# Inheritence and Abstraction (Disaheim2)

# Øvelse 1: Terminologi

Når hele teamet er klar:

* Del teamet op i to mindre grupper, og brug **Møde på midten** i hver gruppe til at diskutere begreberne (sørg for, at alle får mulighed for at tale):
  + Gruppe 1: ”foreach”, ”overriding – virtual/override”, “abstract classes” og ”class library”.
  + Gruppe 2: ”nedarvning”, ”base class”, ”derived class” og ”sealed class”.
* Saml teamet igen, og lad hver gruppe præsentere de begreber, de har alt om, til den anden gruppe.

*Tidsramme: ca. 20 minutter*

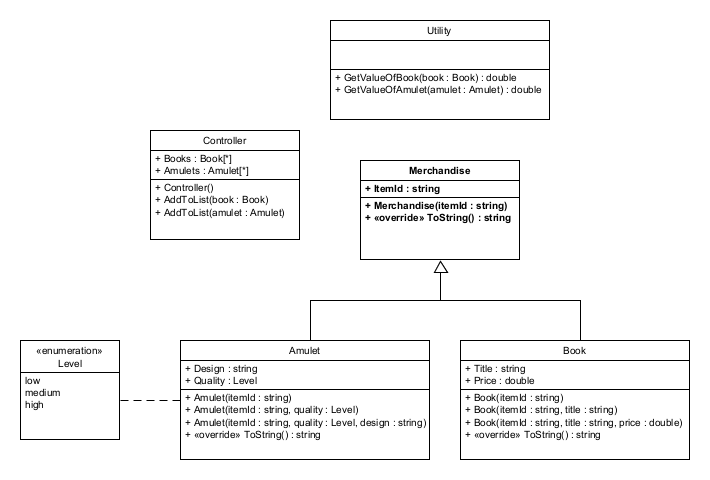
**Benyt parprogrammering til alle resterende øvelser**.

# Øvelse 2: Nedarvning - Merchandise

Du skal fortsætte med at bygge videre på koden i Disaheim-casen, så tag udgangspunkt i den løsning, du lavede sidste gang i opgaven Uge 15\_1.

## Øvelse 2.1: Merchandise-klassen

I DCD’et fra den forrige opgave har klasserne Book og Amulet begge et *ItemId*, og de er begge repræsentanter for merchandise, der sælges. Denne sammenhæng kan vi modellere i et arve-hierarki i en videreudvikling af DCD’et (det er kun Merchandise-klassen, der er ny, samt dets relationer til Amulet og Book):



**Nedarvning i UML** (kaldet enten generalisering eller specialisering) angives med en pil med en ikke-udfyldt trekant. Ovenfor læses Merchandise som en generalisering af Amulet og Book, og de to sidstnævnte hver betegnes som en specialisering af Merchandise.

Den fælles ItemId-property er nu repræsenteret i Merchandise-overklassen (overklassen betegnes også superklassen eller ’base class’ på engelsk; kært barn har mange navne) og er tilsvarende fjernet fra Amulet og Book.

Udfør følgende:

* Se nøje på DCD’et, så du helt forstår den ændring, der er foretaget
* Inden du begynder at ændre dit nuværende Disaheim-projekt, da **gem den nuværende version i GitHub**, så du altid kan vende tilbage til denne version (og gør det for hver ny version af Disaheim, du laver fremover)
* Implementér klassen *Merchandise* i C#, og tilret klasserne *Book* og *Amulet*
  + Bemærk, at Amulet- og Book-klasserne selv skal sørge for, at Merchandise-klassens ene constructor bliver kaldt eksplicit, da der ikke længere er en default constructor, der kaldes automatisk (vink: benyt base-nøgleordet i Amulet- og Book-klassernes constructors)

## Øvelse 2.2: Test af Merchandise-klassen

Udfør følgende:

* Tilføj nedenstående testmetode MerchandiseConstructorWorkProperly() sidst i ”*UnitTest1.cs*” og kontrollér, at alle dine tests stadig kører succesfuldt:

[TestMethod]

public void MerchandiseConstructorWorkProperly()

{

// Arrange

Merchandise m = new Merchandise("100");

// Assert

Assert.AreEqual("ItemId: 100", m.ToString());

}

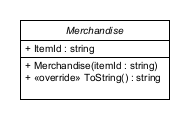
* Tilret, indtil testmetoden (og de andre testmetoder i denne test) kører uden fejl.

