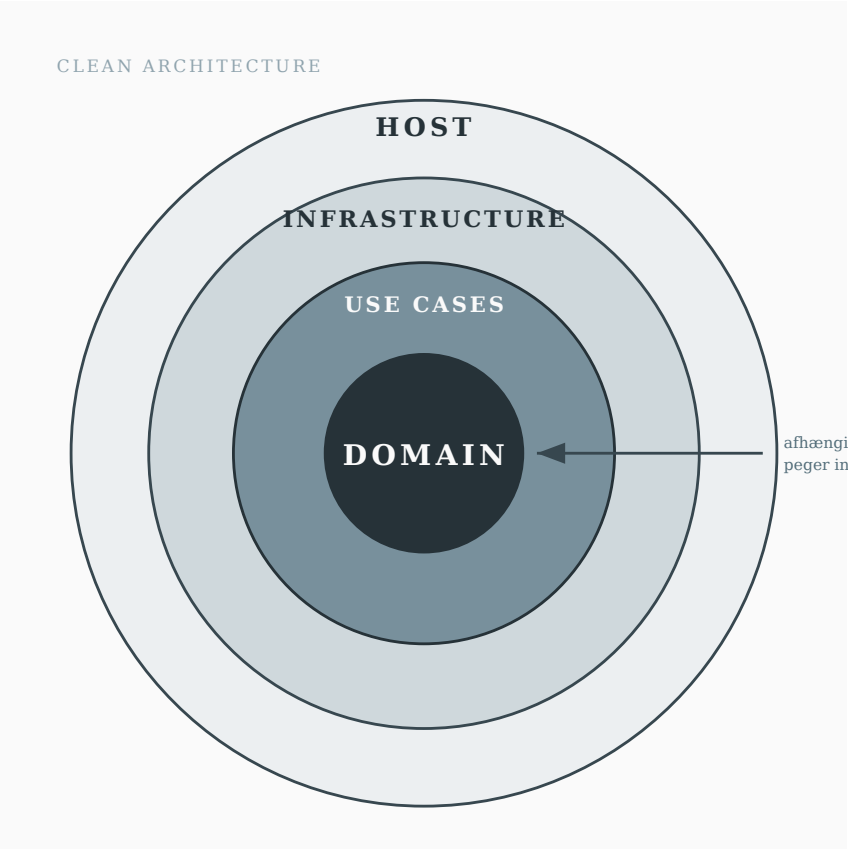
Software der holder

Professionel C#-udvikling med Clean Architecture, DDD og parallelisme



Figur 1: Clean Architecture i fire lag — Domain inderst, Host yderst. Pile peger indad: ydre lag kender de indre, aldrig omvendt.

Kaj Bromose

UCL Erhvervsakademi og Professionshøjskole

Førsteudgave · 2026

”Ikke bare kode der kompilerer, men kode der kan vedligeholdes, testes, skaleres, og forklares for andre.”

C# 13 · .NET 10 · EF Core 10 · xUnit v3 · Moq · Blazor Interactive Server

Kolofon

Software der holder — Professionel C#-udvikling med Clean Architecture, DDD og parallelisme

Førsteudgave, 2026

Forfatter: Kaj Bromose, UCL Erhvervsakademi og Professionshøjskole

Forlag: Bromose Publishing

Udgivelsesplatform: LeanPub (`leanpub.com`)

ISBN: 978-87-976951-1-1

Sat med: pandoc + xelatex. Kildetekst i Markdown.

Forside-grafik: Clean Architecture-diagram efter Robert C. Martin's koncept (*The Clean Architecture*, 2012). Tegnet til denne udgivelse.

Tekniske valg: C# 13, .NET 10, EF Core 10, xUnit v3, Moq, FixtureBuilder (Dennis Johnsen), Blazor Interactive Server, SQLite/SQL Server.

Reference-kode: `github.com/bromose/minklinik` (MIT-licens). Hvert kapitel har sit eget Git-tag (`kap-01` … `kap-25`).

Kontakt og bidrag: Hvis du finder fejl, har spørgsmål, eller vil bidrage med forbedringer, er du velkommen til at kontakte forfatteren via udgivelsesplatformen. Bogen vedligeholdes løbende — hver ny version dokumenterer ændringer i et changelog.

Indhold

Forord

Software der holder — professionel C#-udvikling med Clean Architecture, DDD og parallelisme.

Hvem er bogen til?

Denne bog er skrevet til dig der vil bygge professionel C#-software — ikke bare kode der kompilerer, men kode der kan vedligeholdes, testes, skaleres, og forklares for andre. Den er udviklet til datamatiker-uddannelsen ved UCL Erhvervsakademi og Professionshøjskole, men er anvendelig for tre slags læsere:

Datamatiker-studerende på 2. semester der har programmering og teknologi som fagområder. Hvert kapitel afsluttes med en *Hands-on*-boks der lægger op til praktisk anvendelse — typisk på et gruppeprojekt der bygges sammen med underviserstøtte. Konkrete acceptkriterier og uge-baseret progression for et 12-ugers kursusforløb finder du i `Bilag E – Underviser-noter`. Hvis kursusforløbet afsluttes med en individuel mundtlig eksamen, er `Bilag C – Læringsgæld` obligatorisk læsning for jeres studiegruppe.

Junior-udviklere i industrien der har C#-grundkendskab fra et bootcamp, et tidligere studie, eller selvstudie, og som vil lære at bygge enterprise-software efter moderne arkitektur-mønstre. Bogen er bygget op fra fundamenterne (indkapsling, SOLID), gennem domæne-modellering (DDD), arkitektur (Clean Architecture), brugergrænseflade (Blazor Interactive Server), til parallelisme og algoritmisk analyse. Hver Hands-on-boks fungerer som en øvelse du kan anvende på et eget projekt.

Selvstuderende der følger sin egen rejse — fx en hobby-udvikler der har lært C# alene og nu vil professionalisere sin praksis. Bogen kan læses lineært eller som opslagsværk; hvert kapitel henviser eksplicit til de tidligere kapitler det bygger på.

Forventet forudsætning er at du kender C#'s kontrolstrukturer, klasser og objekter, properties og konstruktører — typisk det stof man dækker i et 1. semester eller en introduktion til OOP.

Hvad lærer du?

Bogen er bygget op i seks dele over 25 kapitler:

1. **Fundament for programkvalitet** (kap. 1-5): indkapsling, SOLID, Dependency Inversion, kontrakter, testing.
2. **Domain-Driven Design** (kap. 6-8): hvorfor domænet er centrum, Entity/Value Object/Aggregate Root, always-valid model.
3. **Clean Architecture** (kap. 9-13): lagdeling, Use Cases og CQS, Facade-laget, EF Core 10, LINQ.
4. **Brugergrænseflade med Blazor Interactive Server** (kap. 14-18): HTML5 og CSS, komponenter, forms, CRUD, Strategy-mønstret.
5. **Parallelisme og samtidighed** (kap. 19-23): tråde, synkronisering, async/await, parallel udførelse, integration med Strategy.
6. **Helhedsbillede** (kap. 24-25): Big O og algoritmisk analyse, plus en samlet refleksion over hvad du har lært.

Bogen bygger op om to gennemgående eksempler:

- **MinKlinik** — det færdige reference-system du *læser om*. En klinik-bookings-applikation bygget i Clean Architecture med Domain, Use Cases, Facade, Infrastructure (EF Core 10), Console, Web API og Blazor Interactive Server. Kildekoden ligger på `github.com/bromose/minklinik` med tag pr. kapitel (`kap-01` … `kap-25`). Inden du går i gang med kapitel 1, anbefales det at læse `MinKlinik – bogens reference-domæne.md`, der introducerer aggregaterne, forretningsreglerne og lag-strukturen så du har den store helhed på plads.
- **Et eget projekt** — det system du *bygger*. Hands-on-boksene i hvert kapitel inviterer dig til at anvende kapitlets stof på et projekt du selv vælger: et personligt sideprojekt, en kodebase fra arbejdet, eller — hvis du følger et kursusforløb — det fælles SportZone-domæne fra Bilag E.

Bogens konventioner

Pædagogiske greb

Hvert kapitel åbner med **kapitel-ILOs** (Intended Learning Outcomes) markeret med SOLO-niveauer (*Multistructural*, *Relational*, *Extended Abstract*). SOLO-taksonomien er en pædagogisk model fra Biggs & Tang der beskriver dybden i en lærings-aktivitet. Hvis du underviser, giver markeringerne en ramme for vurdering. Hvis du selv læser, kan du læse målene som en checkliste over hvad du bør kunne efter kapitlet.

Hvert kapitel følger samme 8-sektions-skabelon:

1. Kapitel-ILOs
2. Forudsætninger fra tidligere kapitler
3. **N.1 Motivation** — concrete first, et konkret problem fra MinKlinik der motiverer kapitlets stof
4. **N.2 Begrebsapparat** — definitioner i tabel-form
5. **N.3 Kerneindhold** — den primære gennemgang med kodeeksempler
6. **N.4 Anvendelse i MinKlinik** — hvordan stoffet ser ud i den færdige reference-implementation
7. **N.5 Sammenhæng** — hvad kapitlet bygger på, og hvad der bygger på det
8. **N.6 Hands-on** — praktisk øvelse til et eget projekt
9. **N.7 Øvelser** — 3-5 opgaver med stigende SOLO-niveau (facit i Bilag B)
10. **N.8 Litteratur og videre læsning**

Tonalitet

Bogen henvender sig til *du* — altid, undtagen i citerede dialoger. Kort, aktiv, forklarende. Eksempler før definitioner. Når noget er vigtigt, gentages det.

