyekti Paint metoduna ötürülür, o da öz növbəsində TextArea class obyekti olan Paint metodunu çağırır, sonda isə işiqlandırılmış mətnin təsvirinə cavab verən DrawColoredText metodunu da çağırır. Çağırışların ardıcıllığı aşağıdakı ardıcıllıq diaqramında illustrasiya olunmuşdur.



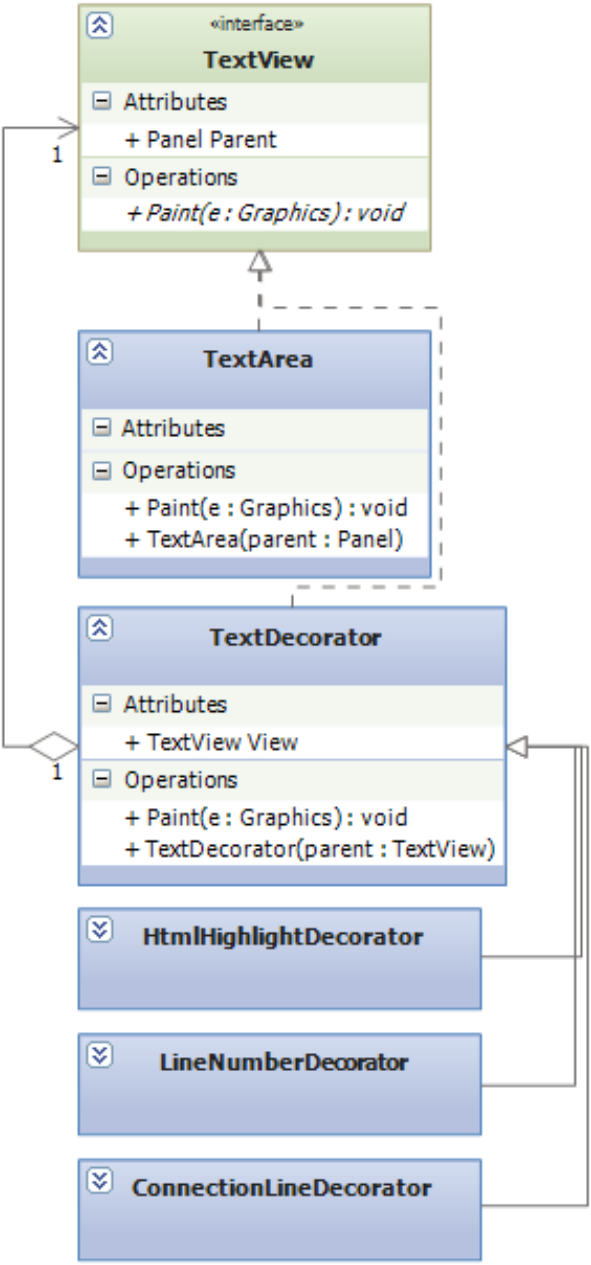
Beləcə, pənəcərə class-ı Paint, hadisəni emal edənə idarənin qayıtmasından sonra, emalçıda yaradılmış təsvirdə, işiqlandırılmış sintaksisli mətn təsvir olunacaq. Sonuncu icra ilə biz mətn təsvirinin əlaqədə olduğu alınmış təsviri pəncərənin qrafik kontekstində təsvir edirik.

```
4       <component name=
        "Microsoft-Windows-International-Core-WinPE"
        processorArchitecture="x86" publicKeyToken=
        "31bf3856ad364e35" language="neutral" versionScope=
        "nonSxS" xmlns:wcm=
        "http://schemas.microsoft.com/WMIConfig/2002/State"
        xmlns:xsi=
        "http://www.w3.org/2001/XMLSchema-instance">
5           <SetupUILanguage>
6             <UILanguage>ru-RU</UILanguage>
7           </SetupUILanguage>
8           <InputLocale>en-US; ru-RU</InputLocale>
9           <SystemLocale>ru-RU</SystemLocale>
```

Bu cür realizə sonda bizə mətn redaktorlarına xas müxtəlif dekorasiya edən effektləri çevik əlavə etməyə və silməyə imkan verəcək. Məsələn: qoşa element xəttlərini birləşdirən sətirlərin nömrələri, yuxarıda göstərilən şəkildəki kimi.

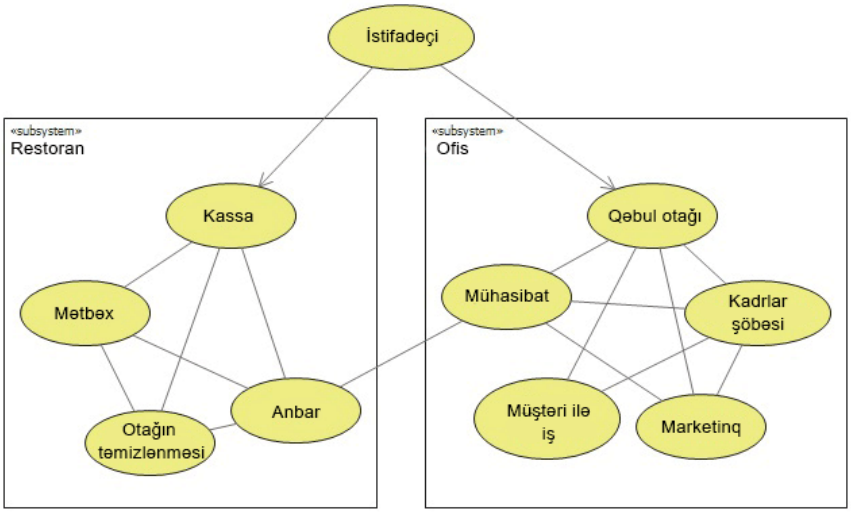
Bunun üçün bir neçə dekoratorları əlavə etmək və onları mətn təsviri ilə əlaqələndirilən pəncərə class-ında onların mətnli təsviri üçün əlavə etmək kifayətdir.

Ümumi model aşağıda göstərilib.


«interface»
TextView
Attributes
+ Panel Parent
Operations
+ Paint(e : Graphics) : void
1
TextArea
Attributes
Operations
+ Paint(e : Graphics) : void
+ TextArea(parent : Panel)
TextDecorator
Attributes
+ TextView View
1
Operations
+ Paint(e : Graphics) : void
+ TextDecorator(parent : TextView)
HtmlHighlightDecorator
LineNumberDecorator
ConnectionLineDecorator

6. Pattern Facade

Pattern Facade (fasad) daha çox mündəricatın inkapsulyasiyası (gizlədilmə) və məntiqi hissələrin müstəqil alt sistemlərə bölünməsi üçündür. Fasad patterninin bir çox yerdə tətbiq olunan "real həyatda" ən yaxşı nümunəsi kimi fast food restoranlarıdır, burada siz sadəcə kassaya yaxınlaşıb, yemək sifariş edirsiniz, restoranın sonrakı strukturu bir qayda olaraq sizi maraqlandırmır. Lakin restoranda həmçinin ofis var və siz gəlib "dar" interfeysdə müraciət edə bilərsiniz (misal üçün daimi müştəri olmaq). Beləcə, restoran iki bir-birindən asılı olan alt sistemlər əldə edir və bunlar ikisi bir yerdə bir böyük sistemə yığılır.



6.1. Patternin məqsədi

Fasad patterninin məqsədi altsistemlərə ayırd edilmiş class strukturizasiyasıdır və hər birinə keçid bir interfeysdən baş verir. Həmçinin tətbiq arxitekturasının qurulması zamanı bir-birindən maksimal müstəqil olan ayrıca alt sistemlərə bölmək məqsədə uyğundur.

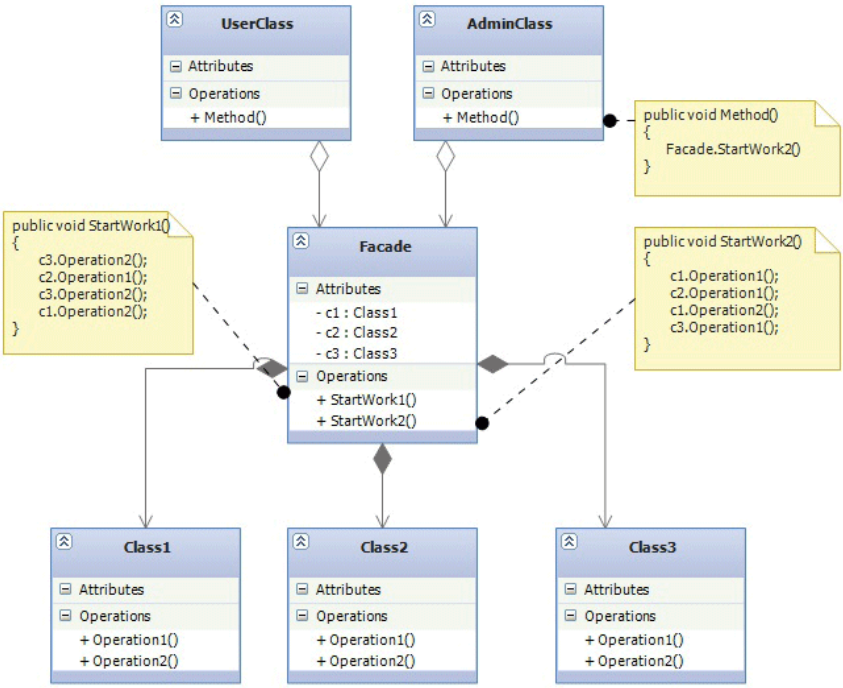
Proqramın (altsistemin) hər hissəsinin öz fasad class-ı olacaq və buradan proqramın bu hissəsinin idarəsi icra ediləcək.

6.2. Patternin yaranma səbəbləri

"Birdən" layihələndirib, sonra çox iri tətbiqi qurmaq çox çətindir, arxitektorlar bu məsələni proqramın daha kiçik altsistemlərə bölünməsi ilə həll etdilər və bu, proqramın hansısa bir hissəsini onun digər aspektlərinə diqqətini yayındırmadan qurmağa, qurduqdan sonra isə növbəti altsistemin yazılışına keçməyə imkan verir.Sistemləri bir-birindən müstəqil şəkildə yazsaq, onun komponentlərinin "təhlükəsiz" əvəzedilməsini əldə etmək olar, həmçinin müstəqil alt sistem əldə edərək siz onu təkrar olaraq digər layihələrdə də istifadə edə bilərsiniz və sonda əgər proqramçıların komandasında bir nəfərdən çox adam varsa bu onlardan hər birinə (yaxud bir neçəsinə) ayrıca altsistemin yazılmasını və sonra onların hamısının bir tam əlavəyə yığılmasına imkan verəcək.

Lakin, sizə tanış olmayan altsistemdən istifadə etmək üçün bu struktur barədə çoxlu sənədləşməni oxumaq və ondan necə istifadə etməyi öyrənmək lazım gəlirdi.

