# Disaheim 4

|  |  |
| --- | --- |
| **Læringsmål** | Du kan:   * **1Pf1:** anvende centrale metoder til at specificere og konstruere algoritmer [...] * **1Pf2:** anvende centrale faciliteter i programmeringssproget til realisering af algoritmer, designmønstre, […], datatyper, datastrukturer, designmodeller […] * **1Pf3**: anvende et i professionen udbredt, integreret udviklingsværktøj, herunder versionsstyringssystem […] til at designe og konstruere praksisnære applikationer baseret på en lagdelt arkitektur * **1Pk3**: i en struktureret sammenhæng tilegne sig ny viden, færdigheder og kompetencer inden for programmeringssprog, udviklingsværktøjer, programmeringsteknikker og programdesign * **1Pf6**: anvende moderne teknikker og værktøjer til afvikling af test […] |
| **Forventet læringsudbytte [SOLO]** | Designsporet:   * [Unistrukturel] Du kan huske enkelte kendetegn ved **ansvarshåndtering (en: responsibility management)** – Controller (GRASP), Single Responsibility Principle (SOLID) * [Unistrukturel] Du kan huske enkelte kendetegn ved **afhængighedshåndtering (en: dependency management)** – lagdeling (en: layering), pakkediagram (en: package diagram) * [Unistrukturel] Du kan genkende enkelte kendetegn ved **Designmønstre (en: design patterns)** – Repository pattern   Programmeringssporet:   * [Unistrukturel] Du kan huske enkelte kendetegn ved styring af **program flow** * [Unistrukturel] Du kan huske enkelte kendetegn ved styring af **datahåndtering** * [Unistrukturel] Du kan genkende enkelte kendetegn ved simple **C# datatyper** –  […], ’is’- og ’as’-operator (safe type casting) * [Unistrukturel] Du kan genkende enkelte kendetegn ved **C# klasse (objekt)** * [Unistrukturel] De kan nævne enkelte kendetegn i forhold til **OOP-principper** – indkapsling, polymorfi, nedarvning |
| **Din forberedelse** | Programmeringssporet:   * Skimlæs [Larman] kapitel 13 (s. 197-212) Logical architecture and UML package Diagrams   + Bemærk, de tre øverste lag i figur 13.4 (s. 203) * [Larman] 17.3 (s. 302-314) Controller   + Hav fokus på, hvilket problem controlleren løser * [GRASP](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design))   + Med fokus på Controller * [SOLID, GRASP, and Other Basic Principles of Object-Oriented Design](https://dzone.com/articles/solid-grasp-and-other-basic-principles-of-object-o)   + Læs kun om Single Responsibility Principle (SRP) og Controller for nu * [Repository pattern in C#](https://codecompiled.com/?s=Repository+pattern+in+c%23+2015)   + Fokusér på forklaringen af Repository-mønstret i starten af artiklen, og skimlæs det sidste med interfaces fra afsnittet ”How Repository Pattern is implemented in C#” * [Teste om et objekt er en bestemt type med ‘is’-operatoren](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/is) * [Safe Type Casting with IS and AS Operator](https://www.dotnettricks.com/learn/csharp/safe-type-casting-with-is-and-as-operator) |

Du er nu kommet så langt med Disaheim (og nok også dine andre projekter), at det bliver nødvendigt at introducere nye vigtige designelementer i arbejdet med at organisere et it-system med mere end nogle få klasser for at sikre overblik og minimere risikoen for et uhensigtsmæssig kollaps i designstrukturen.   
I denne opgave introduceres du til nogle få vigtige og centrale designprincipper og -mønstre fra både SOLID og GRASP, som skal hjælpe dig med at strukturere Disaheim-løsningen på en ny og bedre måde. Du kommer også til at arbejde med Repository-mønstret og begynder så småt at se på den logiske arkitektur af et it-system via en simpel lagdelt arkitektur.

