Kuidas materjalidega töötada?

Käesoleva õppematerjaliga töötamiseks soovitab materjalide koostaja võtta korraga ette üks muster ning see endale selgeks teha. Disainimustritest aru saamine võib olla nende kõrge abstraktsuse tõttu keeruline ja seega ei ole soovitatav neid esialgu omavahel segamini õppida.

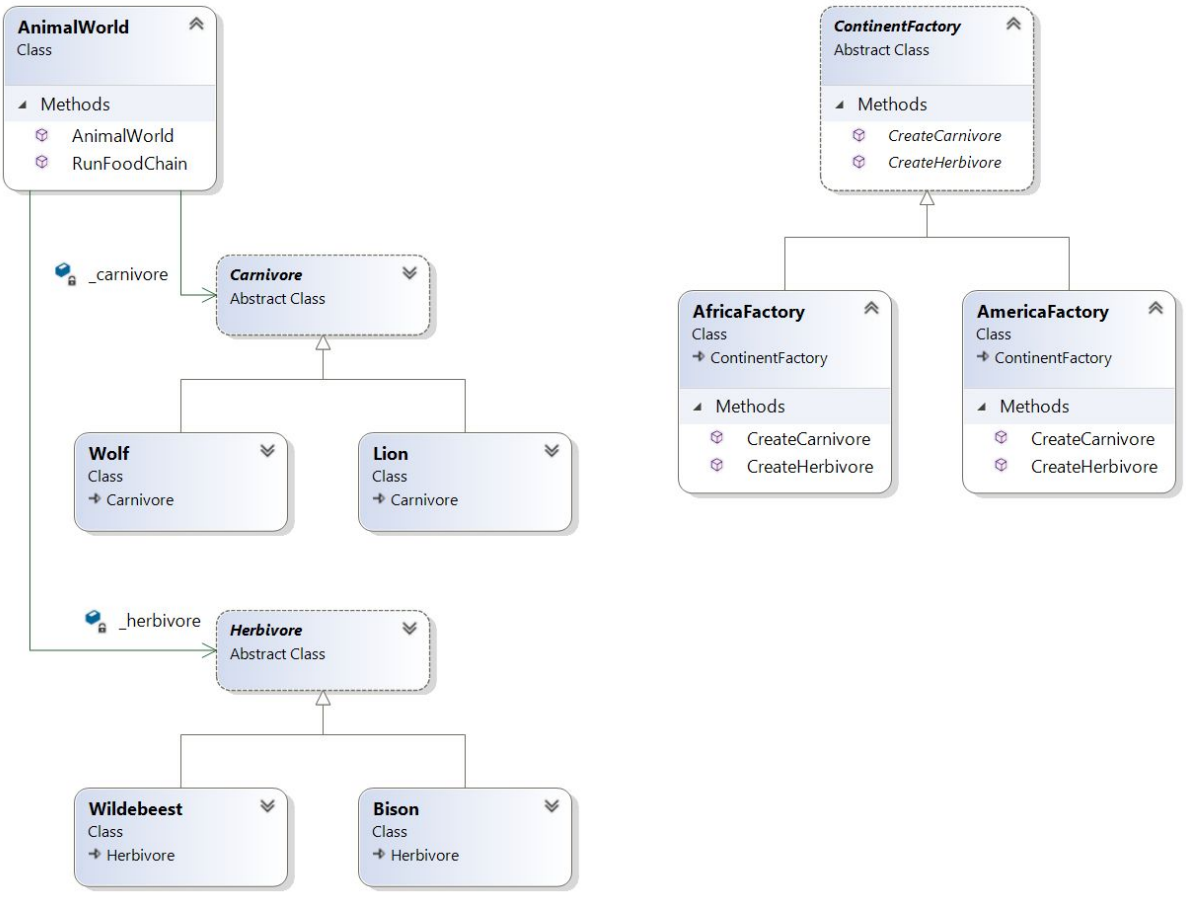
Järgnevalt on loeteluna välja toodud soovitused mustrite õppimiseks.

1. Üks muster korraga. Mis järjekorras mustreid õppida ei ole tähtis - need ei ole omavahel seotud.
2. Käesoleva dokumendi järgi õppides on hea võtta kõrvuti lahti mustri kirjeldus ning repositooriumis olev mustri lähtekood. Projektis **GoF** asub mustrite kood ja klassidiagrammid (samad, mis siin dokumendis) ning **Tests** projektis on vastavad testid. Selleks, et kiiremini aru saada, mis toimub tasub enne uurida, mida teevad testid ja seejärel juba süveneda mustri juurde kuuluvatesse klassidesse.
3. Iga mustri juures on selle definitsioon ja viide leheküljele raamatus **Design Patterns: Elements of Reusable Object-Oriented Software** (GoF) - Raamat on saadaval ülikooli raamatukogu kaudu Safari Books Onlinest: https://proquest.safaribooksonline.com/book/software-engineering-and-development/patterns/0201633612
4. Kui tundub, et dokumendi järgi õppides jäi asi segaseks, siis järgnevalt on välja toodud kohad, kus täiendavat infot hankida.
    a. https://www.dofactory.com/net/design-patterns - Mustrite lähtekoodi algallikas. Lisaks dokumendis olevale koodile on siin ka iga mustri struktuurne kood ja inglise keelne kirjeldus.
    b. https://proquest.safaribooksonline.com/0596007124 - **Head First Design Patterns**, mis seletab iga mustri üksikasjalikult lahti, kasutades rohkelt illustratsioone ja näiteid.
    c. https://www.youtube.com/watch?v=v9ejT8FO-7I&list=PLrhzvIcii6GNjpARdnO4ueTUAVR9eMBpc - Head First Design Patterns raamatu põhjal loodud õpetusvideote esitusloend YouTubes.

Loomismustrid

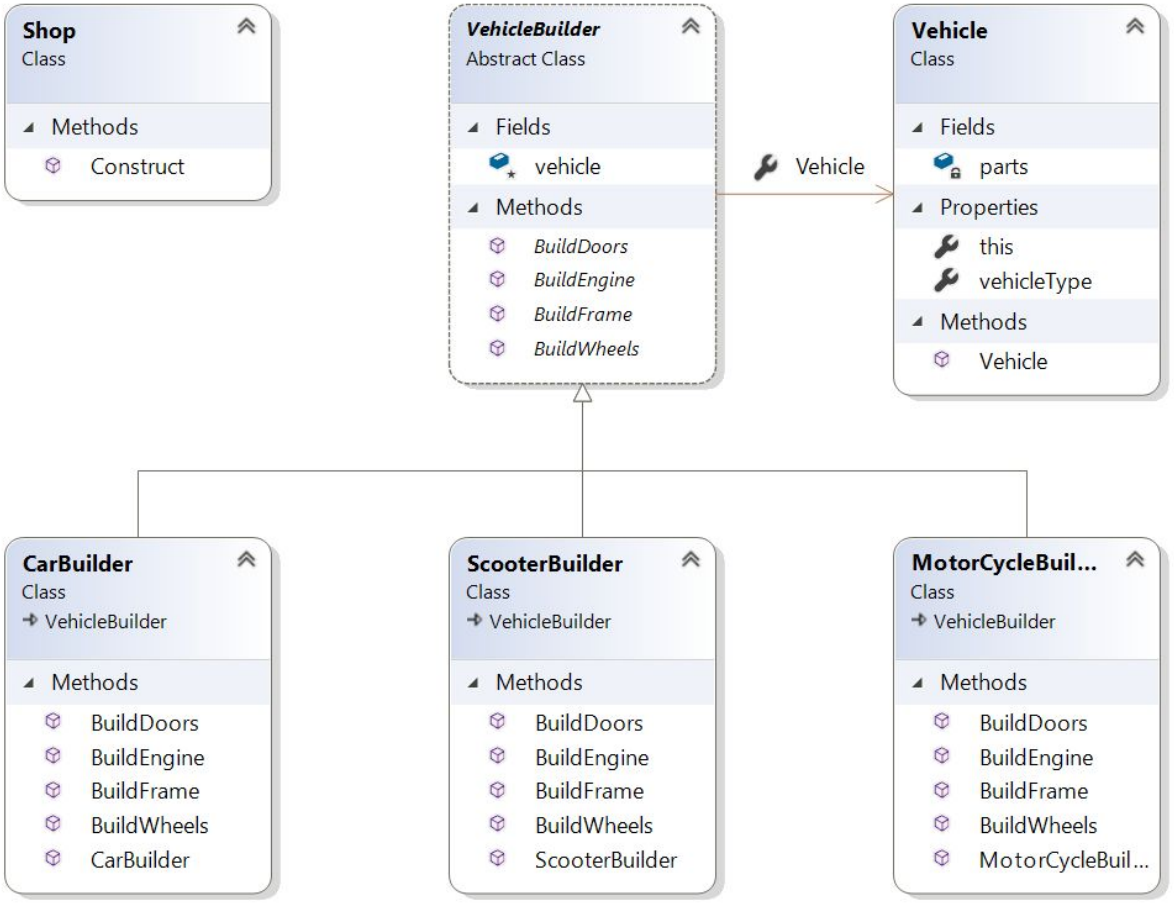
Abstract Factory

Definitsioon: Pakub liidest sarnaste või sõltuvate objektide perekondade loomiseks, ilma nende konkreetseid klasse täpsustamata. [GoF, lk 87]



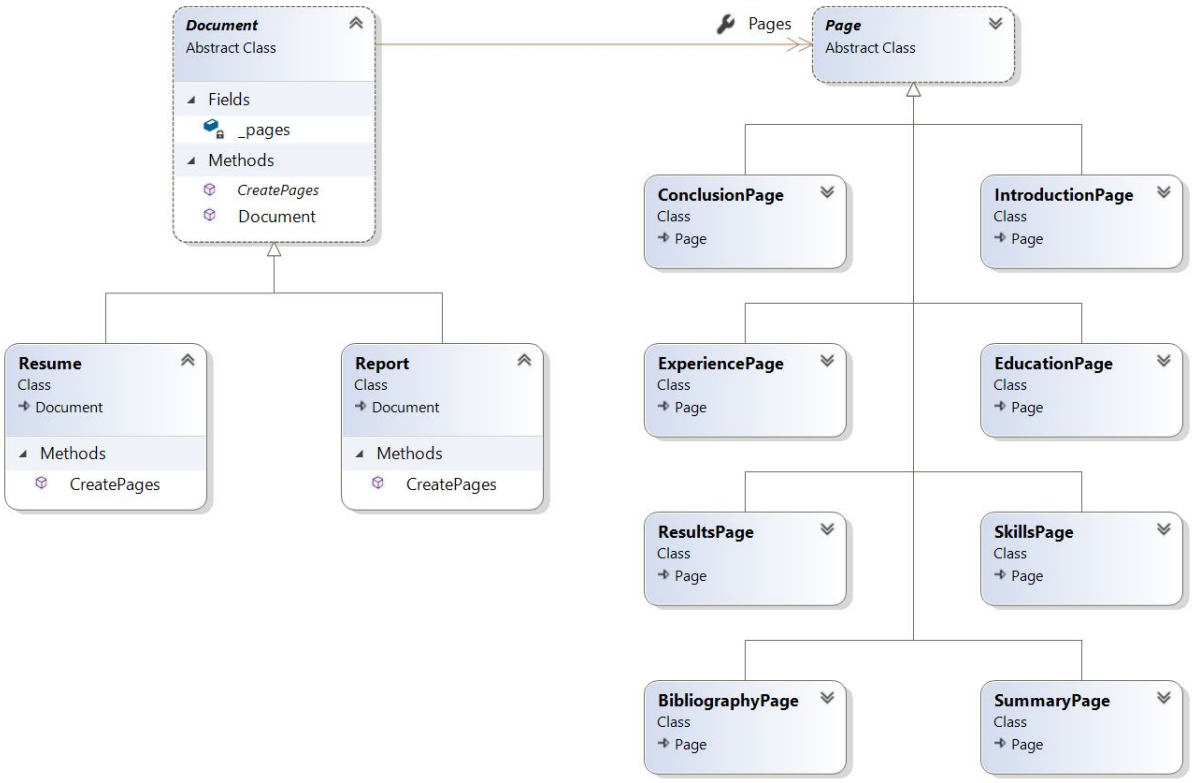
Builder

Definitsioon: Eraldab keeruka objekti loomise selle esitusest, nii et sama ehitusprotsessi abil on võimalik luua erinevaid esitusi. [GoF, lk 97]



Factory Method

Definitsioon: Defineerib liidese objektide loomiseks, kuid laseb alamklassidel otsustada, millist klassi instantsieerida. Võimaldab klassil instantsieerimise edasi anda alamklassidele. [GoF, lk 107]



Factory Method mustri mõte on luua meetod objektide loomiseks (Document klassi CreatePages), mida alamaklassid (Resume ja Report) realiseerivad. Antud näites koosnevad alamklassid Resume ja Report erinvat tüüpi lehtedest. Resume koosneb lehtedest SkillsPage, EducationPage ja ExperiencePage - need lehed lisatakse Resume objekti loomisel konstruktori poolt kutsutud CreatePages meetodi poolt automaatselt, kui luuakse uus seda tüüpi objekt. Uue Report tüüpi objekti loomisel lisatakse sellele sama põhimõtte järgi teised viis Page alamklassi (vaata all olevat koodi).

```
abstract class Document
{
    protected Document() => CreatePages();

    public List<Page> Pages { get; } = new List<Page>();

    public abstract void CreatePages();
}
```

```
class Resume : Document
{
    public override void CreatePages()
    {
        Pages.Add(new SkillsPage());
        Pages.Add(new EducationPage());
        Pages.Add(new ExperiencePage());
    }
}

class Report : Document
{
    public override void CreatePages()
    {
        Pages.Add(new IntroductionPage());
        Pages.Add(new ResultsPage());
        Pages.Add(new ConclusionPage());
        Pages.Add(new SummaryPage());
        Pages.Add(new BibliographyPage());
    }
}
```

Tänu Factory Method disainimustrile võib objektide loomisel olla kindel, et sarnast tüüpi objektid luuakse järjepidevalt sama põhimõtte järgi ning hilisemad muudatused saab teha ühes kohas. Samuti on selle malli järgi lihtne luua uusi dokumendi või lehe tüüpe. Näiteks võime luua uue klassi Thesis (mis pärib Document klassilt), mille lehekülgedeks oleks Title, Abstract, TableOfContents, Introduction, jne.