Bu problemin həlli üçün Fasad (Facade) patterni yaradılmışdı. O hər bir altsistem üçün öz class-ının yaradılmasını nəzərdə tutur və o da özlüyündə altsistemi saxlayacaq. Bu cür class-dan strukturla qarşılıqlı əlaqədə olmaq daha asan olacaq.

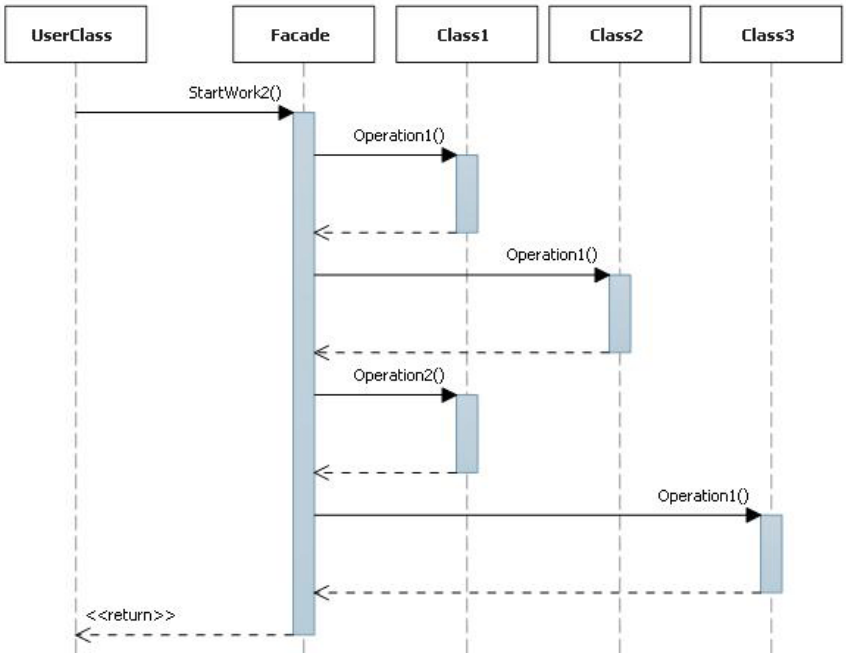


6.3. Patternin strukturu

Fasad patterni fasadın onun üçün hazırlandığı proqramda hər hansı bir alt sistemin varlığını nəzərdə tutur. Fasadın class-ından altsistem class-ları əməliyyatlarının icrası inisializasiya edilir.

Qeyd etməyə lazımdır ki, müştərilərin (altsistemin istifadəçiləri) altsistemin class-larına keçidi olmamalıdır.

Altsistemin bütün obyektləri əgər mümkündürsə fasad class-ında saxlanılmalıdır. Müştərilərin Fasad obyektindən metodun çağırılması zamanı o altsistemin class-ları ilə işləməyə başlayır.



Yuxarıda göstərilən nümunə qayda deyildir. Fasad patternində altsistem daxilində üçdən az və çox class ola bilər. Fasad class-ının obyekti metodları yalnız altsistem elementlərindən çağırmır, həmçinin onları sazlaya və onları qiymətlərin xüsusiyyətlərinə mənimsədə bilər və s.

Fasad class-ının, altsistem class obyektlərini özündə saxlamağı vacib deyil.

Altsistem obyektlərinə keçidi Proxy patternindən də yerinə yetirmək olar. Həmçinin hərdən proqramçılar fasadın class-ını statik edirlər ki, ona proqramın istənilən nöqtəsindən müraciət etmək mümkün olsun.

6.4. Patterndən istifadə nəticələri

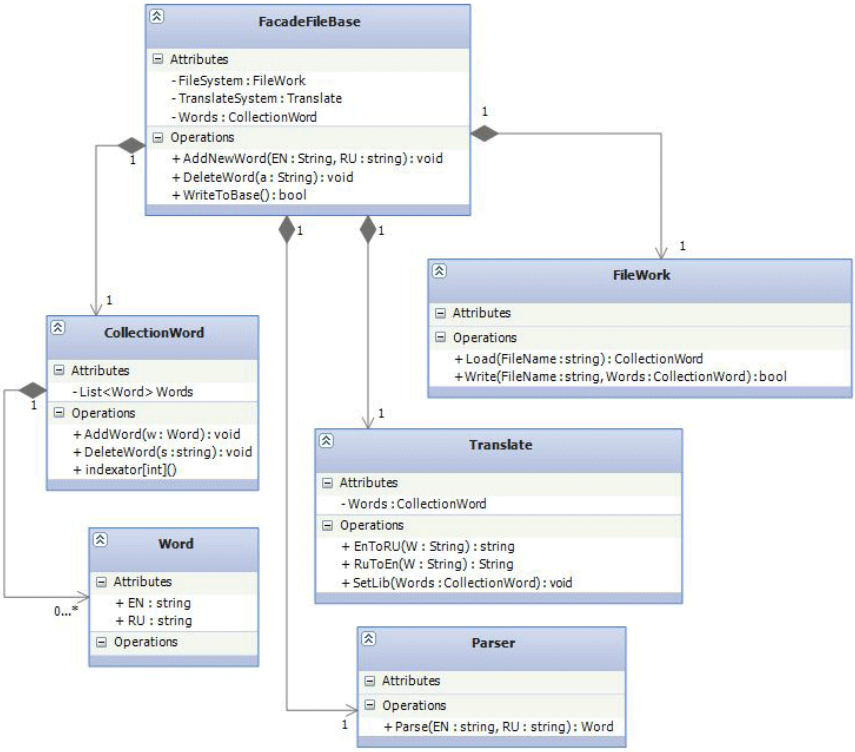
Proqramı o dəqiqə bütün tətbiqi yazmaqdansa, ayrı-ayrı müstəqil altsistemlərin yazılmasından başlayaraq qurmaq daha asandır. Əlavə olaraq, biz proqramın başqa tətbiqlərdə istifadə edə biləcəyimiz blokunu əldə edirik. Həmçinin altsistemin bir komponentinin əvəzlənməsini tətbiqin ümumi strukturunu pozmadan edə bilərik. Əgər hər altsistem üçün öz fasad class-ını yaratsaq, bu cür sistemdən istifadə etmək asan olacaq, belə ki, strukturun bütün işi fasad class-ında inkapsulyasiyalanmışdır (gizlədilmişdir). Digər patternləri Facade patterniylə birgə istifadə edərək tətbiqin daha böyük məhsuldarlıq, çeviklik və təhlükəsizliyinə nail olmaq olar.

6.5. Praktiki nümunə

Fasad pattern class-ına növbəti tətbiqin nümunəsində nəzər salaq: sözlərin sadə tərcüməçisi, istifadəçi sözü ingiliscə daxil edir və onun rusca olan tərcüməsini alır.

Tətbiq, fayldan əvvəlcədən yazılmış (yadda saxlamaq, yükləmək) sözlərin lüğəti ilə işləmə imkanını təqdim etməlidir. Həmçinin əlavə yeni sözləri mətn məlumat bazasına əlavə etməyi bacarmalıdır.

Proqramda lüğztlə işləmək üçün altsistem nəzərdə



tutulmuşdur, ona keçid fasad class-ı obyektindən icra ediləcək.

Yuxarıda bu tətbiqin class diaqramları təsvir edilmişdir. İstifadəçi sitemi (FacadeFileBase) fasad class-ından idarə edə biləcək və o da öz növbəsində altsistemin obyektlərini çağıracaqdır.

7. Pattern Flyweight

7.1. Patternin məqsədi

Patternin məqsədi kompüter yaddaşının daha qənaətli istifadəsidir, bu çoxsaylı xırda obyektlərlə düzgün işin nəticəsində əldə edilir.

Patternin anlayışı üçün obyektin daxili və xarici xüsusiyyətlərini ayırmağı öyrənmək lazımdır. Obyektin daxili xüsusiyyəti daxilində obyekti saxlamaqdır, digər sözlərlə bu daxilində (dəyişkən) xassə yaradılmış və obyektin "yaşadığı" vaxt ərzində daxilində saxlandığı class ekzemplyarıdır. Xarici xüsusiyyət elə bir xüsusiyyətdir ki, obyektə bir və ya bir neçə metodun arqumenti kimi ötürülür və metod icra edildiyi vaxt mövcud olur.

Patternin məğzi daxili xüsusiyyətləri xaricilərə "köçürtmək" sonra isə çoxlu obyekt yox, müxtəlif cür "təsvir olunan" bir obyekti ona hansı xarici xüsusiyyətlərin tətbiq olunmasından asılı olaraq yaratmaqdan ibarətdir.

7.2. Patternin yaranma səbəbləri

OYP – yə "keçiddən" sonra proqramlaşdırmaq daha asan oldu, belə ki, bütün obyektlər "həqiqi" kimidir, lakin bu bir neçə çətinliklərlə bağlıdır. tətbiqin modelini yaradaraq obyektləri obyekt-yönümlü proqramlaşdırma prisnsipləriylə yerləşdirmək arzusu olur, misal üçün, stul oturacaq və ayaqlardan ibarətdir, oturacaq da isə bir neçə detallardan ibarətdir. Əgər biz bu cür arxitekturanı təsvir