Forkerte eksempler først

Mange afsnit viser den naive løsning *før* den rigtige. Markeret med:

> Naivt forsøg

```
// den enkle, men problematiske løsning
```

> Korrekt løsning

```
// den robuste version med begrundelse
```

Den her sekvens er pædagogisk bevidst: du oplever smerten ved den forkerte løsning før du møder den rigtige. Det gør lærdommen sidde.

Stop og tænk

Indlejret i teksten ved nøglemoment står:

> **Stop og tænk:** [refleksionsspørgsmål]

Stop dér. Tag 30 sekunder. Formulér et svar inden du læser videre. Det forskel mellem at *læse* og at *lære*.

Krydshenvisninger

Når et kapitel henviser til en anden del af bogen, bruges kapitelnummer og sektion: *"se §4.2"* eller *"jf. kap. 7 §7.3"*. Aldrig sidetal — bogen kan ændre form (skærm, print, PDF) uden at referencer brækker.

AI som læringsstøtte

Du sidder med stor sandsynlighed med en AI-assistent åben mens du læser. Det er fint — bogen er skrevet til virkeligheden i 2026. Men der er en stor forskel på at *bruge* AI til at lære og at *lade* AI lære for dig. Inden du går i gang med kapitel 1, anbefales det at læse `Bilag A — AI som læringsstøtte`, der giver dig konkrete prompt-skabeloner, anti-mønstre at undgå, og en arbejdsgang pr. kapitel.

Tekniske valg

Bogen er bygget på en konkret stack:

- **C# 13** og **.NET 10** (med `System.Threading.Lock`, `ComplexProperty` i EF Core 10, primary constructors, mv.)
- **xUnit v3** + **Moq** + **FixtureBuilder** (Dennis Johnsen) til testing
- **EF Core 10** til persistens, med SQLite eller SQL Server som backend
- **Blazor Interactive Server** til UI (ikke SSR, ikke WebAssembly)
- **Markdown-kilde** med rendering til både web og PDF

Du kan læse bogen uden at have C# 13 / .NET 10 installeret — koncepterne er principielle, ikke version-specifikke. Men kode-eksemplerne forudsætter den nye syntaks (fx `Lock`-typen og `ComplexProperty`).

Tak

Lærebogen står på skuldrene af mange. Særlig tak til:

- **Mark Lauridsen** — kollega på UCL — for de uddybende guides der refereres i flere kapitler (især om EF Core, LINQ, samtidighed, parallelisme, async/await og Big O). Hans materiale er indtegnet i bogens struktur.
- **Dennis Johnsen** for `FixtureBuilder`-NuGet-pakken der løser et reelt problem med always-valid aggregates i tests (kap. 5).
- **Kollegerne på datamatiker-uddannelsen ved UCL** for løbende sparring og pilotering af materialet.
- Forfatterne der har formet de underliggende ideer — særligt **Robert C. Martin** (Clean Architecture, SOLID), **Eric Evans** (Domain-Driven Design), **Vaughn Vernon** (Implementing DDD), **Vladimir Khorikov** (Unit Testing Principles), **Stephen Cleary** (Concurrency in C#), og **Mark Seemann** (Dependency Injection).

Bogens fejl og udeladelser er forfatterens egne.

Kontakt og bidrag

Forfatter: Kaj Bromose, UCL Erhvervsakademi og Professionshøjskole.

Hvis du finder fejl, har spørgsmål, eller vil bidrage med forbedringer, er du velkommen til at kontakte forfatteren. Bogen vedligeholdes løbende — hver ny version dokumenterer ændringer i et kort changelog.

Held og lykke med læsningen. Hvis du sidder med en specifik kodebase mens du læser, så afprøv hvert kapitel der. Den hurtigste vej til at lære at bygge software er at bygge software.

MinKlinik — bogens reference-domæne

> Hvert kapitel i *Software der holder* bruger MinKlinik som gennemgående eksempel. Det her dokument introducerer domænet samlet — så du har konteksten *før* du møder de første kodeeksempler i kapitel 1. Det er den slags introduktion en kollega ville give dig på kontoret den første morgen: *"Vi bygger et system til en klinik. Lad mig vise dig hvad det handler om…"*

Hvad er MinKlinik?

MinKlinik er et bookingsystem til en mindre lægeklinik. Receptionisten opretter konsultationer for patienter hos klinikkens behandlere. En patient kommer ind, vælger en tidspunkt og en behandlingstype, og får sin booking. Hvis hun ikke kan komme alligevel, kan bookingen aflyses. Når konsultationen er gennemført, afsluttes den med et journal-notat.

Systemet er bygget i C# 13 / .NET 10 efter Clean Architecture-mønstret. Det fungerer som *reference-implementation* for alt det bogen lærer dig — fra grundlæggende indkapsling i kap. 1 til parallel rabatberegning i kap. 23. Du behøver ikke køre koden mens du læser, men hvis du gør, vil du genkende stoffet trinvis efterhånden som det introduceres.

MinKlinik er bevidst lille. Den skal kunne læses og forstås i sin helhed inden for et semester — ikke være en realistisk produktions-applikation med alle de kompromiser der følger. Den fokuserer på at gøre arkitekturen, domænet og kvalitets-principperne *synlige*.

Domænet

Klinikken har fire centrale begreber:

- **Patient** — den person der bookes til en konsultation. Har et navn, et CPR-nummer og kontaktinformation.
- **Behandler** — den person der udfører konsultationen. Kan være en læge, sygeplejerske eller anden klinisk personale. Har et navn og et speciale.
- **Behandlingstype** — den slags konsultation der bookes (almindelig konsultation, vaccination, recept-fornyelse, kontrol). Hver type har sin egen pris.
- **Konsultation** — selve bookingen. Forbinder en patient, en behandler og en behandlingstype til et bestemt tidspunkt. Har en livscyklus: *Planlagt* → *Afsluttet* eller *Planlagt* → *Aflyst*.

I DDD-termer er alle fire **Aggregate Roots** — de er hver især selvstændige entiteter med egen identitet og livscyklus. Konsultation er den centrale rod der binder de tre andre sammen via Guid-references.

Plus ét vigtigt **Value Object**:

- **Tidsinterval** — et `Fra-Til`-par der repræsenterer et tidsrum. Implementeret som en C# `record` med en `OverlapperMed`-metode der bruges til at tjekke at to konsultationer ikke kolliderer.

Og ét **Enum**:

- **KonsultationStatus** — `Planlagt`, `Afsluttet` eller `Aflyst`. Tilstandsmaskinen styres af `Konsultation`'s metoder.

Forretningsreglerne

MinKlinik håndhæver fem domænereglerne, og du møder dem alle i bogen:

1. **Tidspunkt skal ligge i fremtiden.** En konsultation kan ikke oprettes med et tidspunkt der allerede er passeret. Tjekkes i `Konsultation.Opret`.
2. **Ingen overlap pr. patient.** En patient kan ikke have to konsultationer der overlapper i tid. Tjekkes med `Tidsinterval.OverlapperMed`.
3. **Ingen overlap pr. behandler.** Tilsvarende — en behandler kan kun være ét sted ad gangen.
4. **Status-overgange er ensrettede.** En afsluttet konsultation kan ikke aflyses. En aflyst kan ikke afsluttes. En allerede afsluttet konsultation kan ikke afsluttes igen.
5. **Notat er påkrævet ved afslutning.** Når en konsultation afsluttes, skal der være et journal-notat — ikke tom.

Alle fem regler bor i `Konsultation`-klassen i Domain-laget. De er håndhævet med guard clauses og kaster `DomainException` hvis de brydes. Det her er kerne-eksemplet på den *always-valid model* du lærer i kap. 8.

Arkitektur — Clean Architecture i fem lag

MinKlinik er organiseret i fem .NET-projekter plus tre alternative host-projekter:

```
MinKlinik/
├── src/
│   ├── MinKlinik.Domain/          ← inderst, ingen afhængigheder
│   │   ├── Entities/              (Konsultation, Patient, Behandler, Behandlingstype)
│   │   ├── ValueObjects/          (Tidsinterval)
│   │   ├── Enums/                 (KonsultationStatus)
│   │   └── Exceptions/            (DomainException, NotFoundException)
│   │
│   ├── MinKlinik.Facade/          ← leaf — ingen afhængigheder
│   │   ├── UseCases/              (interfaces — IOpretKonsultationUseCase osv.)
│   │   ├── Queries/               (interfaces — IKonsultationQueries osv.)
│   │   └── DTOs/                  (records — kun primitiver)
│   │
│   ├── MinKlinik.UseCases/        ← orkestrerer (refererer Domain + Facade)
│   │   ├── Konsultationer/        (OpretKonsultationUseCase osv.)
│   │   └── IRepositories.cs       (Repository-interfaces)
│   │
│   ├── MinKlinik.Infrastructure/  ← persistens (refererer Domain + Facade + UseCases)
│   │   ├── Persistence/           (AppDbContext)
│   │   ├── Repositories/          (KonsultationRepository osv.)
│   │   ├── QueryHandlers/         (KonsultationQueriesImpl osv.)
│   │   └── SeedData.cs
│   │
│   ├── MinKlinik.Api/             ← REST-host (composition root)
│   ├── MinKlinik.Blazor/          ← Blazor Interactive Server-host
│   └── MinKlinik.Console/         ← konsolapp-host
│
└── tests/
    ├── MinKlinik.Domain.Tests/    (xUnit + FixtureBuilder)
    └── MinKlinik.UseCases.Tests/  (xUnit + Moq)
```

De tre host-projekter (Api, Blazor, Console) er alternative ydre skalere. Kun ét bruges ad gangen i en konkret deployment, men de deler alt det indre. Det illustrerer den centrale Clean Architecture-værdi: *forretningen ændres ikke når UI'en gør.*

Du møder lagdelingen formelt i kap. 9 (Lagdeling og Dependency Rule). De foregående kapitler (1-8) bygger gradvis fundamenterne *uden* at navngive arkitekturen — så den fremstår naturligt som konsekvensen, ikke som en udefra-pålagt tvang.

