# Disaheim

# Øvelse 1: Terminologi

Del teamet op i to mindre grupper, og brug **Møde på midten** i hver gruppe til at diskutere begrebet ”enumeration”. Husk, at alle i gruppen skal have taletid.

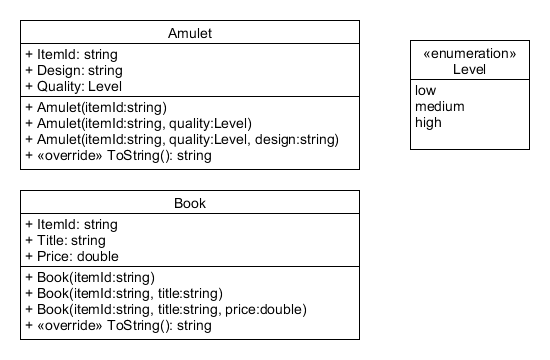
*Tidsramme: 20 minutter*

# Øvelse 2: Disaheim

Disaheim ønsker at sælge bøger om wellness, meditation og levitation af uorganiske materialer. De har netop udvidet deres sortiment med sølvamuletter mod negative kosmiske stråler. Amuletterne er i flere design og forskellige kvaliteter (lav, middel og høj). Deres revisor tager sig af alt regnskab og betaling af moms og skat – det forstår Disaheim sig overhovedet ikke på og mangler derfor et overblik over deres forretning. De ønsker sig derfor en simpel applikation til at give dem dette.

Du skal anvende **parprogrammering** gennem hele opgaven. Du starter *agilt* i det små og bygger på efterhånden. Disaheim-casen vil blive udviklet over flere opgaver, så det er vigtigt, at du og din makker får løst den, da næste opgave er afhængig af denne. Du får brug for at *refaktorere* flere gange, så **sørg for, at dit projekt er lagt på GitHub**.

Et design firma har første omgang designet nogle få designklasser til DCD’en (designklassediagrammet for hele systemet), som du skal implementere i C#:



**Navnekonvention:**

* Klassens navn skal være i UpperCamelCase/PascalCase og i ental
* Attributter, der er public (+) og starter med et stort bogstav (UpperCamelCase/PascalCase), skal implementeres i C# som properties med stort startbogstav.
* Attributter, der er private (-) og starter med et lille bogstav (lowerCamelCase), skal implementeres i C# som et felt (en: field) med lille startbogstav. **Bemærk:** Hvis et privat C# felt fungerer som et ’backing field’ for en property, da skal feltnavn foranstilles med en underscore (\_).
* En operation med samme navn som klassen (og uden returtype) skal implementeres i C# som en constructor (dvs. stereotypen «constructor» udelades), ellers som en metode

**Denne navnekonvention benyttes fremover i alle kommende opgaver med mindre andet nævnes.**

## Øvelse 2.1: Implementér Disaheim-løsning

Udfør følgende:

* Opret et konsolprojekt med navnet ”Disaheim”.
* Implementér først Level-enumeration som en C# enum (i en separat C#-fil).
* Implementér dernæst de to designklasser Amulet og Book i hver sin C#-fil. Husk at gøre klasserne public.
  + Erklær klassernes properties (inkl. interne datastruktur)
  + Kontroller også, om du har implementeret en ’no-arguments’ default constructor. Hvis du har, så skal du måske kigge på diagrammet en ekstra gang og overveje, om det er rigtigt.   
    Ret din kode om nødvendigt.
  + Implementér ToString()-metoden, som er en override.  
    Overvej, hvor ToString()-metoden kommer fra. Hvad er det, som bliver ’overrided’?

## Øvelse 2.2: Videndel i teamet

Videndel (vælg selv CL-struktur):

* Sammenlign din og din makkers kode med de andre par i teamet med fokus på implementering af din enum-datatype, properties, constructor-koden, og ToString()-metoden.
* Ret i din kode, om nødvendigt.