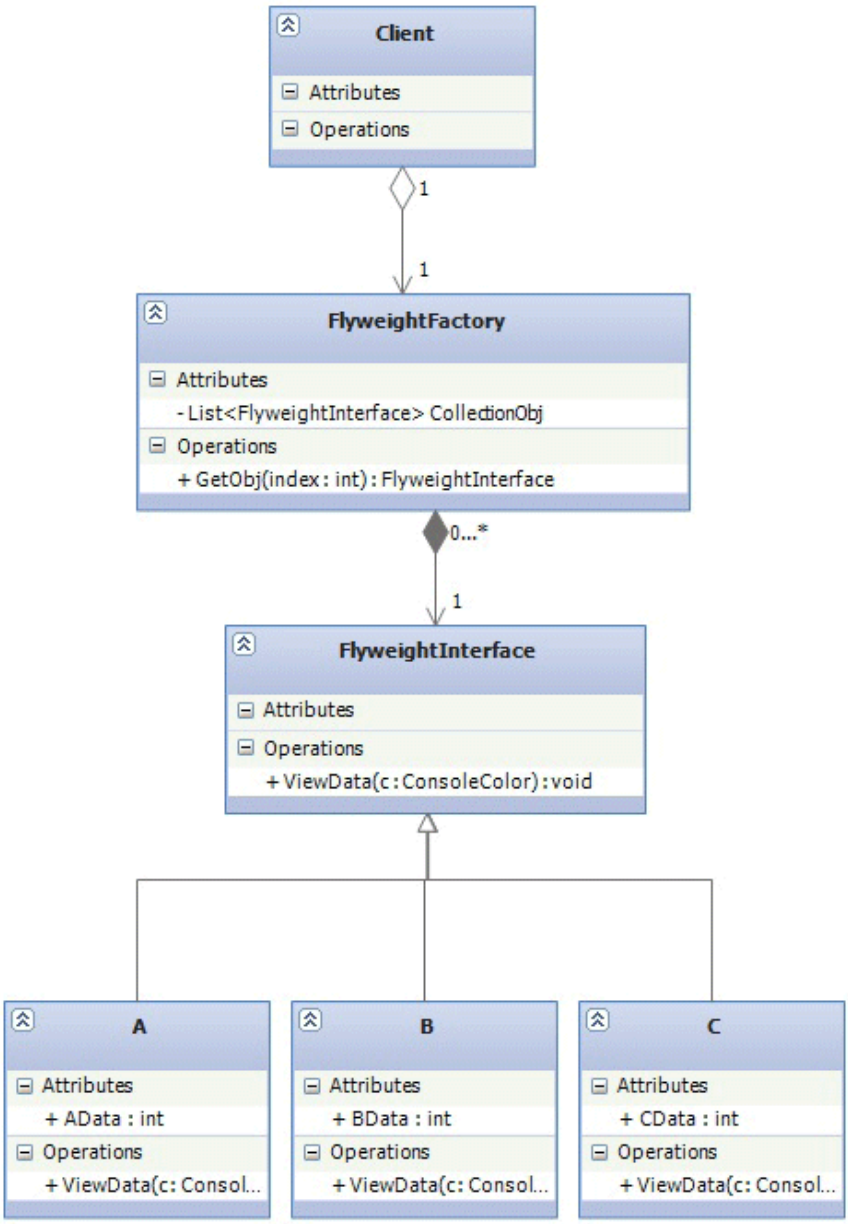
etsək belə bir "dərin" vəziyyət yarana bilər ki, obyektlər həddən çox olarsa, bu cür arxitekturalı tətbiq "tormozlama" verəcək. Lazım olan "dərinlik" kompüterə əlçatmaz olarsa nə etməli? Cavab: tətbiqin məntiqi anlayışını fizikidən ayırmaq, çünki real həyatdan fərqli olaraq, proqramlaşdırmada eyni bir varlığın bir və ya iki (və ya bi çox) yerdə "özünü göstərdiyi", yerləşdiyi anlarla qarşılaşmaq olar. Məsələn, bizdə müxtəlif yerlərdə olan ayaqlar və oturacaqdan ibarət stul var. Obyektin bir neçə yerdə olması üçün obyektden metodun çağırılması zamanı ona elementi harada və necə təsvir olunduğunu göstərən bir neçə parametrləri ötürmək lazımdır. Bu cür yanaşmadan arxitekturanın qurulması üçün istifadə edərək proqrammçı kompüterin resurslarını kifayət qədər qənaətlə istifadə edə bilən tətbiqi əldə edəcək.

7.3. Patternin strukturu

Bir neçə yerdə eyni vaxtda olan obyekt "uyğunlaşan" adlanır. Bu cür obyektlərə keçid fabriklərdən olur. Müştəri, uyğunlaşanın obyektini geriyə qaytaran və ona geri qaytarmaq üçün lazım olan obyektin identifikatorunu qəbul edən metodu çağıraraq, uyğunlaşanların obyektlərini fabrikin köməyilə yaradır.

Fabrik özündə uyğunlaşanların obyektlərini (kolleksiya, massiv, ayrıca) saxlayır. Obyektlərin yaradılması müştərinin ilk sorğusu zamanı vacibdir, sonrakı sorğularda isə onlara yuxarıda dediyimiz kimi fabrikin içində saxlanmalı olan yaradılmış obyekt qaytarılmalıdır.

GetObj metodunun çağırılması zamanı müştəri ona əldə etməyə lazım olan obyektin identifikatorunu ötürür, əgər bu cür obyekt artıq yaradılmışdırsa, onu metodun işinin nəticəsi kimi geri qaytarmaq lazımdır, ancaq, əgər, obyekt yaradılmayıbsa onu yaratmaq və fabrikin anbarına yazmaq, bundan sonra onu metodun işinin nəticəsi kimi geri qaytarmaq lazımdır.


Client
Attributes
Operations
1
1
FlyweightFactory
Attributes
- List<FlyweightInterface> CollectionObj
Operations
+ GetObj(index : int) : FlyweightInterface
0...*
1
FlyweightInterface
Attributes
Operations
+ ViewData(c : ConsoleColor) : void
A
Attributes
+ AData : int
Operations
+ ViewData(c: Consol...
B
Attributes
+ BData : int
Operations
+ ViewData(c: Consol...
C
Attributes
+ CData : int
Operations
+ ViewData(c: Consol...

Tanışlıq üçün növbəti olaraq aşağıda GetObj metodu təsvir olunmuşdur.

```
public IFlyweightInterface GetObj(int Index)
{
    // Obyekti kolleksiyadan indeksinə uyğun götürürük.
    IFlyweightInterface D = CollectionObj[Index] as IFlyweightInterface;

    // Yoxlama, əgər obyekt mövcud deyilsə.
    if (D == null)
    {
        // Obyektin yaradılması.
        switch (Index)
        {
            case 0:
                D = new A();
                break;
            case 1:
                D = new B();
                break;
            case 2:
                D = new C();
                break;
        }
        // Obyekti yaratdıqdan sonra onu kooleksiyaya əlavə edirik.
        CollectionObj.Add(D, Index);
    }

    // Koleksiyadan alınmış və ya indicə yaradılmış obyekti qaytarırıq.
    return D;
}
```

A, B, C class-ları, uyğunlaşanların bütün tipləri ilə qarşılıqlı əlaqə interfeysini təsvir edən baza class-larından varislik alır.

7.4. Patterndən istifadə nümunələri

Bu patterni öz tətbiqinizə tətbiq etsəniz, hətta obyektə keçidin icazəsinin alınması zamanı, fabrikin metoduna vaxtın itirilməsini nəzərə alsaq belə (bizim halda GetObj), siz güclü yaddaş qənaəti əldə edəcəksiniz. Lakin, yaddaşın qənaəti yalnız o halda əldə oluna bilər ki, əgər obyekt-

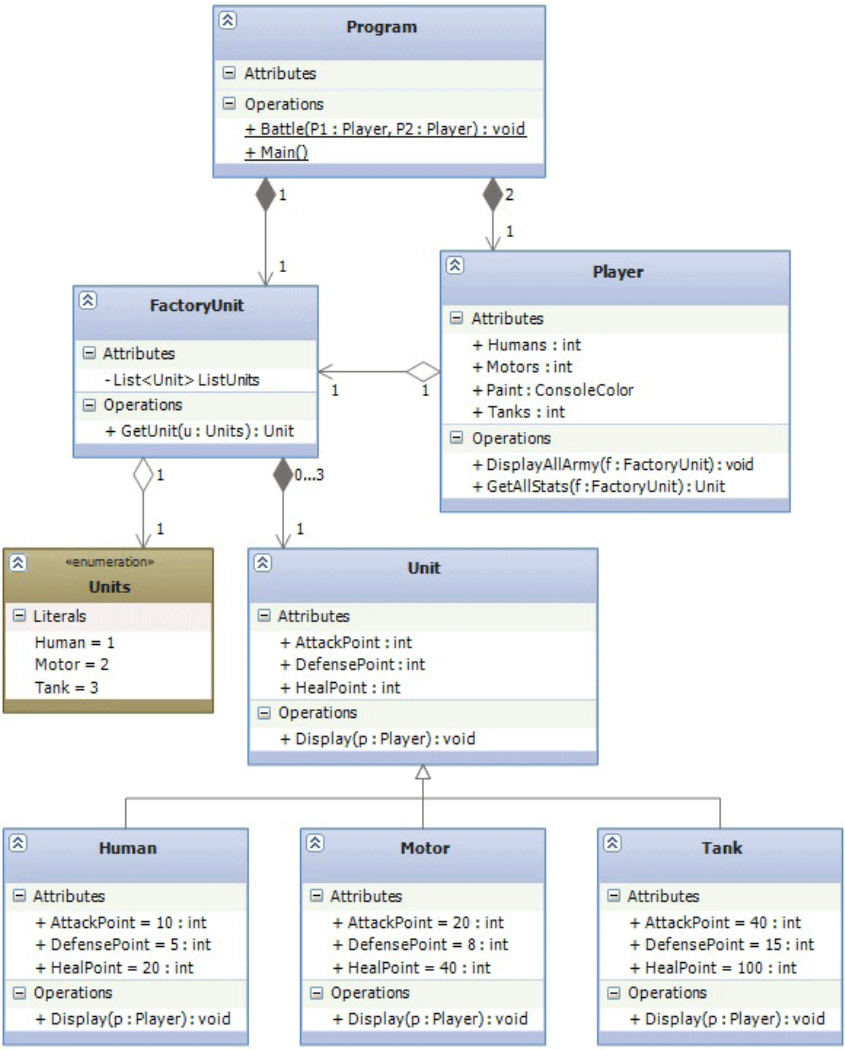
lərdə daxili xüsusiyyətlər minimum qədər olsun, çağırılma zamanı isə xarici xüsusiyyətlər ötürülsün. Xarici xüsusiyyətlər alqoritmlərlə hesablanmalıdır , patternin əsas ideyası bundan ibarətdir ki, "yaddaşa qənaəti, hesablama gücü hesabına etmək".

Uyğunlaşanların obyektlərinə keçidi ancaq fabrikin metodları ilə almağa dəyər, çünki bu, obyektə çeviklik əlavə edəcək və fabrika metodlarında sazlamağa imkan verəcək.

Patterni sizin tətbiqinizdə obyektlərin sayı həqiqətən çox olduğu və onların vəziyyətini xarici xüsusiyyətlərlə təsvir etmək olduğu, xarici xüsusiyyətləri hesablamaların köməyilə təsvir edib və mənimsəmək olduğu zaman tətbiq etmək lazımdır.

7.5. Praktiki nümunə

İki ordunun döyüş nəticəsinin hesablanmasını təsvir edən konsol tətbiqinə nəzər salaq.



Human, Motor və Tank class-ları bir-birindən daxili xüsusiyyətlərin qiymətləri və döyüş vahidini ekranda təsvir edən metodla fərqlənir.

FactoryUnit class-ının obyekti, özündə Unit şəklində sıfırdan üçə qədər obyekti xüsusi olaraq bunun üçün yaradılmış ListUnits kolleksiyasında saxlayacaq. Müştərilərin class-ları obyektə keçidi fabrikdən metodu çağırıb ona hesablama obyektini ötürərək keçid alacaqlar, bu da metodun çağırılma nəticəsi kimi konkret hansı Unit-i qaytarmaq lazım olduğunu göstərəcək.

Oyunçu class-ı (Player) fabrikanı obyektlərin alınması və təsvir olunması üçün istifadə edəcək. Lakin, oyunçuların bütün obyektləri Unit nəslinin class-larının yalnız 3 nümunəsindən istifadə edəcək, bu da vaxt və yaddaşa qənaət etməlidir.