Tekniske valg (låst)

For at give bogens kodeeksempler en konkret reference er MinKlinik bygget på:

Lag	Teknologi
Sprog	C# 13
Runtime	.NET 10
Persistens	EF Core 10 (med SQLite eller SQL Server som backend)
Web	Blazor Interactive Server (ASP.NET Core 10)
API	ASP.NET Core 10 + Scalar.AspNetCore (OpenAPI)
DI	`Microsoft.Extensions.DependencyInjection`
Test	xUnit v3 + Moq + FixtureBuilder
Synkronisering	`System.Threading.Lock` (.NET 9+ / C# 13)

Hvis du arbejder på en lidt ældre stack (fx .NET 8) kan du følge med i bogens *principper*, men nogle konkrete eksempler — `System.Threading.Lock` (kap. 20), `ComplexProperty` (kap. 12), primary constructors og kollektion-expressions — kræver C# 13 / .NET 10.

Hvor finder du hvad?

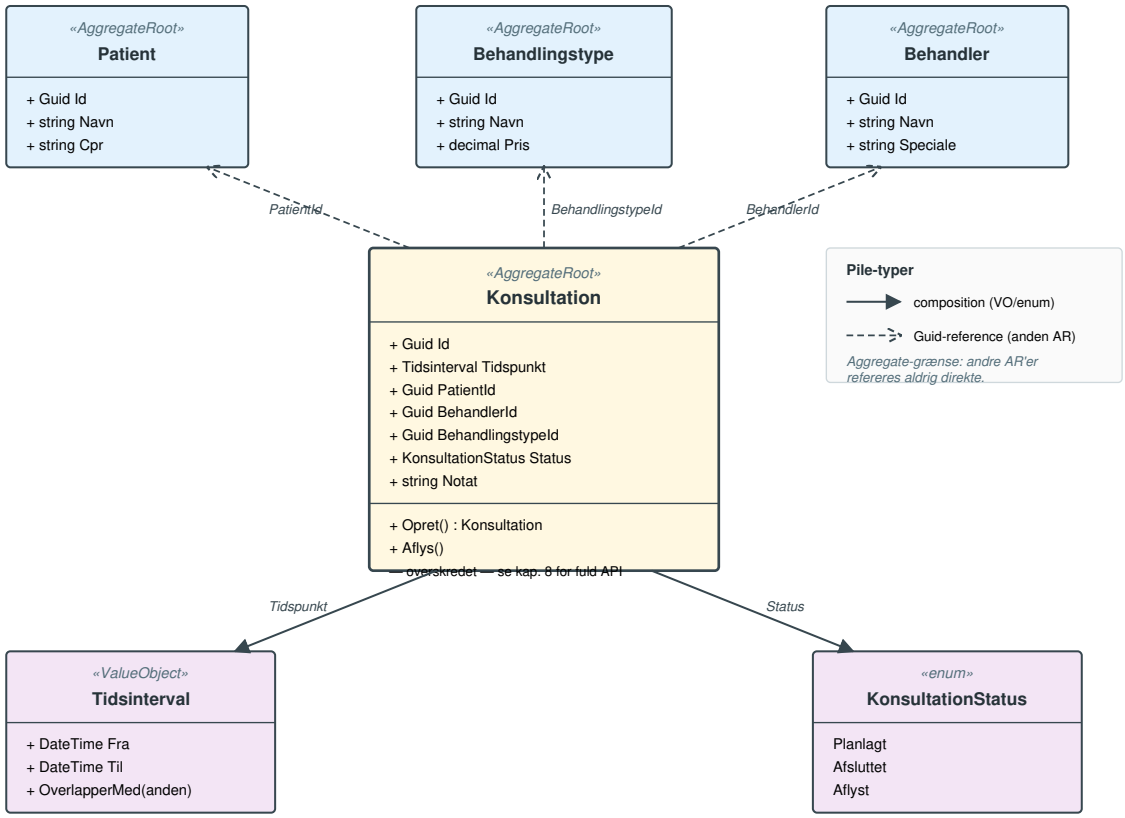
Kapitlerne afslører MinKlinik trinvis. Den her tabel viser hvor i bogen hver del af systemet introduceres:

Del af MinKlinik	Først introduceret i	Bygges færdig i
`Konsultation` med `private set` og guard clauses	Kap. 1 §1.4	Kap. 8 (factory + always-valid)
`INotifikation` + `EmailNotifikation` (interface og polymorfi)	Kap. 1 §1.3.9-10	Kap. 18 (Strategy)
`KonsultationRepository`-implementation med EF Core	Kap. 12 §12.4	Kap. 13 (LINQ-tuning)
`IKonsultationQueries` + `KonsultationQueriesImpl` (CQS)	Kap. 11 §11.4	Kap. 13 (AsNoTracking)
`MinKlinik.Domain` som leaf-projekt	Kap. 9 §9.4	Kap. 12 (DbContext-mapping uden domain-forurening)
`MinKlinik.Facade` som leaf-projekt	Kap. 11 §11.4	(uændret)
`MinKlinik.Blazor` med Interactive Server	Kap. 15 §15.4	Kap. 17 (komplet CRUD)
`MinKlinik.Api` med Scalar	Kap. 12 §12.4 (composition root)	Kap. 17 (alternative host)
`MinKlinik.Console` som primær DI-eksempel	Kap. 3 §3.4	(uændret)

Del af MinKlinik	Først introduceret i	Bygges færdig i
`BeregnRabatterParallelt`-use case	(introduktion)	Kap. 23 (parallel Strategy)
`BehandlingstypeCache` (Singleton + lock)	Kap. 20 §20.4	(uændret)

Du behøver ikke at have hele billedet i hovedet for at komme i gang. Bogen samler trådene op trinvist. Men når du undrer dig over *hvor* en bestemt klasse i MinKlinik bor i den større struktur, kan du komme tilbage til det her dokument.

Visualisering — domæne-modellen



Figur 2: MinKlinik domæne-model: Konsultation som central aggregate root med Tidsinterval (value object) og KonsultationStatus (enum) som komposition, samt Guid-references til de øvrige aggregate roots Patient, Behandler og Behandlingstype.

Bemærk pile-typerne:

- **Direkte arrow (`-->`):** Konsultation har et `Tidsinterval` som value object og en `KonsultationStatus` som enum-værdi. De er en *del* af Konsultations-aggregatet.
- **Dotted arrow (`..>`):** Konsultation har Guid-references til Patient, Behandler og Behandlingstype — ikke direkte object-references. Det er aggregate-grænsen fra DDD: hver aggregate referer andre aggregates via deres Id, ikke deres instans (kap. 7 §7.3.3).

Hvor finder du koden?

Kildekoden til MinKlinik bor i sit eget GitHub-repository:

> **`github.com/bromose/minklinik`**

Klon det med:

```
git clone https://github.com/bromose/minklinik.git
```

Repositoriet er versions-styret med Git, og hvert kapitel har sit eget tag (`kap-01`, `kap-02`, … `kap-25`) der lader dig checke kodebasens tilstand ud som den ser ud ved kapitlets afslutning:

```
git checkout kap-08      # se koden som den ser ud efter kapitel 8
git checkout main        # tilbage til seneste version
```

Det giver dig mulighed for at *læse koden trinvis* sammen med bogen, eller hoppe direkte til den endelige version (`kap-25`) hvis du vil se det færdige system. Hvis du følger et undervisningsforløb, vil din underviser have peget dig på en specifik branch eller release-tag.

Et beslægtet projekt: SportZone

Hvor MinKlinik er det system du *læser om*, er SportZone (en fitness-klub) et parallelt sample-domæne der inviterer dig til at *bygge sammen* med din studiegruppe — typisk med underviserstøtte i form af ugentlige sparringsmøder og code reviews. Hands-on-boksen i hvert kapitel (§N.6) kan udføres på SportZone, eller du kan vælge dit eget projekt.

SportZone-domænet er beskrevet kort i `Bilag E — Underviser-noter` for de der følger et organiseret kursusforløb. Hvis du læser bogen som selvstuderende, kan du tilpasse hands-on-aktiviteterne til hvilket projekt du nu sidder med — fx en kodebase fra arbejdet eller et personligt sideprojekt.

