# Delegates

|  |  |
| --- | --- |
| **Læringsmål** | Du kan:   * **1Pf1**: anvende centrale metoder til at specificere og konstruere algoritmer […] * **1Pf2**: anvende centrale faciliteter i programmeringssproget til realisering af algoritmer, designmønstre, abstrakte datatyper, datastrukturer, designmodeller og brugergrænseflader. |
| **Forventet læringsudbytte [SOLO]** | Programmeringssporet:   * [Unistrukturel] Du kan genkende enkelte kendetegn ved **designmønstre** –Observer Pattern * [Unistrukturel] Du kan genkende enkelte kendetegn ved **Delegates** |
| **Din forberedelse** | Programmeringssporet:   * [YB]: 4.12.6 (s. 136 – 139) Using Delegates * [Delegates, Events and Lambdas](https://www.linkedin.com/learning/c-sharp-delegates-events-and-lambdas/overview-of-c-sharp-delegates-events-and-lambdas) (video: 2:47; bemærk: du behøver ikke at forstå events til denne opgave, og slet ikke lambda-udtryk) * [Common delegate and event scenario](https://www.linkedin.com/learning/c-sharp-delegates-events-and-lambdas/common-delegate-event-and-lambda-scenarios?u=57075649) (video: 2:14) * [Creating C# delegates](https://www.linkedin.com/learning/c-sharp-delegates-events-and-lambdas/creating-c-sharp-delegates?u=57075649) (video: 6:54) * [Using anonymous delegates](https://www.linkedin.com/learning/c-sharp-delegates-events-and-lambdas/using-anonymous-delegates?u=57075649) (video: 2:44) * [Using composable delegates](https://www.linkedin.com/learning/c-sharp-delegates-events-and-lambdas/using-composable-delegates) (video: 7:15) * [C# delegates](https://www.tutorialspoint.com/csharp/csharp_delegates.htm) * ”Jon Skeet - C# In Depth – 2.1 Delegates.pdf” (6 sider, se i planen i ItsLearning) * Genlæs:   + [YB] 3.1.8 (s. 55 – 56) Parameter Passing By Reference |

Du skal i denne opgave arbejde med delegates. Det kan i starten være noget af en hjernevrider at forstå brugen af delegates, bl.a. at man kan overføre en metode som parameter i en metode (dette skal du arbejde med i en øvelse i denne opgave). Ikke desto mindre er delegates en særdeles vigtig del af C# med et stort anvendelsesområde. De danner grundlaget for events, som du allerede har stiftet bekendtskab med i WPF-applikationer, hvor event-handlere kan håndtere brugerens handlinger i en GUI. Delegates står også bag lambda-udtryk, som er et emne, du snart skal se på.

I denne opgave kommer du også til at arbejde med delegates til at notificere ”observers” i Observer-mønstret sammen med interfaces; delegates er nærmest skræddersyet til denne opgave.

**Sørg for som minimum at have lavet øvelse 3 i denne opgave, da du skal arbejder videre med den i næste uge.**

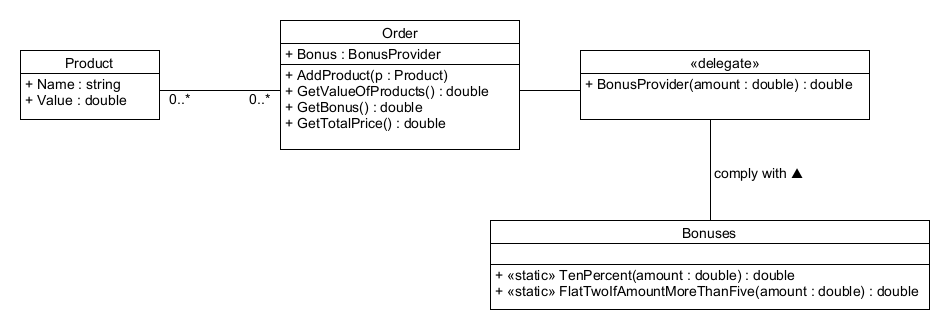
# Øvelse 1: Terminologi

Del teamet op i mindre grupper, og brug **Ordet rundt** til at reflektere over begreberne ”delegate”, ”anonymous delegates”, ”composable delegates” og ”pass by reference” .  
Sørg for, at alle får mulighed for at tale.