Selleks, et kontrollida, kas Report ja Resume objektid on korrektselt instantsieeritud, on loodud mõlema jaoks testid. Need asuvad Tests > Creational > **FactoryMethodTests.cs**

```
[TestMethod]
public void ReportFactoryTest()
{
    var report = new Report();

    Assert.AreEqual(report.Pages.Count, 5);
    Assert.IsInstanceOfType(report.Pages[0], typeof(IntroductionPage));
    Assert.IsInstanceOfType(report.Pages[1], typeof(ResultsPage));
    Assert.IsInstanceOfType(report.Pages[2], typeof(ConclusionPage));
    Assert.IsInstanceOfType(report.Pages[3], typeof(SummaryPage));
    Assert.IsInstanceOfType(report.Pages[4], typeof(BibliographyPage));
}
```

```
[TestMethod]
public void ResumeFactoryTest()
{
    var resume = new Resume();

    Assert.AreEqual(resume.Pages.Count, 3);
    Assert.IsInstanceOfType(resume.Pages[0], typeof(SkillsPage));
    Assert.IsInstanceOfType(resume.Pages[1], typeof(EducationPage));
    Assert.IsInstanceOfType(resume.Pages[2], typeof(ExperiencePage));
}
```

Võtame vaatluse alla ReportFactoryTesti. Kõigepealt luuakse Report objekt.

```
var report = new Report();
```

Seejärel kontrollitakse, kas objektile sai lisatud korrektne arv lehekülgi.

```
Assert.AreEqual(report.Pages.Count, 5);
```

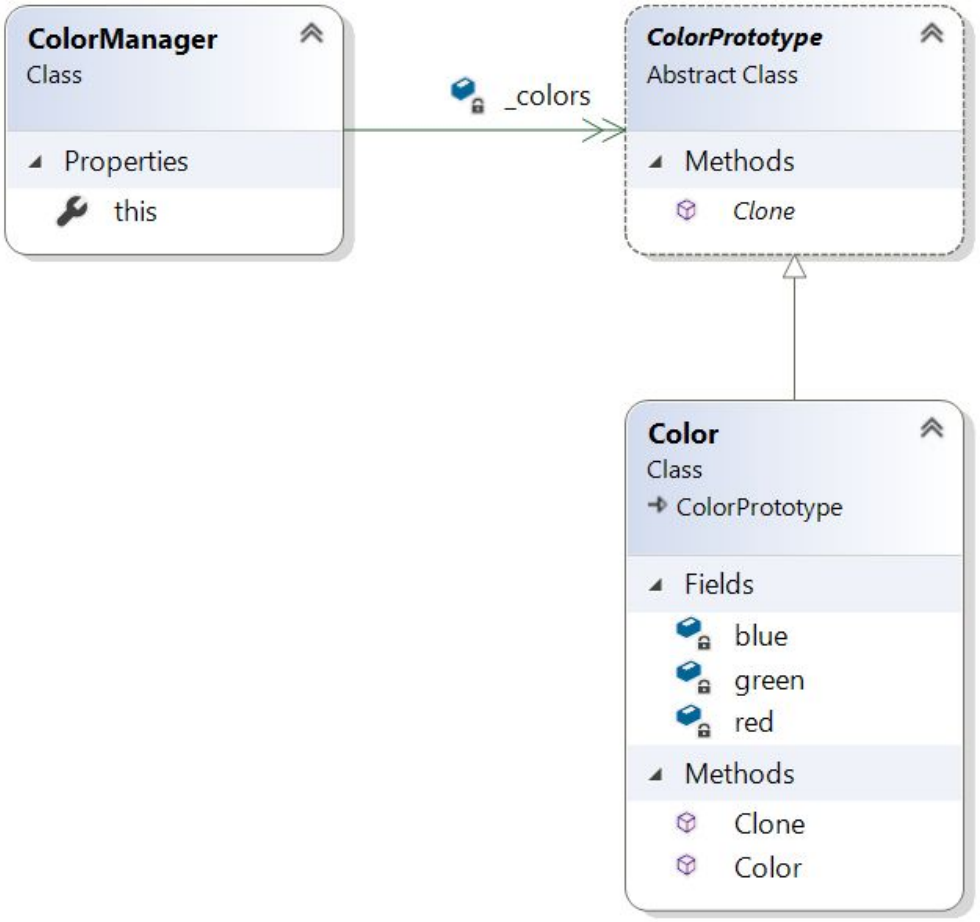
Viimaks, et iga lehekülje objekt oleks õiget tüüpi.

```
Assert.IsInstanceOfType(report.Pages[0], typeof(IntroductionPage));
Assert.IsInstanceOfType(report.Pages[1], typeof(ResultsPage));
Assert.IsInstanceOfType(report.Pages[2], typeof(ConclusionPage));
Assert.IsInstanceOfType(report.Pages[3], typeof(SummaryPage));
Assert.IsInstanceOfType(report.Pages[4], typeof(BibliographyPage));
```

Sama põhimõtte järgi toimib ka ResumeFactoryTest. Testide käivitamisel saame tulemuseks mõlemal testi läbimise.

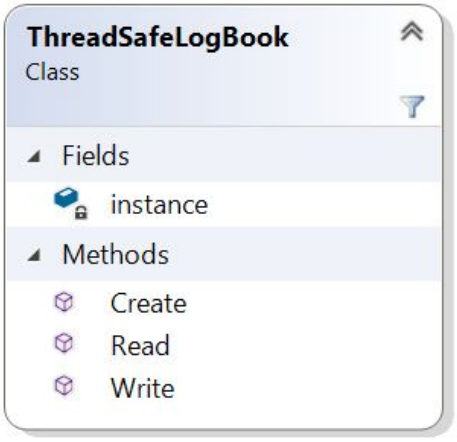
Prototype

Definitsioon: Täpsustab mis tüüpi objekte luuakse, kasutades prototüübilist isendit ning loob uued objektid kopeerides seda prototüüpi. [GoF, lk 117]



Singleton

Definitsioon: Tagab, et klassil oleks ainult üks isend ning pakub sellele globaalset ligipääsu. [GoF, lk 127]



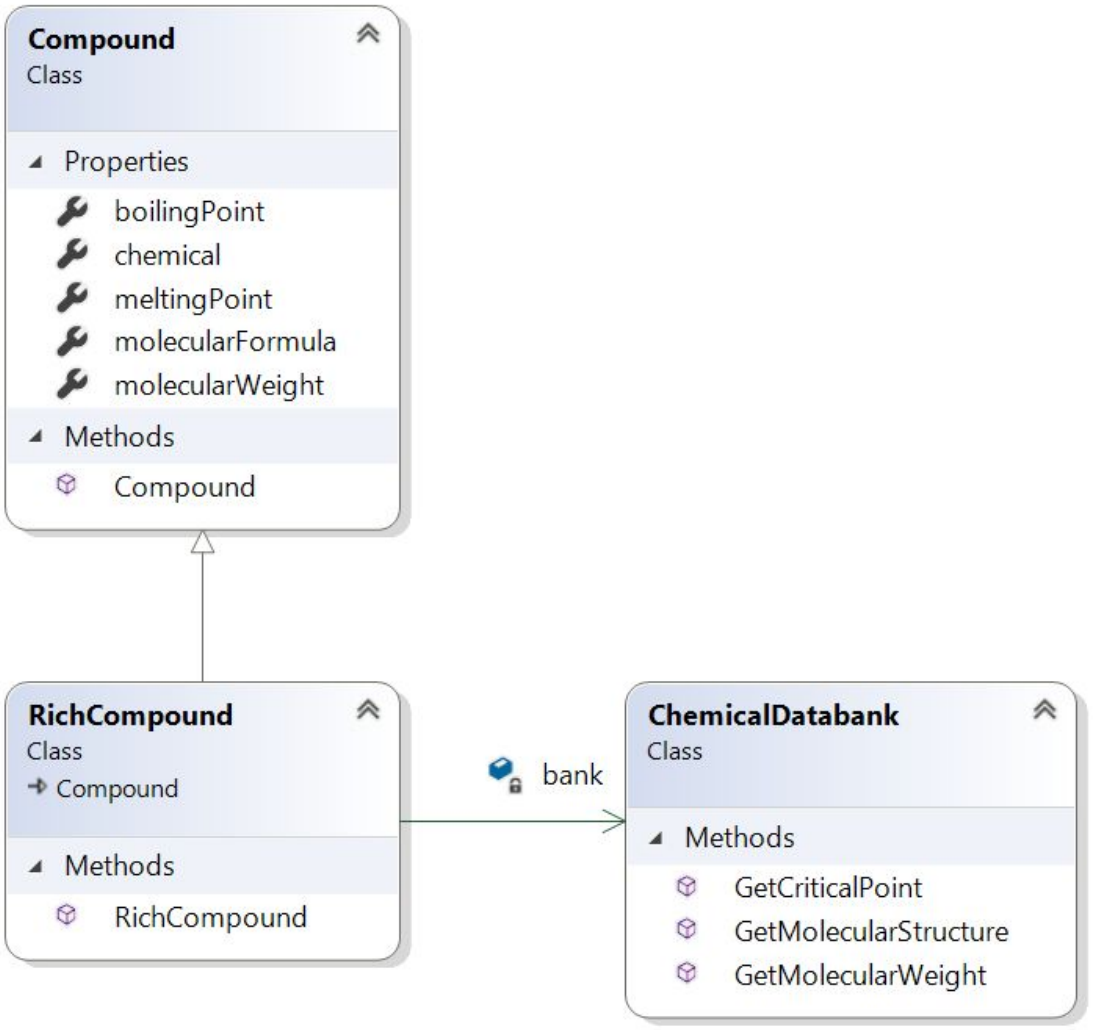
Antud juhul on kasutatud Singleton mustrit selleks, et realiseerida ühtne lõimeturvaline logiraamat.

Singletoni korral on olemas ainult üks klassi eksemplar, mida siis kõik, kellele selle klassi objekti on vaja, saavad kasutada.

Struktuurimustrid

Adapter

Definitsioon: Muundab klassi liidese kliendi poolt oodatavaks liideseks. Võimaldab klassidel koos töötada, mis muidu omavahel ei ühildu. [GoF, lk 139]



Antud näite puhul on tegu keemiliste ühendite infosüsteemiga, kus on tarvis kasutada pärandsüsteemis ChemicalDataBank paiknevaid andmeid. Andmed ei ole Compound klassile sobivas vormingus ning seepärast on andmetele ligipääsemiseks kasutatud RichCompound liidest, mis teisendab andmed sobivasse vormingusse.