Hvis du følger et kursusforløb der afsluttes med en eksamen, bygges eksamensprojektet selvstændigt af eksamensgrupperne under hensyntagen til *læringsgæld* — princippet om at viden og kompetencer ikke må samles hos få i gruppen, fordi den individuelle eksamen tester transfer hos hver enkelt. Læringsgæld er beskrevet i `Bilag C — Læringsgæld`.

Pointen er at *du bygger noget*. Bogen lærer dig værktøjerne; det er først når du anvender dem at de bliver dine.

God læselyst

Når du nu går i gang med kapitel 1, vil den første kodeeksempel være en `Konsultation`-klasse med `public set` på alle properties — den naive version. Du vil måske genkende den fra et eget projekt. I løbet af bogen ser du den vokse, blive disciplineret, blive testet, blive lagdelt — indtil du i kapitel 25 har set hele rejsen. Velkommen til MinKlinik.

Kapitel 1 — Indkapsling, kohæsion og kobling

Kapitel-ILOs

Efter at have læst dette kapitel kan du:

- *(Multistructural → Relational)*: **Redegøre** for indkapsling som princip og **forklare** hvorfor `private set` er fundamentet for objektorienteret kvalitet.
- *(Relational)*: **Identificere** brud på indkapsling i eksisterende kode og **omskrive** koden så indkapslingen overholdes.
- *(Relational)*: **Forklare** kohæsion og kobling som begrebspar og **vurdere** om en given klasse har høj eller lav kohæsion.
- *(Relational)*: **Anvende** `private set`, konstruktør-validering og guard clauses til at beskytte en klasses tilstand mod ugyldige tilstande.
- *(Relational)*: **Forklare** hvordan interface og polymorfi reducerer kobling og åbner for at udvide kode uden at modificere eksisterende klasser.

  Forudsætninger: Du har gennemført 1. semester og kender C#'s kontrolstrukturer, klasser og objekter, properties og konstruktører.

1.1 Motivation

Forestil dig at du arbejder på en klinik. Receptionisten skal oprette en booking til en patient. Hun klikker "Opret", men taster ved en fejl `2024` i stedet for `2026`. Booking'en oprettes — i fortiden.

Senere ringer patienten og spørger hvorfor hun ikke har fået en bekræftelse. Receptionisten kigger i systemet og kan ikke finde ud af hvad der gik galt. Tre timer senere opdager I problemet i loggen: booking-objektet var blevet oprettet med en `Tidspunkt`-værdi i fortiden, og scheduleringen havde derfor sprunget den over.

Hvis vi kigger i koden, ser den måske sådan her ud:

```
public class Konsultation
{
    public DateTime Tidspunkt { get; set; }
    public Guid PatientId { get; set; }
    public Guid BehandlerId { get; set; }
    public KonsultationStatus Status { get; set; }
}
```

Den ser uskyldig ud. Bare en datapose med fire properties. Men problemet er at *alle steder* i kodebasen kan skrive hvad som helst til `Tidspunkt`. Der er ingen regel der håndhæves. Der er ingen "dørmand". Klassen *kan* være i en ugyldig tilstand — og den var det.

Det her kapitel handler om hvordan man bygger klasser så *de aldrig kan være i en ugyldig tilstand*. Det er fundamentet for alt det vi kommer til at bygge i de næste kapitler: SOLID, DDD, Clean Architecture.

Det hele begynder med at vi tager kontrollen tilbage over vores data.

1.2 Begrebsapparat

Begreb	Forklaring
Indkapsling *(encapsulation)*	At skjule en klasses interne tilstand og kun give kontrolleret adgang til den udefra.
Tilstand *(state)*	De værdier en klasses felter og properties har på et givet tidspunkt.
Invariant	En regel der altid skal være sand for et objekt, uanset hvilken metode der lige er kaldt.
Guard clause	En tjek-sætning øverst i en metode eller konstruktør der afviser ugyldige input før de når at gøre skade.
Kohæsion *(cohesion)*	Hvor sammenhørigt indholdet i en klasse er. Høj kohæsion = klassen handler om én ting.
Kobling *(coupling)*	Hvor afhængig en klasse er af andre klasser. Lav kobling = klassen kender få andre.

1.3 Kerneindhold

1.3.1 Det naive forsøg — public set

Den nemmeste måde at lave en klasse på er den vi så i åbningen:

Naivt forsøg — virker ikke

```
public class Konsultation
{
    public DateTime Tidspunkt { get; set; }
    public Guid PatientId { get; set; }
    public Guid BehandlerId { get; set; }
    public KonsultationStatus Status { get; set; }
}
```

Det her er en *anæmisk* klasse — en datapose. Den har ingen regler. Den ved ikke hvad den selv handler om. Resten af koden kan tilgå den frit:

```
var k = new Konsultation();
k.Tidspunkt = new DateTime(2024, 1, 1); // i fortiden — fint
k.PatientId = Guid.Empty;               // ingen patient — fint
k.Status = KonsultationStatus.Aflyst;   // aflyst før den blev oprettet — fint
```

Resultatet: Du kan ikke stole på at en `Konsultation` du henter fra databasen overhovedet er meningsfuld. Du må tjekke alle felter for hånd, alle steder, hver gang.

1.3.2 Hvad indkapsling betyder

Indkapsling siger: *"Klassen ejer sin egen tilstand. Andre må gerne kigge, men ikke ændre direkte."*

I C# realiseres det med to skift:

1. `set` bliver til `private set` — så ingen udefra kan skrive til en property.
2. Klassen får *metoder* der beskriver hvad man må gøre, og som validerer reglerne.

Korrekt løsning

```
public class Konsultation
{
    public DateTime Tidspunkt { get; private set; }
    public Guid PatientId { get; private set; }
    public Guid BehandlerId { get; private set; }
    public KonsultationStatus Status { get; private set; }

    public Konsultation(DateTime tidspunkt, Guid patientId, Guid behandlerId)
    {
        if (tidspunkt < DateTime.UtcNow)
            throw new ArgumentException("Tidspunkt må ikke ligge i fortiden.");
        if (patientId == Guid.Empty)
            throw new ArgumentException("PatientId er påkrævet.");
        if (behandlerId == Guid.Empty)
            throw new ArgumentException("BehandlerId er påkrævet.");

        Tidspunkt = tidspunkt;
        PatientId = patientId;
        BehandlerId = behandlerId;
        Status = KonsultationStatus.Planlagt;
    }

    public void Aflys() => Status = KonsultationStatus.Aflyst;
}
```

Forskellen ser måske lille ud. Den er ikke lille. Det her er forskellen mellem *"jeg håber data er rigtig"* og *"data ER rigtig — ellers eksisterer objektet ikke"*.

> **Stop og tænk:** Hvad sker der i den nye version hvis nogen prøver at skrive `k.Tidspunkt = new DateTime(2024, 1, 1)`? Hvad er fejlmeddelelsen?

1.3.3 Konstruktøren som dørmand

Konstruktøren er objektets fødestue. Hvis du gør det umuligt at fødes ugyldig, eksisterer der aldrig et ugyldigt objekt i din kode. Det er ekstremt kraftfuldt.

De små `if (...) throw`-linjer øverst i konstruktøren kaldes **guard clauses**. De er aktive *før* objektet bliver til. Hvis de fejler, kastes en `ArgumentException`, og objektet oprettes aldrig.

Det betyder også at *du senere i koden ikke behøver tjekke at felterne er gyldige*. Hvis du har en `Konsultation` i hånden, så ER den gyldig. Det er en garanti.

1.3.4 Metoder som dørmænd

På samme måde som konstruktøren håndhæver gyldighed ved oprettelse, skal metoder håndhæve gyldighed ved ændring.

```
public void Afslut(string notat)
{
    if (Status != KonsultationStatus.Planlagt)
        throw new InvalidOperationException("Kun planlagte konsultationer kan
        ↪ afsluttes.");
    if (string.IsNullOrWhiteSpace(notat))
        throw new ArgumentException("Notat er påkrævet ved afslutning.");

    Status = KonsultationStatus.Afsluttet;
    Notat = notat;
}
```

Bemærk at metoden hedder `Afslut(notat)` — ikke `SetStatus(...)`. Forskellen er pædagogisk. Med

`public set` siger du *"sæt status til X"*. Med en metode siger du *"afslut konsultationen, her er notatet"*. Det første er en teknisk operation. Det andet er en forretningshandling.

> **Stop og tænk:** Læs disse to linjer: `- k.Status = KonsultationStatus.Afsluttet; - k.Afslut("Patient anbefales kontrol om 6 måneder.");`
>
> Hvilken af de to fortæller en historie? Hvilken er nemmest at logge eller validere?

1.3.5 Hvad har vi opnået?

Indkapslingen giver os tre konkrete fordele:

Dataintegritet. Vi kan stole på at en `Konsultation` altid er i en gyldig tilstand. Tidspunkt er aldrig i fortiden. Status-overgange følger reglerne.

Læsbarhed og hensigt. Koden fortæller en historie i stedet for at flytte data. `k.Afslut(notat)` er forretningssprog. `k.Status = ...` er teknisk ledningsføring.

Vedligeholdbarhed. Hvis reglen *"konsultationer skal være mindst 15 minutter fra oprettelsestidspunktet"* tilføjes senere, ændrer vi den ét sted: konstruktøren. Med `public set` ville vi skulle finde alle steder i kodebasen hvor `Tidspunkt` blev sat.