*Tidsramme: 20 minutter*

# Øvelse 2: BonusApp

Følgende DCD beskriver klasserne for BonusApp-applikationen:



*Navngivningskonvention:*

* *Public (+) attribut startende med stort bogstav skal implementeres som en C# property*

Klassen *Bonuses* indeholder de statiske metoder, som skal overholde delegate-typen BonusProvider:

* *TenPercent(double amount)* returnerer 10 % af amount
* *FlatTwoIfAmountMoreThanFive(double amount)* hvis amount er større end 5, returneres 2.0, ellers returneres 0.0

Klassen *Order* repræsenterer en ordre med tilknyttede produkter. Den har en *Bonus*-property af delegate-typen *BonusProvider* samt fire metoder:

* *AddProduct(Product p)* tilføjer et *Product*-objekt til ordren (husk at implementere *en-til-mange* associationen mellem *Product* og *Order* med en passende datastruktur)
* *GetValueOfProducts()* returnerer den samlede værdi af de tilknyttede *Product*-objekter
* *GetBonus()* benytter ordrens delegate til at udregne bonus af den samlede værdi af de tilknyttede *Product*-objekter
* *GetTotalPrice()* returnerer den samlede værdi af de tilknyttede *Product*-objekter minus den beregnede bonus

Klassen *Product* repræsenterer et simpelt produkt med navn og værdi.

## Øvelse 2.1: Implementér BonusApp

Udfør følgende:

* Opret en konsol-applikation med navnet BonusApp
* Implementér ovenstående model

Hvis det driller, kan det anbefales at tage et kig på unittesten.

## Øvelse 2.2: Test BonusApp

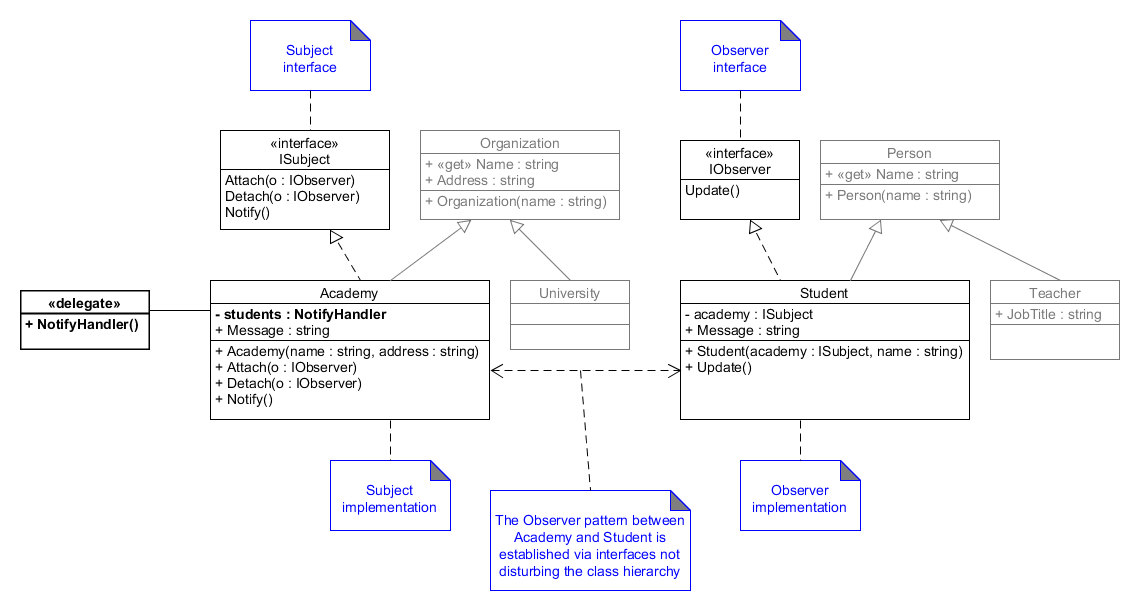
Udfør dernæst:

* Tilføj et ”Unit Test Project” med reference til projektet BonusApp til din solution
* Indsæt testen i ”Ex33-Delegates.UnitTest.docx” (fra git materiale-mappen)
* Kør testen, og tilret, indtil det virker

**Bemærk,** at Assert.AreEqual(double1, double2) af og til fejler, selvom de to double-værdier tilsyneladende er ens. For at undgå, at det er afrundingsfejl, der spiller ind, kan en tredje parameter tilføjes. Den repræsenterer et delta (et meget lille tal, f.eks. 0.00000001), og så længe forskellen på de to double-værdier ikke overstiger dette delta, vil Assert.AreEqual betragte dem som ens.

# Øvelse 3: Meddelelsessystemet (Observer Pattern)

Udviklerne har tænkt over implementeringen af meddelelsessystemet og har besluttet sig for at benytte delegates til at realisere Observer-mønstret. DCD’en er blevet opdateret til nedenstående:



Udover at delegate-typen NotifyHandler() er tilføjet til DCD’en, er det kun Academy-klassen, som er ændret; begge ændringer er fremhævet med fed skrifttype. I Academy-klassen er det private felt ’students’ nu af typen ’NotifyHandler’, dvs. en delegate-type, og ikke længere en collection (f.eks. List eller array).

Bemærk, at angivelsen af delegate-typen NotifyHandler i UML implementeres som følger i C#:

public delegate void NotifyHandler();

## Øvelse 3.1: Refaktorér Academy-klassen

Overvej, hvad denne ændring betyder for implementeringen af Observer-mønstret, og hvordan delegates kan realisere det.