Ava fail **ChemicalDataBank.cs** - siin failis olevad andmed on kättesaadavad pöördudes keemilise elemendi omadusele vastava meetodi poole. Näiteks, vee molekulaarstruktuuri saamiseks tuleb pöörduda meetodi poole GetMolecularStructure(“water”), mis tagastab

“H2O”. Vee molekulmassi, sulamispunkti ja keemispunkti jaoks tuleb samuti kutsuda välja vastavad meetodid parameetriga “water”.

```
public double GetMolecularWeight(string compound)
{
    switch (compound.ToLower())
    {
        case "water": return 18.015f;
        case "benzene": return 78.1134f;
        case "ethanol": return 46.0688f;
        default: return 0d;
    }
}
```

Meie infosüsteemis on aga tarvis, et ühe keemilise elemendi kõik omadused oleksid ühes kohas. Soovitud kuju näeb failis **Compound.cs**

```
public class Compound
{
    public string chemical { get; set; }
    public float boilingPoint { get; set; }
    public float meltingPoint { get; set; }
    public double molecularWeight { get; set; }
    public string molecularFormula { get; set; }

    public Compound(string chemical)
    {
        this.chemical = chemical;
    }
}
```

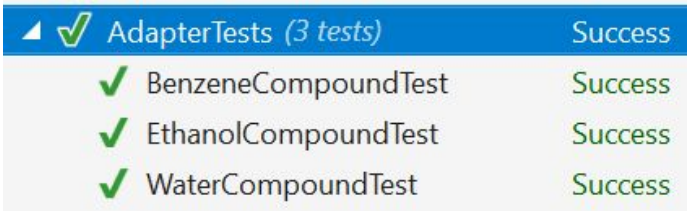
Selleks, et andmed sobivasse vormingusse saada, kasutame Adapter klassi **RichCompound.cs**, kus toimub teisendus.

```
public class RichCompound : Compound
{
    private ChemicalDatabank bank;

    public RichCompound(string name)
        : base(name)
    {
        bank = new ChemicalDatabank();

        boilingPoint = bank.GetCriticalPoint(chemical, "B");
        meltingPoint = bank.GetCriticalPoint(chemical, "M");
        molecularWeight = Math.Round(bank.GetMolecularWeight(chemical), 4);
        molecularFormula = bank.GetMolecularStructure(chemical);
    }
}
```

Tänu RichCompound Adapterile on nüüd võimalik luua uus Compound objekt vee jaoks, mis võtab andmed ChemicalDataBank klassist. Sellele saame kinnitust, kui avame **AdapterTests.cs** faili.

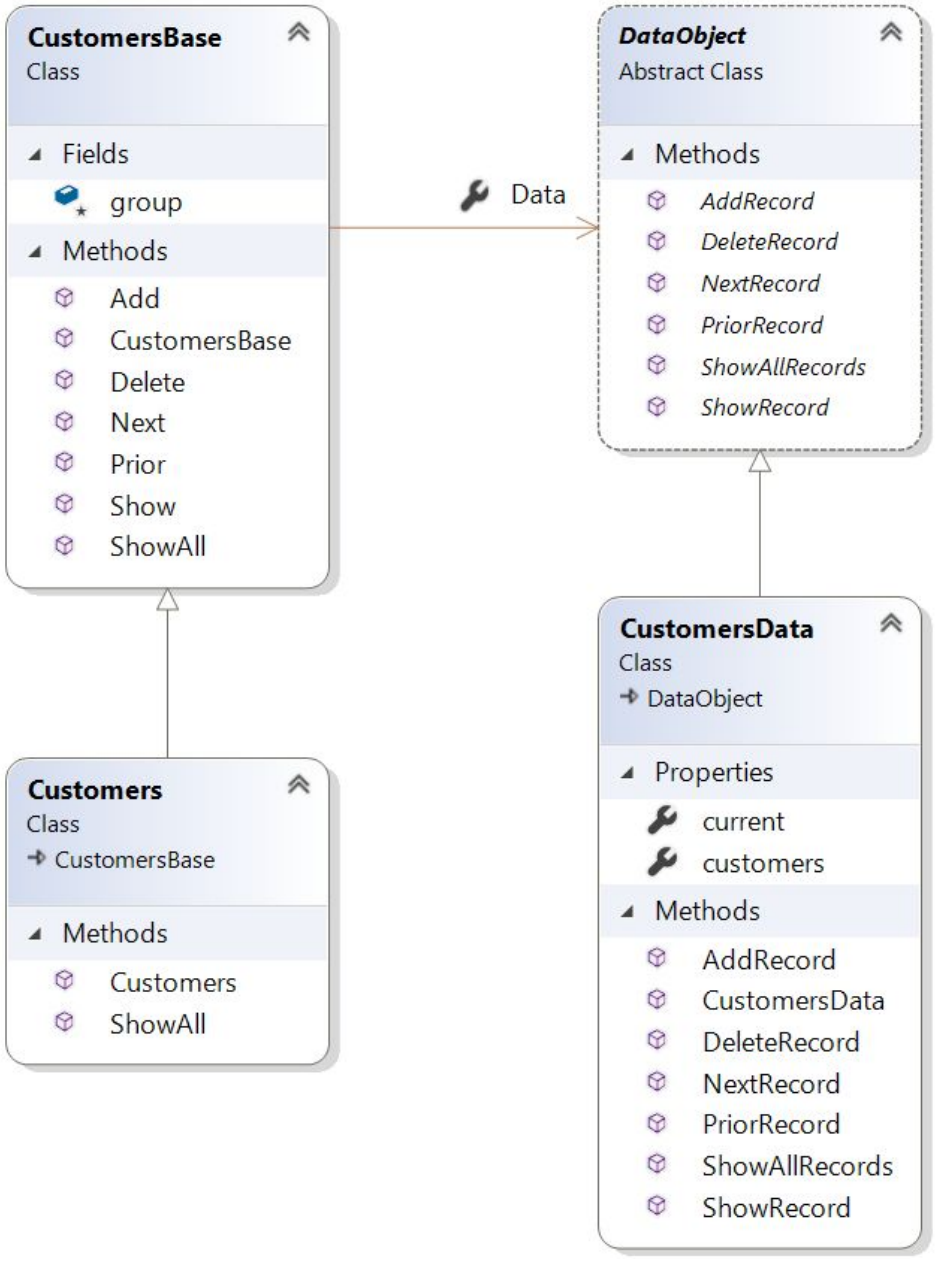
```
[TestMethod]
public void WaterCompoundTest()
{
    Compound water = new RichCompound("Water");

    Assert.AreEqual("H2O", water.molecularFormula);
    Assert.AreEqual(18.0150, water.molecularWeight);
    Assert.AreEqual(0.0f, water.meltingPoint);
    Assert.AreEqual(100.0f, water.boilingPoint);
}
```

Käivitades selle testi, näeme, et veele loodud muutujad on korrektselt väärtustatud.



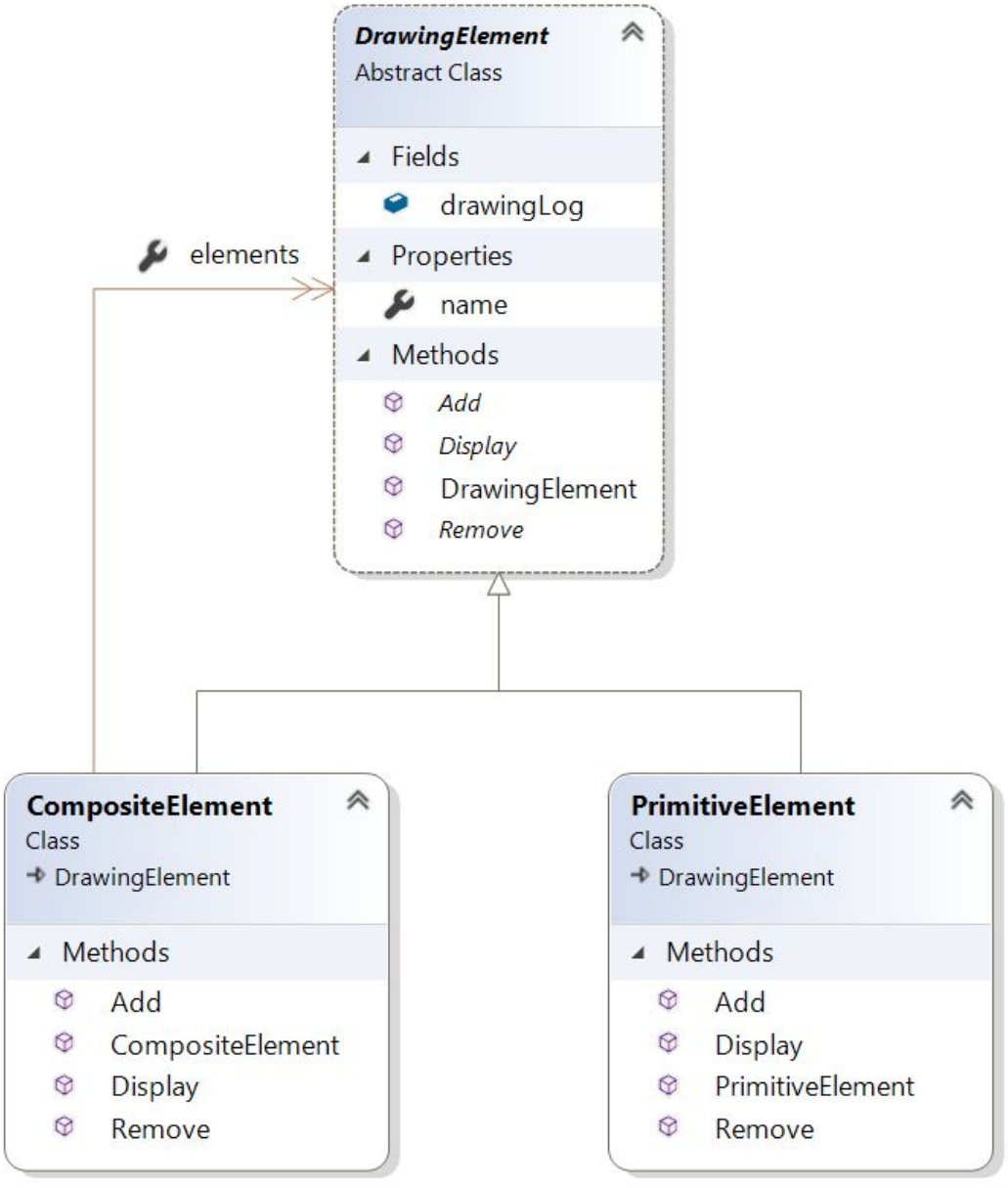
Bridge

Definitsioon: Lahutab abstraktsiooni selle implementatasioonist, nii et need saavad teineteisest sõltumatult muutuda. [GoF, lk 151]



Composite

Definitsioon: Esitab objektid puu struktuurina ja näitab nende kuuluvussuhteid. Võimaldab kliendil üheselt kohelda individuaalseid objekte ja objektide kompositsioone. [GoF, lk 163]



Antud näite puhul on kasutatud Composite mustrit, et luua joonistuselementide (DrawingElement) puu hierarhia. Puu koosneb kahte tüüpi tippudest: liittipud (CompositeElement) ja lihttipud (PrimitiveElement). Lihttipud on näiteks joon, ring, ruut vms. Liittipud on grupp lihttipu elemente, näiteks kaks ringi.

Pöördudes CompositeElementi Display() meetodi poole, kuvatakse see element ning kõik selle elemendi lapsed (ja rekursiivselt omakorda nende lapsed, kui tegemist on liittippudega).
Vaatame lähemalt koodi. Nii CompositeElement kui ka PrimitiveElement realiseerivad abstraktset klassi DrawingElement. Neil eksisteerivad samad meetodid, kuid reaalsuses elementide lisamine ja eemaldamine toimivad ainult CompositeElemendi puhul.

```
class CompositeElement : DrawingElement
{
    public List<DrawingElement> Elements { get; } =
        new List<DrawingElement>();

    public CompositeElement(string name)
        : base(name)
    {
    }

    public override void Add(DrawingElement d)
    {
        Elements.Add(d);
    }

    public override void Remove(DrawingElement d)
    {
        Elements.Remove(d);
    }

    public override void Display(int indent)
    {
        drawingLog.Add(new String('-', indent) +
                       "+ " + Name);

        foreach (DrawingElement d in Elements)
        {
            d.Display(indent + 2);
        }
    }
}
```

```
class PrimitiveElement : DrawingElement
{
    public PrimitiveElement(string name)
        : base(name)
    {
    }

    public override void Add(DrawingElement c)
    {
        throw new Exception("Cannot add to a PrimitiveElement");
    }

    public override void Remove(DrawingElement c)
    {
        throw new Exception("Cannot remove from a PrimitiveElement");
    }

    public override void Display(int indent)
    {
        drawingLog.Add(
            new String('-', indent) + " " + Name);
    }
}
```

PrimitiveElementi Add ja Remove meetodid annavad veateate, sest antud klassil ei ole alamelemente.

