

Refaktorering mod MVP og MVVM

Delphi Guide

Af Bent Normann Olsen

NORM4NN

©
Copyright
Bent Normann Olsen,
2023 Dianalund

Den danske udgave er gratis,
i modsætning til oversættelser
til andre sprog. Udgaven er stadig
beskyttet af lov om ophavsret.

Historiske og seneste versioner af denne håndbog kan findes på norm4nn.dk:
<https://www.norm4nn.dk>

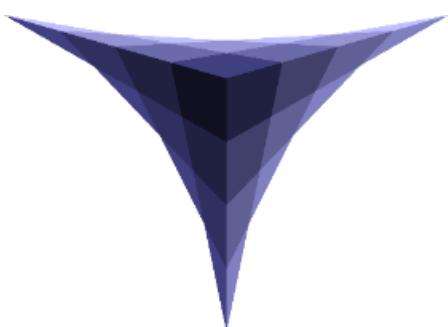
1. udgave, 1. udkast, skrevet med Community Edition af

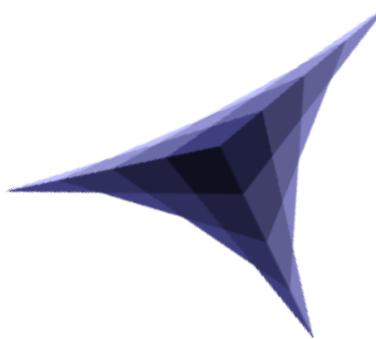
Delphi 10.4 – Sydney

Copyright © 2023, Bent Normann Olsen, Certified Delphi Developer og Component Writer. All rights reserved. Dokumentlayout og grafik af Bent Normann Olsen.

Der er ikke læst korrektur på dansk.

Varemærket "Delphi" tilhører Embarcadero Technologies, Inc.





Indhold

INDHOLD	4
FORORD	6
K A P I T E L 1 INTRODUKTION	7
ARCHITECTURAL PATTERNS.....	8
DEN OPRINDELIGE MVC	10
DEN MODERNE MVC	11
MODEL-VIEW-CONTROLLER.....	12
View.....	12
Controller	13
Model.....	14
BEHAVIORAL PATTERN	14
Entry Point.....	15
MODEL-VIEW-PRESENTER.....	16
View.....	16
Presenter	17
Model.....	17
MODEL-VIEW-VIEWMODEL	18
View.....	18
ViewModel	19
Model.....	19
HVORFOR MVP ELLER MVVM?	20
REFAKTORERING.....	21
TILNÆRMELSE TIL MVP OG MVVM	22
Anbefalinger før refaktorering.....	22
Fordele og ulemper ved patterns.....	23
K A P I T E L 2 SHOW, ACT, AND WORK..	25
Filer og Navne	28
Forskellige tilgange.....	30
Fuld kvalificerede navne.....	33
Adskillelse.....	35
K A P I T E L 3 REFAKTORERING	43
Definering af et interface.....	39
OPSUMMERING AF SAW, ACT, OG WORK	40
K A P I T E L 4 ACTIONS OG MULTIPLATFORM.....	64
INDLEDENDENDE SKRIDT	44
FØRSTE FASE	45
NÆSTE SKRIDT	45
ANDEN FASE.....	47
TREDJE FASE.....	54
FJERDE FASE.....	55
DET FÆRDIGE RESULTAT.....	59
MULTIPLATFORM.....	60
K A P I T E L 5 DATA-AWARE OG REFAKTORERING	75
DATA-BINDING MELLEM VIEW OG MODEL.....	76
ACTIONS OG LINKING.....	78
MODEL-VIEW-VIEWMODEL.....	82
K A P I T E L 6 OBSERVER PATTERN	84
TÆT KOBLING	87
NOTIFIKATIONER OG EVENTS	89
ÉN-TIL-MANGE OBSERVERE	90

KNAP SÅ TÆT KOBLING	90	INSTANTIERING AF VIEWS.....	110
LØS KOBLING	92	STRUKTUERING AF FILERNE.....	113
OBSERVER PATTERN OG ARCHITECTURAL PATTERNS	93	K A P I T E L 9 GENERISK OBSERVER PATTERN.....	116
K A P I T E L 7 REFAKTORERING MOD MVVM.....	94	OBSERVABLES.....	120
OBSERVER PATTERN.....	95	OBSERVERS	120
STOPWATCH MVVM	96	OBSERVER METODE	121
REFAKTORERING FRA MVP TIL MVVM	97	K A P I T E L 10 ARCHITECTURAL PATTERNS	123
VIEWMODEL	101	A P P E N D I K S A	124
SETTINGS - MVVM.....	102	NON-VISUAL KOMPONENTER	124
MANGE VIEWS	107		
K A P I T E L 8 STRUKTUREN AF ET MVVM- PROJEKT	108		

Forord

Der findes ikke ret mange bøger, og kun få blog-indlæg og webinarer, der handler om Delphi og MVC eller MVVM, og mindre om MVP. De kilder har imidlertid alle det til fælles, at det oftest handler om nye projekter i Delphi med et af disse architectural patterns, og ikke så meget om hvad vi kan gøre med bestående projekter, der er udviklet uden et bestemt pattern for øje og med en RAD-tankegang i bedste fald. Derudover kan vi finde bøger og andre kilder om refaktorering, ligesom Delphi har features, som understøtter refaktorering, omend vi ofte må acceptere, at bøgerne ikke specifikt er skrevet til Delphi, men som dog trods alt vil give os et indblik i hvad refaktorering er for en størrelse.

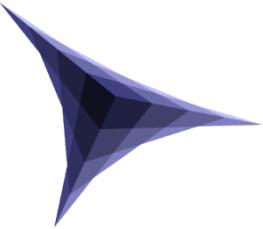
Det helt centrale i refaktorering handler om at foretage strukturelle ændringer af en kode uden at ændre på funktionaliteten i konstruktionen. Og grundlæggende kan dette udnyttes til at give bestående kode et tilsnit henimod et pattern. Det er selvfølgelig for at kunne drage fordele ved architectural patterns, som vi vil kunne finde i separering af view, eller form, fra en model, uddover andre fordele, der er tiltrækkende ved refaktorering.

Mange Delphi projekter lider af det simple forhold, at både kode og form er gemt i samme unit, eller fil, og det gør det umuligt at følge principper som *Separation of Concerns* og andre tilgange, som handler om at adskille det visuelle, som brugeren kan se, og den bagvedliggende kode, som oftest har fokus på forretningslogik og data. Vi har alle oplevet, hvor svært det er at ændre en form, som måske består af tusindvis af kodelinjer og masser af komponenter. De nævnte architectural patterns har alle det til formål at adskille View fra Model med et mellemliggende lag, og bare en simpel separering af form i en View-Model-arkitektur vil ikke alene give os nogle af fordelene ved SoC, men også et tankesæt, vi vil have brug for, for at kunne arbejde med MVP og MVVM.

Denne bog vil forsøge at give inspiration til en systematisk eller metodisk tilgang til refaktorering henimod et architectural pattern. En refaktorering kan i mange tilfælde give et resultat, der svarer til et projekt, der følger et pattern fra start, som for de fleste Delphi-projekter alligevel kun kan være et tilnærmet pattern. Vi vil under alle omstændigheder stadig kunne drage nytte af dette. Samtidig kan refaktorering og separering foretages i så små skridt, at det er muligt f.eks. at fastholde en *Continuous Integration*, ligesom testop-sætninger ikke nødvendigvis behøver at blive berørt, om end det måske ikke gælder for unit tests. Men sikkert er det, at bare en tilnærmelse til en arkitektur vil åbne op for nye muligheder for et system, som jo vil være hele formålet.

Bent Normann Olsen

Danmark,
April 2023.



1

Kapitel

Introduktion

Som redskab er Delphi ikke designet til at påskynde nogle af de populære patterns såsom Model-View-ViewModel (MVVM), Model-View-Controller (MVC) eller lignende, hvis vi lige ser bort fra Rapid Application Development (RAD). Delphi blev udviklet i kølvandet på oprøret mod bl.a. den klassiske vandfaldsmodel, der krævede at alt var projekteret. I 90'erne tog mange af os værktøjet til sig uden at tænke over ideerne bag RAD eller andre tilgange, og vi tvang måske ovenikøbet en vandfaldsmodel ned over projekterne, fordi det var vi uddannet i. Det resulterede ofte i nogle af de ulemper, der er associeret med RAD i dag. Vi behøver ikke at tage de ofte dårlige design som eksempel, men også at RAD er så fleksibelt, at det kan blive på bekostning af kontrol, som måske vil være knap så hensigtsmæssigt, hvis vi f.eks. arbejder med kritiske systemer.

Uanset fordele og ulemper ved RAD, så skal vi først og fremmest hæfte os ved, at det anses som dårlig praksis, når både kode og den grafiske brugergrænseflade (forms) er blandt sammen i én stor fil (unit), der i mange tilfælde kan spænde over flere tusinde kodelinjer. Sædvanligvis har det ikke været et bevidst valg, men det er den tilgang, som Delphi især er designet til at understøtte, og det har været den mest oplagte at følge og lige til at gå til. Idéen med Rapid Application Development markedsføres også gennem udviklingsværktøjer fra Embarcadero, som f.eks. i Delphi RAD Studio, uden at de går i dybden med RAD som udviklingsmodel. Der er nogle fordele ved RAD, men også ulemper udover det fornævnte, f.eks. at det ikke virker så godt i store teams, at de ofte løber ind i problemer i større projekter, ligesom det er nødvendigt at udviklere har stor erfaring med RAD.

RAD-modellen kræver en iterativ udvikling af prototyper gennem feedback fra brugere, men det er som regel heller ikke noget ejerne af projektet har særligt stort fokus på – det forventes som sædvanligt, at den første ”prototype” også er det endelige design, og som tit ender med at være temmelig rigid. Og så er der udviklere, der forventer at se et design før udvikling af software, så RAD-tankegangen hos disse tilfælde slet ikke er til stede i et Delphi-projekt. Det kræver erfaring at konstruere kode, der er nem at transformere gennem en konstant *build-demonstrate-refine*, indtil interesserne er tilfredse.

Selvom Delphi ikke er målrettet nogle af de populære patterns, så er det ikke umuligt at følge disse eller andre tilgange, modeller, metoder, patterns m.m. Det kræver måske bare

en tilnærmet tilgang fremfor en ren tilgang. Det skal også fremhæves, som med mange patterns, at der vil være meget store variationer i en praktisk implementering af et pattern og dets anvendelse, der afhænger af programmeringssprog og frameworks, samt de værktøjer, der bliver brugt, og ikke mindst typen af forretning og opgaver, hvori de skal indgå. Derfor vil vi også finde variationer af de enkelte patterns, der er afstemt efter forskellige miljøer, som nødvendigvis ikke er tilpasset, fordi nogle synes ”det skal se sådan ud”, men fordi teknologien eller andre aspekter måske sætter rammerne over de oprindelige ideer bag et pattern. Vi vil f.eks. finde tekniske forskelle mellem Delphi MVC Framework og ASP.NET Core MVC, eller praktiske forskelle i implementeringen af et pattern for et *embedded* system i forhold til en webapplikation, men ideerne vil være de samme.

Architectural Patterns

Når vi taler om MVC, MVVM, m.fl., så hører de til den gren af patterns, der specifikt er rettet mod arkitekturen af en applikation. Ofte skal et architectural pattern ses i sammenhæng med en problemstilling i softwareudvikling, fordi de helt sikkert er skabt med et formål for øje. Når vi snakker om arkitekturer, så skal vi tænke dem i de sædvanlige problemstillinger, når vi udvikler webapplikationer, men i helt andre velkendte problemer, når vi udvikler desktop-applikationer. De enkelte patterns forsøger ofte at løse et bestemt problemfelt i softwareudvikling, ligesom et bestemt pattern kan være indarbejdet i et framework, som benyttes i samme felt. Vi kan finde kilder, der demonstrerer nogle af disse patterns, og hvordan vi laver vores eget framework med et af disse patterns.

Indenfor architectural patterns taler vi også om *architectural style* (arkitekturstil), som egentlig dækker over en gruppe af patterns, der minder om hinanden og f.eks. deler de samme begreber, ligesom de kan have samme måde at illustrere deres arkitektur på. Og det er i den kategori, vi kan finde MVC, MVP, MVVM, men også MVA (Model-View-Adapter), PAC (Presentation-Abstraction-Control) og mange flere. Architectural style kan også dække over en gruppe af patterns, som de specifikt er udviklet til, f.eks. webapplikationer, desktop-applikationer, server-applikationer osv. Vigtigst er det, at stilart indenfor software-arkitektur handler om patterns, som har noget tilfælles, på samme måde med de fælles begreber vi vil se med MVC og alle varianter af samme.

Indenfor design patterns vil vi også kunne finde hele grupper af anbefalede architectural patterns, software design patterns, solution patterns, og øvrige patterns, som er målrettet helt særlige forretningsspecifikke felter, f.eks. *Analytics and Business Intelligence*. De specifikke anbefalinger kan dække over flere architectural patterns, ligesom de kan indeholde anbefalinger til software design patterns og solution patterns. Hovedpointen er, at et pattern skal ses som en løsning på et problem, mens en stil dækker over noget mere generelt og ikke et specifikt problemstilling.

Udover Rapid Application Development (RAD) for Delphi, støder vi tit på udtrykket ”client-server applikationer”, især i de tidligste udgaver, som bl.a. blev markedsført som Delphi Client/Server Suite. Client-server omtales ofte som ”model”, men har dog en lang forhistorie, og hører egentlig til blandt *Multitier Architecture*. Derfor er det heller ikke

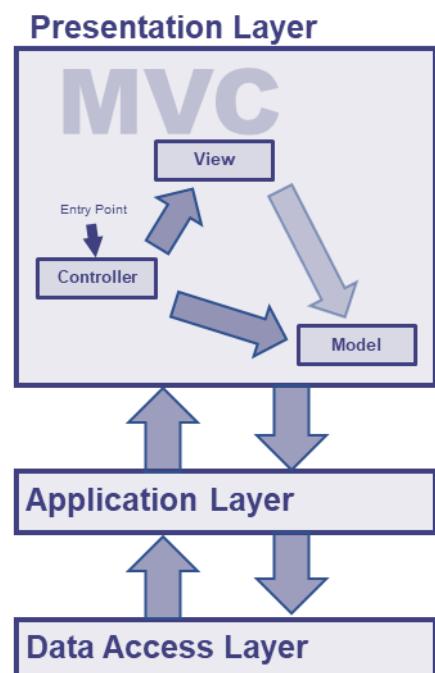
usædvanligt at se Delphi-projekter, hvor man rent faktisk har gjort et forsøg på at adskille de typiske præsentationslag fra applikationslaget og forretningslaget fra datalaget. Multitier architecture, herunder Client/Server, handler om at adskille de enkelte lag fra hinanden rent fysisk, men almindeligvis ser vi blot en server-applikation, der kommunikerer med en række client-applikationer.

I Multitier Architecture kender vi også problemet med, at nogle lag ikke er adskilte, f.eks. hvis forretningslaget ikke fysisk er adskilt fra præsentationslaget, og så har vi faktisk implementeret den traditionelle opfattelse af en Client/Server model – eller et såkaldt two-tier i en Multitier Architecture. Men i modsætning til gængse architectural patterns, så dækker Multitier Architecture over en række software, der er adskilt fra hinanden. Dvs. at two-tier sædvanligvis vil dække over én client-applikation og én server-applikation, som måske bare er en database-server. Data-tier er nemlig det lag, hvor data hentes og gemmes, og det vil et database-system på en server sagtens gå ud for at være.

Three-tier fra multitier architecture er i de fleste tilfælde den implementering, vi sigter efter, hvor vi har en service-applikation kørende på en server, der tager imod forespørgsler fra klienter og sender responser tilbage. Og som en afstikker, så minder two-tier lidt om situationen med Delphi forms og al kode i samme unit, og som på længere sigt godt kan skabe lignende problemer i forhold til en three-tier-løsning eller f.eks. MVC.

En gennemgang af den klassiske client-server-model er ikke helt uden grund, fordi Multitier Architecture dækker over en række uafhængige applikationer, der udfører hver deres funktion. Hver applikation kan vi udvikle efter et architectural pattern, vi finder mest passende til lige netop den situation og forretning, som hver applikation nødvendigvis må dække over. Det vil sige, at vores client-server-løsning ikke behøver at blive afmonteret og heller ikke står i vejen for, at vi tager et architectural pattern til os. Vi står måske med en two-tier løsning og en client-applikation, der trænger til et løft forud for en opgradering, som så vil det være den del, vi vil refaktorere, frem mod et pattern. I så fald vil det være Model, der kommunikerer med Data Access Layer i en overordnet two-tier Client/Server-model, eller med Application Layer i en three-tier model.

Kort sagt, så dækker en Multitier Architecture, eller en client-server-model, ikke over de patterns, der evt. vil have været brugt i udviklingen af de enkelte applikationer i en client-server-løsning. Og ikke desto mindre, så kan de enkelte applikationer også følge en Multitier Architecture, hvorfor vi til gengæld vil have en overordnet såkaldt N-tier Architecture. Vi kan formentlig finde endnu flere variationer af Multitier Architecture, end vi kan finde variationer af Model-View-Controller pattern, ligesom de mange forskellige lag eller *tiers* kan hedde



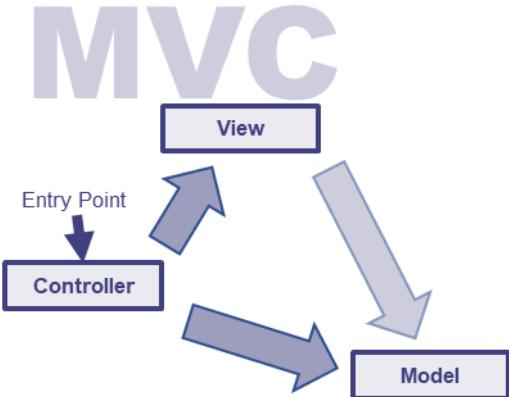
andre navne, f.eks. Top Tier og Bottom Tier, Service Layer og Business Layer, osv. Og for nogle værktøjer og frameworks vil det helt klart også handle om at beskrive en bestemt variation, som de lige præcis understøtter, ligesom de også vil være særligt rettet mod f.eks. webapplikationer, server- eller desktop-applikationer, embedded software m.fl., som vil kræve forskellige tilgange.

Den oprindelige MVC

Det kan kun blive en god idé at omtale de spæde tanker bag MVC, når det pattern nu er stammoder til de øvrige varianter, der alle kan sammenskrives til én arkitekturstil. En kort gennemgang vil ikke bare vise forskellen mellem MVC dengang og de moderne fortolkninger af MVC i dag, den vil også fremhæve de små variationer i de øvrige patterns, og på den måde give en idé om hvilke patterns, der vil være mest fordelagtig at følge i et givent projekt. Denne gennemgang vil også påpege de skiftende og til tider modstridende afbildninger af MVC, som vi ser i dag, og som helt klart skal ses i lyset af at MVC blev til i slutningen af 70'erne og også blev brugt i de indledende udviklinger af grafiske brugergrænseflader i 80'erne. Billedet herover viser de enkelte deles referencer til de øvrige dele.

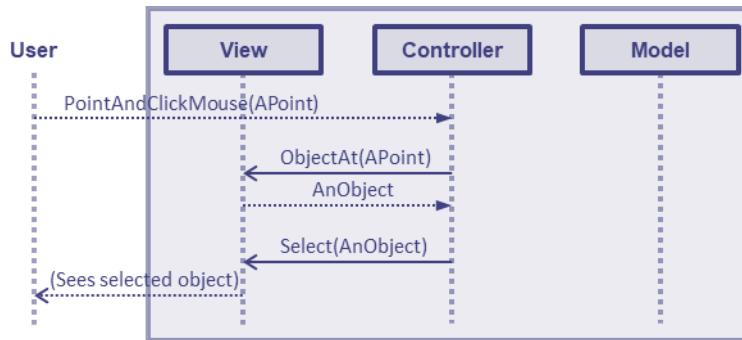
Mest iøjnefaldende er det, og i modsætning til nyere varianter, at det er Controller i MVC pattern, der tager imod og håndterer inputs fra brugeren, og det strider jo imod al logik i dag, fordi vi ved, at input rent teknisk sker i View. Men den er god nok – og der er en simpel forklaring på denne "uoverensstemmelse". MVC kom til verden i en tid, hvor LCD-displays blev en mere og mere integreret del af en brugergrænseflade i rigtig mange hardwareprodukter, f.eks. kopimaskiner. De var hver især koblet til en mikrocontroller, ligesom tilhørende input-enheder, mest knapper, og måske relæer til at aktivere et eller andet osv. View var således noget kode, der skulle håndtere en perifer enhed og primært havde til opgave at forme et billede for et sådant display – der var ikke noget input. I dag kan det bedst sammenlignes med de udfordringer, vi vil få med, hvis vi havde et View til en desktop-applikation, hvor vi bare havde en Canvas, vi kunne tegne på, og at vi derudover skulle sætte en *callback* op i Controller til at lytte på mouse-events. I gamle dage ville vi have tal om *handlers* til *hardwareinterrupts*.

De tilhørende input-enheder var sædvanligvis nogle navigationsknapper, som var tilsluttet en mikrocontroller, og så blev det en selvfølge at Controller fik ansvaret for at fortolke disse inputs og fortælle Model og View, hvad der skulle bearbejdes og præsenteres. Banalt set ville en bruger trykke på en fysisk start-knap, som Controller vil registrere, hvorfor den vil sende en instruks til Model via en start-metode, som så vil sende en notifikation til View om, at noget var startet, der så opdatere et skærbillede og på den måde bekræfte



brugeren i sit valg – hvis altså View er sat op til at lytte til Model på netop denne start/stop-mekanisme, for ellers er det Controller, der også må håndtere den del. Den sidste detalje er vigtig, fordi Model ikke kender noget til View, men View kender Model.

Controller var på den måde en central del af arkitekturen, og denne tankegang fulgte med i de allertidligste udviklinger af grafiske brugergrænseflader i 80’erne – altså også med de tidligste computere med tastatur og mus. Herunder kan vi se en ganske almindelig situation i 80’erne, hvor det var helt normalt at bede om det objekt på et punkt på skærmen, hvor brugeren har klikket med musen.



Det var ikke usædvanligt at se ovenstående eller lignende sekvensdiagrammer i 80’erne, som i eksemplet bare har til opgave at fremhæve det objekt, som brugeren har klikket på. Brugerens klik var altså noget en Controller lyttede på, og ikke en egenskab som de enkelte objekter var udviklet til at håndtere selv. Det er naturligvis den måde de tidligste operativsystemer var indrettet på, og den enkelte Controller måtte derfor selv kalde f.eks. ovenstående funktion `ObjectAt()`, en art API-kald til operativsystemet, som de enkelte dele kunne trække på.

I dag håndteres den del af operativsystemet og af de klasser, som et udviklingsværktøj tilbyder. Operativsystemet kender de enkelte objekter på en form, eller i et View, og mange objekter er konstrueret til at kunne registrere og håndtere forskellige inputs fra brugeren – efter udviklerens anvisninger. Fordi denne opgave nu formidles direkte fra applikationen til de enkelte objekter, så er det nu definitionen af klasser, der bestemmer om et objekt får et klik-event eller ej, men de klasser vil stadig have brug for en funktion, der minder om `ObjectAt()`. I Delphi har vi en klasse, der hedder `TWinControl`, der bl.a. varetager denne type af opgaver, og den har da også en funktion, der hedder `ControlAtPos()`. Det er altså en funktion, der har overlevet siden de tidligste grafiske brugergrænseflader.

Den Moderne MVC

I dag er der også mange, der helt naturligt antager, at håndtering af inputs fra brugeren sker i View i MVC, selvom det ikke var den oprindelige tanke. Det er ekstremt svært at komme udenom, at input fra brugeren i dag rent teknisk går igennem et View, eller en Delphi-form gennem f.eks. et `TWinControl`. Selv hvis vi kiggede ned i et ASP.NET Core MVC-projekt, så må vi finde os i, at input fra brugeren registreres gennem en reference i

et View, men at selv det framework i øvrigt forsøger at dirigere handlingen direkte videre til en Controller. På det område har ansvaret hos en Controller flyttet sig en lille smule, men det er ikke altid at baggrunden er lige tydeligt for alle implementeringer af MVC i de forskellige frameworks og platforme, der tilbydes.

Den største forskel er nok, at vi, i stedet for at modtage et klik-event fra brugeren i form af koordinater til en position og selv må finde et objekt, nu modtager et klik-event fra objektet med en reference til objektet selv. Og en anden udvikling er, at MVC er gået hen og blevet et populært pattern indenfor udvikling af webapplikationer, som virker mere indlysende at håndteringen sker i Controller, når View repræsenterer noget, der sker på en browser hos en bruger. Og dette er sket på trods af, at man tidligt forsøgte at tilskynde til MVC i udvikling af desktop-applikationer, så har dette i stedet udviklet MVC henimod nye fortolkninger eller varianter – f.eks. ved at der kan være nogle tekniske forskelle i de enkelte frameworks med MVC, eller at der simpelthen er kommet nye patterns til.

Udover MVC og mere moderne implementeringer i nyere frameworks, så findes der nu andre lige så populære patterns, og vi tænker især på MVP og MVVM. De er som sagt varianter af MVC, og vi vil gennemgå alle tre varianter for at få et overblik over deres forskelligheder. Der findes som bekendt flere varianter af disse patterns og endnu flere fortolkninger af de samme patterns, men vi vil holde os til de tre mest populære og så vidt muligt i en mere moderne sammenhæng.

Model-View-Controller

På nær nogle få tekniske detaljer, så består Model-View-Controller som pattern grundlæggende som oprindeligt, og som helt naturlig hører til gruppen af architectural patterns. Den benævnes også bare som en design pattern, men forskellen er, at en architectural pattern har et bredere omfang, dvs. på alle områder af en applikation, i modsætning til en design pattern, der virker på et mere afgrænset område, f.eks. på klasser.

MVC opdeler kode i tre sammenhængende dele, eller lag, hvor hver del har helt specifikke ansvar. Formålet er, ligesom de andre varianter, netop Separation of Concerns (SoC), som er et design princip i programmering, der skal adskille kode i deres særegne dele. Det handler om en tydeligere deling af de objekter, der kan gøre det ud for vores verden, og de objekter, der skal gøre det ud for en grafisk repræsentation af samme. Det handler i høj grad også om at fjerne begrænsninger fra et projekt og i stedet skabe de bedste muligheder for at videreudvikle samme projekt.

View

Til en given Model er der knyttet et eller flere Views, hvor et View i normale situationer vil fremhæve visse egenskaber ved en Model og undertrykke andre. View er således en visuel repræsentation af en Model, eller dele af den, og kan læse de nødvendige data fra Model, ligesom View sagtens kan modtage notifikationer fra Model. Et View er også i stand til at udføre operationer på en Model i en udstrækning som må være forventeligt af en givent

View, der er knyttet til præcis den Model. Med andre ord, så kender View til en Model, og har til opgave at præsentere data fra Model til brugeren.

I dag er fortolkningen stadig, at en View skal være så tynd som muligt, og hvis der er noget kode i View, så er det udelukkende for at præsentere data. Det kan måske debatteres i hvor høj en grad View har bearbejdet Model gennem tiderne, men platformene har udvikles sig henimod mere komplette og selvstændige objekter, og med data-binding spiller View i dag en større rolle end før. Og fordi flere elementer selv kan håndtere meget mere i dag udenom Controller, så kan vi oftest finde et View med meget lidt eller slet ikke noget kode i. Men det er ikke View, der opretter en Model, idet den blot modtager en reference til en Model fra Controller, når View bliver oprettet. Af samme grund kan der sagtens være flere Views til samme Controller og Model.

Controller

Eftersom Controller fungerer som en såkaldt *entry point*, se senere afsnit, så bestemmer den også hvilken View og Model, der skal arbejdes med, ligesom det er Controller, der har referencer til både View og Model. En Controller er bidelede mellem en bruger og applikationen, og giver brugeren information ved at sørge for, at de rigtige Views præsenteres på passende steder på skærmen. Den oprindelige MVC lagde op til, at et skærmbillede bestod af mange mindre Views, hvor en Controller gav brugeren mulighed for at afgive kommandoer og data på, hvorefter Controller registrerede input fra brugeren, oversatte dem til relevante beskeder, gav View besked eller evt. sendte dem videre til Model, som så kunne notificere View om evt. ændringer.

En Controller skulle ikke være et supplement til Views, ved at Controller f.eks. foretager præsentation af data. Omvendt skulle en View ikke vide noget om inputs fra brugeren, såsom klik på musen og tastetryk. Den del har vi nok affundet os med, at View opfanger og bringer videre enten til Controller eller direkte til behandling af data i Model.

I principippet består den oprindelige beskrivelse af Controller, men i den nyere fortolkning er View et helt skærmbillede, hvilket bl.a. afspejles af at vi opretter et enkelt View-fil for hvert skærmbillede, men at Controller og Model sagtens kan dække over flere Views – særligt i webapplikationer. Den oprindelige tanke var, at et View kunne være et hvilket som helst element på skærmen, ganske enkelt fordi et simpelt input-felt kunne afstedkomme så store mængder af kode, at det kunne struktureres særskilt i sin egen View-fil. Det er heller ikke en useriøs tanke, fordi hver komponent i Delphi, der er nedarvet fra en TWinControl, rent faktisk også er et View i sig selv, med håndtering af events i alle afskygninger, udover optegning af den visuelle del. De elementer kunne sagtens anses som særskilte Views i gamle dage, fordi man bare ikke havde adgang til avancerede komponenter, som vi har i dag. Tænk blot på den udfordring, som vores kode vil ende med at blive, hvis vi selv skulle implementere vores egne TEdit-komponenter i applikationen.

I udvikling af webapplikationer er det også mere naturligt at honorere Controllerens ansvar med at håndtere input fra brugeren, fordi mange elementer i desktop-applikationer kan være direkte linket til en Model gennem data-binding.

Model

En Model skal repræsentere viden, og en Model kan være et enkelt objekt, eller den kan være en struktur af objekter. Model bør udelukkende være en-til-en repræsentation af den virkelige verden. Det var den oprindelige beskrivelse, og i dag betragtes Model som den centrale del i MVC pattern, der er uafhængig af både Controller og View, og repræsenterer status på applikationen, samt består både af forretningslogik og de data, der kan behandles af den, og ikke mindst de regler som en applikation måtte være omfattet af i den virkelige verden. Model får brugerens input fra Controller, når denne har bestemt, hvad der skal ske, og ofte ser vi denne input blot overført direkte til Model.

I dag ser vi også en direkte kommunikation mellem View og Model, f.eks. gennem data-binding, og det er ikke unaturligt at se bindinger, der går begge veje, dvs. at View opdaterer Model gennem en sådan data-binding, og Model notificerer View om ændringer.