Program - Əsas class-ının Main metodu vardır ki, burada bütün proqramın idarə edilməsi baş verir, həmçinin, iki oyunçu ordusunun döyüşünün nəticəsinin göstərən riyaziyyata cavab verməli olan Battle metodu.

8. Pattern Proxy

Proxy patternini çox zaman surroqt pattern adlandırırlar. Bundan sonra haqqında danışılacaq əsas class surroqat adlandırılacaq. Hər iki termin düzgündür və hər ikisi eyni bir class/obyekti təsvir edəcəkdir.

Proksi - digər bir obyektə tranzit giriş iznini təmin edən hər hansı bir obyektdir.

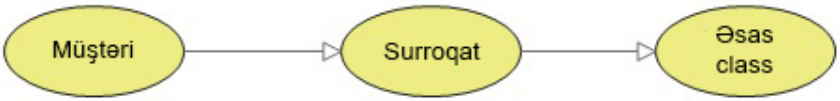
Surroqat - əsas obyektdən xarici görünüşcə heç nə ilə fərqlənməyən, lakin daxili funksionallığı olmayan hər hansı bir obyektdir.

Surroqatın obyekti və ya proksi, təmsil etdiyi obyektdən heç nə ilə fərqlənmir, lakin, onun bütün daxili funksionallığı, onun vasitəsi ilə müştəri class-larının əsas obyektlərə giriş izni ala bilməcəyi bir tuneldən ibarətdir.

Həmçinin surroqat obyektini və ya proksini obyektin müavini də adlandıra bilərik.

8.1 Patternin məqsədi

Patternin məqsədi surroqatın xüsusi obyekti vasitəsi ilə obyektə giriş sistemini yaratmaqdır.



Bu hədəf class-ı obyekti ilə işdə böyük çevikliyə nail olmağa imkan verir.

Əgər hədəf class-ının obyekti yaradılandan sonra çox yer tutacaqsa və obyektin tətbiqinin işində zəruriliyinə zəmanət yodursa, hədəf obyektini onda yaratmaq olar ki, müştəri metodu, surroqatın obyektindən ilk dəfə çağırsın və bu, tətbiqin cəld hərəkətini obyektin istifadə olunmadığı halda sürətləndirəcək. Obyekt əgər yaradılsa, proqramın icra mərhələsində istifadəçi üçün bu elə də "ağrılı" olmayacaq.

Bu cür keçid, sistemi hədəf class-ını böyük müdafiə ilə istifadə etməyə imkan yaradacaq (surroqatda qəbul edilən parametrlərin müxtəlif yoxlamalarını icra etmək olar). Misal üçün, əgər hədəf class-ı kalkulyatordursa, sadəcə metodun arqumetinə iki parametri qəbul edir və riyazi baza əməliyyatları yerinə yetirir, kalkulyator class-lı surroqatda yoxlama etmək olar ki, kalkulyatora "sıfıra bölmək" tapşırığı ötürülməsin.

Proksi patterninin strukturunu adların digər fəzasında "Səfir" adlananın yaradılmasında istifadə etmək olar. Misal üçün, hədəf class-ı adlar fəzasında təsvir olunub və hansısa səbəblər üzündən onu qoşmaq qeyri-mümkündür (ya da narahatdır),b u halda "lazım olan"ad fəzasında yerləşən surroqatı yaratmaq olar. Surroqatın içində isə digər ad fəzasından olan hədəf class-ının obyekti yaradılacaq.

8.2. Patternin yaranma səbəbləri

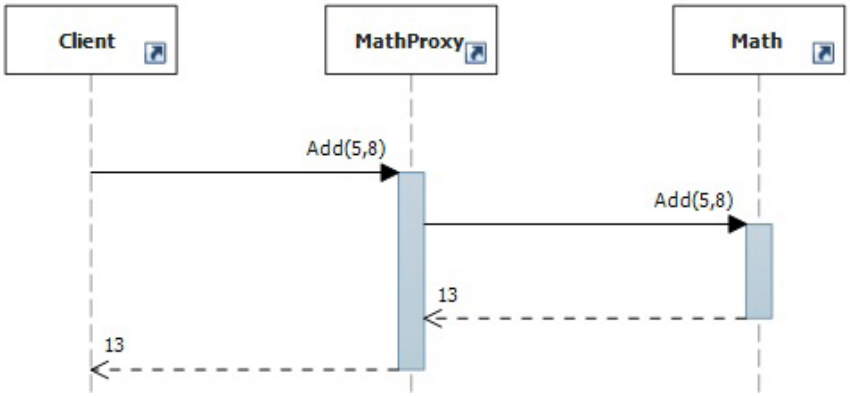
Bir proqramçının kodu, sintaksis və struktur "ənənələr"-i çərçivəsində digər proqramçılar üçün başa düşülən olmalıdır . Tətbiq qurarkən, obyektlərə keçid

üçün proksi patternini istifadə etmək və surroqatların obyektlərində məlumatların yoxlamaları ilə məşğul olmaq, hədəf class-ının özündə isə məntiqi icra etmək olar. Ardınca, digər proqramşıya yoxlamaların harada keçirildiyini və məntiqin harada istifadə olunduğunu başa salmaq kifayət qədər asandır. Ümumiyyətlə, patternlərdən istifadə edərək əlavənin strukturunu anlatmaq çox asandır.

Yəqin, qoşulduğundan sonra uzun müddət "fikirləşib", hardasa 5 və daha çox saniyədən sonra öz interfeysini təsvir edən tətbiqləri yəqin görmüsünüz. Bu tətbiqin start zamanı o dəqiqə bütün öz modullarını yükləməyə başlaması və tətbiqin işində gərəkli olan (və ya gərəkli olmayan) class-ların bütün nümunələrinin yaratması hesabına olur. Bu cür problemin hissəli həlli üçün, "ağır" obyekt surroqatını yaratmaq olar. Tətbiqin startı zamanı belə obyekti yaratmaq, surroqat yalnız mulyaj olduğu üçün hədəf obyekti kimi "ağır" deyildir, buna görə də daha tez yaradılır, "ağır" obyekt isə yalnız zəruri olduğu halda yaradılır. Peyda olma zərurəti isə, müştərinin class-larının surroqat obyektləri ilə işlədiyi zaman, bütün sorğuları indicə yaradılan hədəf obyektinə ötürəcəkdir.

8.3. Patternin strukturu

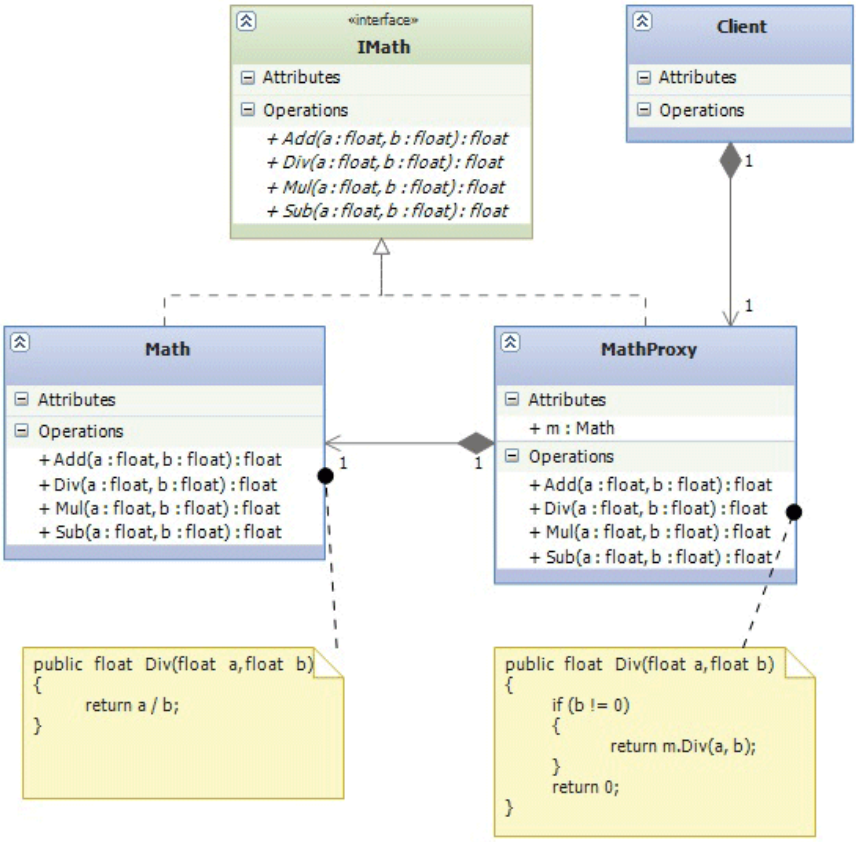
Pattern Proxy hər hansı bir obyektin (ardınca "hədəf class-ı") olmasını nəzərdə tutur, ona keçid xüsusən yaradılmış surroqat obyekti ilə icra olunacaq.



Lakin surroqatın obyekti xarici görünüşdə hədəf class-ından seçilməməlidir, ona görə hər iki class eyni bir interfeysin varisi olmalıdır. Aşağıda göstərilmiş misalda hədəf obyekti Math o an surroqtın obyekti ilə yaradılır, belə ki math class-ı "ağır" deyildir və tez yaradılacaqdır. Nümunə göstərir ki, surroqat obyektinin köməyi ilə müxtəlif yoxlamaları, məlumatların birbaşa hədəf class-ına ötürülməsindən öncə necə etmək olar.

8.4. Patterndən istifadə nəticələri

Patternə keçidi surroqat obyektindən etdikdən sonra, biz hədəf obyekti ilə əlavə iş mərhələsini əldə edirik. Bu mərhələdə biz, hədəf obyektlərni yalnız onların istifadə dərəcələrinə uyğun olaraq yaradaraq optimizasiya ilə məşğul ola bilərik. Məlumatların hədəf obyektinə ötürülməsindən öncə onların yoxlamasını keçirtmək olar. Surroqat obyektində həmçinin obyektə keçidin loqinlenmesini əlavə etmək olar (misal üçün, kalkulyatorda toplama metodunun neçə dəfə çağırılmasını hesablamaq).