Composite mustri tööd illustreerivad testid failis **CompositeTests.cs**

CompositeElement objektile PrimitiveElementide lisamine:

```
[TestInitialize]
public void Initialize()
{
    root = new CompositeElement("Picture");
}

[TestMethod]
public void AddToCompositeElementTest()
{
    root.Add(new PrimitiveElement("Red Line"));
    root.Add(new PrimitiveElement("Blue Circle"));
    root.Add(new PrimitiveElement("Green Box"));

    Assert.AreEqual("Red Line", root.Elements[0].Name);
    Assert.AreEqual("Blue Circle", root.Elements[1].Name);
    Assert.AreEqual("Green Box", root.Elements[2].Name);
}
```

CompositeElement objektile teise CompositeElemendi lisamine:

```
[TestMethod]
public void CreateNestedTreeStructureTest()
{
    var comp = new CompositeElement("Two Circles");
    comp.Add(new PrimitiveElement("Black Circle"));
    comp.Add(new PrimitiveElement("White Circle"));
    root.Add(comp);
    root.Add(new PrimitiveElement("Yellow Line"));

    Assert.AreEqual(2, root.Elements.Count);
    Assert.AreEqual(typeof(CompositeElement), root.Elements[0].GetType());
    Assert.AreEqual("Two Circles", root.Elements[0].Name);
    Assert.AreEqual(typeof(PrimitiveElement), root.Elements[1].GetType());
    Assert.AreEqual("Yellow Line", root.Elements[1].Name);
}
```

CompositeElemendile elemendi lisamine ja kustutamine:

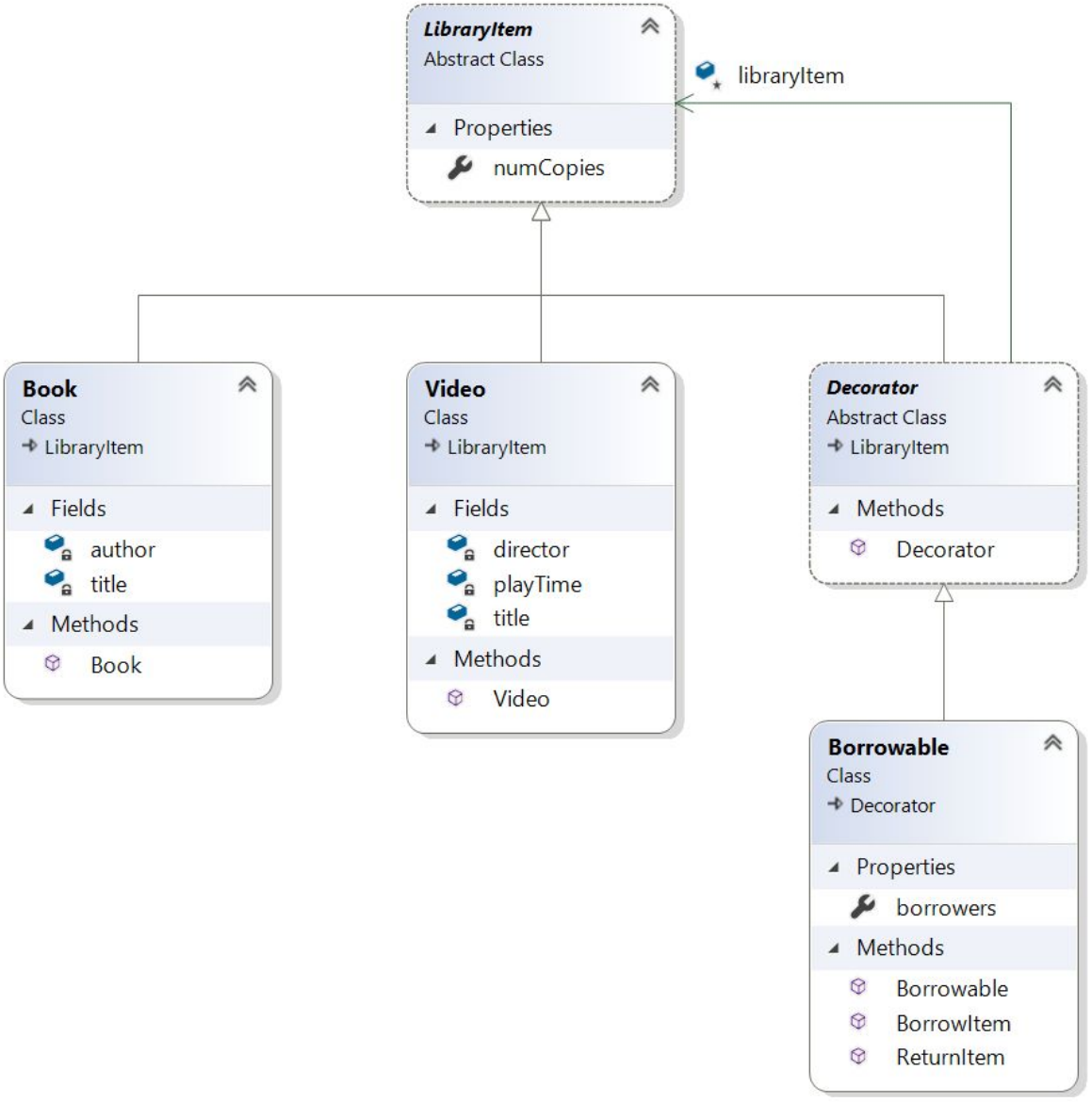
```
[TestMethod]
public void AddAndRemoveFromCompositeElementTest()
{
    var element = new PrimitiveElement("Red Line");
    root.Add(element);

    Assert.AreEqual(1, root.Elements.Count);

    root.Remove(element);

    Assert.AreEqual(0, root.Elements.Count);
}
```

PrimitiveElemendile elementide lisamine ei ole võimalik ja seega peab andma veateate:

```
[TestMethod]
[ExpectedException(typeof(Exception))]
public void AddToPrimitiveElementTest()
{
    PrimitiveElement element = new PrimitiveElement("Red Line");
    element.Add(new PrimitiveElement("White Circle"));
}
```

Decorator

Definitsioon: Määrab objektile dünaamiliselt lisa kohustusi. Pakub alamklasside loomise asemel funktsionaalsuse laiendamiseks paindlikku alternatiivi. [GoF, lk 175]



Antud näites on rakendatud Decorator mustrit, et lisada raamatukogus olevatele raamatutele (Book) ja videotele (Video) laenutamise funktsionaalsus, kasutades selleks Borrowable klassi. Ilma Decoratorit kasutamata peaks laenutamise funktsionaalsuse lisama igale eseme tüübile eraldi, mis võib erinevate klasside arvu tõttu olla ebapraktiline. Samuti ei pruugi alati olla võimalik iga klassi koodi muuta - näiteks teekidest pärit klasside puhul.

```
class Borrowable : Decorator
{
    public List<string> Borrowers { get; } = new List<string>();

    public Borrowable(LibraryItem libraryItem)
        : base(libraryItem)
    {
    }

    public void BorrowItem(string name)
    {
        Borrowers.Add(name);
        libraryItem.NumCopies--;
    }

    public void ReturnItem(string name)
    {
        Borrowers.Remove(name);
        libraryItem.NumCopies++;
    }
}
```

Raamatutel ja filmidel eksisteerib väärtus numCopies, mis näitab kui mitu eset raamatukogus on. Kui raamat või film on dekoreeritud Borrowable klassiga ja kasutada sellega kaasnevat meetodit BorrowItem(), siis väheneb algse objekti numCopies väärtus. Seda illustreerib test failis **DecoratorTests.cs**

```
[TestMethod]
public void BorrowItemTest()
{
    Book book = new Book("Worley", "Inside ASP.NET", 10);
    Borrowable borrowableBook = new Borrowable(book);

    int initialAvailableCopies = book.NumCopies;
    borrowableBook.BorrowItem(customer);

    Assert.AreEqual(initialAvailableCopies - 1, book.NumCopies);
    CollectionAssert.Contains(borrowableBook.Borrowers, customer);
}
```

Esemete tagastamist illustreerib järgnev test meetod.

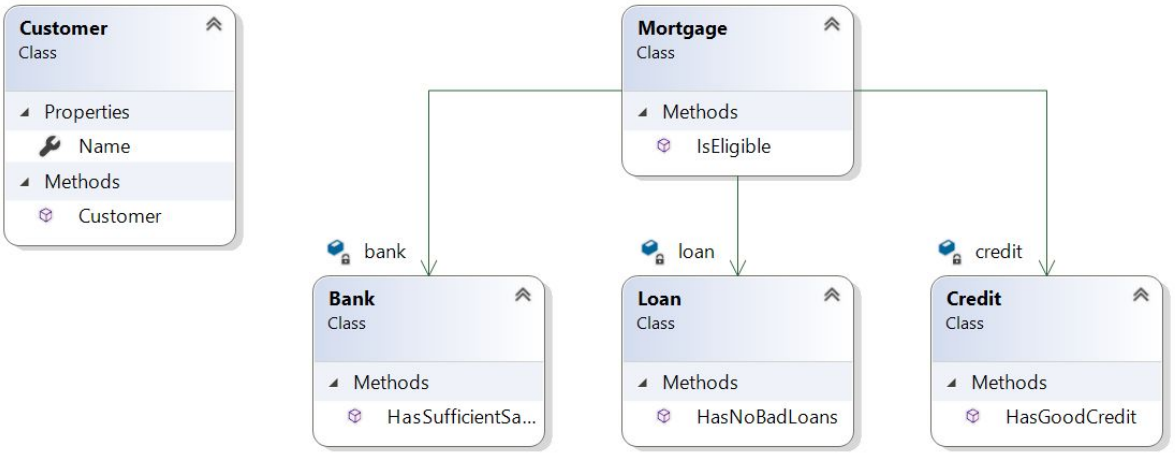
```
[TestMethod]
public void ReturnItemTest()
{
    borrowableVideo.Borrowers.Add(customer);

    int initialAvailableCopies = video.NumCopies;
    borrowableVideo.ReturnItem(customer);

    Assert.AreEqual(initialAvailableCopies + 1, video.NumCopies);
    CollectionAssert.DoesNotContain(borrowableVideo.Borrowers, customer);
}
```

Selleks, et testida Decorator mustri tööd, on loodud abimeetod, mis kontrollib meetodi olemasolu klassis. LibraryItem klass (ilma Decoratorita) ei oma meetodit BorrowItem, kuid dekoreerides selle Borrowable'ga on meetod olemas. See on välja toodud järgnevas testis.

```
[TestMethod]
public void BorrowableItemHasCorrectMethodTest()
{
    Assert.IsFalse(HasMethod(video, "BorrowItem"));
    Assert.IsTrue(HasMethod(borrowableVideo, "BorrowItem"));
}

private bool HasMethod(LibraryItem objectToCheck, string methodName)
{
    var type = objectToCheck.GetType();
    return type.GetMethod(methodName) != null;
}
```

Facade

Definitsioon: Pakub alamsüsteemi liideste hulgale ühtset liidest. Defineerib kõrgema taseme liidese, mis teeb alamsüsteemi kasutamise lihtsamaks. [GoF, lk 185]



Antud näites on tegemist eluasemelaenu kõlbulikkust hindava rakendusega. Siin kasutatakse Facade mustrit, et pakkuda lihtsustatud liidest (Mortgage klass) hulgale alamsüsteemidele (klassid Bank, Loan ja Credit).

```
class Bank
{
    public bool HasSufficientSavings(Customer c, int amount)
    {
        return true;
    }
}

class Loan
{
    public bool HasNoBadLoans(Customer c)
    {
        return true;
    }
}

class Credit
{
    public bool HasGoodCredit(Customer c)
    {
        return true;
    }
}
```

Tänu Mortgage klassile on võimalik sisestada laenuvõtja andmed ühte kohta, selle asemel, et laenukõlbulikkust kolmes kohas eraldi kontrollida.

```
class Mortgage
{
    private readonly Bank bank = new Bank();
    private readonly Loan loan = new Loan();
    private readonly Credit credit = new Credit();

    public bool IsEligible(Customer customer, int amount)
    {
        bool eligible = true;

        if (!bank.HasSufficientSavings(customer, amount))
        {
            eligible = false;
        }
        else if (!loan.HasNoBadLoans(customer))
        {
            eligible = false;
        }
        else if (!credit.HasGoodCredit(customer))
        {
            eligible = false;
        }

        return eligible;
    }
}
```

Mustri töö kontrollimiseks on loodud testid failis **FacadeTests.cs.** CustomerIsEligibleForMortgageTest kontrollib, kas klient on laenuvõimeline.

```
[TestInitialize]
public void Initialize()
{
    customer = new Customer("Ann McKinsey");
    defaultAmount = 125000;
}

[TestMethod]
public void CustomerIsEligibleForMortgageTest()
{
    Mortgage mortgage = new Mortgage();

    bool eligible = mortgage.IsEligible(customer, defaultAmount);

    Assert.IsTrue(eligible);
}
```

Laenuvõimekuse alamosade kontrollimiseks on loodud testid CustomerHasSufficient-SavingsTest, CustomerHasGoodCreditTest ja CustomerHasNoBadLoansTest.