1.3.6 Fra indkapsling til kohæsion

Indkapsling beskytter klassen indefra. Men der er stadig et åbent spørgsmål: *handler klassen overhovedet om det rigtige?*

Se på denne klasse:

Naivt forsøg — virker, men handler om for meget

```
public class Konsultation
{
    public DateTime Tidspunkt { get; private set; }
    public Guid PatientId { get; private set; }
    public Guid BehandlerId { get; private set; }
    public KonsultationStatus Status { get; private set; }

    public void Afslut(string notat) { /* ... */ }
    public void SendBekræftelseEmail(string emailServer) { /* SMTP-kald */ }
    public void GemTilDatabase(string connectionString) { /* SQL */ }
    public string FormatérTilPdf() { /* PDF-rendering */ }
}
```

Klassen "virker". Men den handler nu om fire ting på én gang: forretningsregler, e-mail, database og PDF. Det er en klassisk klump.

Kohæsion er ordet for hvor sammenhørig en klasse er. Høj kohæsion = klassen handler om én ting. Lav kohæsion = klassen er et lager af løs funktionalitet.

Der er to enkle spørgsmål du kan stille:

1. *Kan jeg formulere klassens ansvar i én sætning?* Hvis det kræver "og", "samt" eller "også", er kohæsionen lav.
2. *Kan jeg ændre én ting uden at risikere at ødelægge en anden?* Hvis e-mail-koden og forretningsreglerne lever sammen, vil små ændringer det ene sted påvirke det andet.

Korrekt løsning — én klasse = ét ansvar

```
public class Konsultation { /* kun forretningsregler */ }
public class KonsultationRepository { /* database-kald */ }
public class EmailNotifier { /* e-mail */ }
public class KonsultationPdfBuilder { /* PDF-rendering */ }
```

Det er præcis det mønster du møder igen i Clean Architecture (kap. 9-12), hvor de fire ting endda placeres i hver sit lag.

> **Stop og tænk:** En kollega har lavet en klasse `BookingHelper` der hedder fordi den "hjælper med booking-ting". Hvad er det første tegn på at kohæsionen er lav?

1.3.7 Kobling — hvem kender hvem?

Kobling er det andet halve af kvalitetsbilledet. Mens kohæsion handler om hvad der ligger *inde i* en klasse, handler kobling om hvor mange andre klasser den kender *udadtil*.

Se denne klasse:

```
public class OpretKonsultationService
{
    public void Opret(DateTime tidspunkt, Guid patientId, Guid behandlerId)
    {
        var conn = new SqlConnection("Server=...");
        conn.Open();
        var cmd = new SqlCommand("INSERT INTO Konsultationer ...", conn);
        cmd.ExecuteNonQuery();

        var smtp = new SmtpClient("mail.minklinik.dk");
        smtp.Send(new MailMessage(...));

        File.WriteAllText("c:\\logs\\konsultationer.log", $"Oprettet {tidspunkt}");
    }
}
```

Klassen er afhængig af: - `SqlConnection` (database) - `SmtpClient` (e-mail-server) - `File` (filsystem)

Hvis databasen skifter til PostgreSQL, ændrer klassen sig. Hvis e-mail-serveren ligger et andet sted, ændrer klassen sig. Hvis loggen skal til en cloud-tjeneste, ændrer klassen sig. *Hver* infrastruktur-ændring rammer denne klasse.

Lav kobling betyder at klassen ikke kender til konkrete teknologier — den taler kun til abstraktioner (interfaces). Det er præcis hvad SOLID's *Dependency Inversion Principle* (kap. 3) handler om, og hvad Clean Architecture's lagdeling (kap. 9) gør i stor skala.

For nu er pointen denne: høj kobling gør koden skør. Lav kobling gør den fleksibel. Det er bedre at to klasser kender ét fælles interface end at de kender hinandens implementeringsdetaljer.

1.3.8 Indkapsling, kohæsion og kobling — sammen

De tre begreber er ikke uafhængige.

- **Indkapsling** beskytter klassen *indefra*: ugyldige tilstande kan ikke opstå.
- **Kohæsion** holder klassen *fokuseret*: den handler kun om én ting.
- **Kobling** holder klassen *fri af omverdenen*: den kender få andre.

Tilsammen er de fundamentet for det vi kommer til at kalde *strukturel softwarekvalitet*. SOLID-principperne i kap. 2-3 er fem konkrete regler der styrker præcis disse tre egenskaber. DDD i kap. 6-8 er en tankegang der bruger dem til at modellere forretningens regler. Clean Architecture i kap. 9-12 er en arkitektur der gør dem skalérbare gennem hele systemet.

Men før du går videre til SOLID, mangler du to konkrete C#-byggesten der gør lav kobling muligt i praksis: *interface* og *polymorfi*. De to næste afsnit introducerer dem.

1.3.9 Interface — en kontrakt frem for en konkret type

I §1.3.7 så du at høj kobling gør koden skør. Spørgsmålet er nu *hvordan* du i praksis sænker koblingen i C#. Et af de vigtigste svar hedder **interface**.

Et interface er en *kontrakt*: det beskriver *hvad* en klasse skal kunne, men ikke *hvordan* den gør det. Det er en aftale mellem to parter — én der bruger noget, og én der leverer det.

Forestil dig at en `Konsultation` skal sende en bekræftelse til patienten. Hvis koden direkte instantierer en `EmailNotifikation`-klasse, er den koblet til den specifikke implementation:

Naivt forsøg — direkte kobling til en konkret klasse

```
public class OpretKonsultationFlow
{
    public void Opret(Konsultation k)
    {
        // ... opret konsultation ...
        var notifier = new EmailNotifikation();
        notifier.SendBekræftelse(k);
    }
}
```

Hvis klinikken senere vil skifte til SMS-bekræftelse, skal `OpretKonsultationFlow` ændres. Hvis den vil sende både og, skal flowet ændres igen. Hver leverandør-ændring rammer flowet.

Med et interface vendes afhængigheden om: flowet beder om noget der *kan* sende en bekræftelse, og hvilken slags overlades til den der konstruerer flowet.

Korrekt løsning — kontrakt frem for konkret type

```
public interface INotifikation
{
    Task SendBekræftelseAsync(Konsultation k);
}

public class EmailNotifikation : INotifikation
{
    public Task SendBekræftelseAsync(Konsultation k)
    {
        // SMTP-kald
        return Task.CompletedTask;
    }
}

public class SmsNotifikation : INotifikation
{
    public Task SendBekræftelseAsync(Konsultation k)
    {
        // SMS gateway-kald
        return Task.CompletedTask;
    }
}

public class OpretKonsultationFlow
{
    private readonly INotifikation _notifier;

    public OpretKonsultationFlow(INotifikation notifier)
        => _notifier = notifier;

    public async Task Opret(Konsultation k)
    {
        // ... opret konsultation ...
        await _notifier.SendBekræftelseAsync(k);
    }
```

```
}
```

Tre nye ting at lægge mærke til:

1. **`public interface INotifikation`** — interfacet beskriver hvad en notifier *kan*, ikke hvordan. C#-konvention er at interface-navne starter med `I`.
2. **`: INotifikation`** — en klasse erklærer at den implementerer kontrakten. Compileren sikrer at den faktisk gør det (alle metoder fra interfacet skal være implementeret).
3. **`private readonly INotifikation _notifier`** — flowet kender kun interfacet, ikke implementationerne. Den ved ikke om den modtager en `EmailNotifikation`, en `SmsNotifikation`, eller noget tredje.

Resultatet: `OpretKonsultationFlow` er nu *frikoblet* fra notifikations-teknologien. Skifter klinikken fra e-mail til SMS, ændres flowet ikke. Tilføjes der senere en `PushNotifikation`, ændres flowet ikke. Det er lav kobling i praksis.

> **Stop og tænk:** Hvad sker der hvis klinikken vil have *både* e-mail og SMS sendt? Skal flowet ændres? Tip: hvad nu hvis konstruktøren tog `IEnumerable<INotifikation>` i stedet for et enkelt `INotifikation`?

1.3.10 Polymorfi — forskellige svar bag samme spørgsmål

Polymorfi betyder *"mange former"*. I C# betyder det at samme metode-kald kan have forskellig opførsel afhængigt af hvilken konkret type der ligger bag interfacet på køretidspunktet.

Stop-og-tænk-spørgsmålet ovenfor pegede på løsningen. Hvis klinikken vil sende både e-mail og SMS, lader du konstruktøren tage en samling af notifiers:

```
public class OpretKonsultationFlow
{
    private readonly IEnumerable<INotifikation> _notifiers;

    public OpretKonsultationFlow(IEnumerable<INotifikation> notifiers)
        => _notifiers = notifiers;

    public async Task Opret(Konsultation k)
    {
        // ... opret konsultation ...
        foreach (var notifier in _notifiers)
            await notifier.SendBekræftelseAsync(k);
    }
}
```

Bemærk hvad der sker i `foreach`-løkken. Linjen `await notifier.SendBekræftelseAsync(k)` er *den samme* for alle notifiers, men den udfører forskellig kode afhængigt af om `notifier` faktisk er en `EmailNotifikation`, en `SmsNotifikation`, eller noget helt tredje. Compileren ved ikke hvilken implementation der kaldes — det afgør runtime baseret på den konkrete type.

Det er polymorfi: ét kald, mange svar.

Hvorfor er polymorfi vigtigt?