# Øvelse 3: Abstrakt klasse

Ved nærmere eftersyn giver det ikke mening at kunne oprette instanser af klassen *Merchandise* – den tjener kun det formål at repræsentere fællesdelen (*ItemId*) for de to sub-klasser. *Merchandise* bør derfor være en abstrakt klasse.

## Øvelse 3.1: Abstrakt Merchandise-klasse

Merchandise-designklassen ændres til at være en abstrakt klasse:



Bemærk, at en abstrakt klasse angives i UML med et klassenavn i *kursiv*. Det vigtigste at huske på i forhold til en abstrakt klasse i C# er, at man **IKKE** kan instantiere den, men man kan godt nedarve fra den.

* Ret *Merchandise*-klassen i C#, så den er abstrakt
* Byg din solution igen.

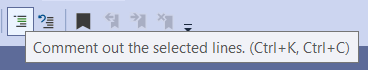
Du får nu en kompileringsfejl i testmetoden MerchandiseConstructorWorkProperly(), du tilføjede i øvelse 2.2 ved linjen:

// Arrange

Merchandise m = new Merchandise("100");

Det er ikke længere muligt at instantiere et Merchandise-objekt.

Testmetoden giver ikke længere mening, og du skal derfor ud-kommentere hele testmetoden - markér hele testmetoden, og tryk dernæst på ud-kommentér ikonet, eller tryk på: ctrl +k, ctrl+ c**:**

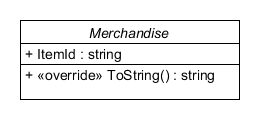


## Øvelse 3.2: Constructor i en abstrakt klasse

Udfør følgende:

* Hvis du inspicerer den abstrakte *Merchandise*-klasse, så bemærker du, at klassen stadig har en gyldig constructor, selvom den er abstrakt
  + Overvej, om det kan være korrekt
* Kør testen ”*UnitTest1.cs*” igen
* Testen kører fint (det skulle den gerne gøre), dvs. alle Book- og Amulet-constructor-tests virker
  + Undrer det dig, at testen kører fint igennem?
  + Overvej, hvorfor det kan give mening, at en abstrakt klasse godt kan have constructors, når man nu ikke kan instantiere den direkte og dermed kalde dens constructor på den måde
  + Overvej, hvordan man så kan kalde constructor’en i en abstrakt klasse (vink: din kode gør det allerede)

Fjern nu constructor’en fra *Merchandise*-klassen. Dvs. designklassen skal reduceres til:



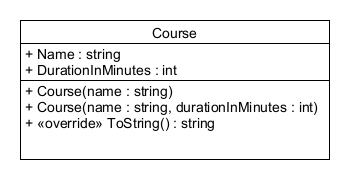
* Implementér denne ændring i *Merchandise*-klassen.
* Byg din solution igen, og observér de fejl, der opstår
* Overvej, hvad problemet er, og ret fejlene, så koden kan kompileres uden fejl, og testen kører fint igen
* Overvej til sidst, hvilken constructor i *Merchandise*, der kaldes nu? Hvordan kan du teste, om din tanke er korrekt?

# Øvelse 4: Course-klassen

Virksomheden beslutter efter mange eksterne opfordringer, at de også vil tilbyde kurser, f.eks.: ”Spådomskunst for nybegyndere”, ”Magi – når videnskaben stopper” og ”Et indblik i Helleristning”.