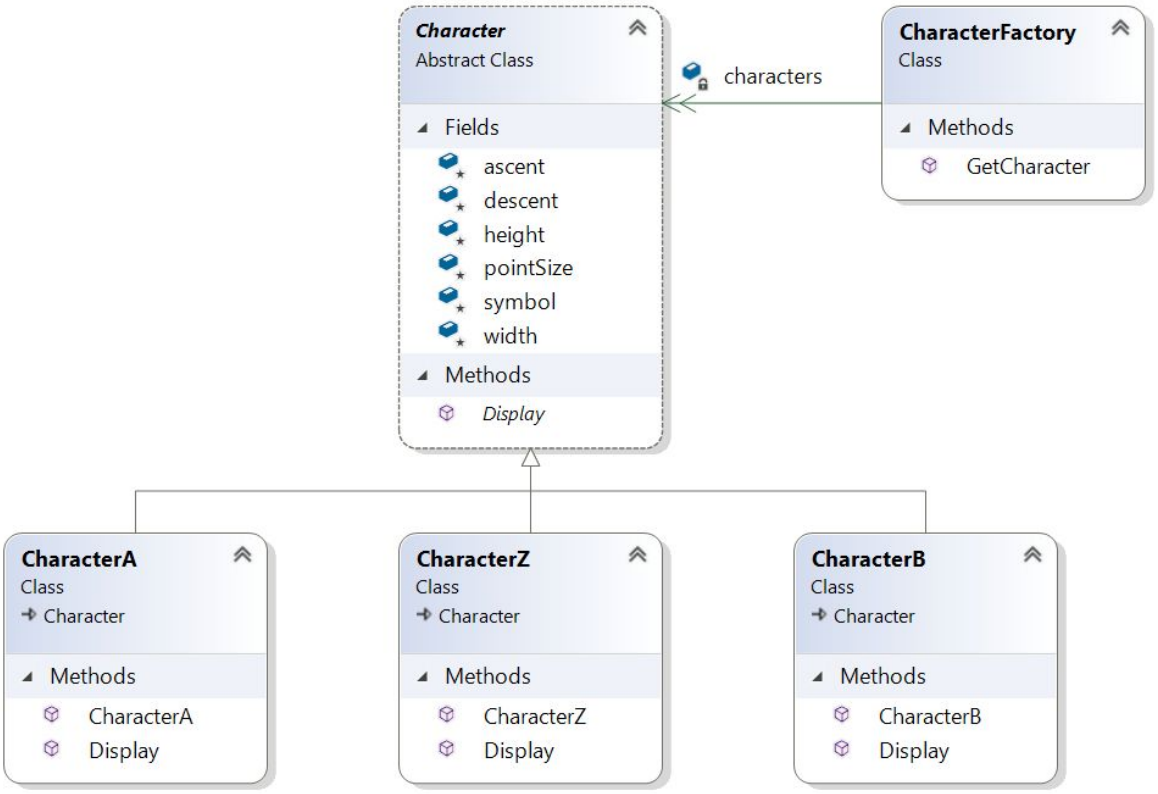
```
[TestMethod]
public void CustomerHasSufficientSavingsTest()
{
    Bank bank = new Bank();

    bool hasSufficientSavings = bank.HasSufficientSavings(customer, defaultAmount);

    Assert.IsTrue(hasSufficientSavings);
}
```

```
[TestMethod]
public void CustomerHasGoodCreditTest()
{
    Credit credit = new Credit();

    bool hasGoodCredit = credit.HasGoodCredit(customer);

    Assert.IsTrue(hasGoodCredit);
}
```

```
[TestMethod]
public void CustomerHasNoBadLoansTest()
{
    Loan loan = new Loan();

    bool hasNoBadLoans = loan.HasNoBadLoans(customer);

    Assert.IsTrue(hasNoBadLoans);
}
```

Flyweight

Definitsioon: Võimaldab suure hulga detailsete objektide efektiivset jagamist. [GoF, lk 195]



Flyweight muster võimaldab objektide jagamist, hoides sellega kokku mälumahtu. Näiteks, dokumendiredaktoris (MS Word, LibreOffice, jt) kasutavaid tähemärke on juba väiksemas dokumendis tuhandeid. Igaühe jaoks eraldi objekti loomine oleks arvutuslikult väga kulukas. Sellisel juhul on abi *flyweight* objektist (Character klass) ehk jagatud objektist, mida on võimalik kasutada erinevates kontekstides samaaegselt.

Käesolevas ülesandes on hulk tähestiku tähti, mida kasutatakse tähtede ükshaaval kuvamiseks. Iga tähe kuvamise eest vastutab temale vastav klass. Tähe klassi objektide loomise eest kannab hoolt CharacterFactory klass. Kui mingit tähte on juba kord kasutatud, siis CharacterFactory tagastab olemasoleva tähe, mitte ei loo enam uut sarnast objekti.

Näide: Eksisteerib dokument sisuga ‘ABBZ’, milles olevad tähed on vaja ükshaaval kuvada. Selle jaoks kasutatakse iga tähe jaoks CharacterFactory klassi meetodit GetCharacter(*character*), kus argument *character* on dokumendis olev täht. Toimub neli meetodit kutset:

1. GetCharacter(‘A’) - CharacterFactory loob uue CharacterA klassi objekti ja tagastab selle.
2. GetCharacter(‘B’) - CharacterFactory loob uue CharacterB klassi objekti ja tagastab selle.

3. GetCharacter('B') - CharacterFactory tagastab äsja loodud CharacterB klassi objekti (ja ei loo uut klassi objekti).
4. GetCharacter('Z') - CharacterFactory loob uue CharacterZ klassi objekti ja tagastab selle.

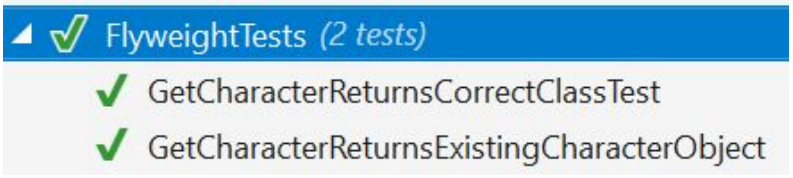
Failis **FlyweightTests.cs** asuvad testid, mis kontrollivad ülaltoodud mustri omadusi. Esimene test GetCharacterReturnsCorrectClassTest kontrollib, et kui luua CharacterFactory'ga objekt, kasutades parameetrina tähte 'A', siis luuakse CharacterA objekt.

```
[TestMethod]
public void GetCharacterReturnsCorrectClassTest()
{
    char charA = 'A';
    CharacterFactory factory = new CharacterFactory();

    Character character = factory.GetCharacter(charA);

    Assert.IsInstanceOfType(character, typeof(CharacterA));
}
```

Teine test, GetCharacterReturnsExistingCharacterObject, kontrollib, et CharacterFactory teiskordsel kutsumisel parameetriga 'A', tagastataks esmalt loodud objekt (ehk ei looda mälus uut sama tüüpi objekti, vaid tagastatakse viide eelmisele).

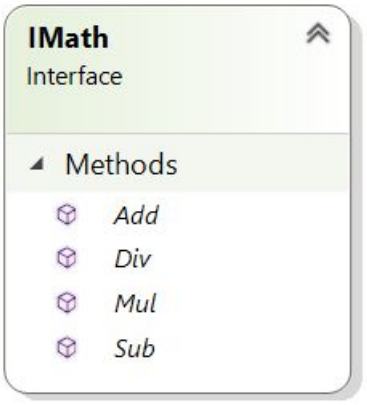
```
[TestMethod]
public void GetCharacterReturnsExistingCharacterObject()
{
    char firstA = 'A';
    char secondA = 'A';
    CharacterFactory factory = new CharacterFactory();

    Character firstResult = factory.GetCharacter(firstA);
    Character secondResult = factory.GetCharacter(secondA);

    Assert.AreSame(firstResult, secondResult);
}
```

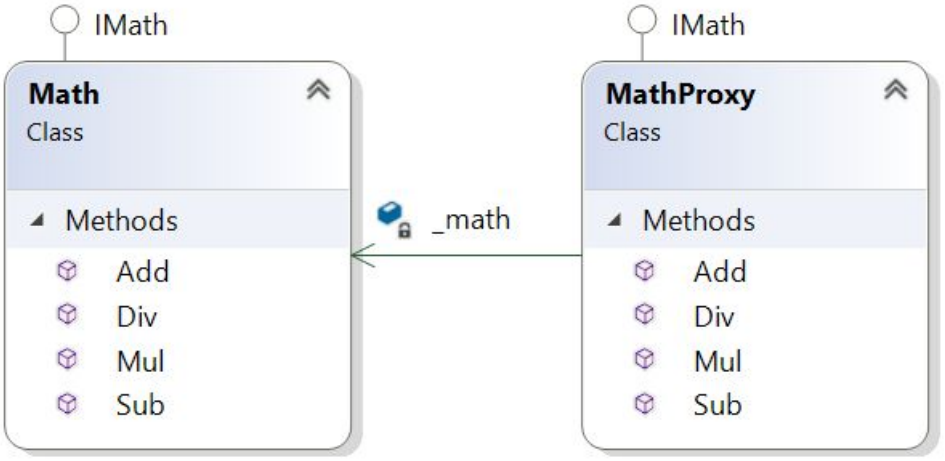
Testide käivitamine annab positiivse vastuse.



Proxy

Definitsioon: Pakub asendust või kohahoidjat teisele objektile, eesmärgiga kontrollida selle objekti juurdepääsu. [GoF, lk 207]

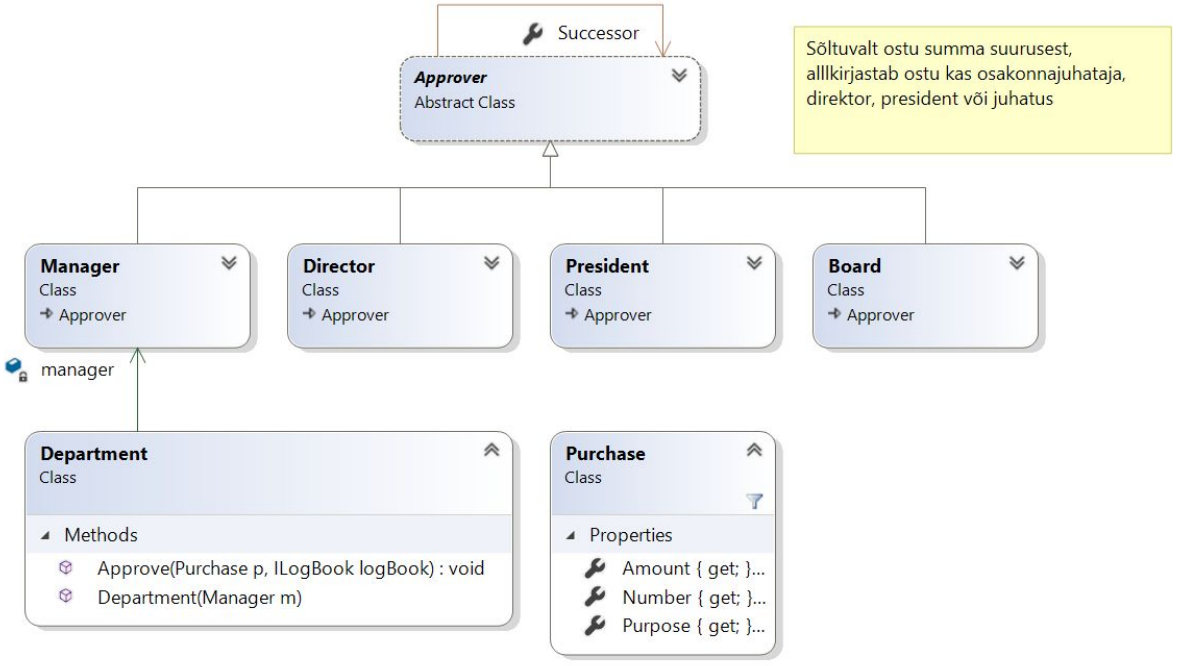




Käitumuslikud mustrid

Chain of responsibility

Definitsioon: Hoiab ära päringu saatja sidumist selle saajaga, andes rohkem kui ühele objektile võimaluse päringut käidelda. Aheldab saaja rollis olevad objektid ja annab päringu mööda ahelat edasi, kuniks leidub objekt, kes päringu lahendab. [GoF, lk 223]



Antud näites kasutatakse Chain of Responsibility disainimustrit tellimuste kinnitamiseks vastavalt tehingu suurusele. Infosüsteemis on neli erineva taseme juhti, kõigil maksimaalne tehingu summa, mida nad tohivad kinnitada. Kõigepealt jõuab tellimus osakonnajuhataja (Manager) kätte ja kui tellimuse summa jääb tema volituse piiridesse, siis ta kinnitab selle. Vastasel juhul suunab ta tellimuse edasi oma ülemusele, direktorile (Director). Kui tellimuse summa jääb direktori volituse piiridesse, siis ta kinnitab selle, vastasel juhul suunab temagi selle enda ülemusele (President). Sama põhimõtte järgi liigub tellimus juhtkonna hierarhiat pidi üles, kuni juhatuseni (Board) välja, kes saavad allkirjastada kõige suuremaid tellimusi.