To grunde.

Lav kobling: `OpretKonsultationFlow` er totalt uvidende om hvor mange notifikations-typer der findes. Hver ny implementation kan tilføjes uden at flowet ændres.

Udvidelse uden modifikation: Vil klinikken senere have en `PushNotifikation`? Tilføj klassen, registrér den hos den der konstruerer flowet (typisk en IoC-container, kap. 3), og det er det. Eksisterende kode rører du ikke.

Den anden idé — *"udvidelse uden modifikation"* — er så vigtig at den har sit eget navn i SOLID: **Open-Closed Principle** (OCP). Du møder den i kap. 2.

Polymorfi via interface vs arv

Polymorfi kan også opnås gennem klasse-arv (`abstract class` med `virtual` metoder). Forskellen er:

- **Interface-polymorfi** (det her afsnit): klasser der implementerer samme kontrakt, kan udskiftes. Klasserne kan være helt urelaterede.
- **Arv-polymorfi**: subklasser der overrider en basetypens metoder. Kun klasser i samme arv-hierarki kan udskiftes.

I bogens øvrige kapitler bruger du primært *interface-polymorfi* — det er den mest fleksible form, og den der giver lavest kobling. Arv-polymorfi vender vi tilbage til i kap. 2 i forbindelse med Liskov Substitution Principle.

> **Stop og tænk:** En kollega har lavet en `EmailNotifikation`-klasse direkte uden interface. Senere vil hun skifte til SMS. Hvor meget kode skal hun ændre — og hvor meget havde hun skullet ændre, hvis hun havde brugt et interface fra start?

1.4 Anvendelse i MinKlinik

Lad os se hvordan principperne ser ud i den rigtige MinKlinik-løsning. Tag `kap-01` i `MinKlinik`-repositoriet og åbn `Konsultation.cs`.

```
public class Konsultation : AggregateRoot
{
    public Tidsinterval Tidspunkt { get; private set; }
    public Guid BehandlingstypeId { get; private set; }
    public Guid PatientId { get; private set; }
    public Guid BehandlerId { get; private set; }
    public string Notat { get; private set; } = string.Empty;
    public KonsultationStatus Status { get; private set; }

    private Konsultation() { } // EF Core

    private Konsultation(Tidsinterval tidspunkt,
        Guid behandlingstypeId, Guid patientId, Guid behandlerId)
    {
        // Guard clauses – håndhæver invariants ved fødsel
        if (tidspunkt.Fra < DateTime.UtcNow)
            throw new DomainException("Tidspunkt må ikke ligge i fortiden.");

        Id = Guid.NewGuid();
        Tidspunkt = tidspunkt;
        BehandlingstypeId = behandlingstypeId;
        PatientId = patientId;
        BehandlerId = behandlerId;
        Status = KonsultationStatus.Planlagt;
    }

    public static Konsultation Opret(/* ... */) { /* factory-metode */ }

    public void Aflys()
    {
        if (Status == KonsultationStatus.Afsluttet)
            throw new DomainException("En afsluttet konsultation kan ikke aflyses.");
        Status = KonsultationStatus.Aflyst;
    }

    public void Afslut(string notat)
```

```
    {
        if (Status != KonsultationStatus.Planlagt)
            throw new DomainException("Kun planlagte konsultationer kan afsluttes.");
        if (string.IsNullOrWhiteSpace(notat))
            throw new DomainException("Notat er påkrævet ved afslutning.");
        Status = KonsultationStatus.Afsluttet;
        Notat = notat;
    }
}
```

Bemærk fire ting:

1. **Alle properties har `private set`.** Ingen kan ændre felter direkte udefra.
2. **Konstruktøren er `private`.** Du kan ikke sige `new Konsultation(...)`. Du *skal* gå gennem `Opret()`-metoden, som vi vender tilbage til i kap. 7-8.
3. **Guard clauses kaster `DomainException`** — ikke `ArgumentException`. Det er en domæne-specifik undtagelse der signalerer at en *forretningsregel* er brudt. Vi udvider det i kap. 8.
4. **Status-overgange er ensrettede.** En afsluttet konsultation kan ikke aflyses. En aflyst kan ikke afsluttes. Reglen er kodet ind i metoderne — ikke i UI'en, ikke i Use Casen, ikke i databasen.

Klassen er **højt kohæsiv**: den handler udelukkende om hvad en konsultation er, og hvilke regler der gælder for den. Den **kender ingen andre klasser** udover sine egne value objects (som `Tidsinterval`) og sine egne enums (`KonsultationStatus`). Koblingen er minimal.

Det er det sted vi vil hen i alle de domæneklasser vi bygger fra her af.

1.5 Sammenhæng

I dette kapitel introducerede du fem fundamentale objektorienterede kvalitetsbegreber: indkapsling, kohæsion, kobling, interface og polymorfi. I de næste kapitler udvides og formaliseres de:

- **Kap. 2** giver dig fire af de fem SOLID-principper. To af dem — Open-Closed Principle og Liskov Substitution Principle — bygger direkte på interface-polymorfi (§1.3.9-10). De to andre — Single Responsibility og Interface Segregation — bygger på kohæsion (§1.3.6) og lav kobling (§1.3.7).
- **Kap. 3** dækker det femte SOLID-princip — Dependency Inversion Principle — som vender afhængigheden om mellem højniveau- og lavniveaukode. Forberedt af interface-introduktionen.
- **Kap. 4** udvider guard clauses til *pre- og post-conditions* — en mere systematisk måde at tale om kontraktbaseret programmering.
- **Kap. 7-8** bygger DDD ovenpå. `private set` og guard clauses bliver til *aggregate roots* og *invariants*. Det er den samme tanke, formaliseret.

Indkapsling, interface og polymorfi er ikke noget du læser om én gang. Det er mønstre du kommer til at bruge i hver eneste klasse fra og med nu.

1.6 Hands-on

> **Hands-on: anvend dette i et eget projekt** Som praktisk øvelse kan du anvende dette på SportZone. I uge 1 bygger I jeres første naive klasser for `Hold`, `Medlem` og `Instruktør`. Sørg for at *alle* properties har `private set`, og at konstruktøren validerer med guard clauses. I uge 2 vender vi tilbage og refaktorerer modellen efter SOLID — men kun fordi grundlaget er solidt.
>
> *Underviser-noter:* konkrete acceptkriterier for uge 1 findes i Bilag E.

1.7 Øvelser

Øvelse 1.1 *(Multistructural — describe, identify)*

Forklar med dine egne ord forskellen mellem:

1. `public int Health { get; set; }`
2. `public int Health { get; private set; }`
3. `public int Health { get; }`

Hvilken kan ændres af klassen selv? Hvilken kan ændres udefra? Hvilken kan slet ikke ændres efter konstruktion?

Øvelse 1.2 *(Relational — apply)*

Du får denne klasse:

```
public class Player
{
    public string Name { get; set; }
    public int Health { get; set; }
    public int MaxHealth { get; set; }
}
```

Refaktorer den så:

1. Alle properties har `private set`.
2. Konstruktøren modtager `name` og `startHealth` og validerer at `startHealth > 0`.
3. Klassen får metoderne `TagSkade(int skade)` og `Heal(int mængde)`.
4. `Health` må aldrig komme under 0 eller over `MaxHealth`.

Skriv også et lille `Main`-program der opretter en spiller, lader den tage 50 skade, healer 30, og udskriver tilstanden.

Øvelse 1.3 *(Relational — analyse)*

Læs denne klasse og vurder:

```
public class BookingHelper
{
    public Guid OpretBooking(DateTime tid, Guid patient) { /* ... */ }
    public void SendSmsBekræftelse(Guid bookingId) { /* ... */ }
    public List<Booking> HentBookingerForUge(int uge) { /* ... */ }
    public decimal BeregnPris(Guid behandlingstype) { /* ... */ }
    public string GenerérPdfKvittering(Guid bookingId) { /* ... */ }
    public bool ValidérCpr(string cpr) { /* ... */ }
}
```

1. Hvor mange forskellige ansvar har klassen?
2. Hvilke ord ville du bruge til at beskrive klassens kohæsion?
3. Foreslå en opdeling i 3-5 mindre klasser. Begrund din opdeling.

Øvelse 1.4 *(Extended Abstract — evaluate, justify)*

To kolleger har designet hver sin version af `BankAccount`:

Version A:

```
public class BankAccount
{
    public decimal Balance { get; set; }
    public string Owner { get; set; }
}
```

Version B:

```
public class BankAccount
{
    public decimal Balance { get; private set; }
    public string Owner { get; private set; }

    public BankAccount(string owner, decimal initialDeposit) { /* validering */ }
    public void Deposit(decimal amount) { /* validering + ændring */ }
    public void Withdraw(decimal amount) { /* validering + ændring */ }
}
```

1. Beskriv tre konkrete bugs der kan opstå i Version A som ikke kan opstå i Version B.
2. Argumentér for hvorfor Version B er sværere at misbruge.
3. Er der nogen situationer hvor Version A faktisk er det rigtige valg? Hvis ja, hvilke?
4. Hvilken version er nemmest at teste? Begrund.

   Facit findes i bilag B.1

1.8 Litteratur og videre læsning

Suppleterende noter

- *Begynderguide til Objektorienteret Programmering i C#* — opfriskning af OOP-grundprincipperne fra 1. semester.
- *It all begins with private set* — uddybende tekst om hvorfor `private set` er fundamentet for indkapsling.