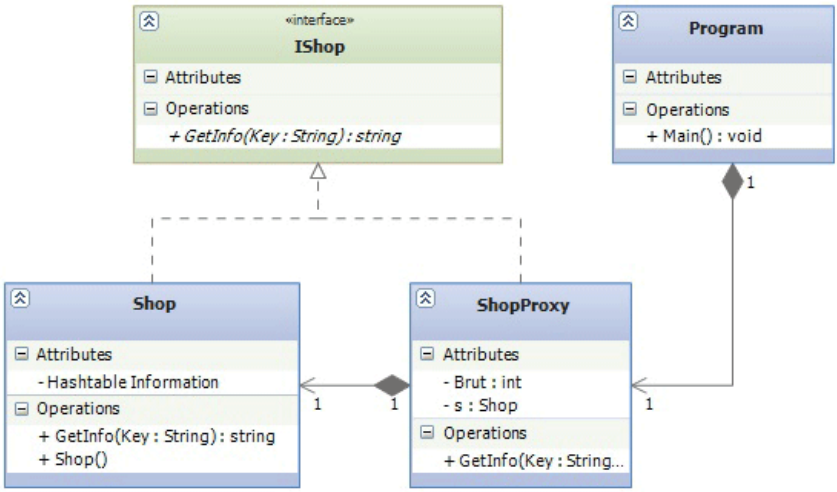


8.5. Praktiki nümunə

Patternin daha yaxşı başa düçülməsi üçün "informasiya mağazası" tətbiqinin strukturu ilə tanış olaq. Tətbiqin arxitekturasında Shop class-ı var, ona keçid ProxyShop class-ından icra edilir. Shop class-ının sətri açar ilə qəbul edən metodu var, sonra isə, əgər mümkündürsə sətri informasiya ilə geri qaytarır. ProxyShop class-ı Shop

class-ını onu “brutforslamaq” imkanından müdafiə edir, həmçinin Shop obyekti, proqrama lazım olan zaman başqa sözlə kimsə açar ilə informasiyanı əldə etmək istəyərkən yaradılır.

Bu tətbiqin diaqramı aşağıda təsvir olunmuşdur.



ShopProxy –dən olan GetInfo metodu obyektin yaranmasını və keçidin nəzarətini icra edir.

```
public string GetInfo(string Key)
{
    if (Brut < 3)
    {
        if (s == null)
        {
            s = new Shop();
        }

        string Ret = s.GetInfo(Key);

        if (Ret == "null")
        {
            Brut++;
            return "Kod düzgün deyil xəta nömrə"  + Brut;
        }
        Brut = 0;
        return Ret;
    }
    return "Sizdə keçid yoxdur" ;

}
```

Shop-dan olan GetInfo metodu isə sadəcə olaraq açara görə dəyəri qaytarır.

```
public string GetInfo(string Key)
{
    String Resoult = Information[ Key]  as string;
    if (Resoult == null)
    {
        return "null";
    }
    else
    {
        return Resoult;
    }
}
```

9. Struktur patternlərinin analiz və müqayisəsi

Siz artıq struktur patternləri haqqında hər şeyi öyrəndiniz:

- Adapter — ona keçid üçün obyekti digər interfeys altına "adaptasiya" etməyə imkan yaradır. Məsələn: obyektdə Operation1 metodu vardır, bizə isə lazımdır ki, o OperationA adlansın. Adapter patterninin məqsədi interfeysin hədəf obyektinə "redaktə" olunmasıdır.
- Bridge – pattern abstrakt şəkildə təsvir olunmuş obyekt ierarxiyası vardırsa və sonra da konkret system altına ierarxiya realizə ediləcəkdirsə, abstraksiyanı realizədən ayırmağa imkan yaradır. Bridge patterninin məqsədi abstraksiya və realizənin ayrıca qurulmasıdır və müştəriyə realizə ilə idarə etmək üçün abstraksiyanın təqdim olunmasıdır.
- Composite – obyektləri ağac tipində düzən struktur pattern. Bu patternin məqsədi obyektlərin saxlanılması üçün ağac şəkilli strukturun qurulmasıdır.
- Decorator – bir neçə obyektin bir obyekt kimi təsvir olunmasını strukturlaşdırmağa imkan yaradır.
- Facade – böyük tətbiqin kiçik müstəqil altsistemlərdən necə qurulmasından bəhs edən struktur pattern. Patternin məqsədi altsistemlə işləmək üçün dar və aydın interfeysin yaradılmasıdır.

- Flyweight – bu pattern proqramda çoxsaylı kiçik obyektlər olduğu halda və onların vəziyyətini alqoritmlərlə hesablanan xarici xassələrə çıxarmaq mümkün olduğu halda yyaddaş yeri qənaət etməyə imkan yaradır.
- Proxy – hədəf obyekti üçün surroqat obyektin yaradılmasını nəzərdə tutur. Bununla da böyük çeviklik, yaddaş qənaəti və müdafiə təmin edir.

Patternlərin istifadəsinə onların əhəmiyyətlərindən uzaqlaşaraq yaxınlaşmaq lazımdlr, çünki ilk baxışdan elə görünə bilər ki, patternlər çox şeylə oxşardırlar, lakin, bu ona görə elə görünür ki, onların bəzlilərində oxşar struktur vardır. Lakin hər patternin istifadə məqsədləri vardır. Məsələn sizə elə gələ bilər ki, fasad və proksi patternləri çox oxşardır, lakin, onlar ayrı məqsədlər üçündür, fasad patterninin məqsədi altsistem ilə işdə dar interfeys yaratmaqdır, proksi isə hədəf obyektini qorumağa imkan verir.

Strukturca Composite və Decorator patternləri də çox oxşardırlar, lakin tətbiq məqsədlərindən uzaqlaşmaq lazımdır, belə ki, Composite patterni obyektləri strukturlaşdırır, Decorator patterni isə funksioanllığın ayrı növlərini yaratamağa və onlardan "bir" obyekt yaratmağa imkan verir.

Tez-tez insanların fasad (Facade) və adapter (Adapter) patternləri arasında fərqin duyula bilmədiyini görmək olur, çünki, elə gəlir ki, onların məqsədi eynidir, bu obyekt və ya obyektlərə keçidin interfeysini dəyişmək və ya modifikasiya etməkdir. Lakin, adapter obyektə keçidin

köhnə interfeysiylə işləməyə və yenisini istifadə etməyə imkan verir, fasad isə daha məqsədyönlü interfeys yaradır.

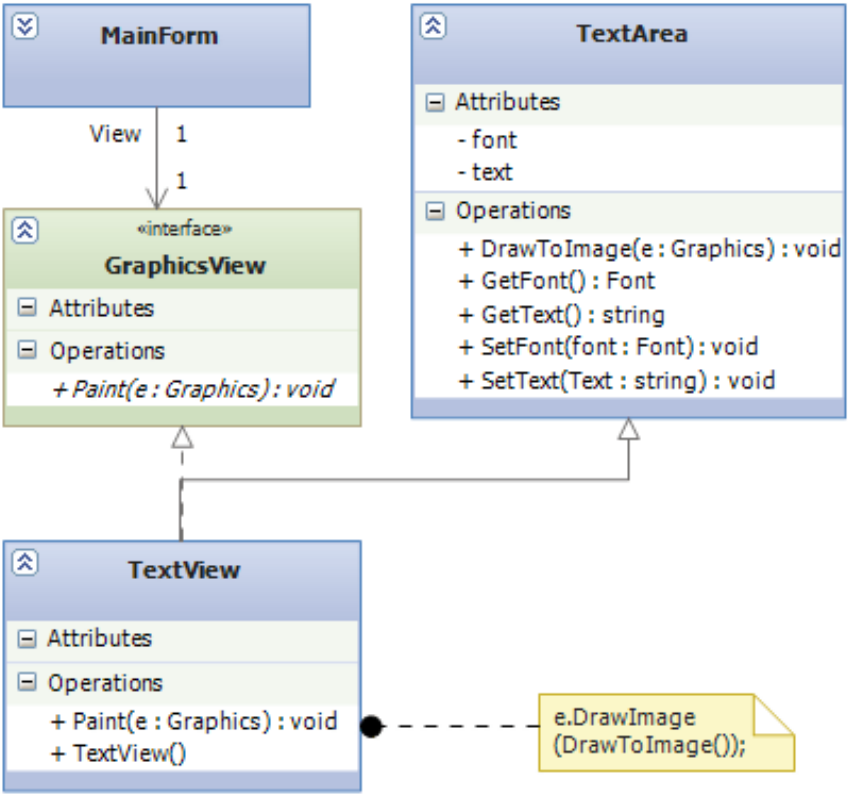
Bir faktı qeyd etmək vacibdir ki, praktikada tez-tez eşitmək olar ki – " bizdə burda Flyweight patternində olan arxitektura var, ancaq, fabrikdə obyektlərin ayrı tipləri ayrı metodlardan geri qayıdır."

10. Struktur patternlərindən istifadənin praktiki nümunələri

İlk nümunə kimi biz Adapter və Decorator patternlərini istifadə edən tətbiqə nəzər salaq. Tətbiq, mətn fayllarının açılma imkanı olan mətn redaktorunu təmsil edəcəkdir. Lakin, o, mətnin redaktə edilməsinə gərəkli olan NET Framework (TextBox və RichTextBox kimi) standart tip sistemi ilə təqdim olunan idarəetmə elementlərini istifadə etməyəcəkdir.

Praktiki nümunənin vəzifəsi, mətn redaktəsini edən idarəetmə elementinin təşkilinə yanaşmanı illustrasiya etməkdir. Adapter patterni mətnlə idarəetməni icra (icra dedikdə almaq, saxlamaq və redaktə etmək başa düşülür) edən class-ı - hər hansı bir qrafik kontekslə - TextView (in-

giliscə - Text View mətn təsviri) adlandırdığımız class-la əlaqələndirmək üçün istifadə oluncaqdır.



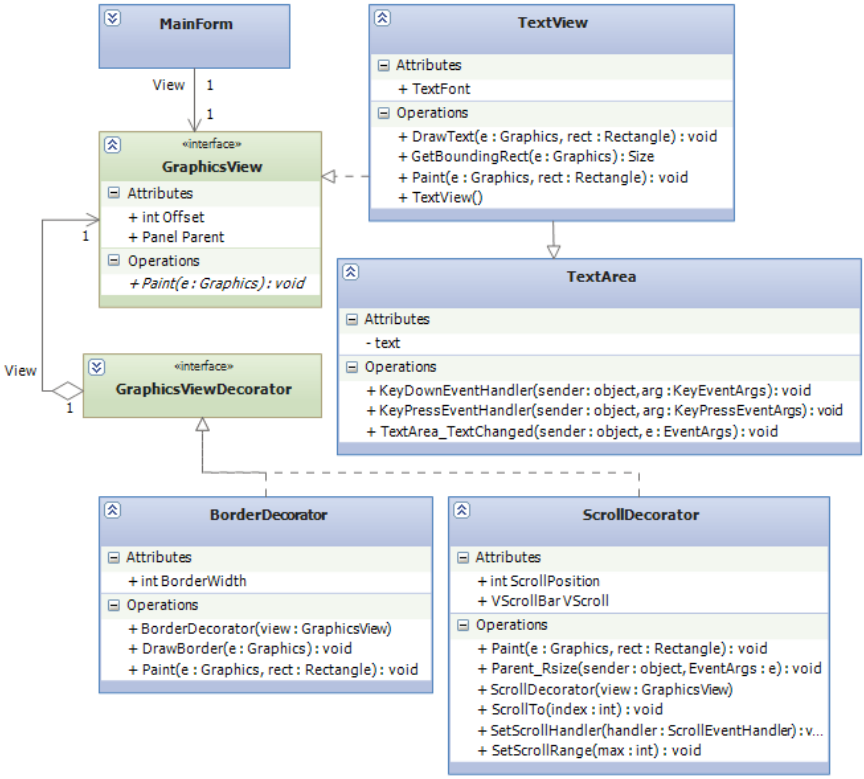
Adapter patterninin istifadəsi zamanı mətn sahəsinin qrafik kontekstlə əlaqəndirmə modeli yuxarıdakı diaqramda göstərilib.