Eelnimetatud juhi klassid on päritud kõik klassist Approver. Vaatame seda lähemalt.

```
public abstract class Approver {

    protected Approver(Approver successor, double liability) {
        Successor = successor;
        Liability = liability;
    }

    public Approver Successor { get; }

    public double Liability { get; }

    public virtual void ProcessRequest(Purchase purchase, ILogBook logBook) {
        if (CanApprove(purchase.Amount))
            logBook.WriteLine(ApprovalMsg(purchase.Number, purchase.Purpose));
        else
            Successor?.ProcessRequest(purchase, logBook);
    }

    internal string ApprovalMsg(int number, string purpose) {
        return $"{GetType().Name} approved request# {number} {purpose}";
    }

    internal bool CanApprove(double amount) { return amount <= Liability; }
}
```

Klass Approver saab parameetrina ülemuse (Successor) ja tellimuse summa ülempiiri (Liability). Vastavalt juhi rolli volitusele, ProcessRequest meetod kas kinnitab tellimuse (kirjutades kande logiraamatusse) või suunab selle hierarhiat pidi üles poole.

Disainimustri testimiseks vaatame testide klassi **ChainOfResponsibilityTests.cs.**

```
[TestClass]
public class ChainOfResponsibilityTests {
    private const double tenThousand = 10000;
    private const double hundredThousand = 100000;
    private const double oneMillion = 1000000;

    private Department department;
    private Board board;
    private President president;
    private Director director;
    private Manager manager;

    private LogBook logBook;
    private string purpose;
    private int number;

    [TestInitialize] public void Initialize() {
        board = new Board();
        president = new President(board, oneMillion);
        director = new Director(president, hundredThousand);
        manager = new Manager(director, tenThousand);
        department = new Department(manager);
        logBook = new LogBook();
        number = GetRandom.Int16(0);
        purpose = GetRandom.String();
    }
```

Testide jaoks on väärtustatud hulk muutujaid. Korrektse kinnitaja kindlaks tegemiseks on loodud privaatne abimeetod TestApproval, mis genereerib etteantud vahemikus (min, max) tellimuse summa ja loob nende põhjal uue tellimuse objekti. Seejärel alustab tellimuse protsessimist edastades selle osakonnale (Department). Selle jaoks pöördutakse muutuja *department* Approve meetodi poole, andes parameetritena loodud tellimuse (muutuja *p*) ning logiraamatu (*logBook*), kuhu kinnitus logitakse.

```
private void TestApproval(Approver approver, double min, double max = double.MaxValue) {
    var amount = GetRandom.Double(min, max);
    var p = new Purchase(number, amount, purpose);

    department.Approve(p, logBook);

    Assert.AreEqual(approver.ApprovalMsg(number, purpose), logBook.ReadLine());
}
```

Esimest testi (ApprovedByDirectorTest) vaadates näeme, et toimub TestApproval meetodi poole pöördumine, parameetriteks *director* muutuja ning minimaalne ja maksimaalne vahemik, kus juhuarv genereeritakse.

```
[TestMethod] public void ApprovedByDirectorTest() {
    TestApproval(director, tenThousand, hundredThousand);
}
```

Jälgime TestApprovalis loodud tellimuse objekti käiku läbi juhatajate ahela. Tellimus saadetakse kõigepealt osakonda. Seal luuakse esimese astme juhi (Manager) objekt ning kutsutakse sellega ProcessRequest meetodit.

```
public class Department {

    private readonly Manager manager;

    public Department(Manager m) { manager = m; }

    public void Approve(Purchase p, ILogBook logBook) {
        manager.ProcessRequest(p, logBook);
    }

}
```

Selle tulemusena jõuab tellimus Manageri kätte. Manager on volitatud kinnitama tellimusi, mis ei ole suuremad, kui kümme tuhat, seega saadetakse tellimus tema ülemusele - direktorile.

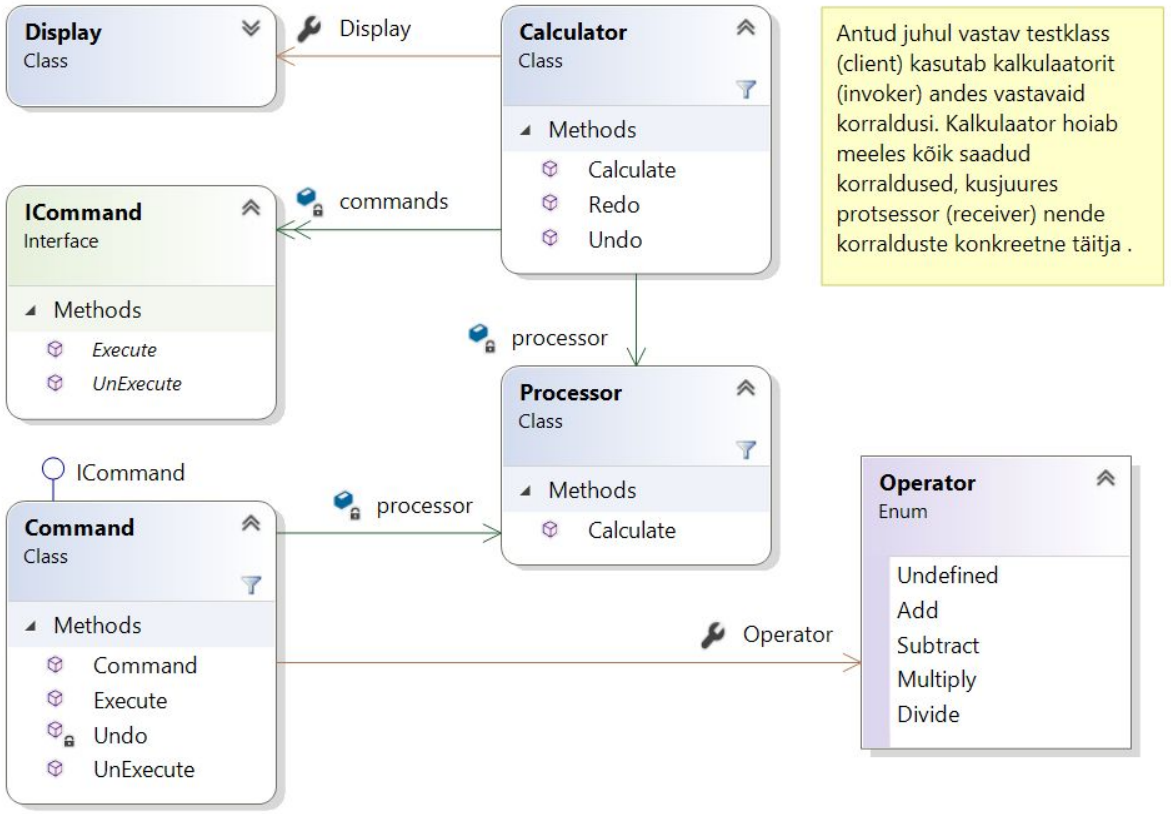
```
public class Manager : Approver {

    public Manager(Approver successor, double liability) : base(successor, liability) { }
}
```

Direktor on volitatud kinnitama tellimusi maksimaalse summaga sada tuhat, seega käesolev tellimus (mis sai genereeritud vahemikus 10000 - 100000) jääb tema kinnitamise piiridesse ja järgmisele ülemusele edasi ei saadeta. Director kirjutab tellimuse kinnitamise logiraamatusse.

```
logBook.WriteLine(ApprovalMsg(purchase.Number, purchase.Purpose));
```

Command

Definitsioon: Kapseldab päringu objektina, võimaldades kliente erinevate päringute parameetritega varustada, päringuid järjestada või logida ja teha tühistatavaid operatsioone. [GoF, lk 223]



Antud näites on kasutatud Command mustrit, et võimaldada kalkulaatoriga teha tagasivõtmise (Undo) ja uuestitegemise (Redo) operatsioone. Iga sellise operatsiooni jaoks luuakse eraldi objekt, mis sisaldab endas kogu vajalikku infot.

Mustri tööpõhimõtte uurimiseks ava **CommandTests.cs**. Vaatame kõigepealt Initialize() meetodit, mis loob vajalikud tingimused testide koostamiseks.

```
private Calculator calculator;

[TestInitialize] public void Initialize() {
    calculator = new Calculator();
    calculator.Calculate(Operator.Add, 100);
    calculator.Calculate(Operator.Subtract, 50);
    calculator.Calculate(Operator.Multiply, 10);
    calculator.Calculate(Operator.Divide, 2);
}
```

Kõigepealt luuakse ja initsialiseeritakse uus Calculator objekt. Seejärel kutsutakse välja selle Calculate meetod parameetritega Operator.Add ja 100. Operator on loend, kus on võimalikud kalkulaatori poolt teostatavad tehted.

```
public enum Operator {
    Undefined,
    Add,
    Subtract,
    Multiply,
    Divide
}
```

Calculator klassi Calculate meetodi poole pöördudes, luuakse uus Command klassi objekt. Seejärel kutsutakse selle Execute meetod, mis salvestatakse Calculator klassi Display-tüüpi muutujasse. Display väärtus on kalkulaatori 'ekraanil' olev väärtus. Edasi lisatakse see Command objekt käskluste loendisse (commands) ja tõstetakse loenduri (counter) väärtust ühe võrra.

Käskluste loendisse lisamine võimaldabki tagasivõtmise ja uuestitegemise operatsioone teha.

```
[TestMethod] public void UndoTest() {
    calculator.Undo(4);
    Assert.AreEqual(0, calculator.Display.Result);
}
```

Calculator klassi Undo meetodit saab kutsuda täisarvulise parameetriga, mis määrab, kui mitu käsklust tagasi võetakse.

```
public void Undo(int levels) {
    for (var i = 0; i < levels; i++) {
        if (counter <= 0) continue;
        var command = commands[--counter];
        Display = command.UnExecute();
    }
}
```

Siit on näha, et liigutakse commands loendis tagasi, valitakse käsklus ning kutsutakse selle UnExcecute meetod. Seda korratakse nii mitu korda, kui mitu sammu (levels) parameetrina sisestati.

```
public class Command : ICommand {

    private readonly Processor processor;

    public Command(Processor c, Operator o, decimal operand) {
        processor = c;
        Operator = o;
        Operand = operand;
    }

    public Operator Operator { get; }

    public decimal Operand { get;}

    public Display Execute() { return processor.Calculate(Operator, Operand); }

    public Display UnExecute() { return processor.Calculate(Undo(Operator), Operand); }

    private static Operator Undo(Operator o) {
        switch (o) {
            case Operator.Add: return Operator.Subtract;
            case Operator.Subtract: return Operator.Add;
            case Operator.Multiply: return Operator.Divide;
            case Operator.Divide: return Operator.Multiply;
            default:
                return Operator.Undefined;
        }
    }
}
```

Command klassi UnExecute meetodist on näha, et kutsutakse Processor klassi Calculate meetod parameetriga Undo(Operator). Undo(Operator) tagastab salvestatud käskluse operatsiooni vastandoperatsiooni ehk kui esialgselt tehti liitmistehe, siis Undo meetodi poole pöördumine tagastab lahutamistehte.

```
public class Processor {
    private decimal result;
    public Display Calculate(Operator o, decimal operand) {
        var r = result;
        switch (o) {
            case Operator.Add:
                result += operand;
                break;
            case Operator.Subtract:
                result -= operand;
                break;
            case Operator.Multiply:
                result *= operand;
                break;
            case Operator.Divide:
                result /= operand;
                break;
        }
        return new Display(r, o, operand, result);
    }
}
```

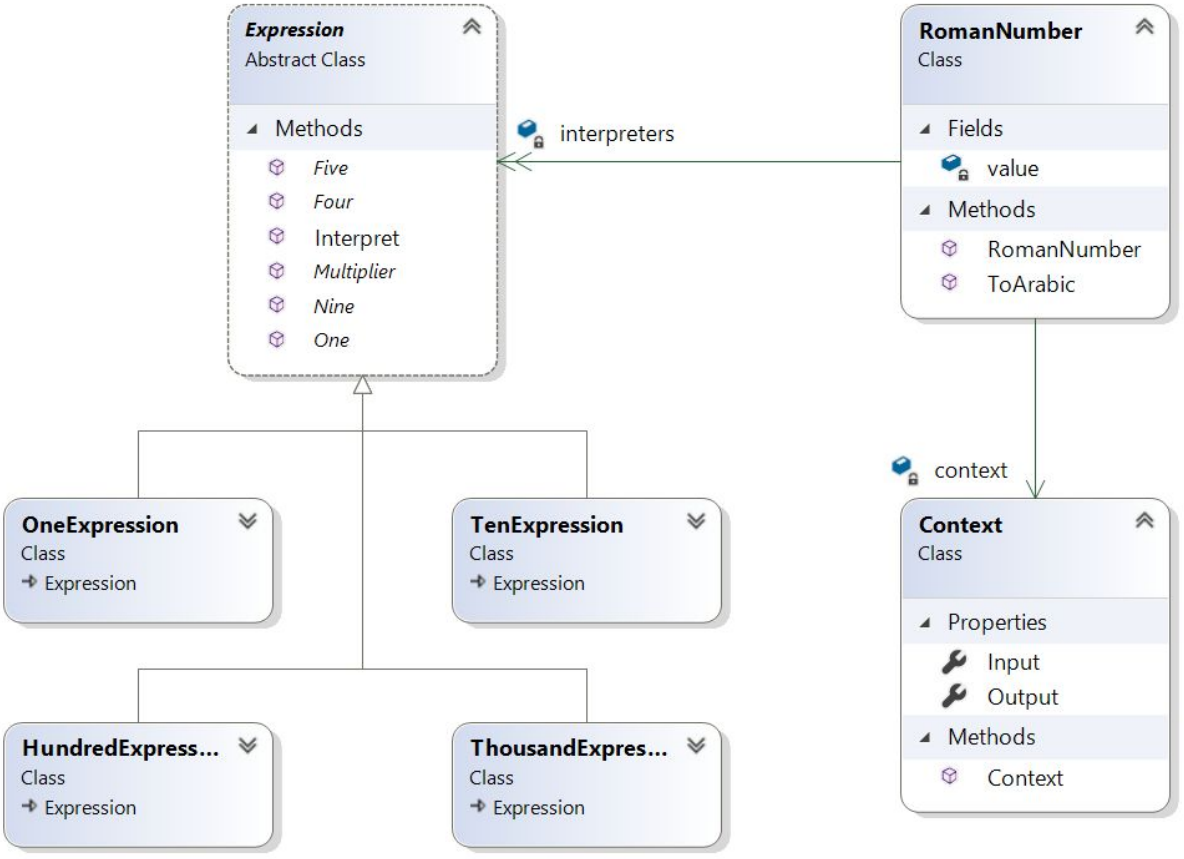
Processor klassi ülesanne on kanda hoolt arvutustehete sooritamise eest. Saanud sisendina sooritatava operatsiooni (Operator) ja arvulise väärtuse (operand), teostatakse vastav tehe ning tagastatakse uus Display objekt koos tulemusega.