## Øvelse 2.3: Testprojekt

Nu er det tid til en test. Udfør følgende:

* Opret et nyt testprojekt ved navn ”DisaheimTest”.
  + **Indsæt projektreference til dit projekt ”Disaheim” (husk også reference til namespace).**
* Erstat indholdet af UnitTest1 (dvs. kun testklassen) med unit-testen ”Uge15\_II*.UnitTest.docx*”.
* Husk at sætte testklassen **ind i** namespacet’s kodeblok.
* Få testen til at køre succesfuldt. Ret din kode om nødvendigt.

## Øvelse 2.4: Debugging

Debug mindst én af testmetoderne i unit-testen:

* Indsæt et breakpoint (F9: Toggle Breakpoint) i starten af Init()-metoden i UnitTest1.cs (dvs. i [TestInitialize]-sektionen).
* Indsæt også et breakpoint (F9) i starten af den udvalgte testmetoden.
* I Test Explorer vælg testmetoden, højreklik den, og vælg Debug-kommandoen.
* Brug Step Into (F11) og Step Over (F10) til at følge din kode trin for trin først i Init()-metoden og derefter i din udvalgte testmetode samt inspicér dine variable, dvs. parametre, properties, felter og lokale variable.
  + Observér hvordan dine constructors kalder hinanden, overfører parametre og sætter dine properties.

Du skal fortsætte med Disaheim-casen, hvor du gennem øvelserne skal arbejde med Collections (da: samlinger) i form af datatypen List<T>, som kan bruges i stedet for arrays, samt method overloading svarende til constructor overloading. Du kommer også til at arbejde med unit-tests og debugging. Du får også et par øvelser om forskellen på reference- og value-typer ved metodekald.

# Øvelse 3: Terminologi

Del teamet op i to mindre grupper, og brug **Møde på midten** i hver gruppe til at diskutere begreberne ”value- og reference type”, ”Collections og List<T>” og ”Method overloading”.   
Husk, at alle i gruppen skal have taletid.

*Tidsramme: 20 minutter*

**Benyt parprogrammering til alle de resterende øvelser**.

# Øvelse 4: Disaheim - Økonomi

Formålet med Disaheim-systemet er at hjælpe virksomheden med at få et overblik over deres aktiviteter – herunder økonomi.

## Øvelse 4.1: Beregning af værdi

Du får derfor brug for at kunne beregne værdien af en bog og en amulet, som udtrykt i nedenstående operationskontrakter:

Operationskontrakt for **GetValueOfBook(book : Book) : double**

Cross reference: Budgetopfølgning

Precondition: Bogen anvendt som parameter eksisterer

Postcondition: Ingen ændringer i systemet

Output: Bogens værdi givet ved Price returneres

Operationskontrakt for **GetValueOfAmulet(amulet : Amulet) : double**

Cross reference: Budgetopfølgning

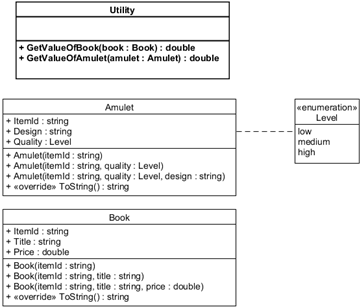
Precondition: Amuletten anvendt som parameter eksisterer

Postcondition: Ingen ændringer i systemet

Output: Amulettens værdi returneres.

Værdien fastsættes ud fra amulettens kvalitet:   
low -> 12.5, medium -> 20.0 og high -> 27.5

I DCD’et placeres disse to operationer i en klasse for sig selv, *Utility*, som tilføjet forneden i DCD’et med fed skrift (bemærk: Husk navngivningskonvention):



Udfør følgende:

* Inden du begynder at ændre dit Disaheim-projekt, da **gem den nuværende version i GitHub**, så du altid kan vende tilbage til denne version (faktisk gør det for hver ny version af Disaheim, du laver fremover)
* **Implementér Utility-klassen**

## Øvelse 4.2: Test af Utility

Nu er det tid til en test. Udfør følgende:

* Opret en ny test-fil – *UnitTest2.cs* – i dit eksisterende test-projekt ”DisaheimTest”, dvs.:
  + I test-projektet tilføj en ny C#-klasse med navnet ”*UnitTest2.cs*”
  + Slet al kode i den nye fil, indsæt i stedet for:

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Disaheim;

namespace DisaheimTest

{

// Indsæt testklasse her!