De tre dele af MVC-pattern går igen i de andre varianter, og selvom det ser ud som om, at det bare er navnet på Controller, der skifter navn, så ændres ansvaret sig for de enkelte dele, som det også indirekte afspejles af referencerne imellem delene. MVC havde oprindeligt det formål at opdele en brugerorienteret applikation i mindre håndterbare dele, og op igennem 80'erne og 90'erne blev den også brugt til desktop-applikationer. Med Internettets fremmarch blev MVC det populær valg, og det afspejles i dag af rigtig mange frameworks har implementeret MVC, herunder men ikke begrænset til Ruby on Rails, ASP.NET Core MVC, Angular blandt mange andre.

Behavioral Pattern

Imellem alle dele af et pattern, så er vi også nødt til at tage stilling til *behavioral patterns*, fordi de enkelte dele nødvendigvis må kommunikere med de øvrige dele, som et architectural pattern består af. Der er ikke noget i MVC pattern, der bestemmer hvilke behavioral patterns, der skal anvendes, men meget tidligt blev MVC pattern krediteret for at bane vejen for Observer Pattern, som gennem årene virkelig har gjort frameworks med MVC ret potente, så især Views og Controller kunne lytte til Model. De forskellige MVC-frameworks, som udviklere ofte får stillet til rådighed igennem deres værktøjer, kan have implementeret et eller flere patterns eller paradigmer, som vil være det helt naturlige at arbejde videre med, ligesom disse implementeringer også kan have haft indflydelse på vores opfattelse af et pattern og hvad vi forstår ved et bestemt pattern.

Event-drevet programmering kan f.eks. være oplagt, når det er den mest dominerende paradigme i desktop-applikationer. Har vi derimod ikke noget MVC-framework til rådighed, så er der ikke noget, der forhindrer os i, at vi anvender et helt andet behavioral pattern. Vi ser f.eks. Observer Pattern i Delphis *data aware*-komponenter, men vi kan lige så godt bestemme os for en Mediator Pattern, f.eks. med definering af abstrakte klasser, dog især i form af en række interfaces, som den bedst mulige tilgang til refaktorering, fordi vi med interfaces netop kan slippe for at ændre på eksisterende kode – når vi ser bort fra selve definering af et eller flere interfaces.

Hvis vi finder en beskrivelse af et bestemt architectural pattern, der bliver kædet sammen med et bestemt behavioral pattern, så er det helt sikkert ikke fordi det ene kræver det andet, men mest sandsynligt fordi det er et pattern, der er blevet implementeret i et framework med MVC. Eftersom vi refaktorerer eksisterende projekter, og vi ikke har et framework til rådighed, så giver denne situation os en klar udfordring med at vælge det eller de patterns, der skal sikre at de enkelte dele af et architectural pattern har mulighed for at kommunikere med hinanden. Til nogle af de øvrige varianter af dette pattern er der dog tradition for eller tænkt på et bestemt pattern, et paradigme, eller en model, hvor det giver god mening, ligesom disse vil være umulige at implementere i andre architectural patterns. Og i nogle få varianter er et behavioral pattern allerede bestemt.

Entry Point

Selvom MVC er stammoder til de øvrige varianter, så er der et bestemt egenskab ved MVC, der kan gøre det svært at følge dette pattern helt til dørs, fordi det netop vil kræve langt mere end blot refaktorering. Det vil kræve så fundamentale ændringer, at vi ikke længere vil kunne honorere tanken om kun at lave strukturelle ændringer ved en refaktorering. Det såkaldte entry point har altid været at finde i en Delphi-form, som jo vil gå ud for at være View i en MVC-pattern, men Controller er entry point i denne pattern, af årsager som vi tidligere har været inde på.

Begrebet entry point er normalt det sted i programmet, hvor eksekveringen begynder, men for varianter af architectural patterns har man brugt udtrykket for at markere det sted, hvor eksekvering af en implementering skal starte. Dvs. at eksekvering af en MVC-pattern starter i Controller, hvormod det vil starte i View for MVP- og MVVM-patterns. Forms i Delphi har traditionelt været "entry points" for eksekvering i denne sammenhæng, og det vil kræve en større indsats at ændre dette forhold, i og med at det er en principiel teknisk indretning, der er svært at ændre på.

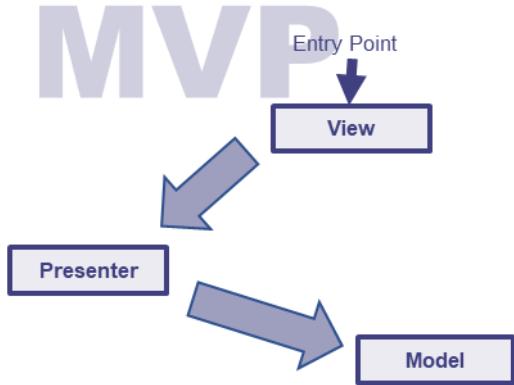
Entry points er vigtige, fordi det, uddover at definere de naturlige referencer til de øvrige dele af et pattern, også dikterer den rækkefølge, som dele af et pattern nødvendigvis må oprettes efter. I MVC er det f.eks. Controller, der bestemmer hvilke Views og Modeller, der skal arbejdes med, og så må entry point ligge i den, ligesom View skal have en reference til Model, hvilken igen betyder at Model må oprettes før View, som så oprettes med en reference til Model. I MVP og MVVM er det View, der defineret som entry point, og så må det blive View, der bestemmer hvilken Presenter eller ViewModel, der skal arbejdes med. Men View i MVP og MVVM har ikke nogen idé om eller reference til Model, som heller ikke har nogen idé om View, og så er det Presenter eller ViewModel, der har den viden om, hvilken Model der skal arbejdes med. Presenter og ViewModel må således have en effektiv definition på, hvordan de så kan kommunikere frem og tilbage med View om status og data fra Model.

Entry point definerer så at sige en patterns mulighed for at referere til de øvrige dele, ligesom de referencer i høj grad definerer de forskellige muligheder, vi har, for at implementere en kommunikation, et behavioral pattern, en model eller et paradigme, mellem de

forskellige dele af en architectural pattern, der i høj grad afhænger af, om delene kender hinanden, eller om kun den ene kender den anden osv.

Model-View-Presenter

I relation til refaktorering kan MVP gå hen og blive et meget spændende valg af pattern til refaktorering af desktop-applikationer udviklet i Delphi. Det er måske et pattern, som vi ellers ikke ville have skelet til i første omgang, fordi der for tiden er megen omtale af MVC og MVVM. Som skrevet før, så er MVP en variant af MVC, der blev til i begyndelsen af 90'erne, dvs. efter en årrække med stor fokus omkring udvikling af desktop-applikationer og på de tidlige generationer af grafiske operativsystemer. Der findes mange fortolkninger af MVP, ligesom der selvfølgelig kan være forskellige implementeringer i frameworks, men de mest interessante udgaver består enten af en Presenter som Supervising Controller eller i form af et Passive View. Desuden er MVP et pattern, som designet til at implementere ét View til ét Presenter, i modsætning til MVC og MVVM, som er designet til at implementere flere Views.



View

Et View kan indtage en meget passiv rolle i MVP, og vi vil f.eks. opleve et øget fokus på framework- og platform-specifikke kode i et View. Entry point findes i View, og på den måde har den en indflydelse på, hvilken Presenter, der får lov til at arbejde med præcis det View. Det betyder, at View har en reference til Presenter, ligesom referencen har en indflydelse på, at Presenter ikke kender View direkte. Det betyder også, at kommunikationen må ske indirekte, f.eks. gennem et interface, som Presenter har defineret og som View skal have implementeret. Det er ikke MVP-pattern, der definerer en bestemt kommunikation, der skal implementeres mellem de to dele – det afgøres først og fremmest af de patterns et framework har implementeret, eller de behavioral pattern, der vil være bedst egnete i en given situation. Interfaces er bare blevet en naturlig tilgang til sådanne situationer, men det kan lige så godt være andre tilgange, f.eks. med abstrakte klasser.

I forhold til MVC, så kender View i MVP ikke noget til Model, fordi det hele må ske igennem Presenter. Dvs. at Presenter kan vælge kun at eksponere dele af en Model overfor View, men den kan jo også vælge at offentliggøre hele data-komponenter i Model, som de er, men det skal ske uden at View har en reference til Model.

Når et View kan indtage en passiv rolle, så er det fordi, der er lagt op til, at vi selv kan tage stilling til, om hvorvidt og hvor meget der skal være af kode i View. Ofte vil et framework have taget denne beslutning for os, og i mange tilfælde kan et View bare være et grafisk design, hvor Presenter så at sige bare skubber data ind i nogle felter. Det er ikke MVP, der

definerer i hvilken grad disse roller er fordelt mellem View og Presenter, og uden et framework vil vi selv skulle tage stilling til dette forhold.

I relation til refaktorering og klassiske Delphi-projekter, så vil den praktiske opgave gå ud på at fordele elementer fra en form, som bliver til et View, ud mellem Presenter og Model. Dette kan glædeligvis foretages i små skridt, også uden at vi behøver at definere et interface på forhånd. Fundamentet til dette interface vil blive synliggjort under den afsluttende fase af refaktoreringen. Dette emne vil blive beskrevet i detaljer og med eksempler senere.

Presenter

En Presenter vil fungere som et bindeled mellem View og Model. Med et passivt View, så er det Presenter, der opdaterer View, når der sker ændringer i Model. Det er Presenter, der henter eller definerer adgangen til data fra Model og foretager de nødvendige formateringer af data for View. View kender ikke Model, og derfor er det også Presenter, der opretter Model. Når View opretter en Presenter, så får Presenter en reference til View, ikke som et konkret objekt, men med en implementation, som Presenter har defineret et View bør have, og som gør det muligt for Presenter at kommunikere med View. Denne definition kan også indeholde properties, som præsenterer data fra Model, i en form som Presenter har defineret.

Vi kan vælge at definere en Presenter, der giver en direkte adgang til Model, så View i stedet kan udnytte et design pattern med f.eks. data-binding direkte, og i de situationer vil en Presenter fungere som en Supervising Controller. Udover at begge tilgange har indflydelse på fordeling af kode mellem View og Presenter, så har det også indflydelse på hvor unit test skal målrettes. Passiv View giver et større testflade, fordi meget af opdateringen af View varetages af Presenter, ligesom Supervising Controller vil kræve mindre, fordi den slipper for simple opdateringer, som View selv kan klare. Det er også muligt at åbne helt op for en reference fra View til Model i en Supervising Controller tilgang, og det vil i nogle tilfælde i refaktoreringen være nødvendigt som en midlertidig overgang.

I relation til refaktorering, så vil det oftest være Presenter, der skal overtage de dele af koden fra View, eller Delphi-forms, som omhandler især *actions*. Som tommelfingerregel vil det være events og handlinger fra View, som ikke kræver *uses* fra hverken VCL eller FMX. De forskellige problemstillinger vil også blive beskrevet med eksempler i senere kapitler.

Model

Presenter opretter en Model og kender den konkrete objekt, og selvom Model ikke kender noget til Presenter, så er der ikke noget, der forhindrer, at det kan blive et klassisk object-orienteret forhold, hvor det er muligt at bruge event-drevet programmering. Det kan også gøre en forskel i refaktorering uden at ændre eksisterende kode. Igen, det er ikke MVP, der definerer et behavioral pattern og forholdet mellem Presenter og Model, og derfor kan vi se klassiske forhold mellem Model og Presenter, men vi vil oftere se Model definere et interface for Presenter, fordi det er ligetil. I frameworks med MVP vil vi også blive tilskynet til at bruge en bestemt tilgang, og i nogle situationer under refaktorering vil mange

finde en tilgang med Supervising Controller som den mest oplagte tilgang, fordi det kan forekomme uoverskueligt at adskille konstruktioner af kodelinjer. Og dette er ikke det samme som at sige, at Supervising Controller er den rigtige løsning.

Model-View-ViewModel

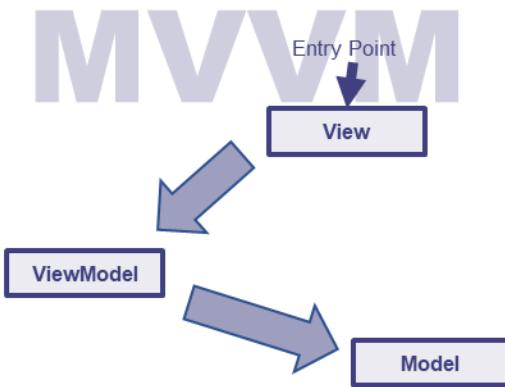
I modsætning til de varianter, så har MVVM en lidt mere spøjs historie, fordi den er en variant af MVC og en konkret fortolkning af Presentation Model af Martin Fowler, som igen er en variant af MVC. Den eneste forskel er, at MVVM foreskriver en eksplisit data-binding, som vi ser det i Windows Presentation Foundation (WPF), medens Presentation Model foreskriver en form for synkroniseringsmekanisme mellem Presentation Model og View, uden at være specifik omkring en løsning.

For begge patterns handler det om at gøre ViewModel en repræsentation af et View, uden de visuelle elementer. ViewModel er et View uden grafik, dvs. ingen VCL- eller FMX-units i *uses clauses*, ligesom ViewModel vil udstille muligheder for data-binding, som et View må gøre brug af. Vi er ikke bundet af samme data-binding mekanisme som i WPF, men oftest vil vi se Observer pattern i implementeringer af MVVM i frameworks. Med MVVM-løsningen i WPF banede den vejen for, at UI-udviklere kunne sidde med et View, definere data-bindinger til en ViewModel, og lade de øvrige udviklere sidde med ViewModel og Model. Det var i hvert fald det behov, der var hos Microsoft.

View

Som i MVC og MVP patterns, så er View det samme. Den viser en repræsentation af data fra Model, som View får igennem en ViewModel. Ideelt set skulle et View defineres gennem Extensible Application Markup Language (XAML) helt uden forretningslogik, men ofte vil et View omfatte implementation, der har høj fokus på det grafiske, der er svær at definere i en XAML. På samme måde kan filen, i en løsning med Delphi-form, indeholde kode, der er centreret omkring det grafiske, eller andet der er specifikt for VCL eller FMX, ligesom .dfm-filen kunne gå ud for at være en XAML-fil.

View håndterer også brugerens input og sender disse gennem en defineret data-binding, og i Delphi kan det også være gennem data-bindinger i data-aware-komponenter, callbacks, actions, properties osv., som allerede findes i komponenterne. Derfor kan der i mange tilfælde være tale om at implementere et simpelt Observer pattern for de dele af en form, som ikke har data-binding. Om det er en simpel løsning, afhænger selvfølgelig af kompleksiteten af en form, men der er mulighed for at ignorere en implementering af en konkret data-binding, indtil refaktorering er på plads, hvorefter man kan arbejdet på at honorere MVVM.



ViewModel

ViewModel er en abstraktion af View, der blotlægger properties og handlinger, og i principippet tilbyder View en måde at binde sig til dem på. I modsætning til Controller i MVC eller Presenter i MVP, så vil en MVVM-implementation specifikt stille et data-binding-mekanisme til rådighed, som automatiserer kommunikationen mellem View og de tilgængelige properties og handlinger. En stor teknisk forskel mellem ViewModel og Presenter i MVP er, at Presenter har en reference (typisk interface eller som en abstrakt klasse) til en View, men det har ViewModel ikke. I stedet binder en View sig til de forskellige properties hos en ViewModel og så at sige lytter til dem, ligesom View kan aktivere en handling i ViewModel gennem en binding. Samtidig vil View kunne modtage notifikationer, når der sker ændringer på status eller data.

Denne pattern kræver en bindingsmekanisme, eller et sammenkog af kode, der gør det ud for data-binding. Denne data-binding kan bedst sammenlignes med data-aware komponenter i Delphi, f.eks. TDBEdit, hvis DataSource property er linket til en data-source, som sagtens kan placeres i en ViewModel – der bliver etableret en bagvedliggende binding mellem de komponenter, hvor TDBEdit er den visuelle komponent og en DataSource er den ikke-visuelle komponent. Det er lidt af samme princip, særligt bindingen, som er af Observer pattern, der sigtes efter i MVVM.

En bindingsmekanisme, der går begge veje, er helt klart den største udfordring i refaktorering mod MVVM, og vil helt klart kræve lidt mere kode end formålet er med en refaktorering. Og under refaktorering mod MVVM kan det helt klart anbefales at følge en refaktorering mod MVP, fordi en sådan mekanisme vil være lettere at implementere, når adskillelsen af delene er sket som i MVP.

En klar gevinst ved en adskillelse af View og ViewModel med binding, er en bedre overskuelighed i implementering af asynkrone metoder i ViewModel og muligheden for at sende notifikationer til View, når disse er færdige. Det giver en bedre oplevelse for brugeren, når der ikke er større blokerende perioderne i View. En anden gevinst ved bindingen er, at det vil være muligt at koble forskellige Views samtidigt til samme ViewModel.

Model

Model vil som regel bestå af såkaldte ikke-visuelle komponenter, dvs. komponenter der ikke er beregnet til en visuel præsentation af data på skærmen, men kan indeholde data i en eller anden form. Opgaven for Model er så at præsentere disse data for de øvrige lag på den bedst tænkelige måde, samtidig med at den kommer til at repræsentere applikationens forretningsmodel. Komponenterne kan enten repræsentere objekter i objektorienteret forstand, eller repræsentere et data-lag og adgangen til en database.

I MVVM er det ViewModel, der opretter en Model, ligesom det selvfølgelig er den, der har en reference til denne, og foretager de nødvendige bindinger. Ligesom i andre patterns, så kan ViewModel sagtens eksponere dele af modellen direkte for View, så der kan skabes data-binding.

Hvorfor MVP eller MVVM?

Vi kunne lige så godt have bestemt os for MVC, så hvorfor er det lige MVP eller MVVM? De er alle sammen varianter af en arkitekturstil, der ligner hinanden, de har samme formål tilfælles, og dog er der nogle tekniske forskelle. Denne bog er særligt målrettet refaktorering af bestående projekter baseret på Delphis Visual Component Library (VCL) – dvs. helt klassiske desktop-applikationer – og det udfordrer valget af de nævnte patterns. Selvom MVC er stammoderen til de nævnte architectural patterns, så er det ekstremt vanskeligt at refaktorere eksisterende Delphi-forms til MVC uden at ændre grundlæggende på koden og i oprettelser af forms, og så taler vi ikke længere om refaktorering. Controller i MVC er *entry point* for en applikation, dvs. at Controller opretter View og Model. Derimod er View i både MVP og MVVM entry points, og gør det oplagt for klassiske Delphi-applikationer, fordi en Delphi-form altid har været entry point.

En Delphi-form vil gå for at være View, og den vil i så fald oprette og have reference til en Presenter i MVP eller en ViewModel i MVVM, hvorimod Controller i MVC ville have haft en reference til en View, hvilket vil indebære en fundamental ændring i tilgangen. Det er den korte udgave i årsagen til, at det er nemmere at refaktorere klassiske Delphi-applikationer frem mod MVP. Det vil være muligt at fortsætte fra MVP mod MVVM, dog med en lidt større indsats. Samtidig kan det gøres i små skridt, og i mange tilfælde ved at udnytte midlertidige referencer og fortsætte refaktorering løbende. MVVM har opnået popularitet for desktop-applikationer efter Microsoft, som nævnt, tilrettede Martin Fowler's Presentation Model med tanker fra MVC-miljøet, og indarbejdet dette pattern i Windows Presentation Foundation (WPF). Denne arkitekturstil er også stærke til event-drevet programmering, som også er kendtegnende ved Delphi og dennes primære RAD-tilgang.

Samtidig er Martin Fowler ikke ringere end manden bag bogen "Refactoring" og populariseringen af *Code Refactoring*, som grundlæggende er en praksis med at restrukturere kode uden at ændre dens funktionalitet. Og dette er udelukkende for at skabe et bedre design, struktur og implementering, ligesom det også kan øge læsbarhed og mindske kompleksitet af et system, som igen kan afhjælpe vedligeholdelsen og give muligheder for at videreudvikle systemet. Og med Martin Fowler som fællesnævner, så bliver MVVM og Code Refactoring ganske tæt forbundet, og der er ikke noget at sige til, at det ikke er umuligt at associere de to tilgange med "Refaktorering mod MVP og MVVM".

Særligt MVVM er dog kendt for at have nogle udfordringer, fordi det kan virke som et overkill i mindre projekter, samtidig med at det kan ramme performance i store projekter. Derfor er der også en stor sandsynlighed for, at de fleste vil nå frem til, gennem erfaring, at MVP er endog særdeles og ganske tilfredsstillende i refaktorering af bestående Delphi-projekter. Mange vil nok synes, at det er fantastisk, at vi ikke behøver at omskrive koden, men "blot" refaktorere koden, før vi kan have en fortælling om, at vi følger et anerkendt architectural pattern, men også at vi kan vise, at vi implementerer *Coding Best Practices* i vores arbejde med kodekonstruktioner.

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."
– Martin Fowler, Refactoring.

Refaktorering

Teknisk set kommer udtrykket fra matematik – det at faktorisere et udtryk til en ækvivalent, der er en renere måde at udtrykke det samme udsagn på. Det vil sige, at produkterne funktionelt set skal være identiske. Det er en anden måde at sige, at vi rydder op i kode, men at refaktorering ikke er det samme som at omskrive kode. Den tekniske forskel er altså at reorganisere kode uden at ændre funktionaliteten, og det er faktisk noget vi gør hele tiden i forbindelse med design af løsninger. Her vil nogle påpege eller genkende, at vi ofte refaktorerer mod patterns og at vi sjældent designer eller udvikler ud fra patterns.

På den anden side findes der er også udviklere, der synes at det helt hen i vejret, at refaktorere kode, der rent faktisk virker, til noget der igen skal virke på samme måde, bare med den risiko at koden kan – så at sige – gå i stykker under processen. Derfor bør vi aldrig refaktorere velfungerende kode bare for at refaktorere, eller bruge det som en undskyldning for at ændre "andre folks dårlige kode" til noget kode, vi selv kan lide – se evt.

"Håndbog i Object Pascal Style Guide". Refaktorering skal ske, fordi det giver mening på sigt at kunne tilpasse fungerende software til nye krav, f.eks. at brugerne har et ønske om at et projekt skal kunne afvikles på macOS, men at vi har at gøre med en VCL-projekt.

Anskuet på den måde, så handler denne bog ikke om de små refaktoreringer, vi foretager på daglig basis, og som måske bedre kan dækkes af en standard kodestil og patterns, men om de større opgaver vi nogle gange må tage på os i forbindelse af en opgradering eller migrering til helt nye generationer af værktøjer og platforme, eller en overgang fra et paradigme til nyt sæt af paradigmer, eller – mest sandsynligt – fra en gammel arkitektur til en ny arkitektur, der skal understøtte en virksomheds overgang til nye markedskrav, f.eks. ved at overgå fra én form-unit separeret til flere units som i MVP eller MVVM.

At vi kan udnytte koncepter som Refactoring, så er det fordi det kan lade sig gøre at flytte kode fra en form-unit til 2 eller flere units, uden at ændre funktionalitet på koden. Og udover at det kan flyttes i små bider ad gangen, så er det også muligt at separere koden i blot 2 units i en Model-View (MV) eller View-ViewModel (VVM), og senere i 3 units (MVP eller MVVM). At separere form-unit i 2 units uden at tilføje yderlige kode antyder også, at der stadig vil være en stærk kobling mellem de to units, men at det vil virke uden at vi tilføjer kode og at de 2 units, især for store filer, kan give et udgangspunkt i den endelige separering i 3 units, hvor fokus vil være en løs kobling og en endelig deling af ansvaret mellem de 3 units (Separation of Concerns).

Allerede ved separering af en form til MV vil den nye view med en VCL-form kunne udskiftes med en FMX-form, og det alene åbner op for nye muligheder. Dvs. at de første

tiltag egentlig handler om at flytte kode, sikre referencer, sikre tests, og ikke så meget om at refaktorere koden i gængs forstand.

Tilnærmet MVP og MVVM

Årsagen til, at vi sigter efter en tilnærmet og ikke en fuldkommen MVP eller MVVM er, at vi godt er klar over at vi selv har skabt behovet for refaktorering af traditionelle Delphi-applikationer, fordi disse ofte består af forms med meget tætte koblinger (*tight coupling*) med alle de tilhørende klasser. MVP og MVVM handler om at afkoble form, eller View, fra koden gennem en effektiv kommunikation, men et MVP- eller MVVM-framework er ikke understøttet af Delphi, og det er en del af udfordringen – i hvert fald ikke på samme måde som andre udviklingsværktøjer, som har haft decidederede architectural patterns in mente.

Når vi taler om tæt kobling, så tænker vi i hvilken grad de forskellige moduler er afhængige af hinanden. Vi vil altid sigte efter, at alle moduler så vidt muligt er uafhængige af hinanden, derfor løs kobling (*loose coupling*). Det er den perfekte scenarie, men det opnår vi ikke ved alene at separere forms til en View og en Model eller en Presenter. Vi er nødt til at bevare den samme grad af kobling, vi finder i forms, blot med referencer mellem View og Model eller ViewModel, ligesom vi måske også må gemme en reference i Presenter, så vi reelt får en delløsning, der er let at veksle til en MVP pattern som det første, før vi kan tage springet og mindske koblingen ved at fjerne nogle af referencerne. På den måde kan denne bog også inspirere til at søge en tilgang, der er tilnærmet MVP fremfor MVVM, hvis dette giver mere mening for et aktuelt projekt. Der er heller ikke noget til hinder for, at vi vælger at stoppe ved MVP, hvis MVP giver den arkitektur, der er fleksibel nok til at tilpasse et system til nye behov.

Om vi vælger først at separere en form til en View og en Model eller en Presenter, eller alle tre lag på samme tid, afhænger i særdeleshed af en forms indhold, og selvfølgelig erfaringerne i teamet med afkobling. Det er også muligt, at indholdet af en forms unit er så lille, at det virker overkomeligt at flytte relaterede dele af formen til et View, en Presenter og Model i samme proces. Det er også tænkligt, at en form består af så mange elementer og kode til manipulering af data, at det giver mening af starte med en View og en Model i små skridt. Hvert skridt skal være lille eller stor nok til, at programmet fungerer som den plejer efter adskilleserne, før vi tager næste skridt.

Det er også muligt, at hovedparten af en forms unit består af en masse brugergrænsefladerelaterede kode og meget lidt manipulering af data, hvorfor det er nærliggende at starte med en Presenter. Og efter nogle få forms, så vil de fleste teams også finde præcis den rytme, der er bedst egnet for den type af projekt, som de sidder med. Hovedfokus skal bare være møntet på, at hvert element eller kode, der er flyttet, stadig virker som det skal, før vi kan tillade os at begynde med det næste element eller kode.

Anbefalinger før refaktorering

Der er i høj grad tale om god praksis at vurdere, om en Delphi-form er egnet til at blive separeret i overensstemmelse med et architectural pattern. Det kan være, at filen består af

tusindvis af kodelinjer, eller meget lange *procedurale* metoder, store klasser, alt for mange parametre, eller en hver anden form for *code smell* og *design smells* i øvrigt, som kan være tegn på, at der er akkumuleret teknisk gæld over tid, og som vil gøre det ekstremt svært at flytte kode efter behov. Det giver næsten sig selv, at enhver form for *anti-pattern* i vores kodebase vil være uforeneligt med et *architectural pattern*.

- Få identificeret design smells i systemet, og overvej om der skal foretages almindelig refaktorering.
- Overvej om små Form-units skal separeres, idet de i mange tilfælde kan være et overkill – benyt evt. krav som multiplatforme som et argument for, at det skal de.
- Forståelse af architectural patterns i et team, herunder MVC, men især MVP og MVVM, er en forudsætning for at kunne implementere patterns korrekt. Hvis ikke, så tilføjer separeringen blot yderligere kompleksitet for udviklerne.

Derudover giver det selvfølgelig mening at benytte værktøjer som versionsstyring (f.eks. Git eller SVN), fordi der vil være situationer, hvor en refaktorering går skævt, og vi altid skal være sikre på, at vi til enhver tid kan gå tilbage til en stabil version af applikationen.

Fordele og ulemper ved patterns

Der er både fordele og ulemper ved architectural patterns, og de bør vejes op imod hinanden før vi beslutter os for at refaktorere hele vores system mod et af disse patterns. De forskellige architectural patterns har nogle tekniske forskelle, men de har en række overordnede fordele og ulemper til fælles. Deres største fordel er uden tvivl, at de sigter efter et design princip om *Separation of Concerns* (SoC), dvs. at hvert område har særskilte kerneopgaver. Princippet i SoC benyttes i mange felter, og de patterns, der sigter efter SoC, bliver således modulært opbygget, hvor der er et veldefineret interface imellem modulerne. I architectural patterns vil vi tale om lag eller lagdeling, og et interface imellem lagene, der definerer den kommunikation, der nødvendigvis må være, men at den teknik, der bruges til at kommunikere, samt hvilke lag, der har referencer til hvilke andre lag, fortrinsvis definerer forskellene mellem de forskelle patterns i arkitekturstilen.

Denne separering af ansvar er ikke uden omkostninger, fordi der netop er en kommunikation imellem, der kan koste lidt i performance i form af overhead. I normale situationer er det ikke en mærkbar overhead, der på nogen måder overtrumper fordelene ved separeringen, men for high-performance systemer vil kommunikationen have et stort fokusområde, hvorfor vi kan have en interesse at have fokus på netop performance.

De teknologier eller design patterns, som de forskellige architectural patterns vil forkynde, har også stærke og svage sider. MVP vil for eksempel være mere fleksibelt med hensyn til at fordele opgaver på forskellige måder, men det gør den også mere sårbar overfor brud på princippet med separation. MVVM vil omvendt helt naturligt tale for separation, men vil også fordré implementering af et Observer pattern imellem lagene. Observer pattern kan bruges i mange henseender, og det er også den teknik, der bruges i Delphis data-aware komponenter. Observer pattern kan implementeres lokalt, men kan også

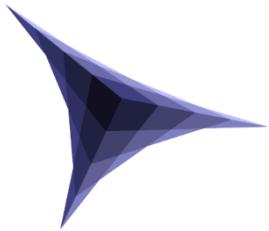
implementeres overordnet, der oftest kan koste noget performance, fordi der vil være en masse observers, der konstant skal notificeres om en masse hændelser, også selvom enkelte observers ikke har nogen interesse i en aktuel hændelse.

De fleste architectural patterns formoder, at View er fuldstændig frakoblet det mellemliggende lag, men i praksis, og med events og bindinger i moderne komponenter, så vil hændelser først og fremmest blive håndteret i View, og i mange patterns må disse hændelser blot sendes videre til det mellemliggende lag. Mange komponenter kan dog godt designes til at binde direkte til en action, der ligger i det mellemliggende lag, og på den måde opfylde tanken om et View, der – i det mindste i sourcekoden – er "frakoblet" det mellemliggende lag, men at komponenten teknisk set bare sender hændelsen direkte videre til en metode i det mellemliggende lag.

Andre væsentlige fordele ved disse separeringer er, at unit tests kan gøres meget nemmere, fordi funktioner i de mellemliggende og underliggende lag kan testes uden at skulle oprette et View først, samtidig med at dette betyder at ethvert behov for redesign af et View også kan blive meget lettere.