Kuna UndoTesti parameetriks oli 4, siis korratakse kogu protsessi 4 korda ning lõpptulemuseks on 4 tehet tagasi olnud väärtus ehk algväärtus 0.

Sama põhimõtte järgi, kuid vastupidiselt, töötab ka kalkulaatori uuestitegemise (Redo) operatsioon.

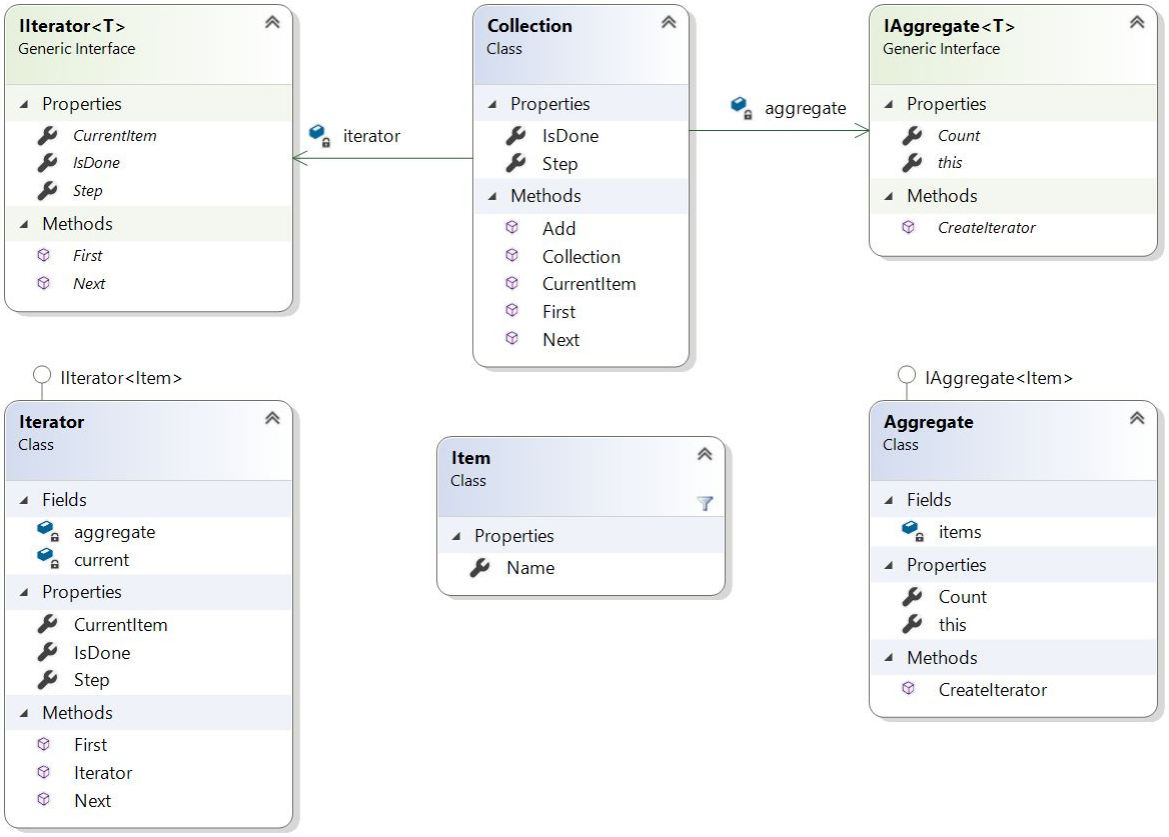
Interpreter

Definitsioon: Defineerib antud keele puhul esituse keele grammatikale ning tõlgendaja, mis kasutab esitust selle keele lausete tõlgendamiseks. [GoF, lk 243]



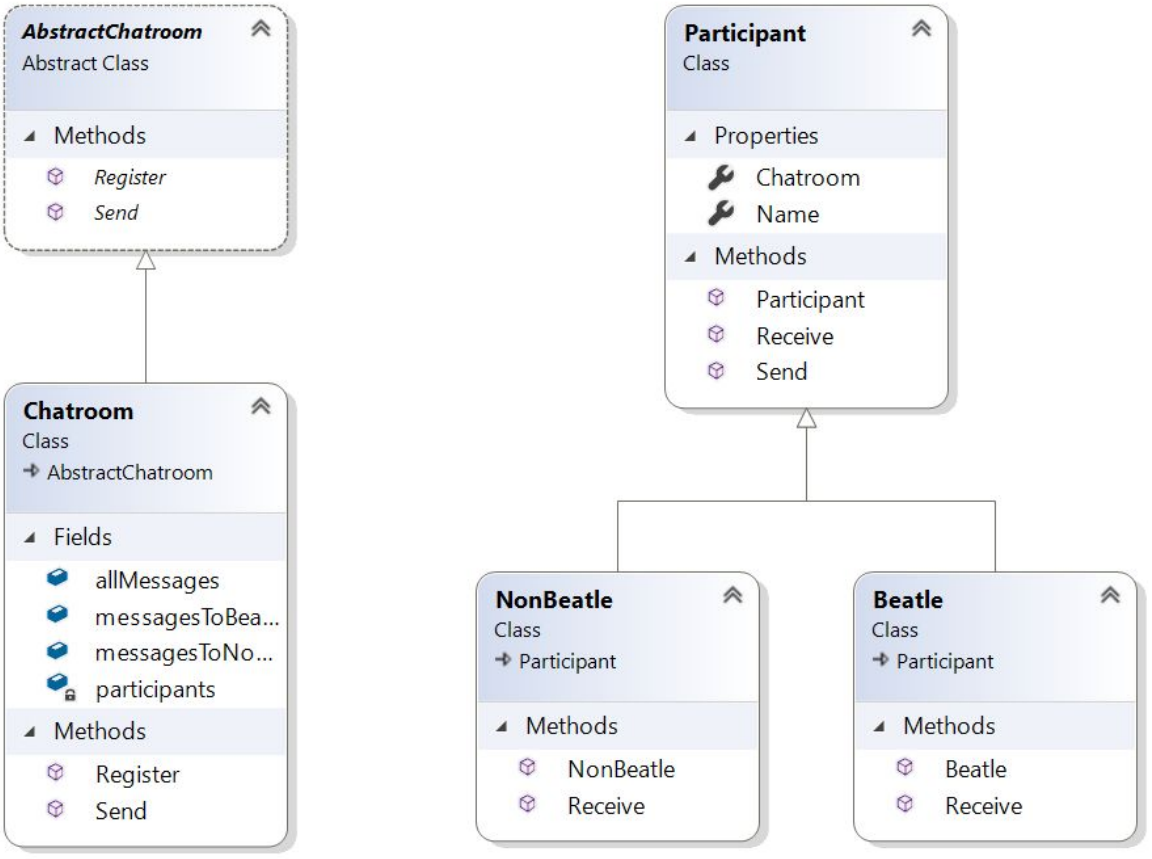
Iterator

Definitsioon: Võimaldab agregaat-objekti elementide poole järjestikku pöördumise, ilma selle aluseks olevat esitust avalikustamata. [GoF, lk 257]



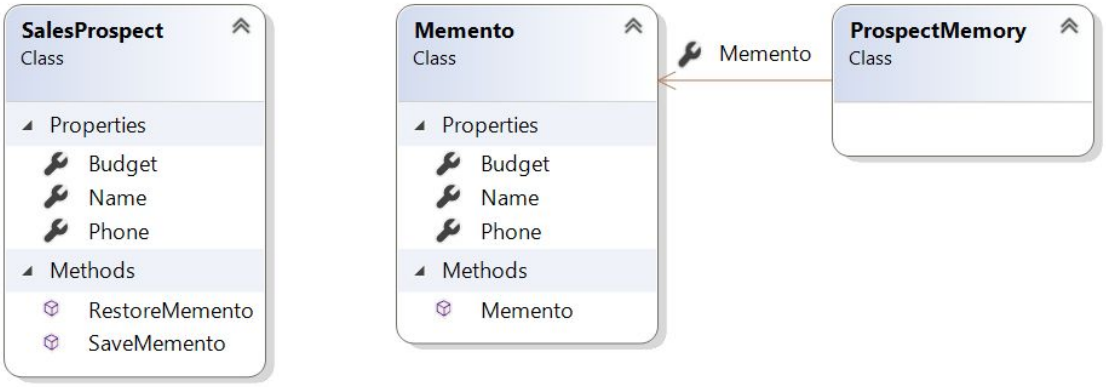
Mediator

Definitsioon: Defineerib objekti, mis kapseldab objektidevahelise suhtluse. Põhineb lõdval sidestusel, väldib objektide omavahelist otsest viitamist ja võimaldab üksteisest sõltumatult muuta nende suhtlust. [GoF, lk 273]



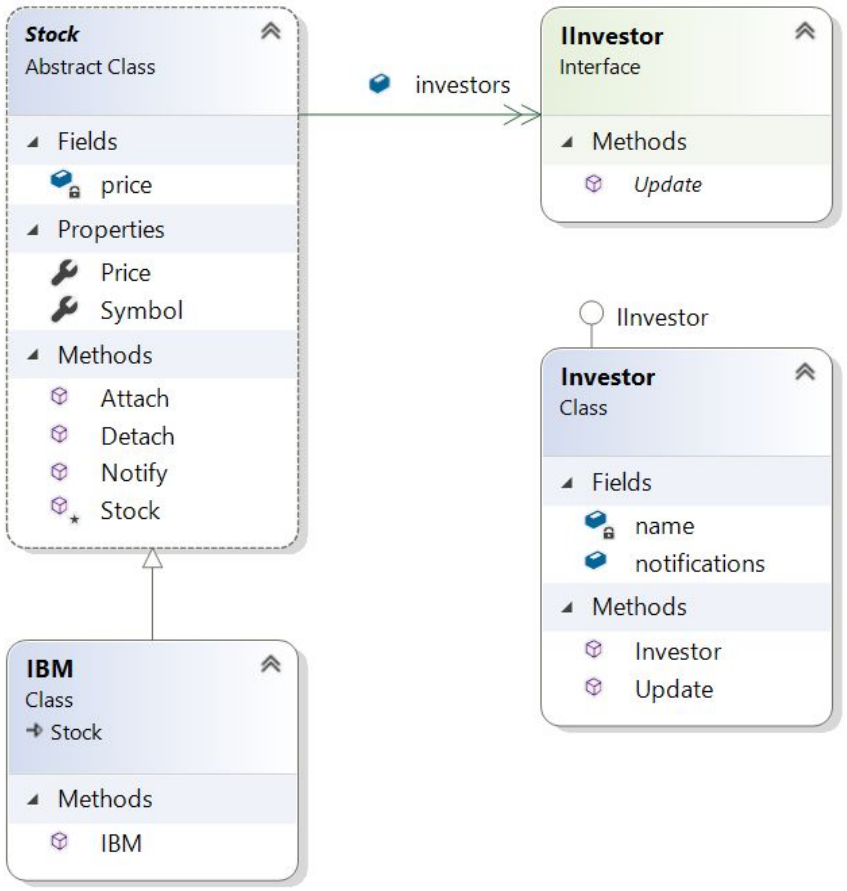
Memento

Definitsioon: Kasutades kapseldamist, salvestab ja väljastab objekti sisemise oleku, nii et objekti oleks hiljem võimalik sellesse olekusse tagasi tuua. [GoF, lk 283]