GraphicsView interfeysi elan edilir – bu, qrafik təsviri olan standart elementin interfeysini müəyyən edir.

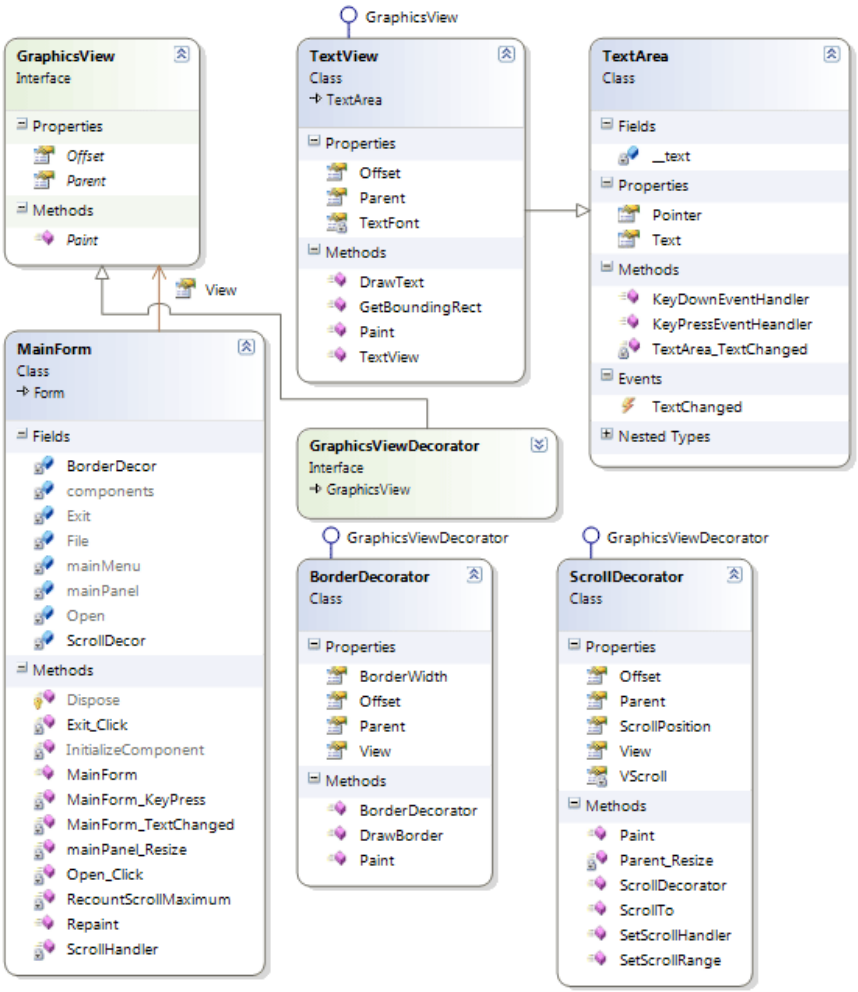
TextView bir tərəfdən GraphicsView, digər tərəfdən TextArea –nın varisi olur.

Ardınca, idarəetmə elementinin müəyyən şəklə salınması üçün biz dekorator patternindən istifadə edirik. Biz iki dekoratoru müəyyən edəcəyik: birincisi, mətn sahəsi ətrafında çərçivənin çəkilməsinə cavabdeh olacaq, ikincisi isə mətn sahəsinə sürüşdürmə zolağının əlavə edilməsini və mətn təsvirinin çəkilişini sürüşdürmə zolağının sürüşməsinin nəzərə alınması ilə təsvir olunmasıdır.

Ümumi model aşağıda göstərilmişdir.



Yuxarıda müəyyən edilmiş modellə class-ların elanının uyğunlaşmasından sonra, biz class-ların növbəti strukturunu alcayıq, bu da aşağıdakı class diaqramında göstərilmişdir.



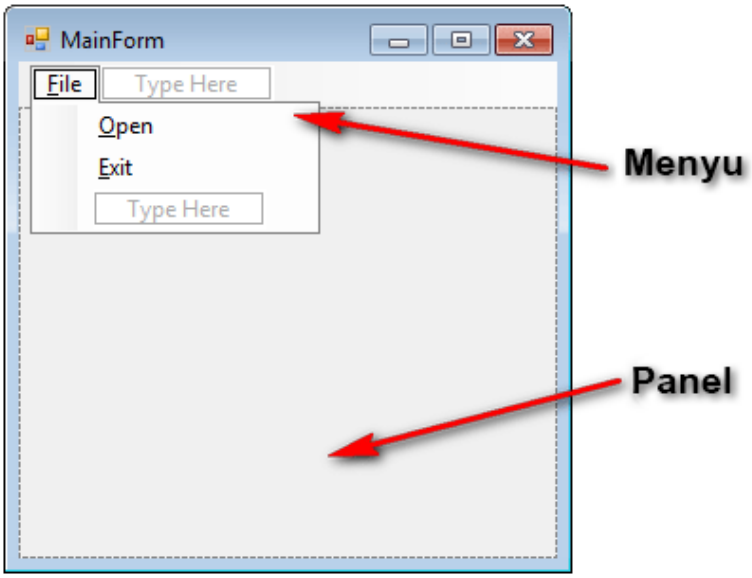
Nəzərə almaq lazımdır ki, model həmişə realizə ilə razılığa gəlmir, buna görə UML – diaqramı və hazır layihənin class diaqramları arasında böyük fərq ola bilər. Amma ümumi yanaşma dəyişməz qalacaqdır. Class

diaqramlarını layihəyə əlavə etmək üçün, SolutionExplorer pəncərəsində layihənin kontekst menyusunu çağırmaq və class diaqramlarının yaradılmasını seçmək lazımdır.

Formanı aşağıdakı kimi tərtib etmək lazımdır:

Menyu və paneli, əlavə etmək, Dock xassəsini mütləq Fill qiymətində quraşdırmaq lazımdır.

Şəkildə göstərilənə oxşar bir şey yaranmaldır.



Ardınca diaqram class-ında təsvir olunan, class-lar üzrə tətbiqin kodu göstərilmişdir.

Aşağıda qrafik kontekstdə çəkilə bilən, qrafik elementin və ya elementin interfeysi göstərilmişdir.

```
public interface GraphicsView
{
    Panel Parent { get; set; }
    int Offset { get; set; }
    void Paint(System.Drawing.Graphics e, Rectangle updateRectangle);
}
```

GraphicViewDecorator interfeysi qrafik sahəyə varislik edir, və onunla fərqlənir ki, View xassəsini GraphicsView tipində müəyyən edir, belə ki, dekoratorun dekorlaşdırılan təsvirə müraciəti olmalıdır.

```
public interface GraphicsViewDecorator : GraphicsView
{
    GraphicsView View { get; set; }
}
```

Ardınca mətn sahəsinin təsviri gəlir, KeyPressed pəncərə tədbirinin emalçısından icra edilən mətnin text sahəsinə əlavə edilməsinin emalı üçün metodları vardır.

KeyDown tədbir emalçısı Left və Right düymələrinin basılmasıyla mətn sahəsininin daxili göstəricisinin hərəkətini emal etməyə lazımdır.

```
public class TextArea
{
    public delegate void TextChangedHandler(object sender, EventArgs e);
    public event TextChangedHandler TextChanged =
                                    new TextChangedHandler(TextArea_TextChanged)
    string __text = "";
    public int Pointer { get; set; }
    public string Text
    {
        get
        {
            return __text;
        }
```

```
            set
            {
                __text = value;
                TextChanged(this, new EventArgs());
            }
        }
        static void TextArea_TextChanged(object sender, EventArgs e)
        {
        }
    public virtual void KeyPressEventHeandler(object sender, KeyPressEventArgs arg)
        {
            if ((int)arg.KeyChar == 8)
            {
                if (Text.Length > 0)
                {
                    Text = Text.Remove(Pointer-1, 1);
                    Pointer--;
                }
            }
            else if (!Char.IsControl(arg.KeyChar))
            {
                Text = Text.Insert(Pointer++, arg.KeyChar.ToString());
            }
        }
        public void KeyDownEventHandler(object sender, KeyEventArgs arg)
        {
            switch (arg.KeyData)
            {
                case Keys.Left:
                    if (Pointer > 0)
                        Pointer--;
                    break;
                case Keys.Right:
                    if (Pointer < Text.Length)
                        Pointer++;
                    break;
            }
        }
    }
```

TextView class-ı adapter – class-ıdır və mətnin qrafik kontekstdə çəkilmə metodunun təsviri, həmçinin sonda dekoratorla çağırılan Paint metodu vardır.

```
public class TextView : TextArea, GraphicsView
{
    public Panel Parent { get; set; }
    public int Offset { get; set; }
    private Font TextFont { get; set; }
    public TextView(string Text, Panel parent)
        :base()
    {
        Parent = parent;
        this.Text = Text;
        TextFont = new Font("Verdana", 10, FontStyle.Regular);
    }
    public SizeF GetBoundingRect(Graphics e)
    {
        return e.MeasureString(
                                Text,
                                TextFont,
                                Parent.ClientRectangle.Width,
                                StringFormat.GenericDefault);
    }
    public void DrawText(Graphics e,Rectangle rect)
    {
        e.DrawString(
                                  Text,
                                  TextFont,
                                  Brushes.Black,
                                  new Rectangle(
                                            rect.X,
                                            rect.Y - Offset,
                                            rect.Width,
                                            rect.Height + Offset));
    }
    public void Paint(System.Drawing.Graphics e,Rectangle updateRectangle)
    {
        DrawText(e, updateRectangle);
    }
}
```

BorderDecorator class-ı qrafik təsvirə (GraphicsView varisi olan class) çərçivənin əlavə edilməsi üçün nəzərdə tutulub. Bunun üçün onun DrawBorder metodu (hansı ki, parametr kimi ötürülən hər hansı bir qrafik kontekstə çərçivəni çəkir) və GraphicsView interfeysinin varisi olan Paint metodu vardır. Paint metodu ardıcıl olaraq qrafik təsvirin anoloji aqreqasiya olmuş metodunu (class

tərəfindən dekor olunmuş təsviri) və DrawBorder metodunu çağırır.

```
public class BorderDecorator : GraphicsViewDecorator
{
    public GraphicsView View { get; set; }
    public Panel Parent { get; set; }
    public int Offset { get; set; }
    public BorderDecorator(GraphicsView view)
    {
        View = view;
        Parent = View.Parent;
        BorderWidth = 2;
        Offset += BorderWidth;
    }
    public int BorderWidth { get; set; }
        public void Paint(System.Drawing.Graphics e, Rectangle rect)
    {
        View.Paint(e, View.Parent.ClientRectangle);
        DrawBorder(e);
    }
public void DrawBorder(Graphics e)
    {
        e.DrawRectangle(
                new Pen(Brushes.LightGray,BorderWidth),
                Parent.ClientRectangle);
    }
```