## Øvelse 4.1: Course-klassen

Et kursus er ikke et håndgribeligt produkt, og det er derfor ikke af typen *Merchandise*, men det er derimod en klasse for sig selv. DCD’et udvides med nedenstående software designklasse for Course:



Udfør følgende:

* Implementér Course-klassen i din løsning
  + Benyt constructor chaining

## Øvelse 4.2: Test af Course

Udfør følgende:

* Tilføj de med gult markerede linjer til starten af ”*UnitTest1.cs*”:

Book b1, b2, b3;

Amulet a1, a2, a3;

Course c1, c2, c3;

[TestInitialize]

public void Init()

{

// Arrange

b1 = new Book("1");

b2 = new Book("2", "Falling in Love with Yourself");

b3 = new Book("3", "Spirits in the Night", 123.55);

a1 = new Amulet("11");

a2 = new Amulet("12", Level.high);

a3 = new Amulet("13", Level.low, "Capricorn");

c1 = new Course("Spådomskunst for nybegyndere");

c2 = new Course("Magi – når videnskaben stopper", 157);

c3 = new Course("Et indblik i Helleristning", 180);

}

* Tilføj dernæst følgende testmetoder sidst i *UnitTest1.cs*:

[TestMethod]

public void CourseConstructorWithOneParameter()

{

// Assert

Assert.AreEqual("Name: Spådomskunst for nybegyndere, Duration in Minutes: 0", c1.ToString());

}

[TestMethod]

public void CourseConstructorWithTwoParameters1()

{

// Assert

Assert.AreEqual("Name: Magi – når videnskaben stopper, Duration in Minutes: 157", c2.ToString());

}

[TestMethod]

public void CourseConstructorWithTwoParameters2()

{

// Assert

Assert.AreEqual("Name: Et indblik i Helleristning, Duration in Minutes: 180", c3.ToString());

}

[TestMethod]

public void AmuletSetPropertiesWorks()

{

// Act

a3.ItemId = "X";

a3.Quality = Level.high;

a3.Design = "Dolphin";

// Assert

Assert.AreEqual("ItemId: X, Quality: high, Design: Dolphin", a3.ToString());

}

[TestMethod]

public void BookSetPropertiesWorks()

{

// Act

b3.ItemId = "Y";

b3.Title = "Smoke on the Water";

b3.Price = 376.45;

// Assert

Assert.AreEqual("ItemId: Y, Title: Smoke on the Water, Price: 376,45", b3.ToString());

}

[TestMethod]

public void CourseSetPropertiesWorks()

{

// Act

c2.Name = "How to Ying-Yang";

c2.DurationInMinutes = 413;

// Assert

Assert.AreEqual("Name: How to Ying-Yang, Duration in Minutes: 413", c2.ToString());

}

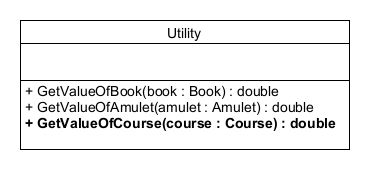
* Se testmetoderne nøje igennem, og få alle tests til at køre succesfuldt ved at tilrette i Course-klassen. Husk, at du aldrig må rette i en unittest

# Øvelse 5: Utility-klassen

Du blev i forrige opgave introduceret til en Utility-klasse. Som du måske kan huske, implementerede du to metoder i denne klasse.

## Øvelse 5.1: GetValueOfCourse

Med introduktion af Course-klassen opdateres Utility-klassen til:



Den nye operation *GetValueOfCourse* gør det muligt at beregne værdien af et kursus, som udtrykt i nedenstående operationskontrakt:

Operationskontrakt for **GetValueOfCourse(course : Course) : double**

Cross reference: Budgetopfølgning

Precondition: Kurset anvendt som parameter eksisterer

Postcondition: Ingen ændringer i systemet

Output: Kursets værdi returneres  
Værdien fastsættes som 875.00 pr. påbegyndt time

Udfør følgende:

* Implementér Utility-klassens nye operation i C#
  + Bemærk, at værdien af et kursus beregnes ud fra påbegyndt time, dvs. 0 minutter svarer til 0 påbegyndt time, og 1 minut svarer til 1 påbegyndt time osv. Overvej, hvordan du algoritmisk kan udregne dette (vink: find heltalsdel og restdel ved division, tjek [operatoren %](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/arithmetic-operators#remainder-operator-))

## Øvelse 5.2: Test af Utility (UnitTest2.cs)

Udfør følgende:

* Tilføj (som før) de med gult markerede linjer til starten af ”*UnitTest2.cs*”:

Book b1, b2, b3;

Amulet a1, a2, a3;

Course c1, c2, c3;

Utility utility;

[TestInitialize]

public void Init()