Den største ulempe er nok, at alle disse architectural patterns kræver viden og noget erfaring hos teams, for at de rigtige implementeringer bliver udviklet de rigtige steder. Hvis et team ikke har erfaring med et architectural pattern, så kan det meget nemt gå hen og blive en meget kompleks struktur at navigere rundt i. Der skal være strikse regler for overholdelse af en arkitektur, for at sourcekoden ikke skal kollapse i et stort rod. Det vil også kræve et erfaren team at vælge den rette arkitektur til lige præcis deres applikation, og det kræver selvsagt både viden og erfaring. Derfor kan det også give mening at have en erfaren udvikler som arkitekt, som har fokus på disse patterns, og samler op på det arbejde, som de øvrige udvikler lægger i refaktoreringen.

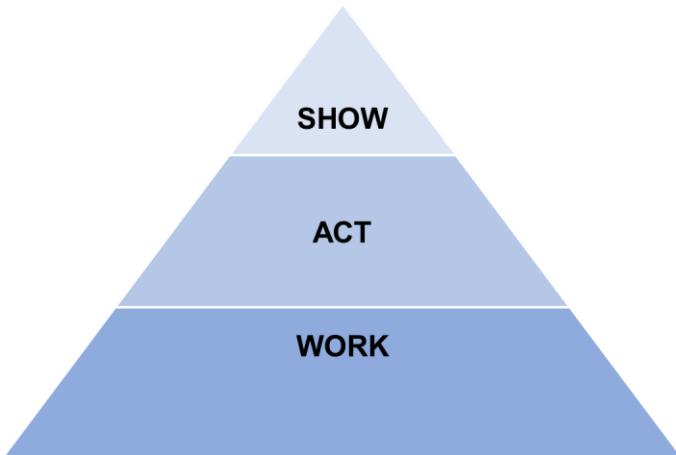
Der er til tider debat blandt udviklere om hvilket pattern, MVC, MVVM, MVP osv., der er bedst egnet generelt, men realiteten er, at nogle patterns vil være egnet til vise projekter eller opgaver og situationer – om det f.eks. er et vindue med faneblade eller dialogboks – og derfor er det mere et spørgsmål om at bruge det pattern, der passer bedst. Af samme grund er man også begyndt at bruge udtrykket MVW, som ikke er et pattern, men står for Model-View-Whatever. Det betyder direkte "whatever works for you" ("hvad end der virker for dig"), som ikke nødvendigvis betyder "at vi bruger det pattern, vi bedst kan lide", men at vi bruger et pattern, der passer til situationen. Det kræver også, at vi ved hvordan de enkelte patterns er skruet sammen, og det ved vi, når vi rent faktisk har prøvet dem. Vi ville nok stå i en anden situation, hvis vi arbejdede med et værktøj og frameworks, som var bundet tæt op af et af disse patterns, ligesom vi vil finde det i "ASP.NET Core MVC".



Kapitel 2

Show, Act, and Work

En ting er *architectural patterns*, noget andet er det praktiske arbejde med at splitte bestående sourcekode op i 3 filer, uanset om vi taler om refaktorering mod MVC, MVVM, MVP eller noget helt fjerde. Det er ekstremt vigtigt, at adskillelsen sker metodisk, eller i det mindste efter planmæssige faser, og efter nogle få men klare principper og koncepter, der kan give en fælles forståelse indenfor et team. Vi kan jo ikke refaktorere på må og få, og hovedudfordringen vil netop være, hvor vi kan lave sammenhængende snit i sourcekoden, og hvor de enkelte stumper af kode i så tilfælde skal flyttes hen. *Refactoring* beskriver allerede, hvordan vi refaktorerer kode og de principper, der gælder for konceptet, men refaktorering fortæller ikke konkret, hvor vi skal skære, og hvad type kode, vi skal flytte hvorhen, når vi har bestemt os for at refaktorere henimod en bestemt pattern.



De fleste architectural patterns af MVC-familien sigter efter et præsentationslag med så lidt kode som overhovedet muligt, som er koncentreret om at generere en brugergrænseflade, der ofte er afhængig af det platform applikationen er udviklet på. De patterns

definerer også et mellem lag, der som regel har til opgave at kontrollere processen og tage stilling til, hvad der skal ske, lige som notifikationer mellem lagene ofte går igennem dette lag. I det sidste lag finder vi data, men især også kode, der er koncentreret omkring forretningsmæssig logik. De sidste to lag vil ofte være uafhængige af et præsentationslag, men også gerne være platformsuafhængige, ligesom det nederste lag også kan være uafhængig af det mellemliggende lag. Det er især behovet for at have samme applikation, eller dele af samme kodebase, på flere platforme, der har skabt en popularitet for især MVC, MVP, og MVVM i forskellige programmeringssprog og frameworks.

Idéen med SAW er at tredele kodelinjer, eller kategorisere hver kodelinje i et af disse lag, og derefter flytte dem til laget. Det havde været rart med et værktøj, der kunne farvelægge de enkelte kodelinjer efter en kategori, men et sådant værktøj har vi desværre ikke til rådighed. Herunder vises et kort eksempel, hvor en grøn markering er forretningsspecifikt, Work, en blå markering er præsentation, Show, og en rød markering er en handling, Act.

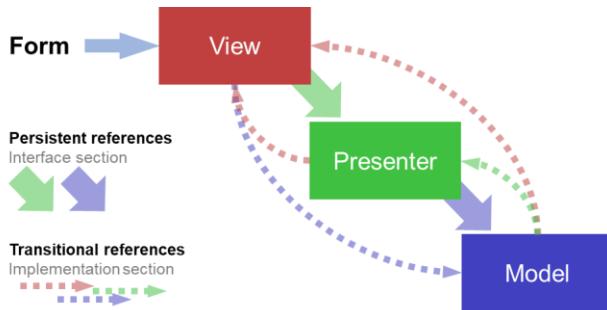
Vi kan inddæle kodelinjer i følgende kategorier; *Show*, *Act*, og *Work* (SAW), fordi de kan give os et fælles sprog for, hvordan vi kan skelne mellem de enkelte variabler, komponenter, metoder og kodelinjer, og hvordan vi kan tyde dem. SAW vil så vidt muligt være pattern-agnostisk, men i hvor høj en grad de enkelte kodelinjer flyttes til de enkelte lag af en arkitektur vil bære præg af det specifikke pattern, vi sigter efter. SAW erstatter på ingen måde en pattern, men er i stedet et redskab for udviklere i arbejdet med at afgøre, hvor en kodelinje hører til. Af samme årsag kan vi også benytte generelle vendinger som øverste lag (såsom View), mellemliggende lag (bl.a. Presenter som i vores eksempler), eller nederste lag (såsom Model), når vi sædvanligvis taler om architectural patterns, og fordi det er naturligt at tale om lag i softwarearkitektur.

I modsætning til ord som View og Model, der er navneord og dækker over en genstand eller noget, der dirigerer eller regulerer ting, såsom Controller, så består delelementerne af SAW af udsagnsord, dvs. at der sker en bestemt handling, eller noget er i en bestemt tilstand. Show, Act og Work har selvfølgelig ikke noget at gøre med en forestilling eller en opgave, men skal ses i sammenhæng med at præsentere, reagere, og udføre et arbejde. SAW skelner mellem kategorier af objekter og metoder, men har langt større fokus på, hvad en konkret kodelinje egentlig gør, og derfor nemmere vil afgøre om en linje mere naturligt vil tilhøre et bestemt lag i et architectural pattern.

Vi kan godt glemme alt om at flytte hele kodekonstruktioner eller hele metoder til et bestemt lag af en pattern – det vil aldrig fungere, fordi hver kodelinje i én kodekonstruktion kan nemlig høre til hvert sit lag af et pattern, og én kodelinje kan tilmed have udtryk, der hører til de øvrige lag af et sådant pattern. Hvis én kodelinje kan omfatte samtlige lag af en pattern, så er det præcis derfor, at vi bør være metodiske, for at en refaktorering kan udføres sikkert og en applikation stadig vil fungere som den skal.

Som vi har nævnt i introduktionen, så bestemmer architectural patterns hvordan lagene bør referere til hinanden, så vi kan opnå en tilsigtet løs kobling, og så bliver det ikke helt uden betydning, hvordan vi flytter koden. Vi vil etablere midlertidige referencer til en

masse tilbageværende variabler og metoder, som vil være i strid med de forskellige patterns, simpelthen fordi mange forms vil være så store, at det vil være umuligt at flytte det hele på én gang. Hvis vi ikke bliver enige om, at en enkel flytning ikke må tage mere end én dag, så bliver det svært at opretholde en regel om, at refaktorering bør ske i skridt, så vi altid har en fungerende applikation til rådighed. Fordi vi sigter efter MVP eller MVVM, så vil vi også kunne bevare permanente referencer i interface-sektionen fra View mod Presenter, og fra Presenter mod Model. Vi kan ikke undgå midlertidige referencer under refaktoreringen i implementationsdelen fra Model mod det mellemliggende lag og videre mod View, men de skal slettes som en del af en fase.



I figuren herover skal de midlertidige (stiplede) referencer fjernes så snart refaktoreringen er gennemført, og vi har etableret en kommunikation mellem lagene uden de referencer. Det mellemliggende lag, som vi i figuren herover har valgt at kalde "Presenter", er ikke en generel betegnelse, men opträder på vegne af de andre. Alle varianter af MVC pattern har et mellemliggende lag, som indtager en eller anden rolle, der udgør det mellemled, der i mere eller mindre grad udgør separeringen af View fra Model, og det sker også med varierende ansvar. Presenter er vores første mål, og bruges derfor i eksempler og figurer.

Der vil forekomme erklæringer i flere eller alle lag, som er afhængig af en type definition, der ikke kan placeres i et af disse lag, fordi lagene i sidste ende ikke bare kan referere til hinanden på kryds og tværs – kun oppefra og ned. Dette vil blive tydeligt, når de midlertidige referencer skal fjernes, og de type-erklæringer må da givetvis høre til en særskilt unit med fælles type-erklæringer. Der er også risiko for, at der opstår behov for referencer fra implementationssektionen af View til Model, men konceptet i dette værktøj vil forsøge at tage højde for, at dette ikke bliver nødvendigt, ved at respektere den rækkefølge de enkelte kodelinjer, objekter og variabler bør flyttes.

Mængden af koder, der bør flyttes i samme ombæring, må således ikke foregå over mere end én dag. Når en arbejdsgang er afsluttet, så skal der være en fungerende applikation til rådighed. Det er ikke kun en god tommelfingerregel, der holder mængden af kode, der flyttes, på et overskueligt niveau, men også at mange teams i forvejen har build- og test-servere, der kører natten over, ligesom de kan have forpligtelser med hensyn til *continuous integration* og *continuous delivery*. Disse tanker om refaktorering mod et bestemt pattern sigter på den måde også på at støtte teams i et agilt miljø, men kan i øvrigt benyttes frit i mere traditionelle teams. Tilgangen sikrer også, at alle udviklere ikke behøver at

arbejde med refaktorering, men at dele af et team kan fortsætte med at koncentrere sig om at fixe bugs og udvikle features i øvrigt.

Filer og Navne

Navne, herunder filnavne, er uhyre vigtige i Software Design Patterns, eller i Coding Best Practices generelt. Det er afgørende for udviklere, at de kan genkende et design pattern ud fra navne alene, og det gælder også for filnavne og de mapper, som filerne bliver en del af. Det er ikke en del af denne tilgang at restrukturere placering af filerne, fordi det kan introducere vidtrækende ændringer, som ikke vil være i tråd med konceptet bag refaktore-ring. Andre udviklingsværktøjer, der understøtter et architectural pattern, udnytter ofte en mappestruktur til at holde styr på filerne, der hører til bestemte lag, men de funktioner har vi ikke til rådighed i Delphi.

Det første skridt bør altså være at omdøbe den eksisterende filnavn til den Delphi-form, der skal refaktoreres, og få resten af projektet til at acceptere, at filen nu hedder noget andet, ligesom den gamle fil skal slettes, hvis et versioneringsværktøj ikke har gjort det for os. Dvs. at referencerne til den gamle fil skal ændres til den nye fil, og at applikationen skal kunne kompileres og virke som den altid har gjort, før vi kan fortsætte til næste skridt. Udover at det er god praksis, at et pattern er synlig i navnene, så vil den eksisterende fil med formen komme til at blive adskilt i 3 filer, og så er det en god idé, at der er et system i de navne.

Hvis det gamle filnavn f.eks. er MainUnit.pas, så overvej Unit Scope Names, eller at det nye navn som det mindste ender med et sammensat navn, en endelse, med det lag, som filen skal repræsentere. MainUnit.pas kan f.eks. blive til MainUnit.View.pas eller Main-UnitView.pas. Hvis vi allerede har et *naming convention* til filerne, så skal disse forhold blot indarbejdes i konventionen. Og hvis vi har indarbejdet en *bad practice* med præfikser til filnavne, så som frmMain.pas, så overvej at ændre den praksis – ikke kun fordi det ikke er standard kodestil for Object Pascal, men fordi det vil give udfordringer med præfikser til både Views, Presenter, ViewModel, Model osv.

Achitectural patterns, ligesom de øvrige Software Design Patterns, hører til Coding Best Practices, og på samme måde hører standard kodestil som Object Pascal Style Guide også til Coding Best Practices. Hvis vi allerede har problemer med at forholde os til navngivning af filer på dette niveau, så bør vi nok overveje, om refaktorering mod en arkitektur er den rigtige vej. Har vi sagt A, så bør vi også sige B, og det handler om, om et aktuelt projekt er modent til refaktorering frem mod et architectural pattern, eller om der er nogle underliggende dårlige praksisser, der bør refaktoreres først.

Omdøbning af filnavne er måske ikke den største udfordring, og der er selvfølgelig andre aspekter som Unit Tests, der skal tages fat om, men også versionsstyring. Med et versioneringsværktøj risikerer vi at miste hele historikken for f.eks. MainUnit.pas, hvis vi bare gemmer som i "Save as" i Delphi, tilføjer den "nye" fil til versionsstyring, og fjerne den gamle fil fra vores *repository*. Hvis vores versioneringssystem understøtter omdøbning af filnavne, og på den måde kan bevare historikken af filen, så er det selvfølgelig langt at

foretrække. Vi må så efterfølgende manuelt ændre referencer af unit-filen i både projekt-filen og de øvrige units.

Det næste skridt er at overveje, hvad der skal ske med formens navn. Der er ikke noget til hinder for, at vi f.eks. bibeholder et forms navn som MainForm: TMainForm. Det skal vuges op imod at vi skal have tilføjet 2 nye hovedklasser som supplement til MainForm, som i MainPresenter og MainModel, og her vil MainView være det mest passende navn, fordi det vil afsløre det pattern, som disse units er bygget efter, og så navne på filerne ikke står alene. Hvis formens navn ikke på nogen måde er bundet, som i *hardcoded*, så bør det bestemt tale for en omdøbning af formen, også selvom tests skal skrives lidt om.

Det tredje skridt er at tilføje de nye units, som skal supplere MainUnit.View.pas, nemlig MainUnit.Presenter.pas og MainUnit.Model.pas. Det er ikke ligegyldigt, hvor de bliver oprettet og i hvilken rækkefølge de bliver oprettet, og ikke mindst hvad det er for en type unit, vi opretter. Eftersom der oftest indgår såkaldte *non-visual components* i forms, der i mange tilfælde skal flyttes til Presenter eller Model, så er det nødvendigt at oprette de supplerende units med en klasse af typen TDataModules. Data moduler i Delphi er ikke længere forbeholdt database-komponenter, men er med Delphi XE2 blevet helt framework-neutrale og benyttes nu også til at samle håndtering af ikke-visuelle-komponenter i en applikation. En nyoprettet og tom data module har kun referencer til System.SysUtils og System.Classes, og er derfor uafhængig af platform og har heller ikke nogen referencer til database-komponenter.

Oprettelse af de nye units sker derfor gennem File / New / Other... og modulet findes under Delphi / Database, og hvis projekt-filen bruges til at oprette formen, så skal de oprettes i nedenstående rækkefølge:

```
begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TMainView, MainView);
  Application.CreateForm(TMainPresenter, MainPresenter);
  Application.CreateForm(TMainModel, MainModel);
  Application.Run;
end.
```

Der er mange projekter, hvor oprettelsen af forms kontrolleres på en anden måde, men hvis vi blot er opmærksomme på, at de oprettes i den rigtige rækkefølge, så er der fri mulighed for at implementere egne konstruktioner. Normalt vil vi også overføre referencer til forskellige lag til relevante lag gennem oprettelsen, ifølge et pattern, men fordi projekt-filen håndteres af Delphi og de autogenerede CreateForm ikke giver sådanne muligheder, så vil overdragelsen af disse referencer ske under konstruktionen af de enkelte forms eller moduler.

Når de enkelte lag er oprettet, så er det vigtigt, at vi skaber referencer i uses, som vil blive nødvendige i refaktoreringen. De referencer, der bliver tilføjet i interface-sektionen vil være permanente, hvorimod de referencer, der bliver tilføjet i implementationssektionen er midlertidige, der vil blive fjernet, når vi er færdige med processen:

Form	View	Presenter	Model
<pre> unit MainUnit; interface uses Winapi.Windows, Winapi.Messages, Sys System.Classes, Vcl.Graphics, Vcl.Co type TForm = class(TForm) private public end; var MainForm: TMainForm; implementation {\$R *.dfm} end. </pre>	<pre> unit MainUnit.View; interface uses Winapi.Windows, Winapi.Messages, Sys System.Classes, Vcl.Graphics, Vcl.Co MainUnit.Presenter; type TForm = class(TForm) private public end; var MainView: TMainView; implementation {\$R *.dfm} end. </pre>	<pre> unit MainUnit.Presenter; interface uses System.SysUtils, System.Classes, MainUnit.Model; type TMainPresenter = class(TDataModule) private public end; var MainPresenter: TMainPresenter; implementation {\$R *.dfm} uses MainUnit.View; end. </pre>	<pre> unit MainUnit.Model; interface uses System.SysUtils, System.Classes; type TMainModel = class(TDataModule) private public end; var MainModel: TMainModel; implementation {\$R *.dfm} uses MainUnit.Presenter; end. </pre>

Hvis vi kigger tilbage på vores figur under side 28 med referencerne, så ser vi konkretiseringen af de referencer herover, som er aftegnet i figuren. Samtidig ser vi *uses* i interface-sektionen af View er proppet med referencer til Winapi- og Vcl-biblioteker, som ikke må optræde i hverken Presenter eller Model. De biblioteker er ikke bare platform-specifikke, men har også i høj grad stor fokus på den visuelle præsentation af data. Derfor er det oftest en simpel regel, at vi holder platform-specifikke referencer i vores View-lag. Det er ikke altid lige til, og der kan være undtagelser, der kan kræve f.eks. en compiler-direktiv, men det er udgangspunktet at de øvrige lag ikke er platform-specifikke.

Når de supplerende units er oprettet og referencerne mellem units er på plads, så kan vi stadig kompilere applikationen, og det vil stadig fungere som altid, men på dette sted vil vi nu klar til at flytte kodelinjer fra den ene View til både Presenter og Model.

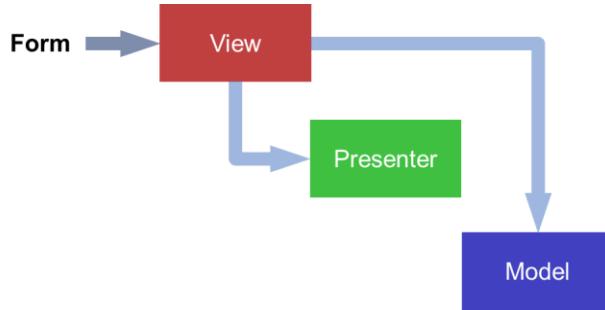
Forskellige tilgange

Udover kategorisering af kodelinjer, så har vi nogle få angrebsvinkler til rådighed, som ikke er baseret ud fra et pattern, vi sigter efter, men især for at holde fast i en systematisk tilgang til refaktoreringen, så det ikke sker tilfældigt. Den mest åbenlyse tilgang ville være at finde de enkelte elementer og flytte dem fra View til Presenter, og fra View til Model.

Det er også den tilgang, der vil give færrest arbejde, men det kan også virke uoverkommeligt ved større units. Selvom det virker ligetil, så kan det måske ved større opgaver, være mere overskueligt at flytte f.eks. database-komponenter til Model først, og stille og roligt arbejde sig videre med kodelinjer til Presenter.

Og i andre sammenhænge kan det virke mere overkommeligt med Presenter først og Model bagefter. Eller at refaktoreringen sker i grupper efter relaterede metoder og komponenter, og flyttes til både Presenter og Model. Her er det dog vigtigt at holde fast ved, at vi først flytter relaterede metoder og komponenter til Presenter, og først derefter flytter relaterede metoder og komponenter fra View til Model, fordi vi i modsat rækkefølge vil være nødt til at have midlertidige referencer til Model i View. Flytter vi først til Presenter, så kan kodelinjer have eksplisitte referencer til View, som senere skal flyttes til Model – de

eksplicitte referencer til View vil helt naturligt blive markeret som ugyldige, hvorfor de er ligetil at ændre med eksplisitte referencer til Model.



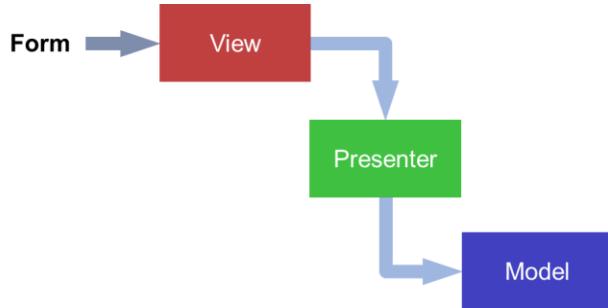
Det er ikke svært at forestille sig, at vi på den måde vil være afhængige af midlertidige referencer, fordi det vil være umuligt at flytte alle komponenter, metoder og kodelinjer på én gang ved større units. Der vil være referencer til dele af de tilbageværende komponenter og metoder. De midlertidige referencerne tjener jo først og fremmest den vigtige funktion, at vi kan sikre at koden virker, som den altid har gjort, i og med vi ikke ændrer noget i koden bortset fra referencer i uses og eksplisitte referencer i kodelinjer. De eksplisitte referencer i kodelinjer vil på et senere tidspunkt tjene som kilde til at definere et interface, før vi endelig kan fjerne referencerne fra uses, og det kan også ske uden at vi behøver at ændre noget i kodelinjerne.

Processen kan være en stor udfordring, og en adskillelse ligger ikke altid lige for, f.eks. hvis kildekoden er massiv og rodet. Hvis sourcekoden i øvrigt synes at bestå af megen *code smell*, så vil det nok være en god idé at få ryddet op i koden, før vi begynder at flytte rundt med koden – det er ikke nogen god idé at rydde op i koden samtidig med at vi flytter det rundt. Det beror alene på, at vi skal sikre at flytning af kode sker med det mål, at applikationen virker som altid, og hvis ikke, så må der ikke sættes spørgsmålstege ved om årsagen er flytning af kode, eller om vi har været lidt for "smarte" med at rydde op i koden.

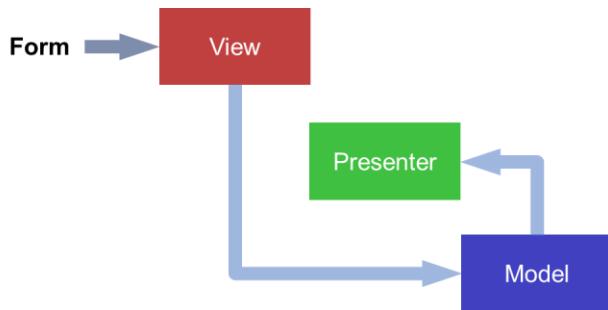
Koden skal, så vidt det er muligt, altid flyttes som den er, og ikke ændres undervejs. Kode, der trænger til at blive kigget på, fordi en anden løsning vil være mere attraktiv, bør noteres som en opgave, og ændringer foretages efter afslutning af refaktoreringen. Samtidig vil kode, der er flyttet som den oprindeligt var, gøre *code reviews* langt nemmere, fordi det vil fremgå af versioneringsværktøjet, hvilket ændringer, der er sket, og de kan sammenlignes kodelinje for kodelinje.

Vi kan også vælge at flytte kode, der ikke har direkte relation med View og flytte dem til Presenter, for så senere at flytte kode til Model, der har relation til data og forretningslogikken. Det kan synes omsonst, men det kan også gøre kode mere overskuelig, når relateret kode i View er separeret. Det er også den tilgang, der giver de mest fornuftige referencer, fordi flytningen sker oppefra og ned, men det giver også lidt ekstra arbejde i og med at kode, relateret til Model, vil blive flyttet to gange. Vurderingen må ske afhængig af de

enkelte forms, om dette er den rette tilgang. Denne tilgang kan også virke mindre udfordrende, fordi vi "kun" skal tage stilling til kodelinjer, der hører til View, hvorfor vi kan vente med at tage stilling til kodelinjer, der hører til Presenter eller Model. Når vi senere skal tage stilling til kodelinjer, der skal flyttes til Model, så er det uden kodelinjer til View, og på den måde mere overskueligt.

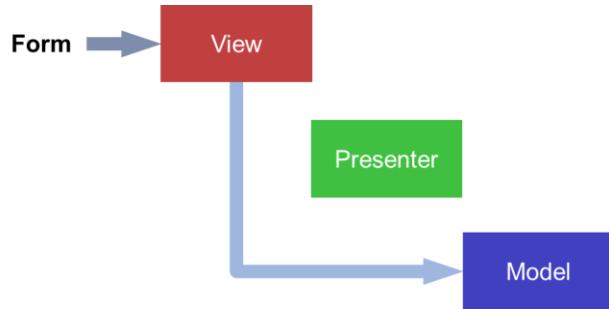


I samme boldgade finder vi en tredje tilgang, nemlig at vi flytter kode til Model, der ikke har nogen relation til View. Herefter kan vi starte med adskillelse af kode, der reelt hører til det mellemliggende lag. Denne tilgang kan i andre situationer virke mere overskueligt, fordi vi oftest har en masse kode, der har relation til data og forretningslogik. Dog er vi startet vi med at skabe referencer fra View til Presenter, ikke fra View til Model, og det vil stride imod det endelige resultat, hvilket vil kræve stor disciplin at fjerne igen. Udenfor selvfølgje, at der vil være noget kode, der flyttes en ekstra gang fra Model til Presenter. Det kan virke ulogisk, men det kan i nogle tilfælde give mening, at vælge denne tilgang, selvom vi klart vil foretrække de andre tilgange.



Ovenstående tilgang baner også vejen for en variant af samme tilgang, som vi nok finder som den simpleste tilgang, nemlig at vi flytter kode, der har relation til data og forretningslogik til Model, og lader således kode, der hører til Presenter, forblive i View, sammen med kode til at præsentere data. Denne tilgang kan også blive til et slutresultat, dvs. at vi "nøjes" med en halv løsning, eller den knap så anerkendte Model-View-arkitektur. Resultatet kan også bruges til at flytte kode til Presenter på et senere tidspunkt, dvs. at kode i View, der ikke har direkte relation til View, kan flyttes til Presenter, så vi i sidste ende med et resultat, der minder om et af de andre patterns.

Idéen om først at flytte kodelinjer, der hører til Model, åbner op for at udnytte den noget mere simplificerede arkitektur, og stadig en variant af MVC, hvor man lægger View og Controller sammen, og som herefter blot er kaldt Model-View. Det er for så vidt ikke et udbredt eller anerkendt pattern, men ikke desto mindre indarbejdet i forskellige miljøer og frameworks. Bl.a. har Qt et Model/View-framework, og selvom de bruger abstrakte klasser og *delegates*, som en teknisk løsning i kommunikationen mellem lagene, så er det ikke nogen hindring for at definere et interface mellem View og Model, som en mere traditionel måde at skabe en løs kobling på. Mange af os har sikkert allerede arbejdet med Delphi-forms, hvor behandlingen af data måske er flyttet over i en Delphi-modul, så tanken om en View-Model-arkitektur ligger derfor heller ikke så fjern.



En Model-View-arkitektur behøver derfor ikke at være en overgangsfase – den kunne også være et acceptabelt resultat for en ellers rigid struktur i en ældre klassisk Delphi-applikation. Omvendt er der ikke særlig langt til en egentlig MVP- eller MVVM-arkitektur, hvis vi først har skåret en Delphi-form op i en View og en Model.

Fuldt kvalificerede navne

Det er vigtigt, at vi ikke lader os rive med, når vi flytter en række kodelinjer eller komponenter, og at dette bevirket, at vi vil være afhængige af at flytte tilstødende komponenter og metoder, for at få det til at virke. Det vil forekomme, og det kræver disciplin at holde sig til umiddelbart at skabe fuld kvalificerede (*fully qualified*) referencer først, for at få kodelinjer til at virke, fremfor at flytte komponenter og metoder indtil det kan kompilere. Hvis vi udelukkende flytter komponenter og metoder, for at det skal virke, så risikerer vi også at flytte elementer, som egentlig ikke skulle flyttes, og så skaber vi et endnu større virvar af kode, som vi nemt kan miste overblikket over.

Når vi har besluttet os for at flytte en blok af kode, eller en hel metode, så er det som udgangspunkt kun den blok, vi skal flytte. Der kan være nogle tilhørende variabler eller definitioner, der skal med, og dem bør vi tage med. Denne fremgangsmåde med fuldt kvalificerede navne er ganske simpel, så vi kan begynde med nogle meget simple eksempler.

```

procedure TStopWatchForm.StartButtonClick(Sender: TObject);
begin
  if StopWatchTimer.Enabled then
    StopTimer
  else
    StartTimer;
end;

```

Eksemplet herover er ikke usædvanligt at finde i Delphi, men hvis vi har en tendens til at skrive lange procedurale koder i én sammenhængende click-event, i modsætning til at splitte koden op i metoder, f.eks. i actions som StopTimer og StartTimer, så har vi gjort arbejdet med at separere koden meget mere besværligt for os selv. Som nævnt i anbefalinger i forrige kapitel, så kan vi foretage almindelig refaktorering og modne vores kode forud for en egentlig refaktorering mod et architectural pattern. Det er overvejelser og beslutninger, vi bør tage med som et team, om den enkelte Delphi-form og dens sourcekode er moden til at blive separeret. Om end koden i eksemplet er simpel, så er den også ren i sit udtryk, og det er ikke altid givet, at vi vil være så heldige – især med ældre kode. Men til formålet giver eksemplet et fint overblik.

StartButtonClick-eventet er en klassisk kode, der skal tage stilling til, hvad der skal ske, når brugeren trykker på en startknap. Det er et typisk klik-event, der er bundet til en OnClick-event på en knap-komponent, og ofte bare efterladt i formens unit, fordi der er metoden autogenereret. I sammenhænge med architectural patterns bør metoden formelt høre til Presenter, fordi koden vil tage stilling til, hvad der videre skal ske, og vil på en betingelse kalde en af funktionerne. Om disse funktioner bliver kaldt i View eller Model er ikke væsentligt i dette eksempel, fordi de er rene referencer og kan blive refaktoreret med fuldt kvalificerede navne. Der er ikke andre kodelinjer i metoden, der giver anledning til, at de hører til View eller Model, og derfor er det på den måde også en ren metode, der tager en stilling til en bestemt handling – *Act*.