# Dagens ord:

Når man designer klasserne i et it-system, skal man sikre sig, at klassernes ansvar bl.a. er nemme at forstå og vedligeholde og ikke mindst teste. Her er **Single Responsibility Principle** (SRP og S’et i SOLID) helt centralt, hvor man i sit design sigter mest muligt efter, at en klasse kun har ét ansvar. Hvis man ændrer i sin klasse, kan dette betyde, at eventuelle sideeffekter (og dermed ændringer) i relaterede klasser minimeres eller fjernes helt. Dette relaterer også til to andre principper i GRASP: Low Coupling og High Cohesion

Ud over anvendelse af principper, så har man gennem tiden observeret visse designmønstre (kan du huske mønstergenkendelse i Computational Thinking?), som ofte går igen i softwareudvikling. Brug af disse mønstre giver ikke kun gode løsninger til problemstillinger, man ofte støder på, men også større læsbarhed og færre fejl. Et mønster, som du allerede har stiftet overfladisk bekendtskab med, er **Controller-mønstret**, hvis ansvar (jf. GRASP) i en **lagdelt arkitektur** er at håndtere hændelser fra brugergrænsefladen og videreformidle disse hændelser til systemets forretningslogik, dvs. dets operationer, kommandoer, m.m., der opretter, ændrer, sletter og gemmer data. Et andet mønster, man ofte støder på, er **Repository-mønstret** fra Martin Fowler. Dette mønster skaber et abstraktionslag med en Repository-klasse, som abstraherer (indkapsler) din konkrete implementering af opbevaring og tilgang til systemets data.

# Øvelse 1: Terminologi

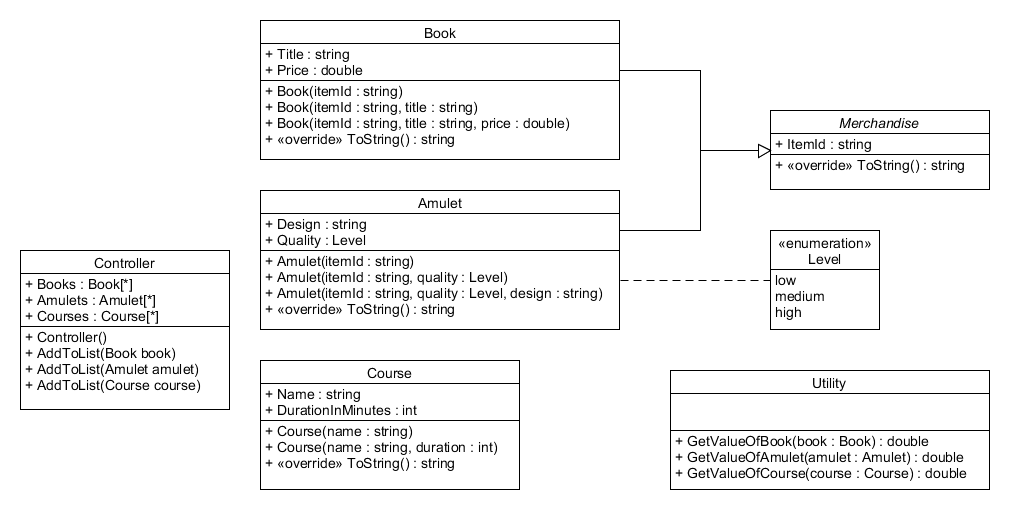
Del teamet op i to mindre grupper (3-4. pers. i hver) og brug **Ordet rundt** til at reflektere over begreberne ”Single Responsibility Principle (SRP)”, ”Repository-mønster”, ”Controller-mønster” og ”lagdelt arkitektur”. Sørg for at alle får mulighed for at tale.

*Tidsramme: 15 minutter*

**Benyt parprogrammering til alle nedenstående øvelser.**

# Øvelse 2: Single Responsibility Principle og Repository-mønstret

Disaheim-systemet kan indtil videre opsummeres i følgende DCD (de samme klasser som før, men i en lidt anden opstilling):



I de forrige Disaheim-opgaver har du håndteret samlinger af bøger, amuletter og kurser via de tre public properties Books, Amulets og Courses i Controller-klassen, alle tre af Collection-typen List<T>, hvor T repræsenterer henholdsvis Book, Amulet og Course.