* Refaktorér de påvirkede metoder i Academy-klassen.
* Kør den tidligere kode i Main-metoden:

static void Main(string[] args)

{

var p = new Academy("UCL", "Seebladsgade");

var s1 = new Student(p, "Jens");

var s2 = new Student(p, "Niels");

var s3 = new Student(p, "Susan");

p.Attach(s1);

p.Attach(s2);

p.Attach(s3);

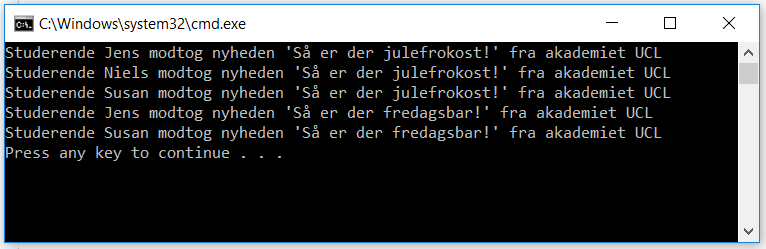
p.Message = "Så er der julefrokost!";

p.Detach(s2);

p.Message = "Så er der fredagsbar!";

}

Som i den forrige opgave får I f.eks. følgende output i konsolvinduet:



## Øvelse 3.2: Udnyt delegate

Selvom du i forrige øvelse 3.1 implementerer Academy-klassen til at anvende delegates, så udnytter løsningen ikke delegates fuldt ud. Der er fire centrale funktionaliteter, som en delegate selv kan håndtere:

1. Den kan tilkobles metoder (Attach) med ’+=’-operatoren (chaining)
2. Den kan frakobles metoder (Detach) med ’-=’-operatoren (unchaining)
3. Den kan kalde alle tilkoblede metoder (Notify) på én gang ved bare at kalde delegaten selv
4. Den kan sættes direkte med ’=’-operatoren

Udfør følgende:

* Omdan nu delegate-feltet students i Academy-klassen fra et ’private’ felt til ’public’ felt Students (med stort startbogstav). Vi ser bort fra, at det burde være en property lige nu
* Med inspiration i din anvendelse af delegate-feltet i øvelse 3.1 i Attach(), Detach() og Notify(), opdatér nu koden i Main(), så den direkte tilgår Students-delegaten og udnytter ’chaining’- og ’unchaining’-muligheden for at tilknytte Update()-metoden i hver observer. Dvs. du kalder ikke længere Attach() og Detach()
* Dernæst kald din Students-delegate direkte i Message-setter’en. Dvs. du kalder ikke længere Notify()
* Kør programmet. Du skulle gerne få samme resultatet i konsolvinduet som før. Hvis ikke, tilret og prøv igen

## Øvelse 3.3: Redesign Academy-klassen

Du har i forrige øvelse 3.2 implementeret samme mønster som før, men via en delegate’s funktionalitet helt omgået de tre metoder i ISubject-interfacet. Dvs. med en delegate behøver du ikke at implementere metoderne Attach, Detach og Notify, som angivet i ISubject-interfacet.

Udfør nu følgende:

* Lad ikke længere Academy-klassen nedarve fra ISubject, og fjern også de to metoder Attach() og Detach() helt fra klassen
* Ret dernæst navnet på delegaten, så den signalerer, hvilken ændring den håndterer. Kald den ’MessageChanged’ i stedet for ’Students’
* Omdåb Notify()-metoden til OnMessageChanged()
* Tilret også resten af din kode, bl.a. så ISubject-interfacet ikke længere anvendes i Student-klassen
* Kør dit program efter din ændring. Det skulle gerne give samme resultat i konsolvinduet som før

## Øvelse 3.4: Problemet med en public delegate

Selvom om en delegate er en smart løsning til at håndtere Observer-mønstret, så er der et stort problem med en public delegate. Den sidste funktionalitet for en delegate nævnt i øvelse 3.2 er tildeling (assignment) med ’=’-operatoren.

* I Main() indsæt linjen   
   p.MessageChanged = null;  
  som angivet forneden (i fed tekst) mellem den anden og tredje observer-tilknytning:

static void Main(string[] args)

{

Academy p = new Academy("UCL", "Seebladsgade");

Student s1 = new Student(p, "Jens");

Student s2 = new Student(p, "Niels");

Student s3 = new Student(p, "Susan");

p.MessageChanged += s1.Update;

p.MessageChanged += s2.Update;

**p.MessageChanged = null; 🡨 Indsæt linje**

p.MessageChanged += s3.Update;

p.Message = "Så er der julefrokost!";

p.MessageChanged -= s2.Update;

p.Message = "Så er der fredagsbar!";

Console.ReadLine();

}

* Overvej, hvad effekten af denne ændring er
* Kør programmet og se, om du har ret
* Overvej og argumentér for, hvorfor lige specielt denne ændring kan udgøre et stort problem for en public delegate