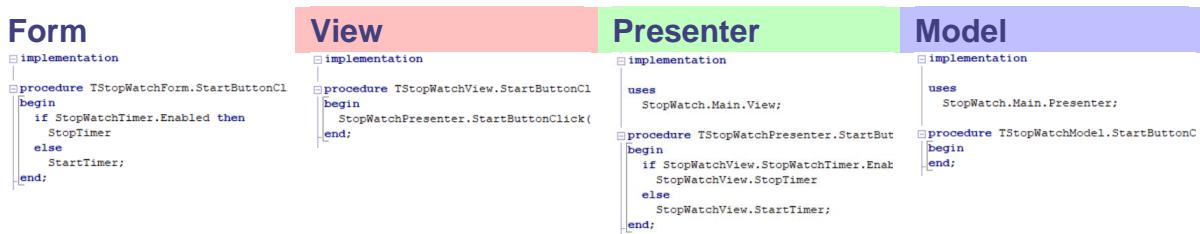
Eftersom metoden er "bundet" i en OnClick event, så har vi to reelle muligheder, men vi vil udelukkende holde os til den ene mulighed efter reglen om, at metoderne som udgangspunkt kopieres til de lag, hvor de flyttes hen, nøjagtig som de er erklæret. Når der ikke er kodelinjer, der skal flyttes til Model, så opretter vi en metode i Presenter, som er fuldstændig identisk med StartButtonClick-metoden. Knappen vil stadig være bundet til metoden i View, men metoden i View vil kalde den samme metode i Presenter. Det er redundant kode, som vi må leve med, når vi har at gøre med værktøj, der ikke understøtter et specifikt architectural pattern.

Det kan ikke rigtig lade sig gøre binde en OnClick-event i View til en metode i Model under design-time, og det gælder for både VCL og FMX. Derfor kan der også være kode, der kun har til opgave at binde især events, viderestille kald osv., men det vil også gøre dem mindre afhængige af frameworks. Hvis vi har et projekt, der udelukkende vil komme til at blive kompileret med enten VCL eller FMX, så er det måske værd at udforske mulighed for at lægge events i actions, fordi de kan bindes som øvrige properties under design-time. Dvs. at actions erklæret i Presenter kan bindes til komponenter i View, ligesom en

datasource i Presenter kan bindes til data-aware-komponenter i View, og fra datasource til datasets i Model osv. Det vil alt andet lige spare os for nogle kodelinjer.

Andre programmeringssprog kan også bruge den mulighed, at de kan erklære namespaces, og i Delphi er den funktion blot en mulighed for at organisere units. Dvs. at vi ikke kan udnytte namespaces til at erklære klasser som værende en del af samme rum, på samme måde som det kan lade sig gøre i andre sprog. Og den sammenhæng er det heller ikke muligt at definere *partial classes* i Delphi. Det ville have gjort en forskel i refaktoringssammenhænge, men i stedet er udgangspunktet er, at metoder bør findes i de lag, hvor kodelinjer skal flyttes hen, og så må vi hjælpe lidt på vej med at nå frem til et architectural pattern, der vil kunne accepteres.

Kodelinjerne i StartButtonClick-metoden vil således kunne flyttes med følgende resultat:



Resultatet er, at kodelinjerne i Presenter foreløbig har fået eksplisitte referencer eller fuldt kvalificerede navne til de metoder i View, som endnu ikke er flyttet. Dvs. at de referencer ikke nødvendigvis er blivende, og at TTimer-komponenten og metoderne StartTimer og StopTimer højest sandsynligt vil ende i Model, så referencer også må ændres på et senere tidspunkt. Men disse midlertidige referencer vil virke, og applikationen vil fungere, som den altid har gjort, og vi har ikke ændret noget som helst i eksekveringen af koden. StartTimer og StopTimer kan ligge i *private* sektionen af formens klasse, og derfor må de løftes op i *public*, ligesom de vil finde tilbage deres rette *access level*, når refaktoreringen nærmere sig sin afslutning.

View vil allerede have en permanent reference til Presenter i sin interface-sektion, hvorfor det kun er uses clause i Presenters implementation, vi allerede har tilføjet, der senere må fjernes, når vi også nærmer os afslutning af refaktoreringen. Men det er væsentligt, at uses clauses i interface-sektionen af den nye Presenter og Model holdes på et minimum, så vi ikke bare kopierer uses clauses fra View.

Adskillelse

Processen som beskrevet indtil videre er simpel, og det er vigtigt med denne skridtvise refaktorering, så vi kan sikre en fungerende applikation til enhver tid. Desværre vil vi ikke altid have helt så simple kodelinjer, som vi bare lige kan flytte. Vi vil kigge på et andet eksempel, som lige akkurat har nogle kodelinjer, vi kan bruge til at belyse næste problemstilling. Vi fortsætter med denne *Stop Watch*-applikation (stopur), der er udviklet som en klassisk Delphi-form, med en label som display og en enkel knap til at starte og stoppe

stopuret, som bare er en TTimer-komponent. Vores ur fungerer som vores data, fordi den giver os tidsdata i form af starttidspunkt og hvor lang tid, der er gået. Eksemplet er tilgængeligt på hjemmesiden. Vi kan fortsætte med at forestille os den foregående simple kodekonstruktion, som vi typisk kunne finde i en klik-event for en start/stop-knap.

Det er vigtigt at fremhæve, at udtrykket som den i betingelsen i eksemplet og de efterfølgende statements let kunne være kategoriseret som Work, men at hele konstruktionen, set i én sammenhæng med en start/stop-knap, må skulle kategoriseres som en Act med en betingelse, der refererer til noget arbejde i vores data. Referencer til noget arbejde, skal ikke nødvendigvis kategoriseres som Work, og er på den måde ikke usædvanlige i kodekonstruktioner, men det er en detalje, der er værd at fremhæve. Der er ikke noget teknisk til hinder for, at konstruktionen kan ligge i alle lag af en pattern, men koden udløses som en reaktion på, at brugeren har trykket på en knap, og konstruktionen skal derfor tage stilling til, hvad der skal ske og hvilke arbejder, der skal udføres, som naturligt vil ligge i den mellemliggende lag af en pattern. Det er ikke ensbetydende med, at alle kodelinjer i konstruktionen skal kategoriseres som Act, og det er selvfølgelig også værd at pointere, at denne handling vil være udløst i det øverste lag.

Hvis vi konkret skal adskille koden i vores næste eksempel og flytte linjerne til de øvrige lag, så tager vi stadig udgangspunkt i, at vi erklærer samme metode i samtlige lag – eller i hvert fald skal tænke os til, at vi i sidste ende kan ende med at have samme metode erklæret i samtlige lag. I og med, at hver af de enkelte kodelinjer kan kategoriseres som enten Show, Act og Work, så har vi en hel metode, der skal flyttes til samtlige lag. Denne situation vil forekomme, og i de fleste tilfælde vil det bare være meget mere opfattende og langt mere komplekst. Dette forhold afslører bare, hvordan en metode i en klassisk Delphi-form, som mange af os vil betragte som ganske normal praksis, kan ende med at være noget så uforenelig med en anerkendt architectural pattern.

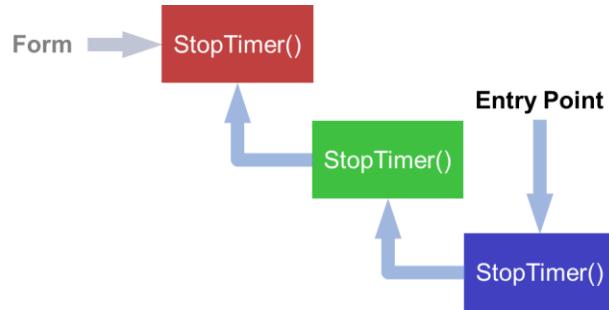
I eksemplet herunder er hver kodelinje farvelagt, så rød repræsenterer Show, grøn er Act, og blå er Work. De farver vil være gennemgående i nogle eksempler, og bruges blot til at illustrere eksemplerne, men det kunne være rart med et værktøj, der kunne benyttes til at kategorisere kodelinjer, for derefter at aktivere en form for automatiseret refaktorering.

```
procedure TStopWatchForm.StopTimer;
begin
  StopWatchTimer.Enabled := False;
  StartButton.Caption := 'Start';
  UpdateDisplay(Now - FTimeStart);
end;
```

Hvis timeren i vores eksempel går ud for at være vores data, så afslører referencen i første linje, at det er noget arbejde, der bør have udspring i Model. Anden linje har en direkte reference til en VCL-komponent, og præsenterer noget for en bruger, så den kan ikke ligge andre steder end i View. I tredje linje er der taget stilling til, at der er et display i View, der skal opdateres, og fordi der er refereret til data, der normalt hentes fra Model, så bliver det Presenters opgave at bringe den videre til View. Her kommer Presenters rolle til sin ret,

fordi den nu vil fungere som mellemled. Det vil være en anden historie med f.eks. MVC, fordi View kan have en reference til Model, og derfor selv kan trække data frem.

Farverne afslører også, at koden ikke udføres fra Show mod Work, men fra Work mod Show, også på trods af, at Show visualiserer et eller andet før Act kalder på en opdatering. Men farverne kan lige så godt afsløre et virvar af blandede kodelinjer, der reelt kan gøre det helt umuligt at flytte metoden hensigtsmæssigt, fordi eksekveringen ikke bare kan gå op og ned gennem lagene gentagende gange i samme metode, uden at vi bryder konceptet bag Refactoring. Derfor kan farverne også afsløre, om vi reelt har fejlet i modningen af sourcekoden gennem almindelig refaktorering før vi satte os til at separere sourcekoden.



Hvis StopTimer-metodens nye udspring, eller entry method for at blive ved de begreber, er i Model, så er det nu, at vi skal have tungen lige i munden, fordi alle kald til StopTimer i View nu skal kalde StopTimer i Model. Det er et usikkert øjeblik, fordi en metode nu kan skygge for en anden metode af samme navn, før vi når at tilføje fuldt kvalificerede navne eller eksplisitte referencer. Den mest sikre fremgangsmåde er at oprette samme metode i Model først, uden overhovedet at flytte koden, og udkommentere den oprindelige StopTimer-metode i View. Applikationen vil ikke kunne kompileres, fordi alle kald til StopTimer i View vil fejle. Derfor kan vi roligt følge op på samtlige kald til StopTimer og rette referencer eksplisit til Model. Når det er klaret, så kan vi fjerne udkommenteringen af StopTimer i View, oprette en tilsvarende metode i Presenter, og herefter kalde StopTimer fra Model til Presenter, og fra Presenter til View. Selvom vi har ændret referencer til samtlige kald til Models StopTimer, så vil applikationen stadig virke, fordi vi endnu ikke har fjernet eller flyttet kode fra den gamle form, som jo bliver liggende i det nye View.

Form	View	Presenter	Model
<pre> procedure TStopWatchForm.StopTimer; begin StopWatchTimer.Enabled := False; StartButton.Caption := 'Start'; UpdateDisplay(Now - FTimeStart); end; </pre>	<pre> procedure TStopWatchView.StopTimer; begin StopWatchTimer.Enabled := False; StartButton.Caption := 'Start'; UpdateDisplay(Now - FTimeStart); end; </pre>	<pre> procedure TStopWatchPresenter.StopTime begin StopWatchView.StopTimer; end; </pre>	<pre> procedure TStopWatchModel.StopTimer; begin StopWatchPresenter.StopTimer; end; </pre>

Det er måske værd at fremhæve, at navne på de samme metoder ikke nødvendigvis kommer til at hedde de samme i slutresultatet, men at dette er en overgang i refaktoreringen, og at dette fastholdes så længe koderne er ved at blive flyttet.

Den næste udfordring er at flytte de enkelte kodelinjer til de rigtige lag, som vi gennemgik før. Og eftersom vi ved, at entry point er Model, så skal de enkelte kodelinjer flyttes, så de bliver eksekveret i den samme rækkefølge, som de vil være eksekveret i den oprindelige kode. Dvs. at de enkelte kodelinjer kommer foran eller efter de respektive kald til StopTimer, som afbilledet herunder. Når de enkelte kodelinjer er flyttet, så vil applikationen stadig fungere, som den altid har gjort, fordi koden ikke er ændret og rækkefølgen af eksekveringen heller ikke er ændret. Der er kommet et par kald, men det ændrer ikke på eksekveringen – kodelinjerne er bare flyttet.

Form	View	Presenter	Model
<pre>procedure TStopWatchForm.StopTimer; begin 1) StopWatchTimer.Enabled := False; 2) StartButton.Caption := 'Start'; 3) UpdateDisplay(Now - FTimeStart); end;</pre>	<pre>procedure TStopWatchView.StopTimer; begin 2) StartButton.Caption := 'Start'; end;</pre>	<pre>procedure TStopWatchPresenter.StopTime begin StopWatchView.StopTimer; 3) StopWatchView.UpdateDisplay(Now - St end;</pre>	<pre>procedure TStopWatchModel.StopTimer; begin 1) StopWatchTimer.Enabled := False; StopWatchPresenter.StopTimer; end;</pre>

Dette kan lade sig gøre, fordi Model har en midlertidig reference til Presenter i implementationssektionen. Presenter har ligeledes en reference til View i implementationssektionen, og kan kalde samme metode i View. Udfordringen er altså, at kodelinjerne i de forskellige lag skal eksekveres i nøjagtig samme rækkefølge som var de i den oprindelige metode, hvor de på billedet herover er fremhævet med et tal.

Bemærk, at 3) kodelinje har en reference til en field-variabel FTimeStart, som Presenter nødvendigvis må have adgang til i View. Variablen vil højest sandsynligt høre under Model, og hvis den endnu ikke er flyttet, fordi den må flyttes på et andet tidspunkt, så har vi to muligheder, men at vi måske bare vil holde os til midlertidigt at løfte variablen op på en access level public, fordi vi ikke vil tilføje yderligere kode. Den anden mulighed er at oprette en public property med *read* og *write* direkte til variablen – så har vi heller ikke tilføjet kode. Muligheden med en property bør ikke udelukkes, hvis variablen f.eks. skal flyttes til Model, og vi stadig gerne bevare variablen som en private field-variabel. I så fald flytter vi bare property-erklæringen med til Model.

Når vi har separeret kodelinjerne, så bliver det også mere iøjnefaldende, at det kun er StopTimer i Model, der udfører selve arbejdet med at stoppe timeren. De øvrige metoder er snarere metoder til at fange en notifikation om, at timeren er blevet stoppet. Dvs. at metoderne i View og Presenter med rette bør hedde TimerStopped, der nemt kan ændres med Delphis refaktoringsfunktion. At ændre navnet på en metode, der indikerer en handling, f.eks. StopTimer, til en metode, der modtager en notifikation, bør også følge en regel. Navnet skal læne sig op ad de sammensatte ord, der udgør den oprindelige metode. Hvis metoden hedder StopTimer, så bør metoderne til notifikationen om, at timeren er stoppet, simpelthen bare hedde TimerStopped. På samme måde med StartTimer og TimerStarted. Denne regel er vigtig, ikke kun fordi navnene bliver kædet sammen, men fordi disse navne senere hen også vil udgøre navne på metoder til definering af et interface. Er der gamle metoder, der optager disse navne, så er det en opgave under modning og almindelig refaktorering før vi begynder at flytte kode.

View	Presenter	Model
<pre> procedure TStopWatchView.TimerStopped; begin StartButton.Caption := 'Start'; end; </pre>	<pre> procedure TStopWatchPresenter.TimerStopped; begin StopWatchView.TimerStopped; StopWatchView.UpdateDisplay(Now - StopWatchModel.Ti end; </pre>	<pre> procedure TStopWatchModel.StopTimer; begin StopWatchTimer.Enabled := False; StopWatchPresenter.TimerStopped; end; </pre>

Selv efter disse ændringer har vi ikke ændret noget i eksekveringen af koden, og vi har overholdt konceptet med refaktorering, og separerer en metode, så den kommer tættere på en MVP eller MVVM, hvis vi ser bort fra de midlertidige referencer. Kodelinjerne ligger nu i de lag, hvor de bør ligge, uanset om vi taler om MVP eller MVVM, eller med god vilje også MVC, og det er kun kommunikationen mellem lagene vi mangler at løfte op på et niveau, som kan leve op til et af disse architectural patterns. Det er klart, at der nu er tre metoder, fremfor én metode, der skal eksekveres, og det skaber selvfølgelig noget overhead jo mindre metoderne er, men det er prisen for Separation of Concerns, og til gengæld vil vi vinde på andre faktorer.

Definering af et interface

Når samtlige kodelinjer er flyttet, og de nye metoder har fundet deres pladser, så mangler vi bare at fjerne de midlertidige referencer i implementationssektionen, dvs. fra Model til Presenter, og fra Presenter til View. Det kan først lade sigøre, når vi har defineret en kommunikationsform, så Model kan kalde Presenter, og Presenter kan kalde View. Den nemmeste og mest åbenlyse tilgang er interfaces, som kan implementeres uden at vi behøver at ændre i koden. Og definering af hvilke metoder, der skal med i disse interfaces afslører sig selv ved de eksplisitte referencer eller fuldt kvalificerede navne som i **StopWatchPresenter.TimerStopped** i Model og **StopWatchView.TimerStopped** og **StopWatchView.UpdateDisplay** i Presenter.

Det er alle de referencer, som vil ophøre med at virke, når vi fjerner referencer til View og Presenter i implementationssektionen af Presenter og Model. Delphi-værktøjet vil helt automatisk fremhæve de referencer, der nu ikke længere er gyldige, når referencerne til de units er væk. Derfor giver det helt sig selv, at de metoder, der refereres til, optræder i de nye definitioner af interfaces, som View skal have implementeret og som Presenter har defineret, og som Presenter skal have implementeret og som Model har defineret. Det mesterlige er, at metoderne allerede er implementeret i View og Presenter – det er kun definitionerne af interfaces, der mangler.

I vores eksempler vil interfaces være ganske simple, men i bestående projekter kan sådanne interfaces være omfattende, og også her vil det afsløre om projektet, eller den ene form, er moden til at blive refaktoreret mod et pattern, eller om der er brug for at foretage almindelig refaktorering, så en separering kan foretages mere hensigtsmæssigt.

Når vi har defineret et interface i Presenter, som View skal implementere, så er det nok at tilføje interfacet til erklæringen af View – metoderne er der jo allerede. Det samme gør vi med definering af et interface i Model, som Presenter skal implementere – det er nok at tilføje interface til erklæringen af Presenter, fordi metoder er der allerede. Vi har stadig

ikke ændret noget i koden. Derefter mangler vi bare at få referencerne eller de fuldt kvalificerede navne til at kompilere igen.

På afbildningen herunder skal vi notere os, at der nu er erklæret en klassevariabel for henholdsvis View i Presenter-klassen og for Presenter i Model-klassen. De klassevariabler vil ganske enkelt få samtlige referencer til View og Presenter til at blive fuldgylde referencer igen, stadig uden at vi har ændret noget i koden, men klassevariablerne mangler at blive initialiserede. Dvs. at når View bliver oprettet, så skal den under sin konstruktion (FormCreate) som det første sætte klassevariablen i Presenter til sig selv. Og når Presenter bliver oprettet, så skal den også under sin konstruktion (DataModuleCreate) som det første sætte klassevariablen i Model til sig selv. Herefter vil samtlige referencer virke som de altid har virket. Det er denne lille detalje, som adskiller Delphi fra andre værktøjer, som understøtter et architectural pattern, mens Delphis projekt-fil er autogenereret til at oprette forms på sin egen klassiske måde.

View	Presenter	Model
<pre>TStopWatchView = class(TForm, IStopWatchView) private procedure TimerStopped; procedure UpdateDisplay(const ATimeElapsed: string); end;</pre>	<pre>IStopWatchView = interface['{342F3A0C-ABDC-4DE6-85AD-F5 procedure TimerStopped; procedure UpdateDisplay(const ATimeElapsed: string); end; TStopWatchPresenter = class(TDataModule, IStopWatchPres public procedure TimerStopped; class var StopWatchView: IStopWatchView; end;</pre>	<pre>ISTopWatchPresenter = interface['{498325A6-E9D7-45D6-94 procedure TimerStopped; end; TStopWatchModel = class(TDataModule) public procedure StopTimer; class var StopWatchPresenter: IStopWatchPresenter; end;</pre>

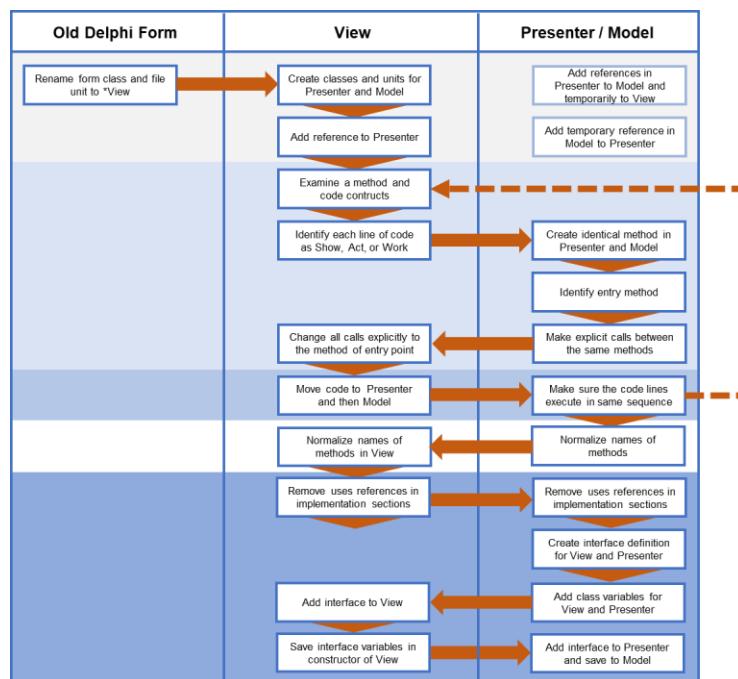
På dette tidspunkt har vi nu tilføjet et par defineringer af interfaces, vi har tilføjet et par klassevariabler, og vi har initialiseret dem i de respektive lag, og vi har stadig ikke ændret på noget kode, så det vil eksekvere som det altid har gjort, og vi har opnået målet om Separation of Concerns. Med et interface kender Presenter ikke til noget konkret View – og med interface-variablen, så kan View praktisk taget være hvad som helst, f.eks. en klasse til unit tests. Model kender heller ikke noget til Presenter, som, bortset fra interfacet, kunne være hvad som helst, f.eks. helt anden Presenter og View.

Vi kan følge denne tilgang for samtlige Delphi-forms, vi har i en hel applikation, og selvom det overhovedet ikke er en opgave, der er lige til at gå til, så kan det blive et resultat, der virkelig kan gøre en forskel. Vi vil i de næste kapitler tage mindre Delphi-projekter og refaktorere dem på samme måde som beskrevet i dette kapitel. Den mere detaljerede trin-for-trin beskrivelse vil blive fulgt op med Delphi-projekter, der er tilgængelige på hjemmesiden, så det er muligt at give sig i kast med at refaktorere selv. Hvis vi står uden erfaring med denne form for refaktorering, så kan det sagtens betale sig at forsøge sig med selv små projekter, fordi vi får en rigtig god fornemmelse for, hvad det indebærer af udfordringer ved separering en Delphi-form i tre filer, og hvilke typer af kodelinjer, der skal flyttes til de forskellige lag.

Opsummering af Saw, Act, og Work

dfadsf

- **Show** – denne type af kode handler om at vise noget på skærmen, og som for de nævnte patterns vil betyde View. I Delphi vil kode, der er i berøring med visuelle komponenter fra VCL eller FMX helt klart høre i denne kategori, ligesom Canvas og fonte, især fordi de refererer til VCL-klasser. View vil på den måde være den eneste del af en pattern, der må referere direkte til VCL- eller FMX-biblioteker.
- **Act** – moderne platforme er alle event-baserede operativsystemer, og det præger naturligvis de fleste udviklingsværktøjer, hvorfor vi vil se mange event-metoder. For rigtig mange kodekonstruktioner handler det meget om at reagere på en hændelse og tage stilling til, hvad der skal ske, og aktivere en handling på det, eller en *action*, og på den måde kontrollere eksekveringen. Alle kodelinjer i en kodeblok, der er udløst af en event, er ikke nødvendigvis en type af kode, der hører til denne kategori, hvorfor de ofte også skal skæres til. Denne type af kode vil typisk skulle flyttes til en Controller, Presenter, eller ViewModel m.m., og bør f.eks. ikke have referencer til VCL eller FMX.
- **Work** – denne type af kode har direkte berøring med manipulering af data, eller udfører forretningsrelaterede funktioner. Det er kernen i applikationen, der også vil omfatte domænespecifikke funktioner, og vil ofte have referencer til data og databaser. I det nævnte patterns vil det være Model, og bør f.eks. heller have referencer til VCL eller FMX.



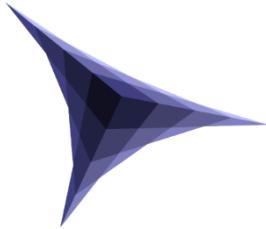
Regler:

- 1) Separering skal altid ske i faser, så det altid er muligt at kompilere applikationen, at applikationen kører som det altid har kørt.

- 2) Separering skal altid ske efter konceptet Refactoring og efter princippet om én-til-én.
- 3) Refaktorering af *identifiers* (navne) må dog ikke ske før separeringen er afsluttet. Dvs. refaktoreringen kan fortsætte uover separeringen, så navnene efterfølgende kan give bedre mening under den nye arkitektur.
- 4) Refaktorering af kodekonstruktioner må heller ikke ske før separeringen er afsluttet. Dvs. ændring af kode indenfor rammerne af Refactoring må først ske, når det nye arkitektur er på plads.
- 5) Hvis én kodelinje består af flere udtryk, der naturligt hører til hver deres lag, så skal det foreløbig flyttes til det lag, der er øverst. Udskillelse af en kodelinje i flere må ske efter separeringen.
- 6) Hvis en flytning kræver midlertidige referencer, der kan sikre en flytning i flere faser, så er det helt i orden.

Baggrunden for, at vi ikke må refaktorere kodekonstruktioner og identifiers, mens vi flytter koden, er, at flytning af kode fra en unit til en anden er et voldsomt indgreb, når vi taler om flere tusinde kodelinjer. Det kan skabe et lige så voldsom en regression, hvis en kodekonstruktion er flyttet og refaktoreret, men at det på et senere tidspunkt viser sig, at det må flyttes tilbage eller i det tredje unit af en eller andet årsag. Koden skal være i stand til at bevare sin oprindelige tilstand og evne til at eksekvere koden, som den har gjort hele tiden, og vi skal bevare friheden til at flytte koden frem og tilbage efter behov.

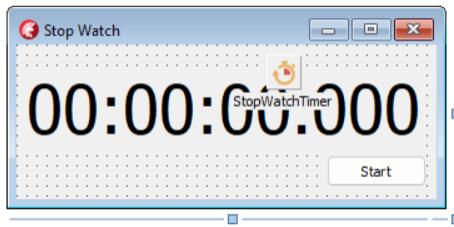
At refaktorere identifiers og kodekonstruktioner øger risikoen for, at vi mister genkendelige pejlemærker i koden, mnemoteknik og gamle patterns om vi vil, når vi skal sikre os, at koden er intakt – også bl.a. ved at sammenligne med det oprindelige udgangspunkt.



3 Kapitel

Refaktorering

Et konkret arbejde med refaktorering ud fra det beskrevne koncept kan bedst fortælles med refaktorering af nogle simple projekter. Det første eksempel vil ikke være et større program, men et lille program, der kan vise separering af en klassisk Delphi-applikation med en form og kode i samme unit til nye units, og vise hvor eksisterende kode flyttes til i et af disse units, uden at koden teknisk set ændres. Dette kan desuden gøres i faser, sådan at der ikke lukkes ned for løbende leveringer af nye versioner. Det første eksempel vil blive fulgt op med andre eksempler, der har fokus på klassiske løsninger på Delphi-applikationer, f.eks. med forhold til databaser og nogle data-aware komponenter.



Det første eksempel er et program med et simpelt stopur, som ikke består af andet end en enkelt start/stop funktion, men lige akkurat har tilstrækkelig kode nok til, at vi kan beskrive hvordan koden kan flyttes, og at det virker nøjagtig som det skal, selvom programmet bagefter vil bestå af 3 filer i stedet for den ene. Samtidig vil processen vise de enkelte faser, og at vi til enhver tid kan stoppe processen og stille os tilfredse med resultatet, fordi det stadig vil virke. Og skulle vi støde på problemer, så er den sidste fase aldrig større, end at vi kan rulle ændringerne tilbage og fortsætte, når vi har fundet en løsning.

Programmet består udelukkende af en label som et display, en knap, og en timer. Og selvom programmet kun består af 5 metoder og 30 reelle kodelinjer, så giver det en forsmag på de udfordringer, som vi står overfor, når vi skal bestemme hvor de enkelte kodelinjer hører til i et af de tidligere beskrevne patterns. Og selvom den største metode består af en kodeblok på 4 simple linjer, så skal de fordeles i alle 3 lag. Og samtidig vil vi vil også fastholde reglen om altid at refaktorere mod MVP pattern først, selvom vores endelige mål skulle være et produkt, som skal være så tæt på MVVM som muligt.

```

unit StopWatchMain;
interface
uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls, Vcl.ExtCtrls;
type
  TStopWatchForm = class(TForm)
    StopWatchDisplay: TLabel;
    StopWatchTimer: TTimer;
    StartButton: TButton;
    procedure StopWatchTimerTimer(Sender: TObject);
    procedure StartButtonClick(Sender: TObject);
  private
    FTimeStart: TDateTime;
    procedure StartTimer;
    procedure StopTimer;
    procedure UpdateDisplay(const ATimeElapsed: TDateTime);
  public
  end;
var
  StopWatchForm: TStopWatchForm;
implementation ...

```

Indledende skridt

Det allerførste skridt er helt klart at sikre sig, at vi har en backup, et rent udtræk fra en *repository*, eller en *branch* vi kan arbejde på. Ved hvert skridt, eller delændringer ved større projekter, skal applikationen ved hvert *commit* stadig kunne kompileres og virke som den altid har gjort. Referencer i unittest eller testværktøjer skal måske efterses, men eksterne tests af applikationen bør stort set bestå som de er sat op til.

Hvis det skulle vise sig, at et stykke refaktorering volder problemer, så må det aldrig blive et større problem end, at vi kan initiere en *revert* af ændringerne og begynde forfra fra før ændringerne, skabe os et nyt overblik, og måske iværksætte en refaktorering fra en anden vinkel eller foretage ændringerne i mindre dele. Det er alfa og omega, at vi så vidt muligt overholder principippet om at foretage refaktorering uden at ændre funktionalitet af en kodekonstruktion. Vi kan omdøbe, flytte og splitte en kodekonstruktion op i flere konstruktioner, uden at vi reelt ændrer funktionaliteten.