}

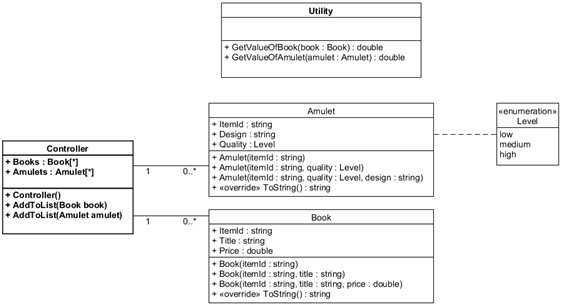
Så har du oprettet grundstrukturen i en ny testklasse.

* Indsæt **inde i** namespacet’s kodeblok indholdet af unit-testen ”Uge15\_II*.UnitTest2.docx*”.
* Få testen til at køre succesfuldt. Ret din kode om nødvendigt

# Øvelse 5: Samlinger af bøger og amuletter

Disaheim ønsker selvfølgelig at have med mange bøger og amuletter at gøre. Der er derfor behov for at repræsentere samlinger (collections) af bøger og amuletter. Du tænker sikkert arrays, når du skal repræsentere en samling i C#, men nu skal du i stedet arbejde med List-datatypen, som på mange måder er nemmere at arbejde med end arrays (på bekostning af lidt hastighed).

Samlingerne af bøger og amuletter skal foreløbig håndteres af en anden klasse, nemlig Controller-klassen, som angivet i DCD’et forneden til venstre:



## Øvelse 5.1: Implementér Controller-klassen

Udfør følgende:

* Implementér Controller-klassen i C#, hvor du anvender List<T>-datatypen i stedet for arrays til de to C#-properties Books og Amulets. Overvej hvad <T> står for, og hvordan det udtrykkes i din kode
* De to properties Books og Amulets skal initialiseres i constructor’en for Controller-klassen. Overvej hvad det indebærer. Hvad er nødvendigt for at initialisere de to properties?
* Implementér de to operationer AddToList() i designklassen som C# metoder. Kigger du nærmere efter, er der faktisk kun én operation, men med to forskellige typer af parametre. Dette kan løses med metode-overloading i C#

**Metode-overloading** er, som tidligere nævnt, en måde at implementere **polymorfi** på, den anden af de fire objekt-orienterede programmeringsprincipper, du indtil nu har set på (kan du huske den første, du lærte?).

## Øvelse 5.2: Test af Controller-klassen

Nu er det tid til at teste din implementering. Udfør følgende:

* Opret en ny test-fil – *UnitTest3.cs* – i dit test-projekt ”DisaheimTest”, dvs.:
  + I test-projektet tilføj en ny C#-klasse med navnet ”*UnitTest3.cs*”

Slet al kode i den nye fil, indsæt i stedet for:

using Microsoft.VisualStudio.TestTools.UnitTesting;

using Disaheim;

namespace DisaheimTest

{

// Indsæt testklasse her!

}

Så har du oprettet grundstrukturen i en ny testklasse.

* Indsæt **inde i** namespacet’s kodeblok indholdet af unit-testen ”*Uge\_15\_II.UnitTest3.docx*”.
* Få testen til at køre succesfuldt. Ret din kode om nødvendigt

## Øvelse 5.3: Debugging

Debug Controller-klassen i unit-testen:

* Indsæt et breakpoint (F9: Toggle Breakpoint) i Init()-metoden i UnitTest3.cs (dvs. i [TestInitialize]-sektionen), hvor controller-feltet instantieres: controller = new Controller();
* I Test Explorer vælg testmetoden, højreklik den, og vælg Debug-kommandoen
* Brug Step Into (F11) og Step Over (F10) til **at følge og tjekke din Controller-kode trin for trin** først i constructor’en i Controller og dernæst AddToList-metoderne, så du får en synlig forståelse for, hvad din kode egentlig gør