ScrollDecorator class-ı dekorasiya edilən mətn təsvirinə sürüşdürmə zolağının əlavə edilməsinə və mətn təsviri sürüşəninin sürüşdürmə zolağındakı hərəkətinin reaksiyasına cavab verir, buna görə tədbir emalçıları qeydiyyat edilmişdir. Həmçinin class-a köməkçi metodların elanları əlavə edilmişdir:

SetScrollRange metodu ona görə lazımdır ki, dekorlaşdırılan mətn təsvirində mətnin həcminin dəyişilməsi zamanı sürüşdürmə zolağının maksimal qiymətini dəyişsin, SetScrollHandler metodu isə sürüşdürmə zolağında sürüşənin vəziyyətinin dəyişmə tədbiri üçün istifadəçi emalçılarının quraşdırılması üçün lazımdır.

```
    public class ScrollDecorator : GraphicsViewDecorator
    {
        VScrollBar VScroll { get; set; }
        public GraphicsView View { get; set; }
        public Panel Parent { get; set; }
        public int Offset { get; set; }
        public int ScrollPosition { get; set; }
        public ScrollDecorator(GraphicsView view)
        {
            View = view;
            Parent = View.Parent;
            VScroll = new VScrollBar();
            Parent.Controls.Add(VScroll);
            Parent.Resize += Parent_Resize;
            Parent_Resize(this, new EventArgs());
        }
        public void SetScrollHandler(ScrollEventHandler handler)
        {
            VScroll.Scroll += handler;
        }

  void Parent_Resize(object sender, EventArgs e)
        {
            VScroll.Location =
new Point(Parent.ClientRectangle.Width - VScroll.Width - View.Offset,View.Offset);
            VScroll.Height = Parent.ClientRectangle.Height - View.Offset*2;
        }
        public void ScrollTo(int location)
        {
            ScrollPosition = location;
            Paint(View.Parent.CreateGraphics(),View.Parent.ClientRectangle);
        }
        public void SetScrollRange(int max)
        {
            VScroll.Maximum = max;
        }
        public void Paint(System.Drawing.Graphics e, Rectangle rect)
        {
            View.Paint(e, View.Parent.ClientRectangle);
        }
    }
```

Daha sonra isə tətbiqin əsas formasının mətn class-ı təsvir edilmişdir. Ona köməkçi istifadəçi interfeysi (pəncərə menyusu nöqtələrinə basılmanın emalçıları), həmçinin, mətn təsvirində mətnin dəyişilməsi tədbirinin emalçısı

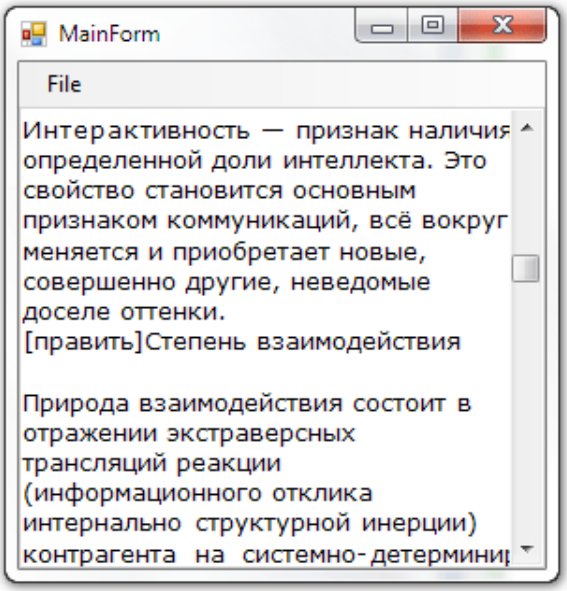
əlavə edilib, buna reaksiya sürüşmə zolağının maksimal qiymətinin dəyişməsi olur. Bu emalçı MainForm class-ının mətninə çıxarılıb, belə ki, ScrollDecorator potensial olaraq istənilən təsvirə birləşdirilmiş ola bilər, bu hərəkət isə tətbiq olunma konteksindən aslıdır.

```
public partial class MainForm : Form
{
    public GraphicsView View { get; set; }
    BorderDecorator BorderDecor;
    ScrollDecorator ScrollDecor;
    public MainForm()
    {
          InitializeComponent();
          this.BackColor = mainPanel.BackColor = Color.White;
          View = new TextView("Hello world", mainPanel);
          this.KeyPress += (View as TextView).KeyPressEventHeandler;
          this.KeyDown += (View as TextView).KeyDownEventHandler;
          this.KeyPress += MainForm_KeyPress;
          BorderDecor = new BorderDecorator(View);
          ScrollDecor = new ScrollDecorator(BorderDecor);
          (View as TextArea).TextChanged += MainForm_TextChanged;
          ScrollDecor.SetScrollHandler(ScrollHandler);
    }
 void ScrollHandler(object sender, ScrollEventArgs e)
    {
          View.Offset = e.NewValue;
          Repaint(this, new PaintEventArgs(
          mainPanel.CreateGraphics(),
          mainPanel.ClientRectangle));
    }
    private void RecountScrollMaximum()
    {
       int max = (int)(View as TextView).GetBoundingRect(
                        mainPanel.CreateGraphics()).Height;
        ScrollDecor.SetScrollRange(max);
        if (ScrollDecor.ScrollPosition > max)
            ScrollDecor.ScrollTo(max);
    }
void MainForm_TextChanged(object sender, EventArgs e)
    {
        RecountScrollMaximum();
    }
    void MainForm_KeyPress(object sender, KeyPressEventArgs e)
    {
```

```
Repaint(
                    mainPanel,
                    new PaintEventArgs(
                        mainPanel.CreateGraphics(),
                        mainPanel.ClientRectangle));
        }
    public void Repaint(object sender, PaintEventArgs e)
        {
            Image img = new Bitmap(this.ClientSize.Width, this.ClientSize.Height);
            Graphics DC = Graphics.FromImage(img);
            DC.Clear(BackColor);
            BorderDecor.Paint(DC, ClientRectangle);
            e.Graphics.DrawImage(img, 0,0);
            DC.Dispose();
            img.Dispose();
        }
        private void mainPanel_Resize(object sender, EventArgs e)
        {
            Repaint(
                      mainPanel,
                      new PaintEventArgs(
                            mainPanel.CreateGraphics(),
                            mainPanel.ClientRectangle));
        }

       private void Open_Click(object sender, EventArgs e)
        {
            OpenFileDialog dlg = new OpenFileDialog();
            dlg.Multiselect = false;
            dlg.Filter = "Text files|*.txt";
            if (dlg.ShowDialog() == DialogResult.OK)
            {
(View as TextView).Text = System.IO.File.ReadAllText(dlg.FileName,Encoding.Default);
                RecountScrollMaximum();
               mainPanel.Invalidate();
            }
        }
        private void Exit_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
```

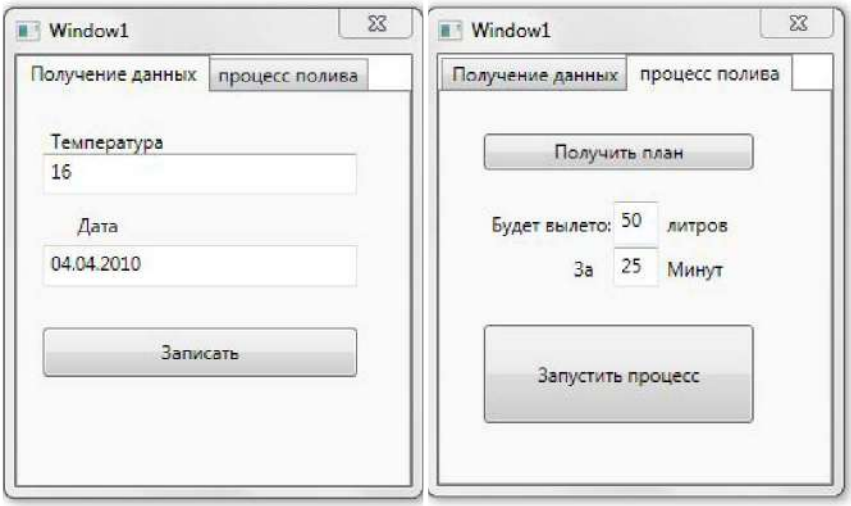
Kodun uğurlu yaradılma nəticəsində, təxminən aşağıdakı təsvirə oxşayan tətbiq alınmalıdır.

Bir faktı qeyd etmək lazımdır ki, patternlər özü-özlüyündə "ciddi fayda vermir". Uğurlu və effektli tərtibatın sirri, patternləri kombinasiya edilməsində və onların tətbiqinin nəticə verdiyi yerdə tətbiq edilməsidir.



İstixanada iqlimin izlənmə proqramı

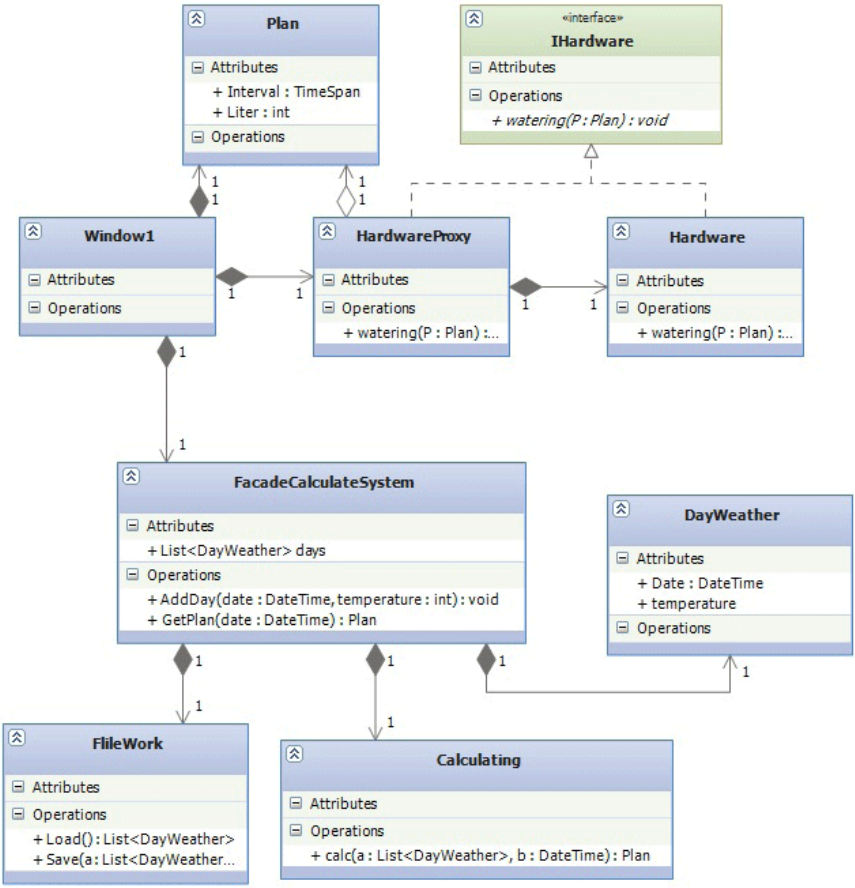
Bu tətbiq iki patterndən istifadə edir, bu proksi patterni, sulama ilə məşğul olan class-a keçid surroqatının yaradılması (harada ki, "tərkiblə" iş olmalıdır) və pattern "hansı temperaturda, nə qədər su tökmək" kimi hesablamalarla məşğul olan altsistemə dar keçid obyektini yaradan fasad patterni.