{

// Arrange

b1 = new Book("1");

b2 = new Book("2", "Falling in Love with Yourself");

b3 = new Book("3", "Spirits in the Night", 123.55);

a1 = new Amulet("11");

a2 = new Amulet("12", Level.high);

a3 = new Amulet("13", Level.low, "Capricorn");

c1 = new Course("Spådomskunst for nybegyndere");

c2 = new Course("Magi – når videnskaben stopper", 157);

c3 = new Course("Et indblik i Helleristning", 180);

utility = new Utility();

}

* Tilføj dernæst følgende til sidst i *UnitTest2.cs*:

[TestMethod]

public void TestGetValueForCourse1()

{

// Assert

Assert.AreEqual(0.0, utility.GetValueOfCourse(c1));

}

[TestMethod]

public void TestGetValueForCourse2()

{

// Assert

Assert.AreEqual(2625.0, utility.GetValueOfCourse(c2));

}

[TestMethod]

public void TestGetValueForCourse3()

{

// Assert

Assert.AreEqual(2625.0, utility.GetValueOfCourse(c3));

}

* Kør testen, og tilret, indtil den gennemføres med succes

# Øvelse 6: Controller-klassen

Du implementerede også en Controller-klasse i forrige opgave, som kan håndtere mange bøger og amuletter i hver sin liste. Den skal nu også kunne håndtere kurser.

## Øvelse 6.1: Courses-property og AddToList()-metode

Controller-klassen opdateres til:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Udfør følgende:

* Implementér Courses-property’en og AddToList()-metoden i Controller-klassen

## Øvelse 6.2: Test af Controller

Ændringerne i Controller-klassen skal selvfølgelig også testes.

Udfør følgende:

* Tilføj de med gult markerede linjer til starten af ”*UnitTest3.cs*”:

Book b1, b2, b3;

Amulet a1, a2, a3;

Course c1, c2, c3;

Controller controller;

[TestInitialize]

public void Init()

{

// Arrange

b1 = new Book("1");

b2 = new Book("2", "Falling in Love with Yourself");

b3 = new Book("3", "Spirits in the Night", 123.55);

a1 = new Amulet("11");

a2 = new Amulet("12", Level.high);

a3 = new Amulet("13", Level.low, "Capricorn");

c1 = new Course("Spådomskunst for nybegyndere");

c2 = new Course("Magi – når videnskaben stopper", 157);

c3 = new Course("Et indblik i Helleristning", 180);

controller = new Controller();

controller.AddToList(b1);

controller.AddToList(b2);

controller.AddToList(b3);

controller.AddToList(a1);

controller.AddToList(a2);

controller.AddToList(a3);

controller.AddToList(c1);

controller.AddToList(c2);

controller.AddToList(c3);

}

* Tilføj følgende testmetode *TestCourseList()* sidst i *UnitTest3.cs*:

[TestMethod]

public void TestCourseList()

{

// Assert

Assert.AreEqual(c1, controller.Courses[0]);

Assert.AreEqual(c2, controller.Courses[1]);

Assert.AreEqual(c3, controller.Courses[2]);

}

* Kør testen, og tilret, indtil den gennemføres med succes

# Øvelse 7: Opret et bibliotek til Utility-klassen

Utility-klassen indeholder reelt et bibliotek af hjælpemetoder, så det giver ikke længere mening, at denne klasse ligger i hovedprogrammet, da hjælpemetoderne også kan anvendes af andre programmer. Du skal derfor flytte klassen til sit eget bibliotek i C#.

Udfør følgende:

* Opret et nyt C# biblioteksprojekt med projekttypen ”Class Library” med navnet ”UtilityLib”
* Flyt Utility-klassen helt over i UtilityLib-projektet (dvs. Utility-klassen eksisterer herefter ikke i hovedprogrammet Disaheim)
* Opdatér projektreferencer i hovedprogrammet, UtilityLib-biblioteket og jeres unittest-projekt (også de tidligere), så der ingen reference-fejl
  + Bemærk, du skal ikke oprette en reference fra din konsolapplikation (Disaheim) til dit nye class library (UtilityLib) – ellers vil du få en cirkulær reference
* Kør alle dine unittests igen for at sikre dig, at alt er gået godt