Det kan lade sigøre at retfærdiggøre en radikal ændring af en kodekonstruktion, så der ikke længere alene er tale om reel refaktorering, men det bør udelades fordi en refaktore-ring mod en architectural pattern som MVC, MVP, MVVM osv., skaber så store struktu-relle ændringer i en applikation, at decidederede omskrivninger af en funktionalitet kan skabe endnu større forvirringer – især hvis der senere viser sig problemer med funktionaliteten og resultaterne af tests. Hvis der er funktionalitet, der skaber problemer med at nå målene med refaktoreringen, så opret en opgave og på den måde gennemtvinge et særligt fokus omkring problemet. Ellers bør opmærksomhed omkring uhensigtsmæssigheder i funktionalitet af kode noteres som *issues* og håndteres på et senere tidspunkt. Også hvis det betyder, at noget kode ikke kan flyttes, før der er fundet en løsning.

Første fase

Den første udfordring er navnet på formens klasse, TStopWatchForm, og variablen, som bør omdøbes til mere passende TStopWatchView og StopwatchView. I vores tilfælde er det ligetil og ændres uden problemer gennem Object Inspector, som ændrer både navn for klassen og variablen, og eventuelle referencer i projekt-filen. I mange tilfælde kan det virke som et voldsomt indgreb i koden, hvis der er referencer på tværs af flere units, men ikke værre end, at en søgning og en oversigt over referencerne kan sikre, at ændring af navnene slår igennem. I andre tilfælde kan det også have indflydelse på både unit tests og eksterne testværktøjer, der er blevet bygget op omkring systemet, der dog i bedste fald blot skal opdateres løbende. Dernæst bør hele unit gemmes i et nyt filnavn, f.eks. fra StopWatchMain.pas til Stopwatch.Main.View.pas, fordi der vil blive skabt rum til nye units, der gerne skal omfatte både Presenter og Model. Samtidig kan vi honorere anbefalingen om at bruge *scoped* filnavne. Det er ikke et krav, men god praksis.

StopWatchMain.pas kan godt gemmes med ny filnavn gennem "Files / Save As...", og Delphi vil forsøge at rette samtlige referencer til unit-filen i hele projektet. Ved større projekter, og især med mange compiler-direktiver, er det ikke altid at dette lykkes, og det er derfor vigtigt, at der ikke er referencer til den gamle unit-fil. Den gamle fil skal simpelthen bare slettes, og dette vil også afsløre referencer, som ikke er rettet. Hvis vi bruger et versioneringssystem, så er det også vigtigt, at filen slettes gennem dette værktøj, ligesom nye filer skal tilføjes versioneringssystemet. Hvis vi kan bevare historikken for den gamle form-fil, så er det også værd at undersøge, om vores versioneringsværktøj understøtter omdøbning af filer, og på den måde bevare hele historikken. I de tilfælde skal vi ikke gemme filen ved hjælp af Delphi, men omdøbe filnavnet med versioneringsværktøjet, og herefter manuelt tilrette navne på units og referencer i filerne. Det er muligt at lave forsøg med midlertidige filer og se, om versioneringsværktøjer understøtter ændring af filnavn.

Det kan virke banalt at navne på både form og fil bør ændres, men det er netop et hovedpunkt i patterns, at der indgår gennemgående og genkendelige navne som View, Presenter og Model i sammensatte navne for både forms og filer, som vil henlede udviklernes opmærksomhed omkring et bestemt pattern – de vil genkende MVP i dette tilfælde. Når navnet på form og fil er ændret, så skal programmet kunne kompileres og eksekveres som vanligt, fordi vi ikke har ændret noget i koden. Applikationen skal køre som den altid har gjort, og det er vigtigt dette bliver testet for hver skridt, vi tager. Eventuelle referencer til formen i testværktøjer eller unit-tests vil således også skulle opdateres.

Det er en regel ved samtlige skridt i refaktoreringen, at ændringerne altid skal kunne kompileres og at applikationen vil virke som altid.

Næste skridt

Næste skridt er at oprette de to units, som et View nødvendigvis må være i følgeskab med, og det betyder at vi vil ende med at have følgende units for hver form vi har efter

refaktorering. Filerne vil repræsentere den fysiske adskillelse af de enkelte dele af et pattern, ligesom de vil omfatte referencer til hinanden, som er defineret af dette pattern:

- 1) StopWatch.Main.View.pas
- 2) StopWatch.Main.Presenter.pas
- 3) StopWatch.Main.Model.pas

Referencerne i filerne vil tilpasse sig i overgangen til et pattern, fordi adskillelse af kodelinjer til de respektive units står øverst som udfordring, og så vil fuldt kvalificerede navne også tilpasses bagefter. De nye units skal oprettes som TDataModule, og kan findes under "File / New / Other" og i "Delphi / Database". Vi skal ikke lade os forvirre af, at TDataModule er placeret under en Database-gruppe, fordi modulet som klasse ikke tager databaseafhængigheder med sig – der er ingen referencer til database-komponenter fra modulet, så de bliver ikke inkluderet i applikationen under kompilering. Tværtom har en TDataModule kun referencer til System.SysUtils og System.Classes, fordi det er en TComponent, der i dag er tilpasset til at håndtere non-visual komponenter på samme måde som på en form, samtidig med at en modul vil kunne oprettes på samme måde som en form i projekt-filen – bare uden Windows-informationer og grafik som for en form. TDataModule er perfekt som basisobjekt til de øvrige units i en pattern, fordi vi kan flytte non-visual komponenter fra formen som de er.

Modulerne oprettes for henholdsvis Presenter og Model, ligesom klasserne i modulerne bør ændres til henholdsvis TStopWatchPresenter og TStopWatchModel, sammen med variablerne til disse klasser, på samme måde som de er sammensat i filnavne som vist foroven. Samtidig vil vi i eksemplet lade Delphi håndtere oprettelsen af de enkelte forms og moduler med CreateForm, selvom instantiering af f.eks. en Presenter normalt vil blive gjort med en View som argument til en Presenters *constructor*. Bemærk rækkefølgen af hvornår delene oprettes, idet View er entry point, som også kan bestemme hvilken Presenter, der skal oprettes, som igen bestemmer hvilken Model, der skal arbejdes med.

```
program Stopwatch;
uses
  Vcl.Forms,
  Stopwatch.Main.View in 'Stopwatch.Main.View.pas' {StopwatchView},
  Stopwatch.Main.Presenter in 'Stopwatch.Main.Presenter.pas' {StopwatchPresent
Stopwatch.Main.Model in 'Stopwatch.Main.Model.pas' {StopwatchModel: TDataMod

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TStopwatchView, StopwatchView);
  Application.CreateForm(TStopwatchPresenter, StopwatchPresenter);
  Application.CreateForm(TStopwatchModel, StopwatchModel);
  Application.Run;
end.
```

Det er kun constructor i de enkelte klasser, der bliver eksekveret, og der da ikke må foregå forretningslogik i en constructor udover initialisering, så bør der ikke være noget problem i rækkefølgen. Viser det sig, at der allerede opstår *dependency* til de øvrige units i formens

constructor, så er det *code smell*, der vil forhindre os i at implementere patterns – det er ikke bare et princip i f.eks. SOLID, men en constructors rolle er beskrevet i paradigmet for *Object Oriented Programming*. I eksemplet er vi tilfredse med ovenstående konstruktion, men i mange situationer vil oprettelsen af Views, Presenters, og Models foregå lidt mere dynamisk og ikke kontrolleret af Delphi, og samtidig være bestemt af interaktioner med brugere og øvrige funktioner og data i en applikation. I et af senere eksempler vil vi beskrive en lidt mere retvisende tilgang til MVP end projektfilen giver os mulighed for.

I vores præliminære øvelser vil referencerne ikke være i overensstemmelse med MVP, men være direkte referencer mellem units, dvs. stadig med en stærk kobling. View vil have en reference til Presenter i interface-sektionen, Presenter vil have en reference til Model i interface-sektionen og en reference til View i implementationssektionen, og Model vil have en reference til Presenter i implementationssektionen. Referencerne respekterer idéen med at adskille delene fra hinanden som i MVP, men at referencerne i implementationssektionerne senere må fjernes for at opnå en løs kobling mellem delene, som vil være det endelige mål. Det er muligt, at der kan opstå situationer, hvor vi midlertidigt må have referencer i implementationen af Model til View, eller af View til Model, men refaktoreringen vil sigte efter direkte referencer mellem lagene og ikke springe det mellemliggende lag over. Presenter vil f.eks. foreløbigt have følgende referencer:

```
unit Stopwatch.Main.Presenter;
interface
  uses
    System.SysUtils, System.Classes, Stopwatch.Main.Model;
implementation
  uses
    Stopwatch.Main.View;
end.
```

Units med TDataModule-klassen i nyere Delphi-versioner oprettes altid med Class-Group-egenskab, som ligner en kompiler-direktiv i implementationssektionen, men er tiltænkt Delphi IDE-værktøjet, så det kan tillade om vi må smide VCL- eller FMX-komponenter på datamodulet. Direktivet er udeladt i det første eksempel, fordi vi vil udelade at flytte platform- eller framework-specifikke komponenter til Presenter og Model:

```
{%CLASSGROUP 'Vcl.Controls.TControl'}
```

Dette er selvfølgelig et aspekt ved separeringen, at vi også har fokus på, hvad vi har af komponenter på de forskellige lag, hvis vi har et multiplatformprojekt.

Anden fase

Overgangen med de midlertidige referencer gør, at det bliver muligt og mere overskueligt at fokusere på, hvilke koder, der skal flyttes hvorhen, samtidig med at vi vil få skabt nogle aftryk af de interfaces, som View og Presenter nødvendigvis må implementere, for at vi

kan skabe en mere løs kobling mellem alle tre lag. De midlertidige referencer i implementationssektionen skal først fjernes i de afsluttende faser af refaktoreringen. Først er vi dog nødt til at få et overblik over den eksisterende kode, og få et blik i, hvilke dele, der skal flyttes hvorhen. Udeover erklæringen af formen, som vi tidligere har skildret, så bestod den oprindelige forms unit af følgende implementation:

```

procedure TStopWatchForm.StartButtonClick(Sender: TObject);
begin
  if StopWatchTimer.Enabled then
    StopTimer
  else
    StartTimer;
end;

procedure TStopWatchForm.StartTimer;
begin
  FTimeStart := Now;
  StopWatchTimer.Enabled := True;
  StartButton.Caption := 'Stop';
  UpdateDisplay(0);
end;

procedure TStopWatchForm.StopTimer;
begin
  StopWatchTimer.Enabled := False;
  StartButton.Caption := 'Start';
  UpdateDisplay(Now - FTimeStart);
end;

procedure TStopWatchForm.StopWatchTimerTimer(Sender: TObject);
begin
  UpdateDisplay(Now - FTimeStart);
end;

procedure TStopWatchForm.UpdateDisplay(const ATimeElapsed: TDateTime);
var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.fff', ATimeElapsed);
  StopwatchDisplay.Caption := LTimeElapsed;
end;

```

Vi kan tænke mellem helt form-afgrænsede kode i View, handlingsspecifikke metoder i Presenter, og data- og forretningsrelaterede metoder i Model. Vi skal først og fremmest lægge mærke til, at View vil omfatte en masse referencer til VCL- eller FMX-units, f.eks. Vcl.StdCtrls, som ikke må flyttes med over i Presenter. Det er en skillelinje, som vi ikke bør bryde, og selvom vi ofte vil høre om MVP med en passiv View og Presenter, der har ansvaret for at opdatere View, så vil vi absolut bevare form- eller platformspecifikke kode i View, særligt når vi taler om klassiske Delphi-applikationer.

På billedet herover har vi med farver markeret de forskellige typer af kode, vi kan finde i den gamle form-unit, som bør blive eller flyttes til et af de nye units – en blå markering betyder View, dvs. at kodelinjen bør blive, rød kodelinje bør flyttes til Presenter, og grøn til Model. Lige præcis denne øvelse er den mest udfordrende og bør på ingen måde betragtes som værende lige til. Det vil stille og roligt begynde at give mere og mere mening efterhånden som koderlinjerne bliver udskilt og flyttet. Og med erfaringen vil det også blive nemmere at pege de kodelinjer ud, der mest naturligt vil høre til Presenter eller Model. I vores lille Stopwatch-applikation er stopuret (en TTimer-komponent) vores data, og alle direkte referencer til denne komponent vil formodes at høre naturligt til Model.

Vores display er også blot en label, og alle direkte referencer til denne komponent vil mest naturligt høre til View. Vores View vil være meget insisterende, fordi vi netop vil holde vores referencer til bl.a. VCL i denne unit, hvilket vil sige, at nogle koder slet ikke kan flyttes, uanset hvor meget vi end måtte synes, at de bør høre til Presenter eller Model. F.eks. bør vi ikke kunne opdatere en labels caption fra Presenter, selvom Presenters opgave netop er at holde View opdateret med data, der f.eks. er behørigt formateret. Det vil kræve, at Presenter kender til Vcl.StdCtrls, og det er vi ikke interesseret i – en Caption vil derfor opdateres på anden vis.

Kodelinjer med røde markeringer bør ligne kodelinjer, der er mere handlingspræget, eller med andre ord *actions*. Med de briller kan vi begynde at følge en tilgang, som vi har beskrevet som SAW (Show, Action, Work), til at skære koder i de tre kategorier, og fordi udfordringen er jo tydeliggjort af farvemarkeringerne. Kode, der bør flyttes til de forskellige lag, kan sagtens optræde i samme kodekonstruktion i én metode, som det fint fremgår på det sidste billede. Denne problemstilling kan løses med en tilgang, som vi kommer til at beskrive efterfølgende. Med erfaring vil tilgangen være en nem måde at flytte kode frem og tilbage på, som også afhænger af at vi efterhånden skal frem til en konklusion om, hvor de enkelte kodelinjer reelt hører hjemme i en MVP-pattern.

Tilgangen er simpel – når vi har udset os, som oftest, en enkel metode, men også en gruppe af relaterede metoder, så opretter vi nøjagtig den samme eller de samme metoder i Presenter, som de er erklæret i View. En enkel metode skaber helt klart det bedst mulige udgangspunkt, et bedre overblik og mindst mulig forvirring. Det er vigtigt, at vi starter med at flytte metoder, der skal flyttes til Presenter, fordi vi i så fald vil undgå at gøre brug for midlertidige reference fra View og helt til Model. Derfor er det også en god tommelfingerregel at begynde med de metoder, der udelukkende består af kodelinjer, der skal flyttes til Presenter, og ikke blande kodelinjer.

Hvis vi starter med at flytte den første kodekonstruktion, der er markeret med rødt i det forrige billede, til en ny StartButtonClick-metode i Presenter, så kommer de første tilføjelser af eksplisitte referencer til elementer i View. Når vi har kopieret en erklæring af en metode i View til public-sektionen af TStopWatchPresenter, så kan vi blot trykke på Ctrl+Shift+C ved en af metoderne, og Delphi vil oprette en implementation af metoden, og vi kan kopiere kodekonstruktionen over:

```
procedure TStopWatchPresenter.StartButtonClick(Sender: TObject);
begin
  if StopwatchView.StopWatchTimer.Enabled then
    StopwatchView.StopTimer
  else
    StopwatchView.StartTimer;
end;
```

Denne kodekonstruktion kalder yderligere to metoder, og uanset om disse metoder på et tidspunkt også ender med at blive flyttet til Presenter, så holder vi fast i, at vi afslutter flytning af denne ene metode først. Dvs. at de to private metoder, StartTimer og StopTimer, løftes til public-sektionen i View, så de i første omgang kan kaldes direkte fra

Presenter. Det er en god idé at udskille løftet med en særskilt kommentar, så det fremgår at løftet er med fuldt overlæg, men også at der kan være en bemærkning om, at metoderne bør sænkes til private på et senere tidspunkt. Kaldene til disse metoder ændres til fuldt kvalificerede navne, eller eksplisitte referencer, i Presenter, samt at kodeblokken i samme metode i View ændres til at kalde den samme metode i Presenter. Applikationen kan herefter kompileres og køres igen uden problemer.

```
procedure TStopWatchView.StartButtonClick(Sender: TObject);
begin
  StopWatchPresenter.StartButtonClick(Sender);
end;
```

Det, vi skal bide mærke i, er, at vi stadig ikke har ændret noget kode – vi har "bare" flyttet lidt rundt på den samme kode og brugt eksplisitte referencer, eller fuldt kvalificerede navne. Den mest "alvorlige" forseelse, vi har foretaget, er, at vi midlertidigt har flyttet et par metoder fra private-sektionen til public-sektionen. Og selvom de ikke ændrer funktionaliteten af koden, så er det et kritisk indgreb, fordi et sådant løft kan skygge metoder af samme navn på tværs af units – derfor er det også god idé at dobbeltsikre os, at der ikke er sammenfaldende navne i vores scope.

Eftersom vi, så vidt dette er muligt, starter med at flytte kodelinjer, der hører til Presenter, så fortsætter vi med at fokusere på de metoder først. Den næste metode er StopWatchTimer, som blot har én kodelinje, der naturligt hører til Presenter. Derfor opretter vi samme metode i Presenter, og flytter den ene kodelinje, samtidig med at den gamle metode kalder den nye metode eksplisit, ligesom vi også må løfte UpdateDisplay-metoden fra private til public. Den ene kodelinje introducerer dog en lille detalje, fordi den tilgår en field-variabel, som ligger i private, men som den stadig skal kunne tilgå. Heldigvis kan vi tilføje et property, som tilgår denne variabel direkte, uden at vi behøver at flytte den field-variabel i public-sectionen.

```
private
  FTimeStart: TDateTime;
{ Elevated from private }
public
  procedure StartTimer;
  procedure StopTimer;
  procedure UpdateDisplay(const ATimeElapsed: TDateTime);
  property TimeStart: TDateTime read FTimeStart;
end;
```

En property med en direkte reference til field-variablen ændrer ikke funktionalitet, eller giver overhead i koden, og er derfor ideelt i denne henseende. Den ene kodelinje vil derfor have et par eksplisitte referencer, men funktionaliteten har ikke ændret sig. Det er heller ikke sikkert, at variablen FTimeStart skal blive i View, og skal den flyttes på et tidspunkt, så er det blot referencen, der skal opdateres.

```
procedure TStopWatchPresenter.StopWatchTimerTimer(Sender: TObject);
begin
  StopWatchView.UpdateDisplay(Now - StopWatchView.TimeStart);
end;
```

Disse ændringer er gemt under projektet "StopWatch - 3", men det var også metoder, der var nemme at flytte, fordi hele kodeblokke kunne flyttes til Presenter. Det næste skridt er straks værre, fordi vi nu må forholde os til kodekonstruktioner, der er blandede. En god tommelfingerregel er også at fortsætte med de metoder, der starter med en kodelinje, der hører til Presenter, fordi vi stadig gerne vil have fokus på. Når en metode starter med en kodelinje, der naturligt hører til enten Presenter eller Model, så er det også tommelfinger-regel, at metodens *entry point* også er Presenter eller model – dvs. at det er den metode i et de tre lag, der kaldes med en eksplisit reference.

Den eneste metode, der begynder med en kodelinje til Presenter, er UpdateDisplay, og det bliver vores første udfordring med at splitte metoden op i mindst 2 metoder. Den primære årsag til, at den første kodelinje hører til Presenter, er udelukkende fordi Presenter har ansvaret for at præsentere data i dets rette format, herunder formatering af klokken. Samtidig transformeres data fra at være af typen TDateTime til string, og det er noget vi bliver nødt til at forholde os til – data modtages i en type og sendes videre i en anden type. Vi starter dog med at oprette samme UpdateDisplay i Presenter, og ændre samtlige kald til UpdateDisplay til eksplisit at kalde metoden i Presenter, fordi det vil være det nye entry method – det gælder også kald til UpdateDisplay, der allerede er flyttet til Presenter.

Den lokale variabel, LTimeElapsed, tilhører den første linje, så erklæringen skal flyttes med over i Presenter sammen med den første kodelinje. Den anden kodelinje kunne flyttes med, fordi det i mange fortolkninger af konceptet er Presenter, der opdaterer View, men også fordi vi kan opdatere Caption fra Presenter uden at vi behøver referencer til Vcl.StdCtrls-unit i uses, som vi ikke ønsker. Men det introducerer et par problemer, fordi Caption kan hedde Text, hvis vi har en målsætning om at bruge Presenter og Model i multiplatform, men også at vi senere skal erstatte referencerne med interfaces, og properties har det med at introducere *getters* og *setters*, som vil føje yderligere metoder til, men hvor vi i højere bør bevare den eksisterende metode, fordi det er mere i tråd med refaktorering. Derfor har vi følgende situation i Presenter:

```
procedure TStopWatchPresenter.UpdateDisplay(const ATimeElapsed: TDateTime);
var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.fff', ATimeElapsed);
end;
```

og i View:

```
procedure TStopWatchView.UpdateDisplay(const ATimeElapsed: TDateTime);
begin
  StopWatchDisplay.Caption := LTimeElapsed;
end;
```

Netop fordi data ændrer type undervejs i de to kodelinjer, så er vi også nødt til at ændre definering af de 2 metoder. Dvs. at argumentet ATimeElapsed i metoden UpdateDisplay i View ændres til typen string, så det bliver muligt at splitte de to kodelinjer, og vi kan nu kalde metoden UpdateDisplay i View fra UpdateDisplay i Presenter. Bemærk, at selv med de ændringer, så har vi ikke ændret funktionaliteten af koden:

```

procedure TStopWatchPresenter.UpdateDisplay(const ATimeElapsed: TDateTime);
var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.zzz', ATimeElapsed);
  StopWatchView.UpdateDisplay(LTimeElapsed);
end;

```

Og i View – ændringerne findes i mappen "StopWatch - 4":

```

procedure TStopWatchView.UpdateDisplay(const ATimeElapsed: string);
begin
  StopWatchDisplay.Caption := ATimeElapsed;
end;

```

De to sidste metoder begynder med en kodelinje, der hører til Model, og derfor vil entry method derfor naturligt være den metode, der bliver defineret i Model. De to metoder består også af kodelinjer, der hører til alle tre lag, og derfor vil vi også erklære den samme metode i både View, Presenter og Model. Det er ikke givet, at de metoder kommer til at hedde StartTimer og StopTimer i samtlige lag, men det er det udgangspunkt, vi bør tage.

For at sikre os, at den rigtige metode – eller entry method – bliver kaldt, så kan vi benytte en tilgang, som vi bør holde os til. Vi starter med at erklære både StartTimer og StopTimer metoderne i Model, som de er erklæret i View, samt opretter metoderne med en tom implementering til at begynde med. De oprindelige metoder i View kan udkommenteres, og samtlige kald til metoder vil fejle i en kompilering, så det er forholdsvis nemt at finde de referencer, der skal ændres til at kalde metoder i Model:

```

procedure TStopWatchPresenter.StartButtonClick(Sender: TObject);
begin
  if StopWatchView.StopWatchTimer.Enabled then
    StopWatchModel.StopTimer
  else
    StopWatchModel.StartTimer;
end;

```

Når de eksplisitte referencer er rettet, så kan vi fjerne udkommenteringen af de oprindelige metoder, velvidende at de metoder ikke længere bliver kaldt direkte, men skal nu kaldes fra de nye metoder. Model kan dog ikke kalde de oprindelige metoder direkte, og derfor skal de samme metoder erklæres i Presenter – også fordi der er noget, der hører til Presenter, der skal flyttes på et tidspunkt. StartTimer og StopTimer oprettes derfor på samme måde, som de er oprettet i Model, og fra entry methods i Model vil de kalde metoderne i Presenter, som igen vil kalde de samme metoder i View:

```

procedure TStopWatchModel.StartTimer;
begin
  StopWatchPresenter.StartTimer;
end;

procedure TStopWatchModel.StopTimer;
begin
  StopWatchPresenter.StopTimer;
end;

```

Og i View:

```

procedure TStopWatchPresenter.StartTimer;
begin
  StopWatchView.StartTimer;
end;

procedure TStopWatchPresenter.StopTimer;
begin
  StopWatchView.StopTimer;
end;

```

Vi har ikke flyttet kodelinjerne endnu, men vi har sikret os, at metoder kaldes i den rigtige rækkefølge og at applikationen stadig vil virke, som det altid har gjort. De oprindelige kodelinjer med ændringer ser således ud:

```

procedure TStopWatchView.StartTimer;
begin
  FTimeStart := Now;
  StopWatchTimer.Enabled := True;
  StartButton.Caption := 'Stop';
  StopWatchPresenter.UpdateDisplay(0);
end;

procedure TStopWatchView.StopTimer;
begin
  StopWatchTimer.Enabled := False;
  StartButton.Caption := 'Start';
  StopWatchPresenter.UpdateDisplay(Now - FTimeStart);
end;

```

Vi skal således have sikret os, at de kodelinjer eksekveres i den rækkefølge, som de oprindeligt er skrevet – dvs. at vi sikrer os, at vi ikke ændrer funktionalitet ved at ændre rækkefølgen. Det betyder ikke noget, at kodelinjer, der skal blive i View, skal eksekveres før kodelinjer, der skal flyttes til Presenter, fordi det er et spørgsmål om, at Presenter kalder metoderne i View før den selv eksekverer kodelinjerne, ligesom Model omvendt vil eksekvere sine kodelinjer først, før Model kalder metoderne i Presenter. Når vi flytter kodelinjer til Model, markeret med grøn, vil vi også bemærke, at der er en direkte reference til field-variablen FTimeStart, og kan hurtigt regne ud, at det er en variabel, der også hører til Model, som derfor også flyttes sammen med dens property.

Vi vil også bemærke, at der er en direkte reference til StopWatchTimer-komponenten, og da den fungerer som vores data, så er det også en komponent, der skal flyttes til Model. En sådan komponent kan have en masse metoder tildelt i dens event-properties, og de er væsentlige at få flyttet med:



Komponent har et event, der skal flyttes med, og det kommer ikke automatisk med over, når vi copy-paster komponenten. Kodelinen i StopWatchTimerTimer kan blot flyttes som den er, fordi den har allerede en eksplisit reference til en metode i Presenter. Herefter vil kodelinjerne være fordelt i Model:

```

procedure TStopWatchModel.StartTimer;
begin
  FTimeStart := Now;
  StopWatchTimer.Enabled := True;
  StopWatchPresenter.StartTimer;
end;

procedure TStopWatchModel.StopTimer;
begin
  StopWatchTimer.Enabled := False;
  StopWatchPresenter.StopTimer;
end;

procedure TStopWatchModel.StopWatchTimerTimer(Sender: TObject);
begin
  StopWatchPresenter.StopWatchTimerTimer(Sender);
end;

```

De næste kodelinjer, der skal flyttes fra View, hører til Presenter, dvs. at StartTimer og StopTimer vil se således, når referencerne er rettet til – bemærk, at kald til metoder i View foretages først, så kodelinjerne eksekveres i deres oprindelige rækkefølge:

```

procedure TStopWatchPresenter.StartTimer;
begin
  StopWatchView.StartTimer;
  UpdateDisplay(0);
end;

procedure TStopWatchPresenter.StopTimer;
begin
  StopWatchView.StopTimer;
  UpdateDisplay(Now - StopWatchModel.TimeStart);
end;

```

Herefter kan applikationen kompileres og køre nøjagtig, som den altid har gjort. Vi har ikke ændre noget funktionelt i kode – vi har flyttet rundt på kodelinjerne, og vi har bevaret den oprindelige eksekvering af koden.

Tredje fase

Når vi har sikret os, at applikationen stadig opfører sig som det altid har gjort, som kan vi efterfølgende normalisere navnene på metoderne, vi har tilføjet, fordi de højest sandsynligt nu har helt andre funktioner og ansvar, end de havde før. Det er vigtigt, at vi giver os den mulig at gennemgå navnene på metoderne inden vi definerer et par interfaces, som View og Presenter skal være en del af. Vi har før nævnt entry method, og den metode vil højest sandsynligt også bevare dets oprindelige, mens nogle de øvrige identiske metoder måske kan ændres som værende kald med notifikationer.

Hvis vi starter med metoden StartTimer, så finder vi entry method i Model, men både Presenter og View har nu StartTimer. Det kan give voldsomme forvekslinger for senere udviklere, at StartTimer i View ikke starter nogen timer. Når StartTimer i Model er entry method, så er det også den rigtige metode, og kigger vi nærmere på de øvrige metoder, så er det kald som alt andet lige er notifikationer om, at timeren er startet. Dvs. at StartTimer i både View og Presenter bør hedde f.eks. TimerStarted, og ændring af navne på metoder er stadig indenfor rammen af refaktorering, og betyder ikke at vi ændrer funktionaliteten af en metode. Det er bare vigtigt, at vi giver os den tid, det tager at give metoder

korrekte og beskrivende navne. Når vi har gennemgået de kopierede metoder, og redefineret deres navne, så kunne fordelingen af kodelinjer se sådan ud:

View	Presenter	Model
<pre> procedure TStopWatchView.TimerStarted; begin StartButton.Caption := 'Stop'; end; procedure TStopWatchView.TimerStopped; begin StartButton.Caption := 'Start'; end; </pre>	<pre> procedure TStopWatchPresenter.TimerStarted; begin StopWatchView.TimerStarted; UpdateDisplay(0); end; procedure TStopWatchPresenter.TimerStopped; begin StopWatchView.TimerStopped; UpdateDisplay(Now - StopWatchModel.TimeStart); end; </pre>	<pre> procedure TStopWatchModel.StartTimer; begin FTimeStart := Now; StopWatchTimer.Enabled := True; StopWatchPresenter.TimerStarted; end; procedure TStopWatchModel.StopTimer; begin StopWatchTimer.Enabled := False; StopWatchPresenter.TimerStopped; end; </pre>

Det er ikke nødvendigt at tage stilling til navnene allerede i denne fase, fordi det også navnene også kan refaktoreres efter definering af interfaces. I nogle situationer vil der allerede være overblik over separeringen på dette stadie, og i andre situation kan en definering af et interface være med til at skabe et overblik, og dermed mulighed for igen at tage refaktorering af navne på især metoder til revurdering, men også placering af komponenter m.m.

Fjerde fase

Når vi har flyttet alle kodelinjer og komponenter til de respektive lag af en MVP-pattern, herunder refaktoreret navne på metoder, så har vi på den måde imødekommet et væsentligt punkt om Separation of Concern, som vi vil se for fleste patterns af denne type. Det er ikke sikkert, at koden ville have set sådan ud, hvis vi havde haft muligheden for at skrive applikationen med MVP-pattern helt fra begyndelsen, men vi kan skabe en separering uden at ændre i kodens funktionalitet. Der er en separation, men på dette tidspunkt er der stadig en stærk kobling mellem de enkelte dele, fordi de har nogle direkte referencer til hinanden. De referencer har gjort refaktoreringen noget nemmere, men det er meningen, at referencer i *uses clauses*, i implementationssektionerne, skal fjernes. Der må kun være en reference fra View til Presenter, og en reference fra Presenter til Model, i interface-sektionerne, og vi vil kunne opnå et ønske om en mere løs kobling.