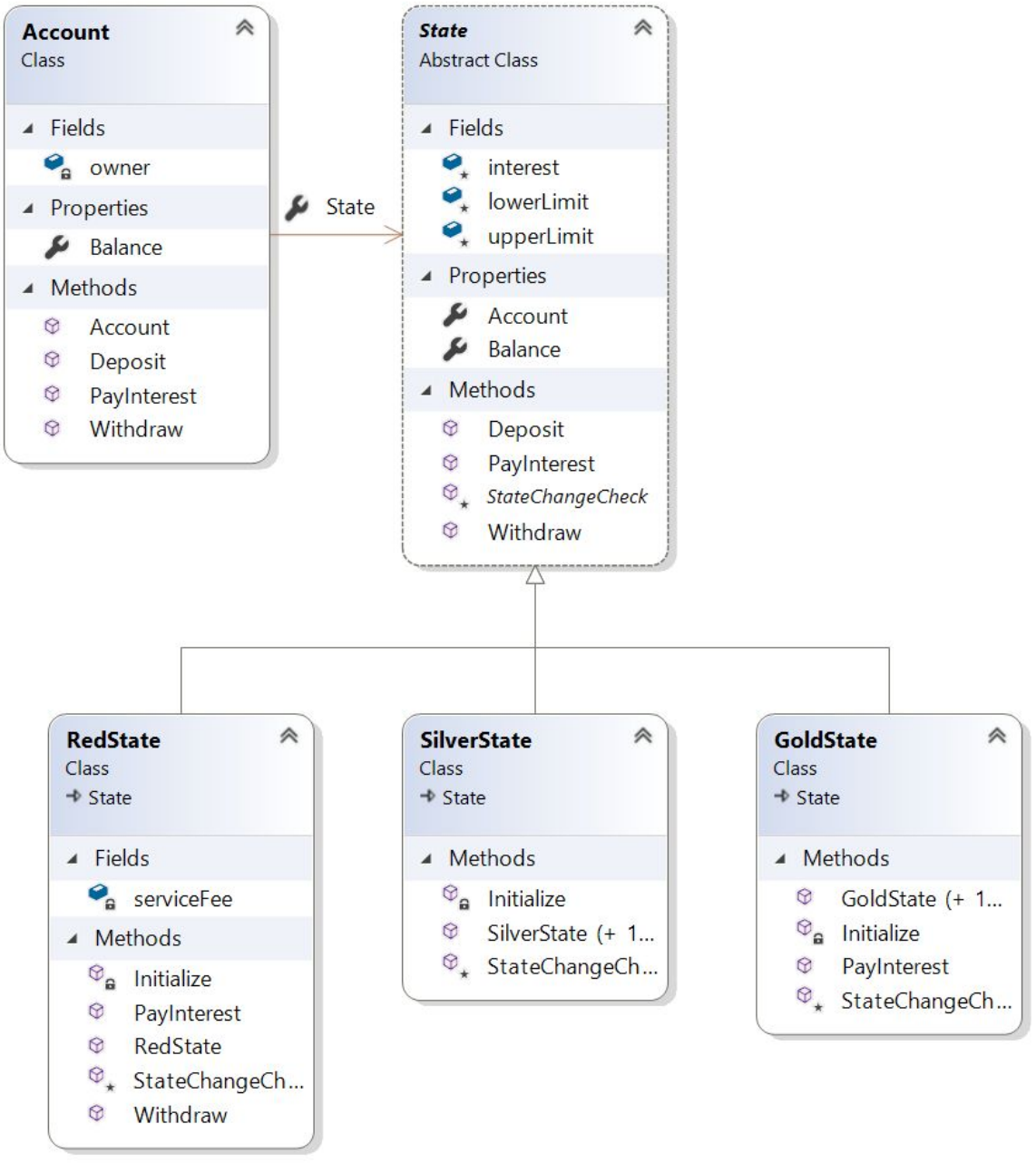
Observer

Definitsioon: Defineerib üks-mitmele sõltuvuse objektide vahel, nii et kui ühe objekti olek muutub, teavitatakse ja uuendatakse kõiki selle objekti jälgijaid. [GoF, lk 293]



State

Definitsioon: Võimaldab objektil muuta enda käitumist vastavat sisemise oleku muutustele. Objekt näib nagu oleks muutnud oma klassi. [GoF, lk 305]



Antud näites on kasutatud State disainimustrit, et võimaldada konto (Account) klassil muuta enda käitumist, sõltuvalt selle kontoseisust. Konto võib olla kolmes erinevas olekus - negatiivse kontojäägiga olek (RedState), tavaolek (SilverState), kuldliikme staatusega olek (GoldState). Iga kord, kui kontojääk muutub, toimub olekumuutuse kontroll (StateChangeCheck meetod) ning vastavalt uuele kontoseisule jääb olek samaks või asendub uue olekuga. Vaatame oleku (State) klassi.

```
public abstract class State {

    protected double interest;
    protected double lowerLimit;
    protected double upperLimit;
    public Account Account { get; set; }
    public double Balance { get; set; }

    public virtual void Deposit(double amount) {
        Balance += amount;
        StateChangeCheck();
    }

    public virtual void Withdraw(double amount) {
        Balance -= amount;
        StateChangeCheck();
    }

    public virtual void PayInterest() {
        Balance += interest * Balance;
        StateChangeCheck();
    }

    protected abstract void StateChangeCheck();
}
```

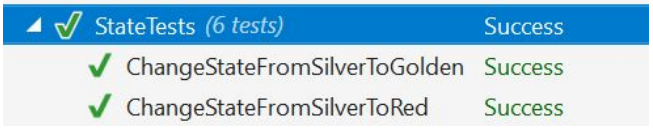
Tegemist on abstraktse klassiga, mida varem nimetatud konkreetsed klassid realiseerivad. Klassi muutujateks on interssimäär (interest) ja konto oleku alumine ja ülemine piir. Deposit ja Withdraw meetoditega toimub kontole raha lisamine ja sealt vähendamine. PayInterest meetodiga toimub intressi väljamakse. StateChangeCheck toimub pärast mistahes äsja nimetatud meetodi kutset ning sellega kontrollitakse, kas konto olek jääb samaks või muutub. SilverState klassi realiseeritud StateChangeCheck näeb välja selline:

```
protected override void StateChangeCheck() {
    if(Balance < lowerLimit) {
        Account.State = new RedState(this);
    } else if(Balance > upperLimit) {
        Account.State = new GoldState(this);
    }
}
```

Seega, kui pärast kontojäägi muutust on bilanss alla SilverState alampiiri (0.0), läheb konto olekusse RedState. Kui see aga jääb üle SilverState ülempiiri (1000.0), siis uueks olekuks saab GoldState. Alampiiri ja ülempiiri vahemikku jäädes olek jääb samaks.

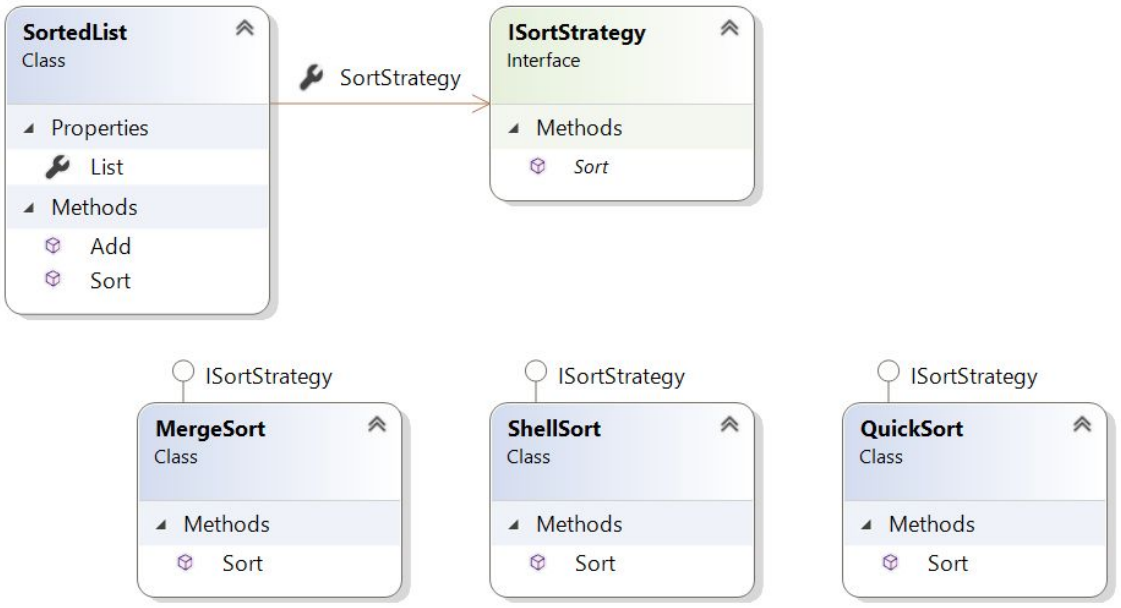
```
[TestMethod]
public void ChangeStateFromSilverToGolden()
{
    Assert.AreEqual(account.State.GetType(), typeof(SilverState));

    account.Deposit(10000.0);

    Assert.AreEqual(account.State.GetType(), typeof(GoldState));
}

[TestMethod]
public void ChangeStateFromSilverToRed()
{
    Assert.AreEqual(account.State.GetType(), typeof(SilverState));

    account.Withdraw(500);

    Assert.AreEqual(account.State.GetType(), typeof(RedState));
}
```

Ülal välja toodud testid kontrollivad oleku muutust SilverState'st GoldState'i ja RedState'i. Testide käivitades saame rohelise tulemuse.



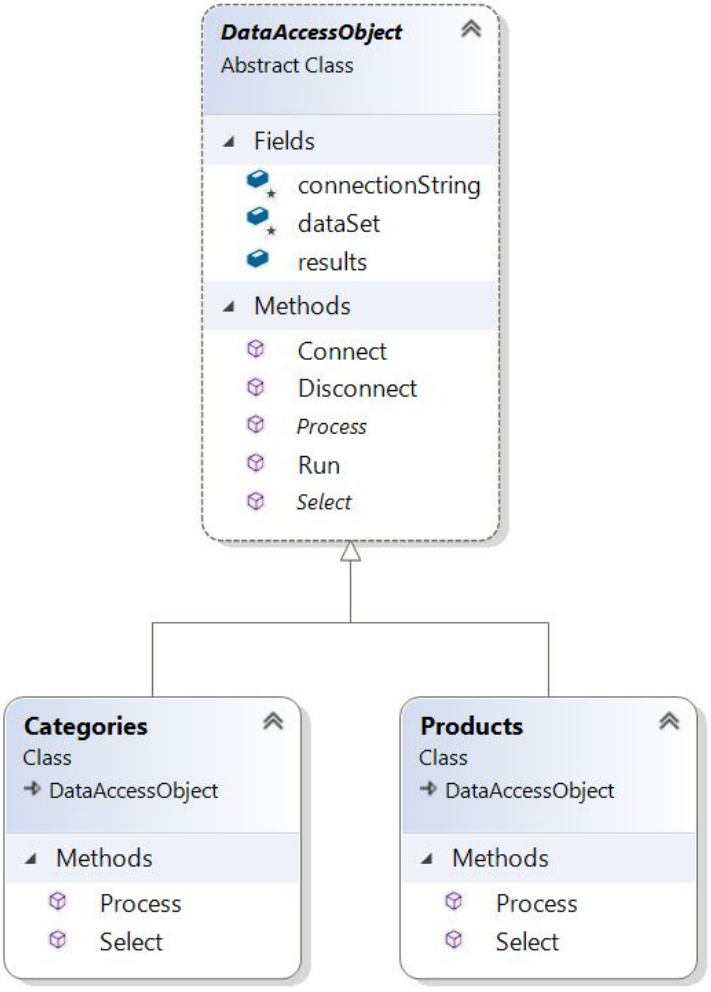
Strategy

Definitsioon: Defineerib algoritmide perekonna, kapseldab algoritmid ja teeb need omavahel vahetatavaks. [GoF, lk 315]



Template Method

Definitsioon: Defineerib operatsioonis algoritmi skeleti, andes mõned teostatavad sammud edasi kliendi alamklassile. Võimaldab alamklassidel defineerida kindlaid algoritmi samme, ilma algoritmi struktuuri muutmata. [GoF, lk 325]



Visitor

Definitsioon: Esitab operatsiooni, mida teostatakse objekti strutkuuri elementidel. Võimaldab defineerida uut operatsiooni, ilma muutmata elementide klasse, millel opereeritakse. [GoF, lk 331]