Der er endnu ikke vist klasser fra brugergrænsefladen i DCDet også betegnet UI-laget (UI = User Interface), f.eks. en Menu-klasse mm., men det vil som nævnt foroven være Controller-klassens ansvar (jf. GRASP) at håndtere hændelser fra UI-laget, og videreformidle disse hændelser til systemets forretnings­logik.

Der er følgende problemer i Controller-klassen, der falder i øjnene:

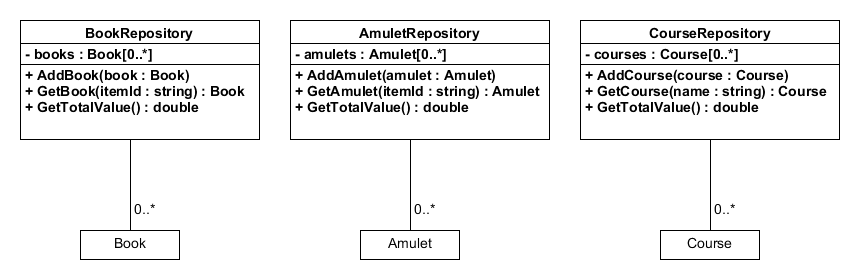
1. Controller-klassen har to ansvar og overholder dermed ikke Single Responsibility Principle (SRP).   
   Disse to ansvar er:
   1. Administrere bøger, amuletter og kurser i den interne datastruktur for de tre samlinger via koden inden i de overloadede AddToList()-metoder.
   2. Videreformidle hændelser fra UI-laget (kald til AddToList()-metoderne) videre til forretningslogikken, **hvilket burde være Controllerens eneste ansvar jf. GRASP**.
2. Den interne datastruktur for de tre samlinger er desuden helt eksponeret (public) i Controller-klassen og kan tilgås uden kontrol; f.eks. er det tydeligt, at List<T> er brugt som datatype for samlingerne (hvilket du måske gerne vil skjule for brugeren af Controller-klassen).

Man kunne løse problemet i pkt. 2 ved at gøre de tre samlinger private i stedet for public. Derved kan brugere af Controller-klassen (UI-laget) kun tilgå samlingerne via AddToList()-metoderne. Men dette løser ikke SRP-problemet i pkt. 1.

## Øvelse 2.1: Repository-klasser

Repository-mønstret kan løse begge udfordringer. Du fjerner først det uønskede ansvar fra Controller-klassen, nemlig at administrere bøger, amuletter og kurser i en intern datastruktur ved at introducere Repository-mønstret som et abstraktionslag, der ikke alene overtager ansvaret for opbevaring og tilgang til systemets data (bøger, amuletter og kurser), men også helt indkapsler den interne datastruktur for disse.

Det skal den følgende tilføjelse til DCD’et med 3 Repository-klasser råde bod på:



Og lige for at genopfriske (bemærk konventionen for felter):

**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

**Bemærk**: Benyt igen List<T> til at repræsentere de private felter *books*, *amulets* og *courses* i hver Repository-klasse foroven.

Her følger operationskontrakterne for *BookRepository*:

Operationskontrakt for **BookRepository.AddBook(book : Book)**

Cross reference: Budgetopfølgning

Precondition: Bogen *book* anvendt som parameter eksisterer

Postcondition: *book* er tilføjet til *books*

Operationskontrakt for **BookRepository.GetBook(itemId : string) : Book**

Cross reference: Budgetopfølgning

Precondition: Ingen

Postcondition: Ingen ændringer i systemet.

Output: Hvis der findes en bog i *books*, der har *itemID* som *ItemId*, returneres denne, ellers returneres *null* .

Operationskontrakt for **BookRepository.GetTotalValue() : double**

Cross reference: Budgetopfølgning

Precondition: Ingen

Postcondition: Ingen ændringer i systemet.

Output: Den samlede værdi af alle bøger i *books* returneres.