For MVP pattern er interfaces det mest almindelige teknik til at definere en kommunikation mellem lagene, og opnå en mere løs kobling. Et interface fra Presenter til View vil også være nem at definere på dette tidspunkt, fordi så snart vi fjerner referencen, så vil der være kodelinjer med fuldt kvalificerede navne, der ikke længere kan kompileres. En søgning på "StopWatchView" i Presenter vil afsløre tre referencer, som i stedet må findes igennem et interface.

```

procedure TStopWatchPresenter.TimerStarted;
begin
  StopWatchView.TimerStarted;
  UpdateDisplay(0);
end;

procedure TStopWatchPresenter.TimerStopped;
begin
  StopWatchView.TimerStopped;
  UpdateDisplay(Now - StopWatchModel.TimeStart);
end;

procedure TStopWatchPresenter.StopWatchTimerTimer(Sender: TObject); ...
end;

procedure TStopWatchPresenter.UpdateDisplay(const ATimeElapsed: TDateTime);
var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.fff', ATimeElapsed);
  StopWatchView.UpdateDisplay(LTimeElapsed);
end;

```

Et interface skal blot erklæres og defineres lige før TStopWatchPresenter i Presenter, og tilføjes til erklæringen af formen TStopWatchView. Tilføjelsen til formen får ikke nogen funktionel indflydelse, fordi metoderne findes jo allerede, og det betyder også at metoderne kan flyttes tilbage fra public-sektionen til private-sektionen.

```

type
  IStopWatchView = interface['{0265C82F-5F42-4609-B9A0-FE221A282CFC}']
    procedure TimerStarted;
    procedure TimerStopped;
    procedure UpdateDisplay(const ATimeElapsed: string);
  end;
  TStopWatchPresenter = class(TDataModule)

```

En ny GUID til interfacet kan vi tilvejebringe ved hjælp af Ctrl+Shift+G (*Inserts a new Globally Unique Identifier*).

```

type
  IStopWatchView = class(TForm, IStopWatchView)
    StopWatchDisplay: TLabel;
    StartButton: TButton;
    procedure StartButtonClick(Sender: TObject);
  private
    procedure TimerStarted;
    procedure TimerStopped;
    procedure UpdateDisplay(const ATimeElapsed: string);
  public
  end;

```

Når vi ikke længere har en direkte reference til View fra Presenter, så må vi definere på anden måde, hvordan Presenter kan komme i kontakt med View. I MVP pattern er View et såkaldt entry point, dvs. at den bliver oprettet først, og har oftest ansvaret for hvilken Presenter, der skal arbejdes med, lige som View normalt vil aflevere en reference til sig selv som et argument til Presenter, når denne oprettes. I vores eksempel har vi dog ladet Delphi håndtere oprettelsen af de enkelte dele, hvorfor vi bare skal sikre, at de bliver oprettet i den rigtige rækkefølge – hhv. View, Presenter, og Model.

I MVP-pattern har vi altid en én-til-én relation mellem View og Presenter under eksekvering, men at vi har frihed til at vælge hvilke Views og Presenters, der kan arbejde sammen, før de bliver oprettet. Derfor passer MVP-pattern også til klassiske Delphi forms, som den 1-1 løsning eller overgang til mere eksotiske patterns. Når projekt-filen i Delphi har kaldt CreateForm på vores View, så kan vi aflevere en reference til Presenter i Views constructor. I klassedefinering af TStopWatchPresenter erklærer vi en klassevariabel med det interface, vi har defineret, med variabelnavnet StopWatchView, der er identisk med navnet på formen, som vi har haft refereret til. Med andre ord, så undgår vi at ændre i kode i implementationsdelen af Presenter.

```

TStopWatchPresenter = class(TDataModule)
public
  class var StopWatchView: IStopWatchView;
private

```

I View genererer vi et FormCreate for formens OnCreate-event. I vores applikation har denne event kun en opgave med at aflevere en reference til sig selv til Presenter:

```

procedure TStopWatchView.FormCreate(Sender: TObject);
begin
  TStopWatchPresenter.StopWatchView := Self;
end;

```

Vi bruger samme fremgangsmåde med referencen fra Model til Presenter, som vi fjerner fra implementationssektionen. En lokal søgning på StopWatchPresenter i Model finder tre eksplisitte referencer, som vi må have med i defineringen af en IStopWatchPresenter interface.

```

procedure TStopWatchModel.StartTimer;
begin
  FTimeStart := Now;
  StopWatchTimer.Enabled := True;
  StopWatchPresenter.TimerStarted;
end;

procedure TStopWatchModel.StopTimer;
begin
  StopWatchTimer.Enabled := False;
  StopWatchPresenter.TimerStopped;
end;

procedure TStopWatchModel.StopWatchTimerTimer(Sender: TObject);
begin
  StopWatchPresenter.StopWatchTimerTimer(Sender);
end;

```

Defineringen af IStopWatchPresenter placeres igen før TStopWatchModel:

```

type
  IStopWatchPresenter = interface['{D8304D16-513E-42A3-A2F2-4E892C077DAC}']
    procedure StopWatchTimerTimer(Sender: TObject);
    procedure TimerStarted;
    procedure TimerStopped;
  end;

```

Og tilføjes i erklæringen af TStopWatchPresenter, men det betyder også, at vi kan rydde op i metodernes access-level, fordi vi tidligere har hævet nogle metoder, så de kunne tilgås på tværs af units. Nu kan metoderne i stedet tilgås gennem et interface, hvilket bl.a. betyder at de ikke behøver at ligge i public-sektionen:

```
  TStopWatchPresenter = class(TDataModule, IStopWatchPresenter)
public
  class var StopWatchView: IStopWatchView;
private
  procedure TimerStarted;
  procedure TimerStopped;
  procedure StopWatchTimerTimer(Sender: TObject);
  procedure UpdateDisplay(const ATimeElapsed: TDateTime);
public
  procedure StartButtonClick(Sender: TObject);
end;
```

I TStopWatchPresenter opretter vi også en OnCreate-event, hvor vi afleverer en reference af Presenter til TStopWatchModel:

```
procedure TStopWatchPresenter.DataModuleCreate(Sender: TObject);
begin
  TStopWatchModel.StopWatchPresenter := Self;
end;
```

På dette stade (i mappen "StopWatch - 7") har vi teknisk set opnået målet om Separation of Concerns med en løs kobling i MVP pattern, som er hele formålet – Presenter har en reference til View gennem et interface, men ved egentlig ikke hvilket View eller om det i det hele taget er et View. Den ved bare, at reference har et foruddefineret interface, og det kunne lige så godt være et helt tredje objekt. En kompilation af koden vil resultere i en applikation, som vil køre som den altid har kørt.

På dette stade kan vi også have taget navnene, på især metoder, op til revurdering, eftersom vi er færdig med at flytte rundt, og hvor vi igen har mulighed at refaktorere. Navnene på metoder i de enkelte dele giver i nogle tilfælde ikke længere mening, fordi de i det ene tilfælde er metode på en handling og i det andet tilfælde en metode til notifikationer. Model vil oftest have fået metoder, der er arbejdsrelateret, f.eks. StartTimer, men de må oftest også notificere, at nu er timeren startet – eller stoppet. Dvs. at det interface, som Model har defineret en Presenter skal implementere, rent faktisk er notifikationer i vores tilfælde. Derfor har vi også tidligere fundet frem til, det ikke hedder StartTimer eller StopTimer, men TimerStarted og TimerStopped, når det er notifikationer. På dette tidspunkt kan vi måske have vurderet, at navnet på eventet StopWatchTimerTimer i interfacet kunne være TimerEvent, fordi navnet vil være lige så passende.

```
IStopWatchPresenter = interface['{D8304D16-513E-42A3-A2F2-4E892C077DAC}']
  procedure TimerEvent(Sender: TObject);
  procedure TimerStarted;
  procedure TimerStopped;
end;
```

Vi skal dog huske at gå til både definitionen af IStopWatchPresenter og benytte os af Delphis refaktoreringsfunktion, Ctrl+Shift+E, samt i implementeringerne, og omdøbe

metoderne. Navnene er lige så vigtige i refaktorering, fordi en udvikler vil forstå StartTimer og TimerStarted som vidt forskellige metoder i en opgave og en notifikation, men det afslører også, hvordan handlinger, opgaver og notifikationer har været blandet sammen i en klassisk Delphi-form, som mange traditionelle Delphi-udviklere er blevet så vant til.

Efter nogle refaktoreringer af forms til View, Presenter og Model, så bliver det også lettere, og måske mere hensigtsmæssigt i nogle tilfælde, at definere interfaces løbende under refaktoreringen, så tilgangen bliver simplere. Hardcore Delphi-udviklere kan måske gøre det i et hug, men refaktorering af tusindvis af kodelinjer på en enkel form-unit kan kræve sin mand, blot for at bevare et overblik. Løbende definering af interfaces er dog ikke tilrådeligt, fordi det er svært at få adgang til komponenter gennem interfaces uden at have properties med getters og setters. Det vil i mange tilfælde blot skabe mere kaos.

Refaktoreringen skal tages i skridt, der altid sikrer, at applikation er fuld funktionsdygtigt i det enkelte skridt. Refaktoreringer må ikke lægge en udvikling ned – i så tilfælde må vi gå tilbage til en tidligere version og revurdering refaktoreringen.

Det færdige resultat

Hvis vi foretager nogle før-og-efter-sammenligninger, så kan det virke temmelig voldsomt, at den samlede sourcekode vokser fra 68 linjer til 183 linjer, eller effektivt fra 14 kodelinjer til 23 kodelinjer, men vi skal have in mente, at MVP vil virke ekstremt for små projekter som i vores eksempel, men ganske befriende i større projekter med masser af kodelinjer. En refaktorering af en klassisk Delphi-form til MVP, eller implementering af MVP generelt, kræver både viden og erfaring for at nå frem til den rette implementation, så derfor vil det også være en god idé med refaktorering af små projekter først.

Form	View	Presenter	Model
<pre> type IStopWatchForm = class(TForm) StopWatchDisplay: TLabel; StopWatchTimer: TTimer; StartButton: TButton; procedure StopWatchTimerTimer(Sender: TObject); procedure StartButtonClick(Sender: TObject); private FTimeStart: TDateTime; procedure StartTimer; procedure StopTimer; procedure UpdateDisplay(const ATim public end;</pre>	<pre> type IStopWatchView = class(TForm, IStopW StopWatchDisplay: TLabel; StartButton: TButton; procedure StartButtonClick(Sender: TObject); procedure FormCreate(Sender: TObject); private procedure TimerStarted; procedure TimerStopped; procedure UpdateDisplay(const ATim public end;</pre>	<pre> type IStopWatchView = interface['1BCF4A1 procedure TimerStarted; procedure TimerStopped; procedure UpdateDisplay(const ATim end: IStopWatchPresenter = class(TDataMod procedure DataModuleCreate(Sender: TObject); private procedure TimerStarted; procedure TimerStopped; procedure TimerEvent(Sender: TObject); procedure UpdateDisplay(const ATim public procedure StartButtonClick(Sender: TObject); class var StopWatchView: IStopWat end:</pre>	<pre> type IStopWatchPresenter = interface['1F procedure TimerStarted; procedure TimerStopped; procedure TimerEvent(Sender: TObject); end: IStopWatchModel = class(TDataModule) StopWatchTimer: TTimer; procedure StopWatchTimerTimer(Sender: TObject); private FTimeStart: TDateTime; public procedure StartTimer; procedure StopTimer; property TimeStart: TDateTime read class var StopWatchView: IStopWat end:</pre>

Som det også fremgår af oversigten herover, så er det væsentligt, at separeringen har genkendelige kodekonstruktioner på tværs af units og den oprindelige unit, som også gælder for implementationssektion herunder. Dermed ikke sagt, at vi ikke kan fortsætte refaktoreringen herfra – vi skal bare lige refaktorere mod et MVP-pattern, og at det virker som det altid har gjort, før vi tænker på at refaktorere slutresultatet frem mod andre mål, som har krævet at vi har opfyldt Separation of Concerns først. I senere kapitler vil vi tage dette slutresultat og refaktorere det mod et tilsnit, der er tættere på MVVM-pattern, hvor vi tage interfaces i vores units herover og bruger dem som skabelon.

Form	View	Presenter	Model
<pre> procedure TStopWatchForm.StartButtonClick(begin if StopWatchTimer.Enabled then StopTimer else StartTimer; end; procedure TStopWatchForm.StartTimer; begin FTimeStart := Now; StopWatchTimer.Enabled := True; StartButton.Caption := 'Stop'; UpdateDisplay(0); end; procedure TStopWatchForm.StopTimer; begin StopWatchTimer.Enabled := False; StartButton.Caption := 'Start'; UpdateDisplay(Now - FTimeStart); end; procedure TStopWatchForm.StopWatchTime begin UpdateDisplay(Now - FTimeStart); end; procedure TStopWatchForm.UpdateDisplay var LTimeElapsed: string; begin DateToString(LTimeElapsed, 'hh:ss'); StopWatchDisplay.Caption := LTimeElapsed; end; </pre>	<pre> procedure TStopWatchView.FormCreate(Sender: TObject); begin TStopWatchPresenter.StopWatchView := nil; end; procedure TStopWatchView.StartButtonClick(Sender: TObject); begin TStopWatchPresenter.StartButtonClick; end; procedure TStopWatchView.TimerStarted(Sender: TObject); begin StartButton.Caption := 'Stop'; end; procedure TStopWatchView.TimerStopped(Sender: TObject); begin StartButton.Caption := 'Start'; end; procedure TStopWatchView.UpdateDisplay(Sender: TObject); begin StopWatchDisplay.Caption := ATimeElapsed; end; </pre>	<pre> procedure TStopWatchPresenter.DataModuleCreate(Sender: TObject); begin TStopWatchModel.StopWatchPresenter := nil; end; procedure TStopWatchPresenter.StartButtonClick(Sender: TObject); begin if TStopWatchModel.StopWatchTimer.Enabled then StopWatchModel.StopTimer else StopWatchModel.StartTimer; end; procedure TStopWatchPresenter.TimerStarted(Sender: TObject); begin StopWatchView.TimerStarted; UpdateDisplay(0); end; procedure TStopWatchPresenter.TimerStopped(Sender: TObject); begin StopWatchView.TimerStopped; UpdateDisplay(Now - StopWatchModel.TimeElapsed); end; procedure TStopWatchPresenter.TimerEvent(Sender: TObject); begin UpdateDisplay(Now - StopWatchModel.TimeElapsed); end; procedure TStopWatchPresenter.UpdateDisplay(Sender: TObject); var LTimeElapsed: string; begin DateToString(LTimeElapsed, 'hh:ss'); StopWatchView.UpdateDisplay(LTimeElapsed); end; </pre>	<pre> procedure TStopWatchModel.StartTimer; begin FTimeStart := Now; StopWatchTimer.Enabled := True; StopWatchPresenter.TimerStarted; end; procedure TStopWatchModel.StopTimer; begin StopWatchTimer.Enabled := False; StopWatchPresenter.TimerStopped; end; procedure TStopWatchModel.StopWatchTime; begin StopWatchPresenter.TimerEvent; end; </pre>

Implementationsdelen kan også ses i mappen "StopWatch - 7". Med sammenligningen herunder vil vi straks bemærke, at der er lang mere kode, når separeringen er fuldført og i forhold til den oprindelige kode, hvilket også betyder at der er lidt overhead i form af kald af ekstra metoder. Og dette skal ses i betragtning af, at koden funktionelt set stadig er den samme. Den lille overhead er prisen, men den pris skal også ses i sammenhæng med, at vi nu vil have lang bedre mulighed for at ændre eller implementere nye features, fordi separeringen har indflydelse på, hvor meget kode vi behøver at ændre, hvor let det er at ændre kode, hvor mindre sandsynligt det bliver at ødelægge eksisterende features, der bruges andre steder, og ikke mindst hvor meget kode vi reelt kan genbruge. Nogle af de mest tungtvejende bevæggrunde, vi kunne have, i forbindelse med refaktorering, vil være, at vi kan have et ønske om netop at genbruge så meget kode som muligt, hvis vi har et mål om at applikationen skal køre på flere platforme – f.eks. en traditionel desktop-applikation til Windows, som vi ønsker at afvikle på macOS.

Multiplatform

Det er ikke en del af bogens hovedmål, fordi vi er nået frem en adskillelse i forhold til en pattern, men fordi refaktorering ofte vil have et formål om at kompilere projektet på forskellige platforme, så er det vigtigt at nævne, at der er yderligere udfordringer med refaktorering fra VCL til FMX. Indtil videre har vi nemlig ikke taget højde for, at det endelige resultat skal kunne kompileres på flere platforme. Det er jo et VCL-projekt, med en masse referencer til VCL-biblioteker, så det vil jo kun kunne kompileres for 32-bit og måske 64-bit Windows. Vi har til vores egen fordel pålagt os den regel, at vi altid vil sigte efter, at kun View har referencer, der er platform-specifikke. Derfor har vi også gjort det nemmere for os selv, at det "kun" er formen, vi skal skifte ud – i bedste fald. Det er bare ikke altid let

at undgå VCL-referencer, fordi selv ikke-visuelle komponenter som TTimer, TActionList m.fl., er VCL-komponenter, der ikke så let lader sig kompilere med et FMX-framework, ligesom tredjepart komponenter kan afstedkomme voldsomme problemer. Her har vi selvfølgelige, forud for refaktoreringen, foretaget en vurdering og test af, om vores brug af komponenter findes i udgaver, der er kompatible med vores slutprodukt.

Ikke alene har vi måske et behov for at få skabt en FMX-udgave af vores View, men vores TTimer i Model har en reference til Vcl.ExtCtrls.pas, som ikke vil gå så godt i spænd med FMX, og som heller vil foretrække en reference til Fmx.Types.pas. Selvom vores Presenter og Model er baseret på TDataModule, der er multiplatform-kompatibelt, så vil der være store problemer med at håndtere et data-modul, som vi gerne vil bruge både i et VCL- og FMX-projekt, under Delphi IDE. Hvis vi tilføjede vores Model til et FMX-projekt, så vil der ganske enkelt bare resultere i følgende:

```
unit Stopwatch.Main.Model;
interface
uses
  System.SysUtils, System.Classes, Vcl.ExtCtrls;
```

Det er af samme grund, at Embarcadero, i autogenerering af et data-modul, har tilføjet et framework-direktiv i koden, der kun tolkes af IDE i units med TDataModule, så IDE-designeren kan se forskel på om modulet er beregnet til VCL eller FMX, og vi fra designeren kun kan tilføje komponenter, der er beregnet til et af disse to frameworks. Uden direktivet er der ingen komponenter fra de to frameworks, der kan tilføjes – det kan vi selvfølgelig manuelt, men ikke med drag-and-drop fra IDE's palette af komponenter. End ikke et compiler-direktiv i uses clauses vil hjælpe designeren, så på dette punkt er en runtime-løsning den sikreste tilgang til at løse op på det problem. Dette framework-direktiv har ikke nogen indflydelse på selve compileren, så den vil under alle omstændigheder forsøge at kompilere med de referencer, der nu er angivet, uanset om vi bruger VCL- eller FMX-framework.

```
TStopWatchModel = class(TDataModule)
  procedure StopWatchTimerTimer(Sender: TObject);
  procedure DataModuleCreate(Sender: TObject);
private
  FTimeStart: TDateTime;
  FStopWatchTimer: TTimer;
public
  procedure StartTimer;
  procedure StopTimer;
  property StopWatchTimer: TTimer read FStopWatchTimer;
  property TimeStart: TDateTime read FTimeStart;
  class var StopWatchPresenter: IStopWatchPresenter;
end;
```

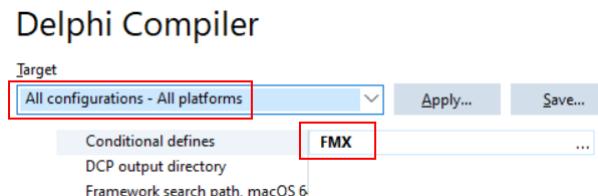
Det betyder, at vi må fjerne vores StopWatchTimer-komponent fra vores Model gennem IDE'en, men først efter vi har noteret os, hvordan dennes properties er sat – især de properties, som er sat anderledes end default-værdien. Vi opretter en field-variabel, FStopWatchTimer, med en tilhørende property, så komponenten fremgår udadtil som

om intet var hændt. Samtidig må vi også tilføje en metode til OnCreate-eventet i modulet, hvor vi kan oprette Timer-komponenten og sætte dennes properties. Vi har ikke ændret noget funktionelt i koden, men det skal måske tilføjes, at Timer-komponent nu oprettes som den sidste i en række af komponenter i modulet, som oprettes under indlæsning af modulet og OnCreate-eventet først efter denne indlæsning er færdig. Vi foretager naturligvis ikke noget arbejde under en constructor, men derfor er dette interessant, fordi vi ofte også vil være nødt til at binde forskellige komponenter sammen, f.eks. datasources i Presenter med datasets i Model, eller komponenter imellem i Model.

```
procedure TStopWatchModel.DataModuleCreate(Sender: TObject);
begin
  FStopWatchTimer := TTimer.Create(Self);
  FStopWatchTimer.Enabled := False;
  FStopWatchTimer.Interval := 10;
  FStopWatchTimer.OnTimer := StopWatchTimerTimer;
end;
```

Det næste problem i rækken er selvfølgelig at vælge den rigtige reference til TTimer-komponenten – enten er det en komponent fra Vcl.ExtCtrls, når vi kompilerer på VCL-frameworket, eller også er det en komponent fra Fmx.Types, når vi kompilerer på FMX-frameworket. Derudover har Delphi heller ikke et compiler-direktiv, der kan skelne om vi kompilerer til VCL eller FMX, fordi der teknisk set ikke er noget, der hedder VCL- eller FMX-applikationer – kun applikationer, der benytter sig af et af disse frameworks. Og selvom det nok ikke anbefales, så kan begge frameworks sagtens benyttes samtidig i en Delphi-applikation.

Til vores behov har vi brug for en compiler-direktiv, som kan fortælle compileren, hvilken af disse to TTimer-klasser, den skal tolke en timer fra, og det sikreste er faktisk, at vi tilføjer vores egen compiler-direktiv til vores FMX-projekt. Vi tilføjer i vores nye FMX-projekt, fordi vi kan bevare VCL-projektet, som det altid har været. Dermed ikke sagt, at det faktisk kan være nemmere at tilføje et compiler-direktiv i det gamle VCL-projekt, fordi det nye FMX-projekt vil have så mange platform-targets, at det måske kan være mere beejligt at adskille det VCL-projektet med et compiler-direktiv. Der er frit slag for at vælge et identifier for direktivet, så det behøver heller ikke at være "FMX":



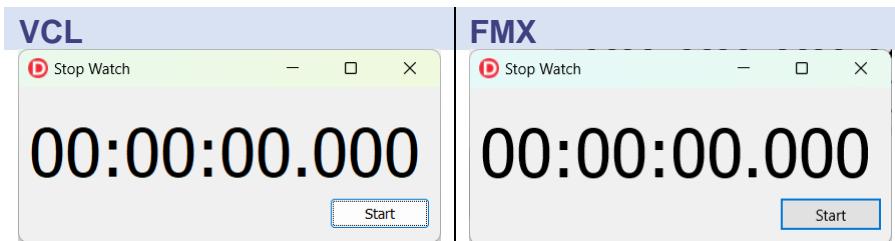
Med dette direktiv, så er det noget lettere at definere og vælge hvilke frameworks, der skal kompileres, for de komponenter, hvor FMX har en komponent, der svarer til en tilsvarende VCL-komponent. Det er ikke altid, at det vil være så nemt, så i andre tilfælde må vi enten ty til flere compiler-direktiver, eller implementere en løsning med et objekt, der håndterer en tredje komponent, der går ud for at være en komponent vi mangler. Det vil

også være den største udfordring ved flere platforme, ganske enkelt fordi operativsystemer har det med at tilbyde API'er, der er vidt forskellige fra hinanden og med vidt forskellige teknikker og tilgange. Det er dog ikke formålet med denne bog, og i eksemplet har vi blot brug for at kunne referere til et af de TTimer-komponenter:

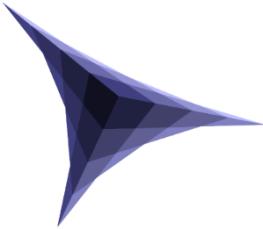
```
uses
  System.SysUtils, System.Classes,
{$IFDEF FMX}
  Fmx.Types
{$ELSE}
  Vcl.ExtCtrls
{$IFEND}
;
```

Hele idéen om at understøtte multiplatform i vores Delphi-projekter, eller at de underliggende biblioteker kan kompileres på tværs af operativsystemer, så er vi alt andet lige bedre stillet med et projekt, der er separeret, som det er sket med dette eksempel. Et klassisk Delphi-projekt, der er forsøgt kompileret til forskellige platforme uden adskillelse, ender altid med så mange compiler-direktiver, at projektet bliver ekstrem sårbar overfor ændringer og særdeles kompliceret, når vi skal have implementeret nye features.

Det er muligt, at det gamle VCL-projekt er ikke længere vil være nødvendigt, fordi FMX-projektet, med Win32/Win64 som target, vil være tilstrækkeligt. Der er dog nogle små visuelle forskelle mellem VCL- og FMX-komponenter, som måske kan være generere, ligesom VCL er langt bedre til at understøtte klassisk Windows-interaktion med brugerne, som pludselig ikke vil kunne lade sig gøre i et FMX-form, selvom det er kompileret til Win32/Win64. Det er overvejelser, vi er nødt til at gøre os, hvis vi gerne vil bevare det produkt, som brugerne har vænnet sig til.



Men det kan lade sig gøre at bevare det gamle VCL-projekt og skabe et nyt View, specifik til andre platforme, og genbruge både Presenter og Model – dvs. handlinger og forretningslogikken vil være det samme, og det vil kun være vinduet, der vil være til forskel. Hvis VCL kunne bortskaffes, så giver dette en ekstra fordel, idet vi vil kunne flytte flere kodelinjer fra View til Presenter, fordi vi ikke vil behøve at forholde til, at der er forskel på TLabel.Caption i VCL og TLabel.Text i FMX.



Kapitel 4

Actions og Multiplatform

Nu har vi gennemgået et meget simpelt eksempel med refaktorering, og hvorfor ikke fortsætte med endnu et eksempel, hvor vi bruger samme teknik, men sætter fokus på andre udfordringer og i øvrigt tage øvelsen med som erfaring i at refaktorere frem mod et MVP-pattern. Dette eksempel vil have fokus på actions, fordi mange projekter bruger komponenten TActionList, og derfor kan vi have behov for at vide, hvordan vi kan flytte disse komponenter til Presenter, og hvordan vi kan gøre brug af dem i Views.

Der er en lille detalje ved TActionList, ligesom TImageList og mange andre non-visual komponenter, der vil skabe udfordringer med refaktorering, fordi det er en VCL-komponent og kan findes i Vcl.ActnList.pas-unit. Det er en detalje, der vil få indflydelse på, om vi ønsker at flytte disse VCL-komponenter, uanset om Presenter vil være det mest ideelle lag at placere actions. Hvis vores refaktorering har et mål om at opnå Separation of Concerns, men at vi i øvrigt ikke har planer om at migrere projektet til FMX-frameworket, så kan vi sagtens flytte komponenterne. Har vi derimod planer om at migrere projektet til FMX-frameworket og bevare VCL-projektet, så kan vi vælge at lade komponenterne blive i View, eller generere komponenterne under runtime. Det sidste vil give drastiske tilføjelser af kodelinjer til håndtering af komponenter og de enkelte actions – også især hvis TActionList gør brug af TImageList, som er en VCL-komponent, der gør brug af resources under load af forms eller moduler.

Uagtet at vi bør undlade at flytte VCL-komponenter til Presenter, så gør vi det, fordi vi skal have en fornemmelse af, hvor mange kodelinjer vi må tilføje, for at vi kan flytte en TActionList-komponent i Presenter, der enten udelukkende skal bruges i et VCL-projekt eller skal bruges i en kombination med VCL- og FMX-frameworket. Der er situationer, hvor det er nødvendigt eller giver mening med begge frameworks, men det vil være at foretrække, hvis denne del af udfordringen tages særskilt, når refaktoreringen mod et pattern er afsluttet. Hvis det er målsætningen at flytte actions til Presenter, fordi det vil være det naturlige sted at placere actions, så vil vi foretrække afslutte refaktoreringen mod et

pattern med VCL-framework, og så tage udfordringen med FMX-frameworket i en migreringsopgave. Og dette gælder også, hvis vi har en målsætning om at refaktorere for at migrere fuldstændigt til FMX-frameworket og skrotte VCL-projektet bagefter. I det tilfælde bør non-visual komponenter som TActionList naturligvis flyttes, og vi vil heller ikke have behov for at generere komponenter under runtime, medmindre VCL-projektet skal bevares som et fallback-projekt.

Det vil sige, at vi kan have ikke færre end 4 forskellige slutresultater som målsætning:

- 1) Separering med ren VCL-projekt.
- 2) Separering med VCL og FMX i View, men uden VCL/FMX i Presenter og Model.
- 3) Separering med VCL og FMX i View, samt Presenter og Model om nødvendigt og med dynamisk generering af komponenter.
- 4) Separering og migrering fra VCL-projekt til ren FMX-projekt.

Selvom denne bog kun handler om separering og refaktorering henimod et pattern, så er disse målsætninger væsentlige, fordi hver målsætning vil have forskellige indflydelse på, hvad vi bør flytte. Hvis vi har sådanne mål i vores tanker, så er denne gennemgang et godt udgangspunkt for de udfordringer, vi vil kunnestå overfor. Nummereringen af målsætningen med separeringen passer også med de afsluttede refaktoreringer i mapperne, dvs. at de endelige resultater kan findes i mapperne "Continents - 1" og op.

Hvis vi går med planer om at separere med henblik på at genbruge vores VCL-projekt til et FMX-projekt, så er der rigtig mange VCL-komponenter, der kan volde rigtig mange udfordringer – enten fordi de ikke findes i FMX eller fordi de er så forskellige, og der skal tages særlige tiltag for at det kommer til at fungere. Derudover har vi ikke taget højde for rigtig mange tredjepartsprodukter, som kan volde lige så mange problemer – hvis der overhovedet findes et FMX-udgave af de komponenter. Vi har tilføjet en liste over standard non-visual komponenter, der følger med Delphi, i appendiks A.

1) Separering med ren VCL

Med et rent VCL-projekt betyder, at vi ikke har nogen ambitioner om, at projektet skal kunne kompileres på andre platforme end Win32/Win64, eller foreløbigt ser bort fra andre platforme, og så kan vi også se bort fra tommelfingerreglen om, at vi ikke bør flytte VCL-komponenter til hverken Presenter eller Model, medmindre det er nødvendigt. Det kan være tungere vejende grunde til, at vi vil flytte en komponent til Presenter eller Model, der har referencer til et VCL-bibliotek, og så må vi indrette os efter det.

Selvom vi kan se bort fra reglen om ikke at flytte VCL-komponenter til Presenter og Model, så bør vi stadig mindske behovet for at referere til flere VCL-biblioteker, fordi det vil hjælpe os i fremtiden, når vi engang skal migrere til nyere versioner eller understøtte andre platform. Derfor kan der være en grad af valg for, hvad vi flytter fra View, og hvad vi eksponerer i View gennem en interface defineret af Presenter. I dette afsnit er der derfor større fokus på de små variationer i refaktoreringen, fremfor de enkelte skridt i processen som i det forrige kapitel, men vi bruger stadig samme teknik.