Bøger

- Robert C. Martin, *Clean Code* (2008), kap. 6 (“Objects and Data Structures”) og kap. 10 (“Classes”).
- Steve Freeman & Nat Pryce, *Growing Object-Oriented Software, Guided by Tests* (2009), kap. 6 (“Object-Oriented Style”).

Web

- Microsoft Learn: *Properties (C# programming guide)*
- Steve Smith (Ardalis): *New is Glue* — vi vender tilbage til denne artikel i kap. 2.

Bilag D — Constructive Alignment og SOLO-taksonomien

> **Formål:** Dette bilag uddyber den pædagogiske ramme bogen er bygget på. Hvis du har bemærket markeringer som *(Multistructural — describe, identify)* ved kapitlernes ILOs og øvelser og spekuleret over hvad de betyder, finder du svaret her. Bilaget er primært for **undervisere og kursusudviklere** der vil forstå *hvorfor* bogen er struktureret som den er — men også relevant for **selvstuderende** der vil kalibrere deres egen progression mod et anerkendt læringsteoretisk rammeværk.
>
> **For den almene læser:** Du behøver ikke læse dette bilag for at få udbytte af bogen. SOLO-niveau-markeringer kan læses som en uformel sværhedsgrad-indikator — *Multistructural* er let, *Extended Abstract* er udfordrende. Hvis du vil dybere ned i hvorfor det betyder noget, så læs videre.

P.1 To begreber, to formål

To begreber bærer bogens design:

1. **Constructive Alignment** (Biggs & Tang) — princippet om at *læringsmål*, *læringsaktiviteter* og *bedømmelse* skal være indbyrdes afstemte.
2. **SOLO-taksonomien** (Biggs & Collis) — en model for *dybden* af forståelse, brugt til at formulere læringsmål med præcision.

Constructive Alignment fortæller dig **hvordan** du bygger et kursus så det hænger sammen. SOLO-taksonomien giver dig **sproget** til at specificere hvor dybt du vil have stoffet forstået.

De to bruges sammen: SOLO-niveau definerer hvad et læringsmål kræver; Constructive Alignment sikrer at aktiviteterne og bedømmelsen rammer det samme niveau.

P.2 Constructive Alignment

P.2.1 Det grundlæggende princip

John Biggs formulerede princippet i 1999 og har siden udfoldet det sammen med Catherine Tang. Den korte version:

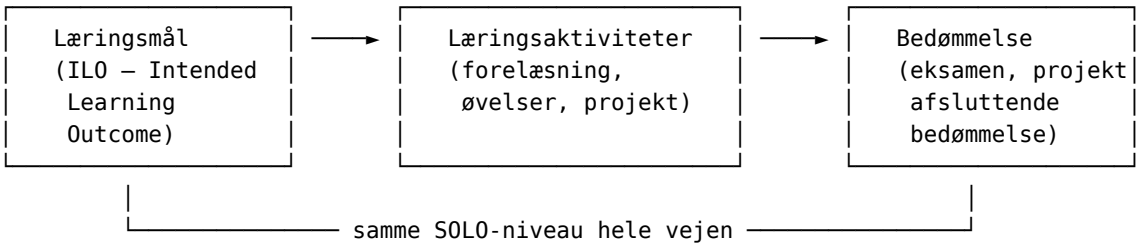
> *Hvis læringsmålet er "kan analysere et arkitektonisk valg ud fra trade-offs", så skal læringsaktiviteten lade studerende analysere arkitektoniske valg ud fra trade-offs, og bedømmelsen skal måle om de kan analysere arkitektoniske valg ud fra trade-offs.*

Det lyder banalt, men er det ikke. I praksis ser man ofte mismatch:

Mismatch-mønster	Eksempel
Mål er på højt SOLO-niveau, aktivitet er overfladisk	Mål: *evaluere*. Aktivitet: udfyldningsopgaver.
Aktivitet er kompleks, bedømmelse er simpel	Aktivitet: byg en hel applikation. Bedømmelse: multiple choice.
Bedømmelse er på højt niveau, ingen aktivitet forberedt det	Eksamen: *kritisér eget design*. Forløb: ingen øvelser i kritik.

Constructive Alignment kræver at *alle tre* er afstemt. Det betyder ofte at ambitiøse læringsmål kræver tilsvarende ambitiøse aktiviteter — og så skal eksamen også være ambitiøs.

P.2.2 De tre elementer



Læringsmål udtrykkes typisk som *Intended Learning Outcomes (ILO)* — sætninger på formen *"Efter kapitlet kan du …"*. Hvert verbum (*beskrive*, *anvende*, *analysere*, *evaluere*) bør være valgt bevidst — det er handlingsverbet der signalerer SOLO-niveau.

Læringsaktiviteter er det studerende *gør*. Læsning er én slags aktivitet, men ofte for passiv. Bogen kombinerer læsning med *Stop og tænk*-bokse, *forkert-først-så-rigtigt*-mønster, øvelser med stigende SOLO-niveau, og hands-on på et eget projekt.

Bedømmelse er hvordan vi måler om læringsmålet er nået. I et formelt forløb er det eksamen; i selvstudium er det øvelses-facit og evnen til at anvende stoffet på sit eget projekt.

P.2.3 Hvorfor det matter

Hvis de tre elementer ikke er afstemt, opstår to klassiske problemer:

Problem 1 — Studerende læser strategisk efter bedømmelsen. Hvis bedømmelsen er multiple choice, læser studerende for at huske facts — selv hvis læringsmålet siger *"kan analysere"*. Bedømmelsen overstyrer altid målet i praksis.

Problem 2 — Aktivitet uden formål. Hvis aktiviteterne ikke er designet til at understøtte et bestemt læringsmål, bliver de øvelser-for-øvelsernes-skyld. Studerende mærker det og kobler af.

Constructive Alignment er modgiften: når mål, aktivitet og bedømmelse peger på samme niveau, opstår en sammenhæng der guider både underviser og studerende.

P.2.4 Sådan er bogen alignet

Bogens design følger Constructive Alignment eksplicit:

Element	Bogens implementation
Læringsmål	Kapitel-ILOs øverst i hvert kapitel, formuleret med SOLO-verber.
Aktiviteter	Læsning + *Stop og tænk* + *forkert-først-så-rigtigt* + hands-on + øvelser. Hver øvelse markeret med sit SOLO-niveau.
Bedømmelse	Øvelses-facit (Bilag B) for selvkalibrering; for kursusforløb: eksamen designet til at teste transfer på et nyt domæne.

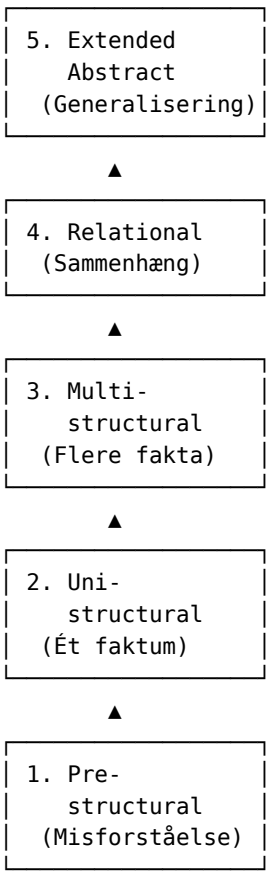
Hver kapitel-ILO hænger sammen med mindst én øvelse på samme SOLO-niveau, og facitten i Bilag B annoterer netop hvilket læringsmål øvelsen tester.

P.3 SOLO-taksonomien

P.3.1 Hvad er SOLO?

SOLO står for *Structure of the Observed Learning Outcome*. Den blev udviklet af John Biggs og Kevin Collis i 1982 som et alternativ til Blooms taksonomi. SOLO fokuserer på *hvor strukturelt sammenhængende* en studerendes svar er — ikke kun på hvilken kognitiv kategori (huske, forstå, anvende) operationen tilhører.

SOLO har fem niveauer, der bygger oven på hinanden:



Bogen bruger primært de tre øverste niveauer som mål: *Multistructural*, *Relational* og *Extended Abstract*. *Prestructural* og *Unistructural* er for lave for et 2. semester-kursus.

P.3.2 De fem niveauer

1. Prestructural *(forstår ikke)*

Studerende har ingen meningsfuld struktur. Svarene er forkerte, irrelevante eller tomme.

Eksempel — DDD: "DDD er et programmeringssprog." Misforståelse.

Dette niveau er ikke et læringsmål — det er udgangspunktet for læring.

2. Unistructural *(ét element)*

Studerende kan identificere ét enkelt aspekt og bruge det isoleret.

Eksempel — DDD: "DDD bruger aggregater." Korrekt, men isoleret.

Verber: *navngiv*, *identificér*, *husk*, *find*, *udfør én simpel handling*.

3. Multistructural *(flere elementer)*

Studerende kan identificere og opliste flere aspekter, men ser dem ikke i sammenhæng.

Eksempel — DDD: "DDD har Entities, Value Objects, Aggregate Roots, Bounded Contexts og Ubiquitous Language." Flere fakta korrekte; ingen sammenhæng.

Verber: *beskriv*, *opliste*, *kombinér*, *udfør flere handlinger*.

I bogen: kapitlets *Begrebsapparat*-tabel (§N.2) er typisk på multistructural niveau.