Der gælder tilsvarende operationskontrakter for AmuletRepository og CourseRepository. For sidstnævnte brug Name som parameter til GetCourse i stedet for ItemId.

Udfør følgende:

* Implementér de 3 Repository-klasser i projektet, hvor du husker at benytte dig af Utility-klassen i dit UtilityLib-bibliotek fra forrige opgave til at beregne den totale værdi for de enkelte produkter.

Du vil opleve, at du støder på et problem i forhold til at benytte Utility-klassen.  
Hvad tror du, problemet er?  
Overvej et par minutter, hvilke løsninger der kunne være til problemet, men gør ikke noget endnu.

Selvom om du har fundet flere måder at løse dette på, så:

* + kopierer du din Utility-klasse tilbage til Disheim-løsningen igen (husk også at tilrette utility-klassens namespace).

Dvs. du benytter dig alligevel ikke af det bibliotek, du har oprettet pga. de problemer, du stødte på.  
Sørg for, at Disaheim-løsningen og eksisterende tests kan kompileres uden fejl.

Vigtig læring: Når man laver et bibliotek, skal man sikre sig, at der **ikke er indbygget uhensigtsmæssige afhængigheder** til klasser uden for bibliotekets egne klasser.

## Øvelse 2.2: Test af repositories

Nu er det tid til at teste dine nye Repository-klasser.

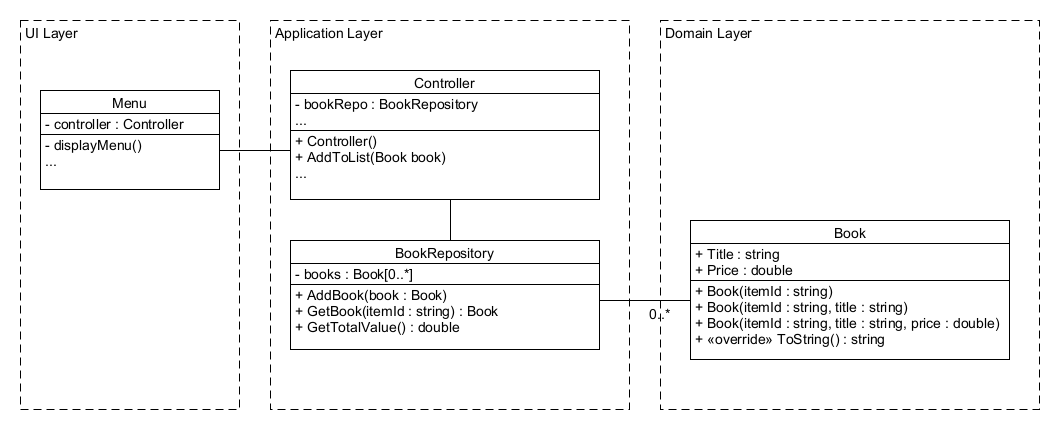
Udfør følgende:

* Opret en ny test-fil – *UnitTest4.cs* – i dit test-projekt ”DisaheimTest” som tidligere vist
* Indsæt indholdet af unit-testen ”*Ex18-Disaheim4.UnitTest4.docx*”, der ligger i materiale-repositoriet for programmering (benyt git pull i materialemappen)
* Få testen til at køre succesfuldt. Ret din kode om nødvendigt

# Øvelse 3: Lagdeling - Inspektion af DCD med Controller og Repository

Introduktionen af Repository-klasser løste de oprindelige problemer med SRP og mangel på indkapsling. Men lige så vigtigt, hvordan påvirkede det så Controller-klassen? I en lagdelt arkitektur spiller Controlleren en bestemt rolle.

Nedenunder er der et uddrag af en lagdelt DCD med en Controller (dog kun med fokus på Book).