De små forskelle kommer især i hvordan vi definere det interface, som et View skal eksponere mod Presenter, og den definerer også de VCL-biblioteker, som Presenter nødvendigvis må referere til, for at definere et interface. Vi har f.eks. et par listbokse (TListBox), som presenter gerne vil have adgang til at manipulere, for ellers er der ikke mange kodelinjer, der kan flyttes til Presenter. Det er derfor også en øvelse i at fremhæve, hvilke opgaver Presenter har i forhold til, bogstavelig talt, at præsentere data i View. Når vi har refaktoret færdig, så bør størstedelen af koden i Presenter, hvorimod View højest vil bestå af kodelinjer til at binde komponenter og events sammen, og nogle *getters*, som interfacet har defineret.

Den første variant virker som den mest åbenlyse løsning, fordi vi kan erstatte referencer til listbokse i View en-til-en med de samme klasser eller fælles klasser, og derfor er vi også nødt til at have referencer til Vcl.StdCtrls-biblioteket. Vi har allerede besluttet os for, at TActionList-komponenten fysisk skal placeres i Presenter, og derfor vil referencer til Vcl.ActnList være nødvendige, og meget svære at komme udenom. Definering af interfacet til View kan også afstedkomme referencer, som det ses herunder:

```

uses
  System.SysUtils, System.Classes, Continents.Selection.Model,
  System.Actions, Vcl.ActnList, Vcl.StdCtrls;

type
  ISelectContinentsDialog = interface['{85899383-81A5-4B38-A02B-1A2129F07977}]
    function GetAvailableListBox: TCustomListBox;
    function GetSelectedListBox: TCustomListBox;
  end;

```

Hvis vi kigger nærmere på koden, så kan vi heldige at udlede, at der udelukkende er referencer til Items og ItemIndex properties i listboksene. Vi bliver jo også klogere på kodelinjerne, når vi har refaktoreret dem til at fungere i Presenter, og resultatet for et af listboksene kan måske se ud som herunder – det endelige resultat kan ses under mappen "Continents - 1 - 1":

```

procedure TSelectContinentsPresenter.SelectOneActionExecute(Sender: TObject);
begin
  with SelectContinentsDialog do
    if GetAvailableListBox.ItemIndex > -1 then
      begin
        GetSelectedListBox.Items.Add(GetAvailableListBox.Items[GetAvailableListBox.ItemIndex]);
        GetAvailableListBox.Items.Delete(GetAvailableListBox.ItemIndex);
        UpdateSelectActions;
      end;
end;

```

Denne refaktorerede version af metoden er ikke så langt fra den oprindelige metoder, der er afbilledet herunder, og er blot tilpasset til at der nu er et interface imellem Presenter og View, hvorfor der vil være eksplisitte referencer, men at kodelinjerne, sammenlignet med de oprindelige kodelinjer, funktionelt set ikke har ændret sig overhovedet.

```

procedure TSelectContinentsDialog.SelectOneActionExecute(Sender: TObject);
begin
  with AvailableListBox do
    if ItemIndex > -1 then
    begin
      SelectedListBox.Items.Add(Items[ItemIndex]);
      Items.Delete(ItemIndex);
      UpdateSelectActions;
    end;
end;

```

Derfor ved vi også, efter gennemgang og refaktorering af kodelinjerne, at der udelukkende tilgås properties som Items og ItemIndex, som enten er en standard klasse fra System.Classes-bibliotek eller af typen Integer. Derfor behøver det heller ikke at være nødvendigt med referencer til Vcl-biblioteker for vores listbokse, og vi kan gøre en indsats for at mindske behovet for Vcl-biblioteker i Presenter. Den første umiddelbare løsning kræver blot 2 metoder defineret i interfacet, og selvom den anden løsning kræver 4 metoder, så er det den pris, der skal vejes op imod at være afhængig af et bibliotek eller et framework.

```

uses
  System.SysUtils, System.Classes, Continents.Selection.Model,
  System.Actions, Vcl.ActnList;

type
  ISelectContinentsDialog = interface['{85899383-81A5-4B38-A02B-1A2129F07977}']
    function GetAvailableListBoxItemIndex: Integer;
    function GetAvailableListBoxItems: TStrings;
    function GetSelectedListBoxItemIndex: Integer;
    function GetSelectedListBoxItems: TStrings;
  end;

```

Hvis vi sammenligner implementationerne i Presenter, så er der så lille et forskel, at det slet ikke er dør, der er en pris, og funktionelt har kodelinjerne heller ikke ændret sig:

Med TListBox i interfacet

```

procedure TSelectContinentsPresenter.SelectOneActionExecute
begin
  with SelectContinentsDialog do
    if GetAvailableListBox.ItemIndex > -1 then
    begin
      GetSelectedListBox.Items.Add(GetAvailableListBox.Ite
      GetAvailableListBox.Items.Delete(GetAvailableListBox
      UpdateSelectActions;
    end;
end;

```

Med TStrings og Integer i interfacet

```

procedure TSelectContinentsPresenter.SelectOneActionExecute
begin
  with SelectContinentsDialog do
    if GetAvailableListBoxItemIndex > -1 then
    begin
      GetSelectedListBoxItems.Add(GetAvailableListBoxItem
      GetAvailableListBoxItems.Delete(GetAvailableListBox
      UpdateSelectActions;
    end;
end;

```

Den store gevinst ved den sidste løsning er, at vi ikke længere er bundet til en TListBox i View – vi kan skifte den komponent ud med en TComboBox, TCheckListBox eller en tredjepartskomponent, der bygger på Items og ItemIndex properties. Der vil ikke længere være nogle eksplisitte VCL-komponenter i View – og det vil i høj grad også lette arbejdet med unit testing, fordi vi specifikt kan sætte et mockup-view op med samme interface og teste samtlige features uden at Presenter og Model nogensinde behøver at vide, at der aldrig var tale om noget rigtigt View. Med andre ord, så skal vi ikke bare flytte VCL-komponenter fra View til Presenter eller Model, fordi vi kan, men fordi det er nødvendigt, og så

vidt muligt refaktorere kodelinjer, så vi søger mod lavere fællesklasser, som mest sandsynligt ikke er VCL-komponenter.

View indeholder ikke længere noget signifikant funktionel kode, og indeholder stort set kun kodelinjer, der binder events og actions, samt *getters*, der blot fungerer som forlængende referencer. View har på denne måde kun en visuel rolle, men har stadig ansvaret for at tilvejebringe en Presenter og etablere bindinger til denne.

Begge løsninger kan ses i mapperne henholdsvis "Continents - 1 - 1" og "Continents - 1 - 2".

2) Separering med VCL-/FMX-udgave af View

Hvis vi har en målsætning om at bruge projektet i et FMX-projekt, hvor Presenter og Model indgår i både VCL- og FMX-projektet, så vil det mest nærliggende være at lade alle VCL-relaterede komponenter blive i View, ligesom FMX-relaterede komponenter vil blive et nyt View i FMX-projektet. Derfor vil vi være nødt til at efterlade rigtig mange kodelinjer i View, men vi vil stadig forsøge at bevare målsætningen om at flytte så mange actions til Presenter og forretningslogik til Model. Dette er en yderligtgående løsning, dvs. at vi ikke vil tillade platformspecifikke referencer i hverken Presenter eller Model. Det er en meget almindelig tilgang i løsninger med multiplatform, men det efterlader også et View, som indtager en større rolle.

Vi kan være heldige, at de VCL-komponenter, vi benytter, er nedarvet fra fællesklasser, og lige præcis kan gøre den forskel, at Presenter kan sin bevare sin rolle som i vores første løsning. I det eksempel vil vi være interesseret i de enkelte actions, fordi vi sjældent interagerer med selve TActionList. De enkelte actions er nedarvet fra TContainedAction, som vi kan finde i System.Actions, og samtidig indikerer, at denne klasse ikke er platformspecifikt, ligesom den også har Execute-metode og Enabled-property. Det betyder, at vi skal have eksponeret disse action-komponenter, vi har i View, til Presenter.

Disse actions er desuden nogle statiske komponenter, på den måde at de bliver skabt, når View bliver skabt, og forsvinder først, når View bliver frigivet. Derfor har mulighed for både at kunne skubbe disse actions i View til Presenter, eller trække dem i Presenter fra View. Den første mulighed kræver en masse *getters* i definering af interfacet til View, som Presenter kan trække fra, og den anden mulighed kræver en field-variabel i Presenter for hver action, som View kan skubbe de enkelte actions til. Den første mulighed er længere definering af et interface og en masse metoder i View, mens den sidste godt nok tilføjer nogle field-variabler og kodelinjer med tildelinger i View, men betyder ikke flere metoder. Derfor er den sidste mulighed lidt mere attraktiv, ligesom den trods alt byder på lidt mindre overheads. Begge løsninger kan findes i mapperne "Continents - 2 - 1" og "Continents - 2 - 2".

Andet eksempel rummer også et FMX-projekt, der genbruger samme Presenter og Model, med et MainForm og View, der er genskabt som et FMX-form. Delphi har ikke et automatisk importering og konvertering af VCL-forms til FMX-forms og omvendt, så det

har altid givet et slavisk arbejde med at skabe en kopi af et form fra det ene platform til det andet. I nogle tilfælde er Delphi IDE i stand til at kopiere VCL-komponenter fra formen til en FMX-form, hvis der ikke er stor en forskel på de komponenter.

Træk af actions fra View

```
type
  ISelectContinentsDialog = interface'{85899383-81A5-4B38
    function GetAvailableListBoxItemIndex: Integer;
    function GetAvailableListBoxItems: TStrings;
    function GetSelectedListBoxItemIndex: Integer;
    function GetSelectedListBoxItems: TStrings;
    function GetDeselectOneAction: TContainedAction;
    function GetDeselectAllAction: TContainedAction;
    function GetSelectAllAction: TContainedAction;
    function GetSelectOneAction: TContainedAction;
    function GetResetAction: TContainedAction;
  end;

  TSelectContinentsPresenter = class(TDataModule, ISelectC
  private
    FDefaultContinents: string;
    FSelectContinentsModel: TSelectContinentsModel;
```

Skub af actions til Presenter

```
type
  ISelectContinentsDialog = interface'{85899383-81A5-4B38
    function GetAvailableListBoxItemIndex: Integer;
    function GetAvailableListBoxItems: TStrings;
    function GetSelectedListBoxItemIndex: Integer;
    function GetSelectedListBoxItems: TStrings;
  end;

  TSelectContinentsPresenter = class(TDataModule, ISelectC
  procedure DataModuleCreate(Sender: TObject);
  private
    FDeselectOneAction: TContainedAction;
    FDeselectAllAction: TContainedAction;
    FSelectAllAction: TContainedAction;
    FSelectOneAction: TContainedAction;
    FResetAction: TContainedAction;
    FDefaultContinents: string;
    FSelectContinentsModel: TSelectContinentsModel;
```

Selvom det betyder, at der er flere getters i View for den første variant, så betyder det ikke flere kodelinjer i Presenter, fordi det ændres blot fra at være field-variabler til forlængende referencer via getters i View. Og selvom der er tilføjet getters i View, så er der ikke ændret noget funktionelt i koden, fordi getters som sagt fungerer som forlængede referencer. Det giver lidt overhead for den første variant, fordi der hentes referencer hele tiden, mens der kun skubbes en enkel gang i den anden variant.

De overvejelser er værd at tage med, selvom det måske ikke betyder noget i implementationsdelen af Presenter, eller Model, så kan det også være et spørgsmål om performance, når vi taler om større kodekonstruktioner og masser af data.

Træk af actions fra View

```

procedure TSelectContinentsPresenter.SelectAllActionExecute
begin
  SelectContinentsDialog.GetSelectedListBoxItems.Text := T
  SelectContinentsDialog.GetAvailableListBoxItems.Text := UpdateSelectActions;
end;

procedure TSelectContinentsPresenter.SelectedListBoxClick(
begin
  UpdateSelectActions;
end;

procedure TSelectContinentsPresenter.SelectedListBoxDblClick(
begin
  SelectContinentsDialog.GetDeselectOneAction.Execute;
end;

procedure TSelectContinentsPresenter.SelectOneActionExecute
begin
  with SelectContinentsDialog do
    if GetAvailableListBoxItemIndex > -1 then
      begin
        GetSelectedListBoxItems.Add(GetAvailableListBoxItems
        GetAvailableListBoxItems.Delete(GetAvailableListBoxI
        UpdateSelectActions;
      end;
end;

procedure TSelectContinentsPresenter.UpdateSelectActions;
begin
  SelectContinentsDialog.GetDeselectAllAction.Enabled := S
  SelectContinentsDialog.GetDeselectOneAction.Enabled := S
  SelectContinentsDialog.GetSelectAllAction.Enabled := Sel
  SelectContinentsDialog.GetSelectOneAction.Enabled := Sel
end;

```

Referencer af actions i Presenter

```

procedure TSelectContinentsPresenter.SelectAllActionExecute
begin
  SelectContinentsDialog.GetSelectedListBoxItems.Text := T
  SelectContinentsDialog.GetAvailableListBoxItems.Text := UpdateSelectActions;
end;

procedure TSelectContinentsPresenter.SelectedListBoxClick(
begin
  UpdateSelectActions;
end;

procedure TSelectContinentsPresenter.SelectedListBoxDblClick(
begin
  DeselectOneAction.Execute;
end;

procedure TSelectContinentsPresenter.SelectOneActionExecute
begin
  with SelectContinentsDialog do
    if GetAvailableListBoxItemIndex > -1 then
      begin
        GetSelectedListBoxItems.Add(GetAvailableListBoxItems
        GetAvailableListBoxItems.Delete(GetAvailableListBoxI
        UpdateSelectActions;
      end;
end;

procedure TSelectContinentsPresenter.UpdateSelectActions;
begin
  DeselectAllAction.Enabled := SelectContinentsDialog.GetS
  DeselectOneAction.Enabled := SelectContinentsDialog.GetS
  SelectAllAction.Enabled := SelectContinentsDialog.GetAva
  SelectOneAction.Enabled := SelectContinentsDialog.GetAva
end;

```

Det fleste vil nok foretrække at skubbe referencer til actions en enkelt gang, og leve med at constructor i View har nogle kodelinjer, der binder de forskellige komponenter sammen den ene gang. Det er muligt at binde action-properties under design-time med Delphi IDE, og på den måde minimere mængden af kodelinjer:

Actions hentes via et interface

```

procedure TSelectContinentsDialog.FormCreate(Sender: TObject)
begin
  FSelectContinentsPresenter := TSelectContinentsPresenter
  DeselectOneAction.OnExecute := SelectContinentsPresenter
  DeselectAllAction.OnExecute := SelectContinentsPresenter
  SelectOneAction.OnExecute := SelectContinentsPresenter.S
  SelectAllAction.OnExecute := SelectContinentsPresenter.S
  ResetAction.OnExecute := SelectContinentsPresenter.Reset
  AvailableListBox.OnClick := SelectContinentsPresenter.Av
  AvailableListBox.OnDblClick := SelectContinentsPresenter
  SelectedListBox.OnClick := SelectContinentsPresenter.Sel
  SelectedListBox.OnDblClick := SelectContinentsPresenter.
end;

```

Actions skubbes til Presenter

```

procedure TSelectContinentsDialog.FormCreate(Sender: TObject)
begin
  FSelectContinentsPresenter := TSelectContinentsPresenter
  DeselectOneAction.OnExecute := SelectContinentsPresenter
  DeselectAllAction.OnExecute := SelectContinentsPresenter
  SelectOneAction.OnExecute := SelectContinentsPresenter.S
  SelectAllAction.OnExecute := SelectContinentsPresenter.S
  ResetAction.OnExecute := SelectContinentsPresenter.Reset
  SelectContinentsPresenter.DeselectOneAction := DeselectO
  SelectContinentsPresenter.DeselectAllAction := DeselectA
  SelectContinentsPresenter.SelectOneAction := SelectOneAc
  SelectContinentsPresenter.SelectAllAction := SelectAllAc
  SelectContinentsPresenter.ResetAction := ResetAction;
  AvailableListBox.OnClick := SelectContinentsPresenter.Av
  AvailableListBox.OnDblClick := SelectContinentsPresenter
  SelectedListBox.OnClick := SelectContinentsPresenter.Sel
  SelectedListBox.OnDblClick := SelectContinentsPresenter.
end;

```

Vi skal ikke lade os skræmme af at skubbe referencer frem, fordi det ikke er en usædvanlig måde at "binde" sig med andre komponenter på, og det ses oftere i sprog, hvor vi f.eks.

ikke har namespaces, eller på anden måde kan binde os på tværs af View, Presenter og Model. Vi kunne også have haft en reference til View i Presenters uses i implementations-del, så vi havde en cirkulær reference, der kunne gå ud for at være en erstatning for vores mulighed for namespaces, men vi vil ikke opnå en løs kobling.

Men det kan give mening at trække actions fra View gennem et interface, fordi vi kan fortsætte med almindelig refaktorering, når vi er færdige med at refaktorere mod et pattern. Hver action havde sin egen getter, men vi kan trimme et interface, så relaterede getters kan få en fælles getter i stedet.

Træk af actions fra View

```
type
  ISelectContinentsDialog = interface[{85899383-81A5-4B38
    function GetAvailableListBoxItemIndex: Integer;
    function GetAvailableListBoxItems: TStrings;
    function GetSelectedListBoxItemIndex: Integer;
    function GetSelectedListBoxItems: TStrings;
    function GetDeselectOneAction: TContainedAction;
    function GetDeselectAllAction: TContainedAction;
    function GetSelectAllAction: TContainedAction;
    function GetSelectOneAction: TContainedAction;
    function GetResetAction: TContainedAction;
  end;
```

Fra eksemplet "Continents 2 - 1"

Efter-refaktoret træk fra View

```
type
  TSelectContinentsAction = (scaDeselectOne, scaDeselectAll,
    scaSelectAll, scaReset);

  ISelectContinentsDialog = interface[{85899383-81A5-4B38
    function GetAvailableListBoxItemIndex: Integer;
    function GetAvailableListBoxItems: TStrings;
    function GetSelectedListBoxItemIndex: Integer;
    function GetSelectedListBoxItems: TStrings;
    function GetAction(const AAction: TSelectContinentsAct
  end;
```

Fra eksemplet "Continents 2 - 3"

Det er ikke bogens hensigt at gennemgå almindelig refaktorering, fordi refaktorering mod et pattern tager udgangspunkt i en-til-en, men det er ikke nødvendigvis slut efter dette, idet det er muligt at fortsætte med ganske almindelige refaktorering, som i optimering, oprydning, kvalitetsløft osv.

3) Separering med VCL/FMX i alle lag

Vi kan også stå i den situation, at vi vil have f.eks. TActionList i Presenter, og at det skal deles mellem et VCL- og et FMX-projekt. Det kan ikke lade sig gøre at have både VCL- og FMX-komponenter i en TDataModule, og derfor skal en komponent som TActionList genereres dynamisk. TActionList-komponenten findes både i en VCL- og en FMX-udgave, og uadtil ligner de hinanden, men de er ikke af samme klasse, og det skyldes navnlig at de hver især bruger en image-list, som også minder om hinanden, men er teknisk forskellige i VCL- og i FMX-frameworket.

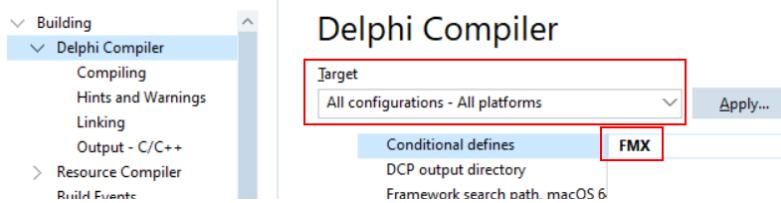
Denne øvelse tager udgangspunkt i løsningen fra "Continents 2 - 2", dvs. at vi har refaktoreret versionen, hvor vi havde en TActionList i en VCL- og FMX-View, så den i "Continents 3 - 1" kan genereres i Presenter, når Presenter også bliver generet. Den største udfordring er ikke så meget TActionList-komponenten, men at holde styr på alle de enkelt actions, som også skal genereres dynamisk.

Der findes ikke en entydig compiler-direktiv, der skelner mellem en VCL- og en FMX-projekt. Det anbefales, at vi selv tilføjer en compiler-direktiv til vores FMX-projekt, der klokkeklat definerer hvilke dele af vores kode, der skal kompileres. I vores eksempel er vi heldige, at vi blot behøver at tilføje et compiler-direktiv i vores uses, der bestemmer

hvilken TActionList compileren skal tage udgangspunkt i. TActionList findes i Fmx.ActnList og Vcl.ActnList:

```
uses
  System.SysUtils, System.Classes, System.Actions,
{$IFDEF FMX}
  Fmx.ActnList,
{$ELSE}
  Vcl.ActnList,
{$ENDIF}
  Continents.Selection.Model;
```

Compiler-direktivet skal vi have defineret i vores FMX-projekt, og det finder vi under menuen "Projekt / Options ..." og under "Building / Delphi Compiler" har vi et punkt, der hedder "Conditional defines". Target skal stå "All configurations - All platforms", fordi uanset hvilket platform det er, og uanset om det er debug eller release, så er det et FMX-projekt. Dvs. at vi kan tilføje et "FMX"-define under "Conditional defines":



Vi behøver ikke at tilføje et "VCL"-define for vores VCL-projekt, medmindre det vil være nødvendigt – ellers vil det også være lige så godt med et {\$ifndef FMX}.

Dette betyder, at vi selv må generere et TActionList, når Presenter bliver oprettet, men også de enkelte actions, ligesom deres events skal bindes til metoder og komponenter skal bindes til actions. Mange af disse kodelinjer, der før fandtes i Views constructor kan flyttes til Presenters constructor, hvor vi tillige har noteret os, hvordan de enkelte actions var konfigureret i View, før vi slettede dem. Derfor kan constructor også se overvældende ud med en mængde nye kodelinjer, men det kan være prisen – kodelinjer kunne refaktoreres ud til et særskilt metode.

Bemærk, at en TAction i VCL har en property, der hedder "Caption", som hedder "Text" i FMX, som er meget normalt i FMX. Caption-property er dog nedarvet fra System.Actions, og i FMX-udgaven af en TAction læser og skriver den også blot Text til Caption i getters og setters. Ydermere er Caption tilgængelig i public-sektionen, og derfor vil vi ikke være nødt til at veksle med en compiler-direktiv for at skelne mellem disse properties:

```

procedure TSelectContinentsPresenter.DataModuleCreate(Sender: TObject);
begin
  SelectContinentsDialog := Owner as ISelectContinentsDialog;
  FSelectActions := TActionList.Create(Self);
  FDeselectOneAction := TAction.Create(FSelectActions);
  FDeselectOneAction.Name := 'DeselectOneAction';
  FDeselectOneAction.OnExecute := DeselectOneActionExecute;
  FDeselectOneAction.Caption := '>';
  FDeselectAllAction := TAction.Create(FSelectActions);
  FDeselectAllAction.Name := 'DeselectAllAction';
  FDeselectAllAction.OnExecute := DeselectAllActionExecute;
  FDeselectAllAction.Caption := '>>';
  FSelectAllAction := TAction.Create(FSelectActions);
  FSelectAllAction.Name := 'SelectAllAction';
  FSelectAllAction.OnExecute := SelectAllActionExecute;
  FSelectAllAction.Caption := '<<';
  FSelectOneAction := TAction.Create(FSelectActions);
  FSelectOneAction.Name := 'SelectOneAction';
  FSelectOneAction.OnExecute := SelectOneActionExecute;
  FSelectOneAction.Caption := '<';
  FResetAction := TAction.Create(FSelectActions);
  FResetAction.Name := 'ResetAction';
  FResetAction.OnExecute := ResetActionExecute;
  FResetAction.Caption := 'Reset';
  FSelectContinentsModel := TSelectContinentsModel.Create(Self);
end;

```

Vi kan vælge at udskille initialisering af actions i en metode, men det er helt klart en type af kode, der kommer til at fylde meget i projekter på tværs af framework, og det kan også introducere fejl, idet vi kan risikere f.eks. at initialisere et objekt med forkert metode, så teknikken kræver stor disciplin. Der vil også være mange situationer, hvor forskellen mellem en VCL- og en tilsvarende FMX-komponent er så stor, at kodekonstruktioner vil bestå af mange compiler-direktiver.

4) Separering til ren FMX

Sidst, men ikke mindst, så kan vi også have en målsætning om helt at skrotte VCL-projektet, når vi engang er færdige med at refaktorer mod et pattern og senere migrerer til FMX-frameworket. Her kan det i også mange tilfælde betale sig at tage udgangspunkt i 2. eller 3. løsning, og rydde VCL-relatedede compiler-direktiver fra, når FMX-versionen er udviklet færdig. Med 2. løsningsmodel skal vi blot have fjernet VCL-projektet og det ene VCL-form, som er knyttet til, men vi kunne måske også omdøbe de tilbageværende filer, så der ikke længere fremstår, at de er specifikke FMX-filer – f.eks. "Continents.Selection.ViewFMX.pas" til "Continents.Selection.View.pas". Det er jo en ren FMX-projekt, og så er der behov for at markere, at det er en FMX-form. Den første løsning er baseret på "Continents - 2 - 2", hvor TActionList findes i View, og som er blevet ryddet for VCL-dele.

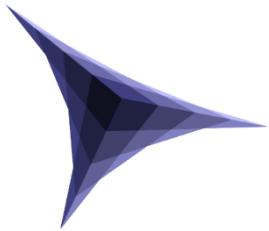
Den anden løsning er baseret på "Continents - 3 - 1", hvor TActionList genereres runtime i Presenter, men refaktoreret så TActionList håndteres under designtime med Delphi IDE. Compiler-direktiver, som skelner mellem VCL og FMX, er også ryddet, så der kun må være FMX-biblioteker i Presenter eller Model, hvis det er nødvendigt.

Opsummering

Som det tydeligt fremgår efter dette kapitel, så betyder det ekstrem meget, at vi har en målsætning med vores refaktorering, fordi det har en altafgørende betydning for, hvordan

og hvad vi flytter fra View til Presenter eller Model. Og selvom vi ikke skulle have FMX i tankerne, så vil der stadig være overvejelser om, om der bør være VCL-komponenter i Presenter og Model. Jo større et projekt, jo mere sandsynligt er det, at vi slet ikke kan komme udenom at skulle overveje VCL-komponenter i Presenter eller Model.

Næste kapitel handler også om andre overvejelser, vi bør gøre, når vi skal separere kodelinjer og komponenter, som vi har gjort de 2 forudgående kapitler. Det næste kapitel tager udgangspunkt i databaser og dataaware-komponenter, som i høj grad er typiske forudsætninger, når vi taler om Delphi-applikationer.



Kapitel 5

Data-aware og Refaktorering

Det næste eksempel vil komme tæt på et typisk Delphi-projekt, som ofte består af forms med nogle data-aware-komponenter og en eller flere datasets bestående af queries eller tables, samt nogle datasources til at binde komponenterne sammen. Data-aware-komponenter er en samlet betegnelse for komponenter, der er indrettet til, på forskellige måder og med forskellige midler, at fremstille en præsentation af aktuelle datasæt, oftest gennem en datasource-komponent, og reagere på statusændringer fra samme datasæt. I tidlige udgaver af Delphi var denne data-binding en simpel implementering af Observer Pattern, men implementeringen har i dag udviklet sig til bagvedliggende datalinks imellem data-aware-komponenter og datasources, der i forvejen fungerer som et binedet mellem data-aware-komponenter og datasæt.

Men netop at data-aware-komponenter typisk forudsætter data-binding, der er bygget ovenpå en Observer Pattern, gør dem vældig interessante på mange måde. Observer Pattern er en typisk tilgang, når vi taler om data-binding, eller binding generelt, og det er også den pattern, som der refereres til i Model-View-ViewModel (MVVM). Hvor vi i Model-View-Presenter (MVP) oftest vil ty til en løsning med et interface, så vil vores applikation ikke følge MVVM, blot i kraft af at vi bruger data-aware-komponenter, hvis resten af applikationen forlader sig på interfaces. MVVM dиктерer eksplisit Observer Pattern som det design pattern, man vil bruge i forbindelse med data-binding, ligesom MVVM vil tillade 1-til-mange Views (observers), i modsætning til MVP og interfaces, der er målrettet 1-til-1 forhold, men at de enkelte lag i øvrigt vil kunne ligne hinanden.

Med data-aware-komponenter kan vi allerede en fornemmelse af, hvor stærk Observer Pattern egentlig er som et design pattern. De enkelte komponenter ikke har nogen idé om, hvad det er for nogle "observatører", der har registreret sig til at modtage notifikationer, når der opstår events. Det skaber lidt overhead, fordi der sendes en masse notifikationer

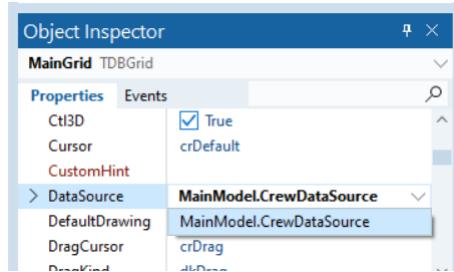
rundt til alle, hvor få måske kun er interesseret i nogle events, mens de ignoreres af andre. Disse detaljer vil vi kigge nærmere på i næste kapitel, men det er den måde Delphi's data-aware-komponenter er indrettet på, som også kan få betydning for, hvordan vi har separeret vores kodelinjer.

Data-binding betyder, at vi, i visse situationer, må åbne op for referencer direkte fra View til Model, enten fordi en komponent på View skal have en datasource eller en decideret dataset i Model som reference. Det er ikke unormalt at se MVC eller MVVM, hvor View har referencer til Model, ligesom det heller ikke vil være usædvanligt at se dette i MVP, hvor vi benytter data-binding i større og større udstrækning. De fleste data-aware-komponenter binder til en datasource, som i mange henseender sagtens kunne have været placeret i Presenter fremfor Model, og på den måde opretholde de naturlige referencer fra View til Presenter, og fra Presenter til Model. Vi kan dog komme ud for konstruktioner, hvor flere datasources indgår som f.eks. mastersources, og derfor nødvendigvis må holde disse komponenter samlet i en og samme Model, fordi det vil være imod disse patterns at referere til en datasource i Presenter som mastersource til en dataset i Model.

Derfor vil eksemplerne tage udgangspunkt i, at database-komponenter som datasources og datasets placeres i Model, og at data-aware-komponenter som TDBEdit og TDBGrids nødvendigvis må findes i View. Dermed har vi også indsnævret vores muligheder med referencer fra View til to muligheder – nemlig at vi enten har direkte referencer fra View til Model, eller at vi skubber og trækker properties til og fra Presenter, der refererer til f.eks. datasources og datasets i Model. Den sidste løsning kræver getters i Presenter, og at vi binder komponenterne med kodelinjer, men til gengæld kan vi opretholde en konsekvent tilgang til separationen, og omvendt vil den første løsning kræve færre kodelinjer, ligesom det kan udnytte data-binding til fulde, hvilket igen betyder at definering af interfaces også kan indsnævres.