4. Relational *(sammenhæng)*

Studerende ser hvordan elementerne hænger sammen og kan anvende dem i kontekst.

Eksempel — DDD: "En Aggregate Root vogter over invariants, og den eksterne kode må kun røre den via metoder. Det er præcis indkapsling fra kap. 1, men anvendt på et helt aggregat — ikke kun en enkelt klasse."

Verber: *anvend*, *analysér*, *forklar*, *integrer*, *sammenlign*, *relatér*.

I bogen: kapitlets *Kerneindhold* (§N.3) og typisk de første øvelser er på relational niveau.

5. Extended Abstract *(generalisering)*

Studerende kan generalisere ud over det stof der er undervist — anvende på nye kontekster, evaluere alternativer, formulere nye principper.

Eksempel — DDD: "DDD's aggregate-grænser handler om transaktionsgrænser. Det samme princip findes i mikroservices, hvor service-grænser typisk er aggregat-grænser. I et hexagonal-arkitektur-projekt kan jeg overveje om mine ports bør spejle aggregaterne — eller om det skaber unaturlige grænser i adapter-laget."

Verber: *evaluér*, *teoretisér*, *generalisér*, *hypotetisér*, *kritisér*, *udvikle nye løsninger*.

I bogen: den sidste øvelse i hvert kapitel og hovedtemaerne (kap. 8 DDD-syntese, kap. 23 parallel Strategy, kap. 24-25 Big O og helhedsbillede) er på extended abstract niveau.

P.3.3 SOLO i bogens markeringer

Hver øvelse er mærket med sit SOLO-niveau plus de typiske handlingsverber:

```
### Øvelse 6.2 *(Relational — apply)*

Refaktorér til en rig model ...
```

Markeringen tjener tre formål:

1. **For dig som læser:** Du ved hvor udfordrende øvelsen er, og kan vælge at springe over indtil du har læst kapitlets stof grundigt.
2. **For underviseren:** Markeringen viser om der er progressiv stigning i øvelses-sværhedsgrad inden for kapitlet (typisk *Multistructural* → *Relational* → *Relational* → *Extended Abstract*).
3. **For bedømmer:** Markeringen kalibrerer hvad et "godt" svar ser ud som — et relational svar skal vise *sammenhæng*, ikke kun korrekt opremsning.

P.3.4 Verbum-katalog (kort)

SOLO-niveau	Typiske verber (engelsk / dansk)
Unistructural	*identify, name, find, list, do simple procedure / identificér, navngiv, find, opliste*
Multistructural	*describe, list, combine, do procedures / beskriv, opliste, kombinér, udfør*
Relational	*apply, analyse, explain causes, compare, contrast, integrate / anvend, analysér, forklar årsager, sammenlign, integrer*
Extended Abstract	*evaluate, theorise, generalise, hypothesise, reflect, create / evaluér, teoretisér, generalisér, reflektér, skab*

Vær forsigtig: nogle verber kan dække flere niveauer afhængigt af kontekst. *"Forklar"* kan være relational (forklar sammenhængen) eller multistructural (forklar definitionen). Konteksten i resten af opgavens formulering afgør niveauet.

P.4 Kalibrering: hvor sværhedsgrad og dybde mødes

Et almindelig misforståelse er at SOLO-niveau er det samme som *sværhedsgrad*. Det er det ikke.

- **Sværhedsgrad** handler om hvor meget *arbejde* opgaven kræver — kompleksitet, mængde, præcision.
- **SOLO-niveau** handler om hvilken *tankegang* opgaven kræver — fra opremsning til generalisering.

En unistructural opgave kan være meget svær (*"Find den specifikke kommando i Linux der …"*) og en extended abstract opgave kan være let formuleret (*"Diskutér i 5 linjer hvornår polymorfi betaler sig"*).

Bogen bestræber sig på at *både* kalibrere SOLO-niveau *og* holde sværhedsgraden inden for et 2. semester-niveau. Konkret betyder det:

- **Multistructural** øvelser er korte og kræver at du *kender* stoffet (læs kapitlet først).
- **Relational** øvelser kræver at du *anvender* stoffet på en konkret kode-situation. De er længere og kræver typisk at du skriver kode.
- **Extended Abstract** øvelser kræver at du *reflekterer* over trade-offs og giver en argumentation. De kan virke "kortere" fordi der ikke er meget kode, men kræver mest tid mentalt.

Når du som læser sidder fast i en *Extended Abstract*-øvelse, så lad være med at føle at du har misforstået noget. Du har ramt det niveau hvor stoffet skal *integreres* — og det er ofte kun muligt efter du har afsluttet flere kapitler.

P.5 Sådan bruger underviseren SOLO i praksis

For en underviser der vil designe sit eget forløb med bogen, er her en praktisk arbejdsgang:

Trin 1 — Formulér ILOs med SOLO-verber. Skriv hvad studerende skal kunne efter forløbet. Brug verber fra tabellen i §P.3.4. Vær eksplicit om SOLO-niveau pr. ILO.

Trin 2 — Tjek at aktiviteter understøtter ILOs. For hver ILO: hvilken læringsaktivitet leder til at studerende kan demonstrere det? Forelæsning alene er sjældent nok for relational eller højere — man skal *gøre* noget.

Trin 3 — Design bedømmelse på samme niveau. Hvis ILO er *evaluér*, så skal eksamen kræve evaluering — ikke opremsning. En multiple choice-eksamen tester kun multistructural; en mundtlig diskussion kan teste extended abstract.

Trin 4 — Annotér øvelser. Marker hver øvelse med sit niveau (som bogen gør). Det hjælper både dig og dine studerende med at se progressionen.

Trin 5 — Vurdér på SOLO-niveau, ikke kun på korrekthed. Et svar der opremser fem korrekte fakta er multistructural. Et svar der viser hvordan de hænger sammen er relational. Begge kan være "rigtige", men de viser forskellig dybde.

P.6 Sådan bruger den selvstuderende SOLO i praksis

For dig der læser bogen alene, er SOLO-markeringer en *selvkalibrering*:

- Hvis du let kan svare på *Multistructural*-øvelserne, har du læst kapitlet.
- Hvis du kan svare på *Relational*-øvelserne uden at slå op, har du *forstået* kapitlet.
- Hvis du kan svare på *Extended Abstract*-øvelserne med en argumentation der peger på flere kapitler samtidig, er du klar til næste kapitel.

Mange selvstuderende stopper for tidligt — de læser, nikker genkendende, og går videre. SOLO-stigningen i øvelserne er din afregningsmekanisme: hvis du ikke kan løse en *Relational*-øvelse, har du ikke forstået kapitlet, uanset hvor godt det føltes.

Brug øvelserne som det de er: *en tjekliste for forståelse*. Lav dem, tjek mod facit i Bilag B, og vend tilbage til kapitlet hvis dybden mangler.

P.7 Hvor passer det med bogens øvrige bilag?

Bogen har flere bilag der udfylder forskellige roller i Constructive Alignment-rammen:

Bilag	Rolle i Constructive Alignment
Bilag A — AI som læringsstøtte	*Aktivitet* — hvordan du bruger AI til at forstærke læringsaktiviteter uden at undergrave bedømmelsen.
Bilag B — Øvelses-facit	*Bedømmelse* — facitter giver selv-kalibrering mod ILOs.
Bilag C — Læringsgæld	*Aktivitet* — hvordan en gruppe sikrer at læring sker hos *alle* gruppemedlemmer, så bedømmelsen (individuel eksamen) faktisk afspejler kollektiv læring.
Bilag D *(dette bilag)* — Constructive Alignment og SOLO	*Mål* — den teoretiske ramme bag ILO-formuleringen.
Bilag E — Underviser-noter	*Aktivitet* — uge-baseret progression der orkestrerer hvordan ILOs nås.

De fem bilag arbejder sammen for at understøtte alle tre dele af Constructive Alignment-trekanten.

P.8 Litteratur og videre læsning

Hvis du vil dybere ind i den læringsteoretiske baggrund:

- **Biggs, J. & Tang, C. (2011).** *Teaching for Quality Learning at University* (4th ed.). McGraw-Hill / SRHE. — Standardværket om Constructive Alignment, skrevet for universitetsundervisere.
- **Biggs, J. & Collis, K. (1982).** *Evaluating the Quality of Learning: The SOLO Taxonomy.* Academic Press. — Den oprindelige SOLO-bog.
- **Biggs, J. (2003).** *Aligning Teaching for Constructing Learning.* Higher Education Academy. — Kort artikel der opsummerer princippet i 5 sider. God indgang.
- **Hattie, J. & Brown, G. T. L. (2004).** *Cognitive Processes in asTTle: The SOLO Taxonomy.* — Empirisk validering af SOLO som vurderings-rammeværk.

For praktiske, lærerorienterede formuleringer af SOLO-verber og rubric-eksempler:

- **Pam Hook's SOLO Taxonomy-website** (`pamhook.com`) — store mængder eksempler og skabeloner. Primært K-12-fokus, men principperne overføres.

  Pædagogisk afsluttende note: Constructive Alignment og SOLO er ikke trolddom. De er sproget der lader dig være *eksplicit* om hvad du vil have studerende til at kunne. Når du kan formulere det præcist, kan du designe forløb der faktisk leder dertil — i stedet for at håbe på at det sker.