Bütün class-lara nəzər salaq:

- Window1 – proqramın idarə olunduğu əsas pəncərə.
- Plan – bü cür class-ın obyekti bitkilərə nə qədər vaxt ərzində, nə qədər suyun tökülməsini təsvir edir.
- Ihardware –Plan class-ını arqumentlərinə qəbul edən watering metodu olan interfeys. Bu metodun köməyi ilə biz bitkilərin sulanma prosesini işə salacayıq.
- Hardware – bitkilərin sulanmasını icra edən qurğuların işinə cavab verir. Bizdə belə bir qurğu olmadığına görə, biz MessageBox görəcəyik.
- ProxyHardware – qurğularla işlə məşğul olan class-a komandaların tranzitini icra edən class. Hardware-Proxy obyekti şəraitlərin yoxlanması ilə məşğul olur.

```
Hardware hard = null;
 public void Watering(Plan a)
     {
     if (hard == null)
     {
     hard = new Hardware();
     }

     if ((a.Litters / a.Interval.TotalMinutes) < 1)
     {
        MessageBox.Show("Səhv : çox kiçik təzyiq");
     }
     else if ((a.Litters / a.Interval.TotalMinutes) > 4)
     {
        MessageBox.Show("Səhv : çox böyük təzyiq");
     }
     else
     {
        // Avadanlıqla işin çağırılması
        hard.Watering(a);
        }

     }
```


Plan
Attributes
+ Interval : TimeSpan
+ Liter : int
Operations
«interface»
IHardware
Attributes
Operations
+ watering(P : Plan) : void
Window1
Attributes
Operations
HardwareProxy
Attributes
Operations
+ watering(P : Plan) :...
Hardware
Attributes
Operations
+ watering(P : Plan) :...
FacadeCalculateSystem
Attributes
+ List<DayWeather> days
Operations
+ AddDay(date : DateTime, temperature : int) : void
+ GetPlan(date : DateTime) : Plan
DayWeather
Attributes
+ Date : DateTime
+ temperature
Operations
FlileWork
Attributes
Operations
+ Load() : List<DayWeather>
+ Save(a : List<DayWeather...
Calculating
Attributes
Operations
+ calc(a : List<DayWeather>, b : DateTime) : Plan

- DayWeather — hər hansı bir vaxtda yazılan hava haqda və ilin fəsilləri barədə obyektlərin məlumatlarını saxlayan xüsusi class.
- FileWork – hava barədə məlumatların yazılması və sərt diskə yüklənmə işini özündə inkapsulyasiya edən class.
- Calculating – bu class-ın obyekti hər bir gün üzrə hava haqda məlumatları saxlayan obyekt kolleksiyalarından irəli gələrək, Plan obyektlərinin yaradılması ilə məşğul olacaq.

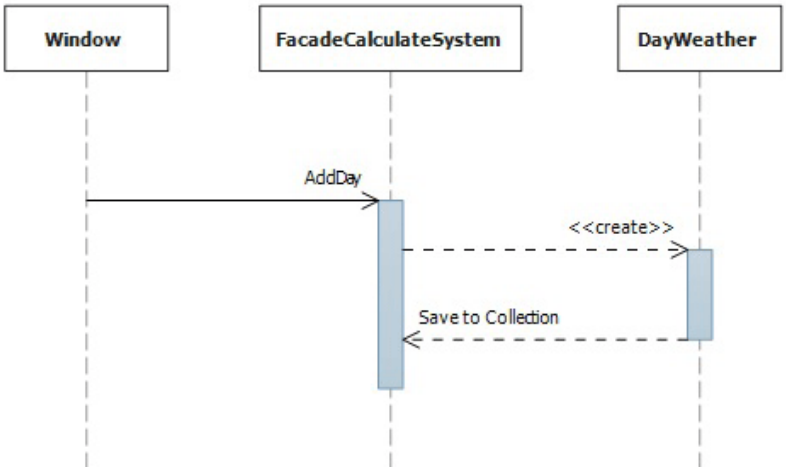
```
/// <summary>
/// Üç ötmüş gün üçün planın heasblanması
/// </summary>
/// <param name="a">Ötən günlərin hava ilə birgə kolleksiyası</param>
/// <param name="date">Bugünkü vaxtlar.</param>
/// <returns>Sulama üçün plan </returns>
public Plan Calc(List<DayWeather> a, DateTime date)
{
    for (int i = 0; i < a.Count; i++)
    {
        if (a[ i].Date == date && i>2)
        {
            int Dt = a[ i].Temperature +
                a[ i - 1].Temperature +
                a[ i - 2].Temperature;
            Dt = Dt / 3;
            int tim = Dt / 2;
            Plan P= new Plan();
            P.Interval = new TimeSpan(0, 0, tim, 0);
            P.Litters – Dt;
            return P;
        }
    }
    MessageBox.Show(" Proqnozlaşdırma səhvi");
    Plan P2 = new Plan();
    P2.Litters = 2;
    P2.Interval = new TimeSpan(0, 0, 1, 0);
    return P2;
}
```

- FacadeCalculateSystem —iqlim barədə məlumatların hesablanması və saxlanılma sistemi ilə istiqamətləndirilmiş işin interfeysini təqdim edən class.

```
class FacadeCalculateSystem
{
    private List<DayWeather> Days;
    private FileWork filework;
    private Calculating Calul;

    public FacadeCalculateSystem()
    {
        filework = new FileWork();
        Days = filework.Load();
        if (Days == null)
        {
            Days = new List<DayWeather>();
            filework.Save(Days);
            MessageBox.Show(" Hava olan fayl yoxdur");
        }
        Calul = new Calculating();
    }

    /// <summary>
    /// Metod gün ərzində olan hava haqqında məlumatları əlavə edir
    /// </summary>
    /// <param name=»Date»>Haqqında danışılan gün </param>
    /// <param name=»Temperature»>O günə olan hava məlumatı </param>
    public void AddDay(DateTime Date, int Temperature)
    {
        DayWeather Day = new DayWeather();
        Day.Date = Date;
        Day.Temperature = Temperature;
        Days.Add(Day);
        filework.Save(Days);
    }

    /// <summary>
    /// Metod tarix haqqında planı qəbul edir
    /// </summary>
    /// <param name=»Date»>Plan lazım olan günün tarixi </param>

/// <returns>Sulama üçün əməliyyat planı</returns>
    public Plan GetPlan(DateTime Date)
    {
        return Calul.Calc(Days, Date);
    }

}
```

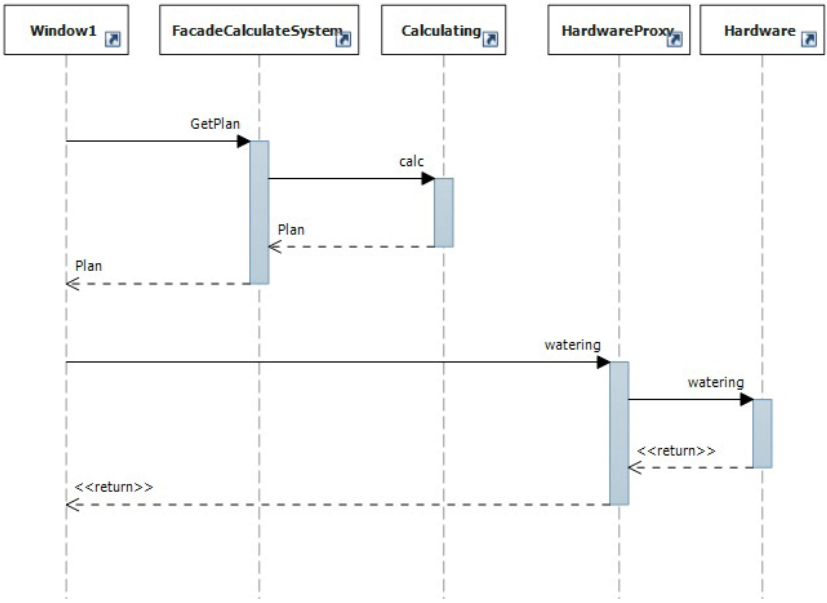
Temperaturun bazaya yazılması zamanı proqramın iş alqoritmi aşağıdakı kimi gedir:

İstifadəçi istixanadakı temperaturun qiymətini proqrama daxil edir, bundan sonra pəncərə bu sorğunu FacadeCalculateSystem class-ına ötürür, o da öz növbəsində məlumatlardan obyekt yaradır, DayWeather yeni obyekti kolleksiyaya yazır və kolleksiyanı faylda saxlayır.



İstifadəçi proqrama sulamanın yerinə yetirilməsini əmr edəndə növbəti alqoritmin icrası baş verir.

Window1 class obyekti sorğunu FacadeCalculateSystem obyektinə GetPlan() metodunu çağıraraq yollayır, fasad class-ı bu vəzifəni Calculating class-ı obyektinə yönəldir, o hesablama aparır və Plan class-ı obyektini geri qaytarır, bundan sonra FacadeCalculateSystem də Planı tətbiqin pəncərəsinə qaytarır. Yuxarıda təsvir olunmuş əməliyyatı yerinə yetirdikdən sonra Window class-ı ProxyHardware class-ına sorğu yollayır ki o, planda təsvir olunmuş hərəkəti icra etsin. Proksi obyekti məlumatları yoxlayır, əgər hər şey yaxşıdırsa çağrını Hardware obyektinə ötürür və sulama məntiqini icra edir.



9. Ev tapşırığı

Ev tapşırığı kimi tətbiqin üç modelini tərtib etmək lazımdır, onlardan hər biri cari dərsdə təsvir edilmiş layihələndirmənin struktur patternindən heç olmasa birini realizə etməlidir. Hər model bu modellə istifadə olunan class-ların strukturunu, həmçinin bu məlumat tiplərinin mövcud olduğu əlaqələri əks edən bir class diaqramı şəklində təqdim olunmalıdır. Yaradılmış modellərdən (seçim üzrə) birini işlək tətbiq şəklində realizə etmək lazımdır.