Benyt dig af CL-strukturen **tænk par del**:

* **Individuelt:** inspicér DCD’et og overvej, hvorfor vi har en interesse i at benytte en Controller og Repository i en 3-lags arkitektur på viste måde (5 minutter)
* **Par**: del dine tanker med din sidemand (5 minutter)
* **Del**: del jeres tanker med de andre par i teamet

**Hint:** Tag udgangspunkt i forberedelsen til i dag (Larman og GRASP).

*Tidsramme: 15 minutter*

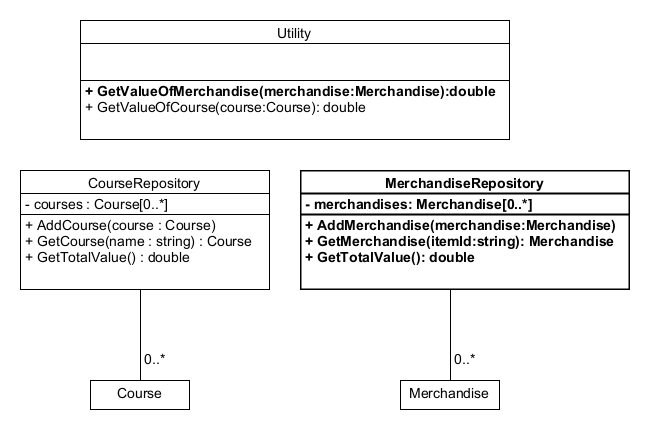
# Øvelse 4: Finjustering af Disaheim

## Øvelse 4.1: MerchandiseRepository

Ved nærmere eftersyn er det ikke heldigt, at *Book* og *Amulet* har hver deres repository samt metode i *Utility* til at returnere værdi. Hvis der kommer flere klasser til i arvehierarkiet under *Merchandise*, vil antallet af *GetValueOfXX* metoder samt repositories stige tilsvarende.

Da begge klasser er af typen *Merchandise*, burde det være muligt at lave et enkelt *MerchandiseRepository*, som erstatter de to: *BookRepository* og *AmuletRepository*.

På samme måde kan metoden *GetValueOfMerchandise(Merchandise merchandise)* erstatte de to metoder *GetValueOfBook* og *GetValueOfAmulet* i *Utility*-klassen. Denne metode skal naturligvis finde ud af, hvilken type parameteren *merchandise* virkelig er og så finde den korrekte værdi i forhold til de regler, der gælder for de pågældende typer. Udvalgte software designklasser fra DCD’et er vist forneden:



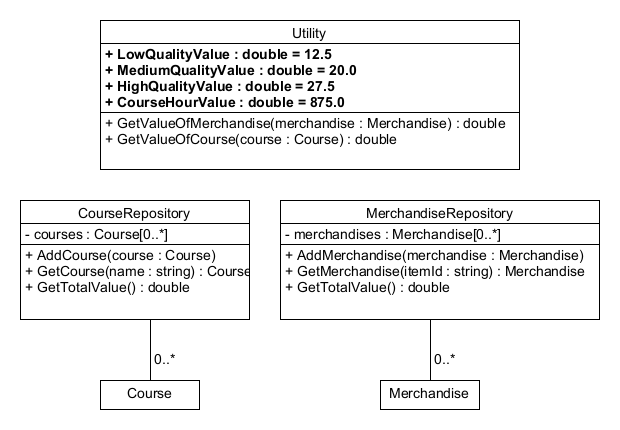
Udfør følgende:

* Implementér disse ændringer
* Refaktorér testmetoderne om nødvendigt (*Ja, du må gerne rette dem til - denne ene gang*)

## Øvelse 4.2: Priser som properties

Det er heller ikke hensigtsmæssigt, hvis priserne for amuletter er afhængige af kvalitet (12.5 for low, 20.0 for medium og 27.5) samt 875,00 pr påbegyndt time for et kursus er hard-coded som ”magic numbers” i klassen *Utility*. De bør være properties, så de kan sættes og ændres efter behov. Lad defaultværdierne for de 3 quality-properties være 12.5, 20.0 og 27.5, og 875,00 for property’en for påbegyndt time.

Nu ser ovenstående klasser således ud:



Udfør følgende:

* Implementér dette ved at initialisere de fire properties i klassen
* Refaktorér testmetoderne om nødvendigt