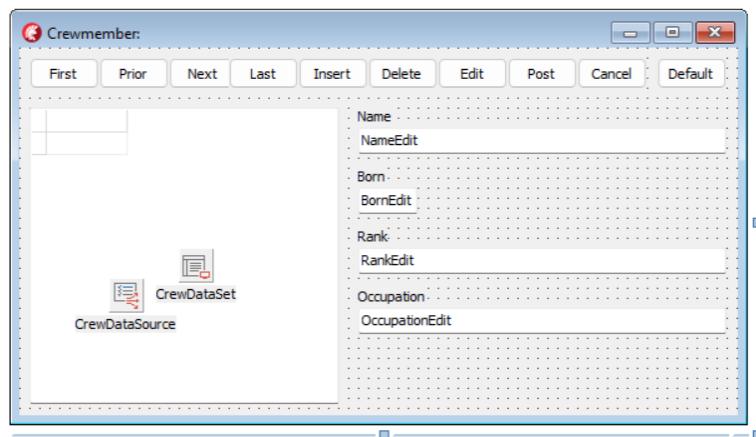
Data-binding mellem View og Model

Det er muligt at binde komponenter fra View direkte til en komponent i Model gennem Delphi IDE. Vi skal blot have en reference til Model i View's interface-sektion, og IDE skal have oprettet en design-instance af Model først – dvs. at Model ikke kun skal have haft sourcekoden åben i editoren, men også den form med komponenter, der udgør TDataModule, har været åben. Delphi IDE er i stand til at finde instancer af komponenter på tværs af units, hvor der er en reference i interface-sektionen, men det gælder desværre ikke for metoder på samme måde, så events i komponenter ikke kan bindes til metoder i andre units fra IDE under design-time.



Denne mulighed betyder, at vi ikke behøver kodelinjer for at binde komponenter sammen på tværs af units, men at de bindes på sædvanlig vis, når forms og moduler oprettes og læses ind. Der er ikke noget teknisk i vejen for, at formen bliver oprettet før modulerne bliver oprettet, fordi bindingen håndteres internt under load, men det betyder stadig at oprettelsen af forms og moduler håndteres fornuftigt, og at deres constructors ikke bruges til andet end at initialisere data i den rækkefølge, som oprindeligt.

Eksemplet med dette projekt finder vi både databinding mellem TDBGrid og TDatasource, men også en masse TButton med traditionelle OnClick-events, både fordi vi ofte ser disse konstruktioner, men også fordi vi kan refaktorere os ud af sådanne OnClick-events og gøre rent bord i View uden at vi ændrer funktionalitet af vores applikation, som vi vil kigge nærmere på senere. I første omgang vil vi fokusere på at separere formen, med metoder og kodelinjer, som de er.



Når vi kigger på det oprindelige projekt i folderen "Datasets", så er der ikke noget usædvanligt i hverken formen eller sourcekoden – der er bevidst brugt knapper, i stedet for en TDBNavigator, for at fremhæve nogle typiske udfordringer i forbindelse med separeringen. Og der er brugt en simpel dynamisk dataset, så vi slipper for at skulle finde en flad database frem. Sourcekoden fylder et par hundrede kodelinjer, og består af typiske konstruktioner, når vi taler om forms med database-interaktioner. Der er ikke brugt actions, og det skal også ses i sammenhæng med refaktorering mod lidt mere klassisk linking, udover data-binding, som kan skabe en kortere vej mod noget, der ligner MVVM.

```

procedure TMainForm.CrewDataSourceDataChange(Sender: TObject; Field: TField);
begin
  FirstButton.Enabled := CrewDataSet.Active and not CrewDataSet.Bof;
  PriorButton.Enabled := FirstButton.Enabled;
  NextButton.Enabled := CrewDataSet.Active and not CrewDataSet.Eof;
  LastButton.Enabled := NextButton.Enabled;
  DeleteButton.Enabled := CrewDataSet.Active and not (CrewDataSet.Bof and Crew
    if Field = CrewDataSetName then
      UpdateCaption;
  end;

procedure TMainForm.CrewDataSourceStateChange(Sender: TObject);
var
  LEditing: Boolean;
begin
  LEditing := CrewDataSet.State in [dsEdit, dsInsert];
  InsertButton.Enabled := CrewDataSet.Active and not LEditing;
  EditButton.Enabled := CrewDataSet.Active and not LEditing;
  PostButton.Enabled := CrewDataSet.Active and LEditing;
  CancelButton.Enabled := CrewDataSet.Active and LEditing;
end;

function TMainForm.CrewMemberExists(const AName: string): Boolean;
begin
  Result := CrewDataSet.Locate('Name', AName, [loCaseInsensitive]);
end;

procedure TMainForm.DefaultButtonClick(Sender: TObject);
begin
  DefaultCrew;
end;

```

Actions og linking

Vores Dataset-eksempel består af nogle metoder, som kan virke banale for de øvede, men er konstrueret på en måde, så de kan tjene som eksempel – de kan helt klart skrives mere fornuftigt og mere formålsrettet. Hvis vores første refaktorering er rettet mod at separere formen til en View, en Presenter og en Model, så vil resultatet se nogenlunde ud som i folderen "Datasets 1".

```

uses
  Datasets.Main.Model;

procedure TMainView.CrewDataSourceDataChange(Sender: TObject; Field: TField);
begin
  FirstButton.Enabled := MainModel.CrewDataSet.Active and not MainModel.CrewDa
  PriorButton.Enabled := FirstButton.Enabled;
  NextButton.Enabled := MainModel.CrewDataSet.Active and not MainModel.CrewDat
  LastButton.Enabled := NextButton.Enabled;
  DeleteButton.Enabled := MainModel.CrewDataSet.Active and not (MainModel.Crew
end;

procedure TMainView.CrewDataSourceStateChange(Sender: TObject);
var
  LEditing: Boolean;
begin
  LEditing := MainModel.CrewDataSet.State in [dsEdit, dsInsert];
  InsertButton.Enabled := MainModel.CrewDataSet.Active and not LEditing;
  EditButton.Enabled := MainModel.CrewDataSet.Active and not LEditing;
  PostButton.Enabled := MainModel.CrewDataSet.Active and LEditing;
  CancelButton.Enabled := MainModel.CrewDataSet.Active and LEditing;
end;

procedure TMainView.UpdateCaption;
begin
  Caption := SDisplayCaption + MainModel.CrewDataSetID.AsString +
    ' ' + MainModel.CrewDataSetName.AsString;
end;

```

Den 1-1 refaktorering, der lå lige til højrebenet, viser det behov, der opstår, når nogle kodelinjer har brug for referencer fra View til Model. Det er ikke fordi verden går under, eller det må vi ikke, men jo færre referencer, jo mindre afhængigheder. Derfor ser vi jo helst gerne, at der var så få referencer fra View til Model så muligt. Én ting er, at vores data-aware-komponenter skal bindes til en datasource i Model, og her har vi brug for en reference i uses, men en anden ting er referencer til Model i kodelinjer. Fordi vi har brugt events i buttons og manipulerer formens buttons afhængig af status i vores dataset, så opstår disse kodelinjer med referencer. Derfor får vi også kodelinjer, der tildeler knapperne de samme metoder til events, der nu er flyttet til Presenter.

```

procedure TMainView.FormCreate(Sender: TObject);
begin
  TMainPresenter.MainView := Self;
  MainPresenter := TMainPresenter.Create(Self);
  FirstButton.OnClick := MainPresenter.FirstButtonClick;
  PriorButton.OnClick := MainPresenter.PriorButtonClick;
  NextButton.OnClick := MainPresenter.NextButtonClick;
  LastButton.OnClick := MainPresenter.LastButtonClick;
  InsertButton.OnClick := MainPresenter.InsertButtonClick;
  DeleteButton.OnClick := MainPresenter.DeleteButtonClick;
  EditButton.OnClick := MainPresenter.EditButtonClick;
  PostButton.OnClick := MainPresenter.PostButtonClick;
  CancelButton.OnClick := MainPresenter.CancelButtonClick;
  DefaultButton.OnClick := MainPresenter.DefaultButtonClick;
end;

```

Vi kan tilføje vores løsning med en TActionList i Presenter, som indkapsler de metoder, som formens buttons gør brug af. TActionList og action-linking fungerer på samme måde som data-binding, i og med at kommunikationen går begge veje, men fremfor data så opererer action-linking med en handling, der kan eksekveres af f.eks. en knap, men også at knappen kan ændre tilstand afhængig af den action, der er linket til knappen. Vi ændrer ikke funktionaliteten af vores metoder, udover at vi for god ordens skyld ændrer navnene på vores metoder, så navnene ikke længere vil være sammensat af "Button" og "Click", men f.eks. "Action" og "Execute".

Presenter før

```

public
  procedure FirstButtonClick(Sender: TObject);
  procedure PriorButtonClick(Sender: TObject);
  procedure NextButtonClick(Sender: TObject);
  procedure LastButtonClick(Sender: TObject);
  procedure InsertButtonClick(Sender: TObject);
  procedure DeleteButtonClick(Sender: TObject);
  procedure EditButtonClick(Sender: TObject);
  procedure PostButtonClick(Sender: TObject);
  procedure CancelButtonClick(Sender: TObject);
  procedure DefaultButtonClick(Sender: TObject);
  class var MainView: IMainView;
end;

```

Fra eksemplet "Datasets - 1"

Presenter efter

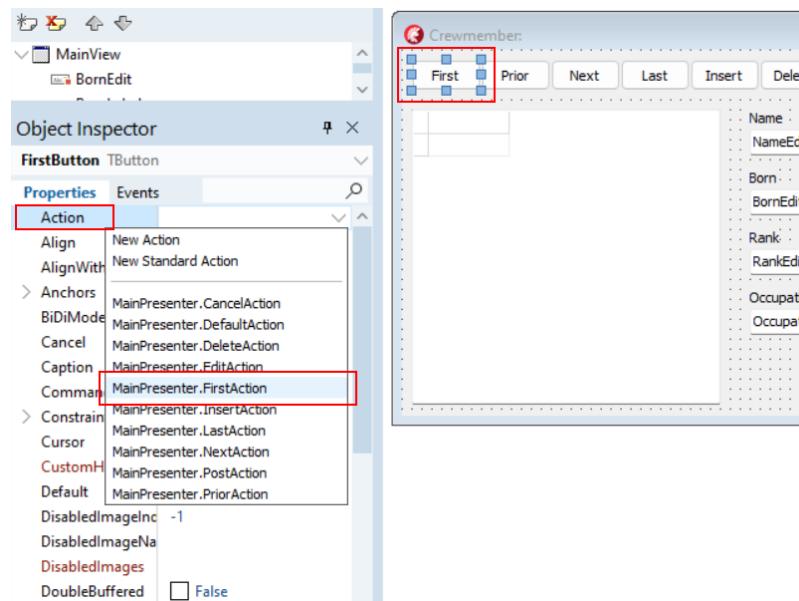
```

public
  class var MainView: IMainView;
published
  procedure FirstActionExecute(Sender: TObject);
  procedure PriorActionExecute(Sender: TObject);
  procedure NextActionExecute(Sender: TObject);
  procedure LastActionExecute(Sender: TObject);
  procedure InsertActionExecute(Sender: TObject);
  procedure DeleteActionExecute(Sender: TObject);
  procedure EditActionExecute(Sender: TObject);
  procedure PostActionExecute(Sender: TObject);
  procedure CancelActionExecute(Sender: TObject);
  procedure DefaultActionExecute(Sender: TObject);
end;

```

Fra eksemplet "Datasets 2"

Det er en god praksis at følge op på de enkelte actions i en TActionList med navne, som matcher de metoder, som vi lige har omdøbt, f.eks. FirstAction, ikke kun fordi vi kan relaterede nye navne med de gamle, men også fordi vi blot kan dobbeltklikke på den enkelte action i vores TActionList-editor, så tildeler IDE dennes OnExecute-event til en matchende metode, fordi navnet er identisk med det navn, som den selv ville have autogenret. Også selvom metoden ligger i public-sektionen, men de skal flyttes til samme sektion som DataModulCreate-metoden eller en published-sektion – ellers vil indlæsningen af modulet fejle, fordi metoderne ikke vil være "synlige" – published. Når vi har tilføjet et tilsvarende antal actions, der svarer til de metoder, vi har, og vi ikke har ændret i koden, så skal vi blot binde de enkelte knapper i vores View til de respektive actions i vores Presenters, og fjerne de kodelinjer, der tidligere bandt knappernes events til metoderne.



Det betyder også, at de kodelinjer, der bearbejder knapperne, kan flyttes til Presenter, og bearbejde de respektive actions i stedet for knapperne. Vi har lagt et lag af action-links imellem, men funktionaliteten er stadig den samme. Vi kunne også have valgt TActionManager, og det vil være helt op til det enkelte projekt, men TActionList var den simple udgave til et simpelt eksempel.

Samme kode i Form før

```

procedure TMainForm.CrewDataSourceDataChange(Sender: TObject);
begin
  FirstButton.Enabled := CrewDataSet.Active and not CrewDataSet.ReadOnly;
  PriorButton.Enabled := FirstButton.Enabled;
  NextButton.Enabled := CrewDataSet.Active and not CrewDataSet.ReadOnly;
  LastButton.Enabled := NextButton.Enabled;
  DeleteButton.Enabled := CrewDataSet.Active and not (CrewDataSet.ReadOnly or CrewDataSet.State in [dsEdit, dsInsert]);
  if Field = CrewDataSetName then
    UpdateCaption;
end;

procedure TMainForm.CrewDataSourceStateChange(Sender: TObject);
var
  LEditing: Boolean;
begin
  LEditing := CrewDataSet.State in [dsEdit, dsInsert];
  InsertButton.Enabled := CrewDataSet.Active and not LEditing;
  EditButton.Enabled := CrewDataSet.Active and not LEditing;
  PostButton.Enabled := CrewDataSet.Active and LEditing;
  CancelButton.Enabled := CrewDataSet.Active and LEditing;
end;

function TMainForm.CrewMemberExists(const AName: string): Boolean;
begin
  Result := CrewDataSet.Locate('Name', AName, [loCaseInsensitive]);
end;

procedure TMainForm.DefaultButtonClick(Sender: TObject);
begin
  DefaultCrew;
end;

```

Fra eksemplet "Datasets - 1"

Samme kode i Presenter efter

```

procedure TMainPresenter.CrewDataSourceDataChange(Sender: TObject);
begin
  FirstAction.Enabled := MainModel.CrewDataSet.Active and not MainModel.CrewDataSet.ReadOnly;
  PriorAction.Enabled := FirstAction.Enabled;
  NextAction.Enabled := MainModel.CrewDataSet.Active and not MainModel.CrewDataSet.ReadOnly;
  LastAction.Enabled := NextAction.Enabled;
  DeleteAction.Enabled := MainModel.CrewDataSet.Active and not (MainModel.CrewDataSet.ReadOnly or MainModel.CrewDataSet.State in [dsEdit, dsInsert]);
  if Field = MainModel.CrewDataSetName then
    MainView.UpdateCaption;
end;

procedure TMainPresenter.CrewDataSourceStateChange(Sender: TObject);
var
  LEditing: Boolean;
begin
  LEditing := MainModel.CrewDataSet.State in [dsEdit, dsInsert];
  InsertAction.Enabled := MainModel.CrewDataSet.Active and not LEditing;
  EditAction.Enabled := MainModel.CrewDataSet.Active and not LEditing;
  PostAction.Enabled := MainModel.CrewDataSet.Active and LEditing;
  CancelAction.Enabled := MainModel.CrewDataSet.Active and not LEditing;
end;

procedure TMainPresenter.DefaultActionExecute(Sender: TObject);
begin
  MainModel.DefaultCrew;
end;

```

Fra eksemplet "Datasets 2"

Efter denne refaktorering, så efterlader det os en meget "tynd" View, som består af ganske få kodelinjer, samt et lille interface til View, som Presenter behøver at definere. Vi kan sikkert finde løsninger, der fjerner de tilbageværende kodelinjer, men hovedvægten er, at View bør stå tilbage med de kodelinjer, som ikke umiddelbart lader sig flytte, men som sikkert kan refaktoreres på anden måde – på et andet tidspunkt. Vi har stadig referencer til Model fra View, og vi kunne sikkert også finde løsninger med at trække de samme data gennem properties i Presenter med nogle getters.

```

procedure TMainView.FormCreate(Sender: TObject);
begin
  TMainPresenter.MainView := Self;
  TMainPresenter := TMainPresenter.Create(Self);
end;

procedure TMainView.UpdateCaption;
begin
  Caption := SDisplayCaption + MainModel.CrewDataSetID.AsString +
  ' ' + MainModel.CrewDataSetName.AsString;
end;

```

Denne løsning kan lade sigøre, fordi vi kan udnytte muligheder for at binde både data til data-aware-komponenter og actions til knapperne. Vi kunne have taget en TDBNavigator i anvendelse i det oprindelige form og fået samme resultat som med knapperne, der nu er bundet til actions, men så havde vi ikke et eksempel, vi kunne refaktorere og fremhæve muligheder med også at binde actions fra View til Presenter.

Denne refaktorering viser også, hvor stor en hjælp både data-binding og action-link egentlig kan udgøre, og at vi faktisk ikke er så langt fra en MVVM-pattern, hvis vi kan inddarbejde den teknik i vores applikationer, fordi vi kan have flere data-aware-komponenter til samme datasource og vi kan have flere knapper og menuer til samme action-link. Vores Separation of Concerns minder mest om MVP, fordi vores separation tager udgangspunkt i en 1-1 forhold mellem lagene, i modsætning til MVVM, som vil kunne håndtere flere Views. Separationen kunne lige så godt have taget sigte i en MVC, men vores View er entry point, og vores Presenter kan godt tage mere traditionelle kodelinjer for præsentering af data om nødvendigt, udover at håndtere actions.

Model-View-ViewModel

Det er ikke helt tilfældigt, at vi refaktorerer henimod at gøre brug af eksisterende features som data-binding og action-linking, fordi det vil give os færre udfordringer, når vi skal til at fortsætte refaktoreringen frem mod Model-View-ViewModel (MVVM). Med refaktoreringen mod MVVM kan vi udelukkende fokusere på de definitioner af interfaces, der især er brugt til at binde View med Presenter, men også mellem Presenter og Model om nødvendigt. Betragtet på den måde, så vil refaktoreringen mod MVP have givet os den separering, der er nødvendig, men også den "opskrift" i form af interface-definitioner, som vil være nødvendige for at binde View og Presenter.

Med MVVM er det standarden med data-binding mere end det er normalen, også med mange Views til en enkel Model, hvilket også betyder at vores interface definition ikke kunne række, da det er en 1-1-løsning, som også er normalen for MVP. MVP er tilstrækkelig for separering af klassiske Delphi-forms, men hvis målet er MVVM, så er MVP et godt udgangspunkt. Udoer data-binding, så har MVVM dog en anden karakteristisk egenskab, nemlig at ViewModel har ansvaret for eksponeringen af data fra Model på en sådan måde, at de let kan håndteres og præsenteres. Dvs. at det ikke vil give meget mening, at View har data-bindinger direkte fra View til Model – det var bekvemt, men det er ViewModel's opgave at tilvejebringe dette.

Hvis vi på forhånd ved, at vi vil fortsætte refaktoreringen mod MVVM, så har vi også mulighed for at tage højde for dette i vores refaktorering frem mod MVP, fordi vi kan oprettholde reglen om, at View ikke må have referencer til Model i sin uses. Dvs. sige at, udoer data-binding af data-aware-komponenter til en datasource i model, vi også skal finde en løsning på de kodelinjer, der har referencer til Model i deres statements.

Vi har to muligheder, nemlig at vi har placeret DataSource i vores Model, som var det mest åbenlyse, og vi skaber nogle properties i Presenter, der har getters, der henter relevant data fra Model – som i løsning "Datasets - 3a". Problemet er bare, at vi ikke kan binde data-aware-komponenterne under design-time, fordi DataSource er en property i Presenter, og ikke en instance i Presenter. Derfor er vi også nødt til at binde alle data-aware-komponenter til en DataSource under View's constructor, og det giver nogle ekstra kodelinjer.

Den anden mulighed er, at vi flytter CrewDataSource-komponenten fra Model til Presenter, fordi denne DataSource reelt ikke har nogen funktion i vores eksempel. Vi kan komme ud for, at en DataSource benyttes som MasterSource i nogle konstruktioner, men i de situationer kan det måske også refaktoreres på anden måde. De kodelinjer, som DataSource eksekverer under data- og status-change, er udelukkende koncentreret i Presenter, hvorfor det er nærliggende at overveje, at DataSource skal ligge i Presenter. Det er muligt at sætte CrewDataSource.DataSet til MainModel.CrewDataSet, så det ændrer ikke på DataSources funktionalitet, og metoderne til data- og status-change findes allerede, så de kan også sættes under design-time.

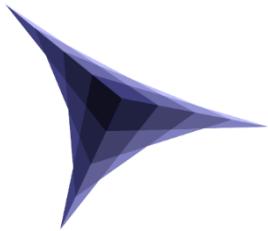
Model's definering af interfacet for Presenter kan skæres ned, fordi data- og status-change ikke længere bliver kaldt fra Model. Skulle det have været nødvendigt, fordi vi havde noget forretningslogik i Model, der skulle eksekveres, så kunne vi også have bundet events i DataSource under Presenter til metoder i Model som en mulighed, men events fra dataset kan som regel være tilstrækkelige.

Bemærk, at vi ikke "inddæmmer" forholdet mellem Presenter og Model på samme måde.

Vi har fjernet referencerne fra View til Model, og det eneste, der forhindrer os i at kalde separeringen for MVVM, er det interface i Presenter, som View nødvendigvis må implementere. Når vi taler om MVVM, så taler vi også om Observer Pattern, og MVVM er en af de få architectural patterns, som foreskriver en bestemt design pattern til kommunikation mellem objekter. Observer Pattern tilføjer noget kompleksitet i en applikation, men de simpleste implementeringer af Observer Pattern er ikke så komplicerede, når vi holder os til det niveau, som vi allerede kender fra interfaces.

De bagliggende biblioteker i Delphi består af mange forskelligartede implementeringer af forskellige udgaver af Observer Pattern, lige fra data-links til action-links, edit-link-observers osv., og deres kompleksitet med fleksibilitet og vokser tværs over klasser og units, og kendes især bedst fra udvikling af komponenter.

Før vi implementerer et Observer Pattern i vores eksempel, så gennemgår vi først principperne bag dette pattern i næste kapitel. Observer Pattern kan bruges i mange sammenhænge, både lokalt og på tværs af en hel applikation.

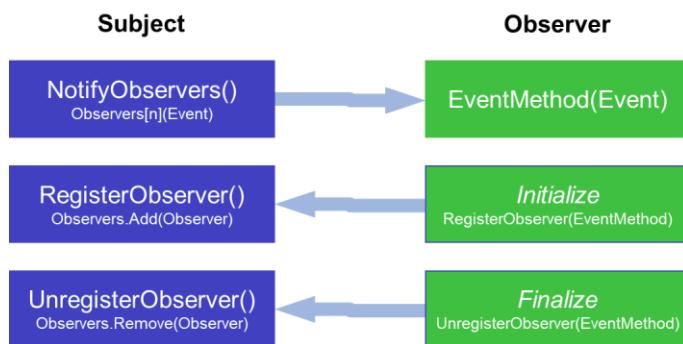


6

Kapitel

Observer Pattern

Før vi fortsætter refaktoreringen mod en MVVM pattern, så skal vi også kunne beherske en ganske bestemt software design pattern, der hører til behavioral patterns. I modsætning til et pattern som MVP, som har en 1-1 relation mellem komponenterne, og som meget hurtigt kan løses med definering af et interface, så kan et pattern som MVVM have én-til-mange relationer. En *subject*, som har implementeret dette pattern, er ofte letgenkendelige ved, at den har brugt nogle gennemgående navne på metoder, som kan afsløre dette pattern. Det er ikke altid, at disse navne benyttes konsekvent, og i forskellige miljøer er der tradition for at benytte et sæt af navne som Attach, Detach og Notify, ligesom navne som AddObserver og RemoveObserver også ses, men i vores eksempel vil vi være helt åbenlyse, som det ses i de blå bokse herunder:



Der er ikke nogen definition på, hvordan en observer skal notificeres, så en observer kan være alt fra en callback, en metode, et interface, en klasse, eller en mellemliggende klasse, som fungerer som en link, eller noget helt sjette. Vi vil ofte se en observer, der blot består af en metode, der bliver kaldt med forskellige events som argument, hvor et interface lige så godt kunne have ligget lige til højrebenet. Med interfaces er der desværre nogle faldgruber i Delphi, når vi mikser objekter og interfaces sammen, f.eks. i en liste over observers i form af referencer til interfaces, som jo ville være nødvendige. En observer vil være et objekt i sin egen ret, men den sædvanlige reference i applikationen indgår ikke i den

normale *reference counting* med dens interface-referencer. Derfor vil vi koncentrere os om en simpel løsning med metoder, men forsøge os med interfaces senere i dette kapitel. Presenter kender ikke View, så klasser er udelukket, og det er af samme årsag, at mange frameworks oftest vil benytte sig af en mellemliggende klasse, der agerer som en link, som vi også ser det mange steder i Delphi.

Metoder er også smidige nok til Observer Pattern, fordi vi kan definere en metode med et argument, der bestemmer hvilken handling, der har udløst notifikationen. En metode kan også suppleres med et eller flere argumenter, der ledsager handlingen, f.eks. en værdi eller et data-objekt, eller begge dele, som betyder noget bestemt ud fra handlingen, hvilket også betyder at de kan være tomme. Til vores eksempler om observere har vi sammensat et projekt, der består af en form og nogle forskellige og tilfældige typer af visuelle komponenter, der i deres helhed går ud for at være et vindue med indstillinger. De indstillinger kan ændres, og målet er så, at der kan kobles et utal af observere, der blot venter på at der sker ændringer. Erklæring af en metode kan f.eks. se således ud:

```
type
  TMainSettingsEvent = (meDestroying, meEnabledChanged,
    meSpinChanged, meTrackChanged, meColorChanged, meDateChanged);
  TMainSettingsObserver = procedure(const AEvent: TMainSettingsEvent) of object
```

Og det, der kan gøre en implementering af Observer Pattern særligt letgenkendeligt, er den del af kodekonstruktionen, der giver adgang til at registrere sin observer:

```
private
  FObservers: TMainSettingsObserverList;
  procedure NotifyObservers(const AEvent: TMainSettingsEvent);
public
  procedure RegisterObserver(const AObserver: TMainSettingsObserver);
  procedure UnregisterObserver(const AObserver: TMainSettingsObserver);
end;
```

Hvordan vi håndterer en liste over observere, og hvordan vi implementerer metoderne, er forså vidt op til den praksis, som applikationen anvender, dvs. at det nødvendigvis ikke behøver at være en generisk liste, men andre teknikker, der er praktisk anvendelige. Eksemplet i bogen er en simpel implementering af Observer Pattern, og det er selvfølgelig muligt at udvikle en implementering, der passer til et bestemt behov, f.eks. at *thread safe*, fordi Observer Pattern også er anvendelig i konstruktioner, hvor en særskilt thread varetager nogle opgaver og sender notifikationer efterhånden som opgaverne er løst. Dette kan også tage udgangspunkt i en simpel implementering, og som stadig har de samme grænseflade udadtil med RegisterObserver og UnregisterObserver.

Observer Pattern er dog designet til at notificere observere synkront, og venter på at observeren er færdig, før subject kan fortsætte. De fleste tilgængelige eksempler på Observer Pattern viser da også implementeringer som synkront og i *main thread*. Det betyder ikke, at der kan ske alt muligt under eksekveringen af en applikation – også uforudsete situationer, som vi også vil give et par eksempler på senere i dette kapitel.

De to nævnte metoder er de vigtigste, ligesom definitionen på den metode, som en observer skal aflevere, og hvordan implementeringen ser ud, kunne en observer være fuldstændig ligeglads med. Observer er interesseret i de events, der kan blive sendt ud, hvornår det sker, og hvad de forskellige events betyder. Derfor vil en implementering stadig være interessant, hvis det er den eneste dokumentation vi har i applikationen. Der er måske ikke så mange konkrete eksempler på Observer Pattern til Delphi, så disse er blot endnu et par tilføjelser i den sammenhæng. En implementering af Observer Pattern for vores eksempel kunne se således ud som herunder.

Lige præcis denne implementering, og mange andre eksempler vi ser med dette pattern, er designet til at køre i *main thread* og synkront, og en sådan implementering vil virke ganske fortrinligt, så længe der ikke opstår uønskede hændelser i applikationen, som implementeringen ikke har haft kontrol over. Derfor vil der også være nogle, som vil være bekymrede for, at en observer kan lave så meget ravage i den, at man vil indkapsle notifikationerne i et try/except-blok, mens andre vil tænke, at der i så fald må være noget andet dyberliggende årsag til, at der går noget galt, og at det ikke er en opgave for dette design pattern.

```

procedure TSettingsForm.NotifyObservers(const AEvent: TMainSettingsEvent);
var
  LObservers: TList<TMainSettingsObserver>;
  I: Integer;
begin
  if EnabledCheckBox.Checked and (FObservers <> nil) then
  begin
    LObservers := TMainSettingsObserverList.Create;
    try
      LObservers.AddRange(FObservers);
      for I := 0 to LObservers.Count - 1 do
        LObservers[I](AEvent);
    finally
      LObservers.Free;
    end;
  end;
end;

procedure TSettingsForm.RegisterObserver(const AObserver: TMainSettingsObserver);
begin
  if FObservers = nil then
    FObservers := TList<TMainSettingsObserver>.Create;
  if FObservers.IndexOf(AObserver) = -1 then
  begin
    FObservers.Add(AObserver);
    AObserver(meEnabledChanged);
  end;
end;

procedure TSettingsForm.UnregisterObserver(const AObserver: TMainSettingsObserver);
begin
  if FObservers <> nil then
    FObservers.Remove(AObserver);
end;

```

Men en metode som NotifyObservers introducerer andre banale udfordringer, bl.a. fordi vi bruger en løkke, der notificerer de enkelte observere i en liste fra den første til den sidste. En observer kan vælge at afmelde sin registrering på et hvilket som helst tidspunkt, også midt i en notifikation, hvorfor tællerne i vores løkke ikke længere vil være retvisende. Eksemplet herover viser, at der er taget højde for at et eller flere observere kan melde fra

eller til, og det er ikke et usædvanligt scenarie. Det betyder bare at der vil være situationer, hvor vi tager vores forholdsregler, ligesom vi ville gøre med subjects og observers, hvis vi vælge at bruge dette pattern med forskellige threads. Vi kunne have talt løkken baglæns, men det vil også være en afvejning af, om de første observere skal have besked først, eller omvendt. Det er normalt, at de første registrerede observere får notifikationerne først.

Det vil også være almindeligt at definere et flag, typisk Enabled, der indikerer om notifikationer er aktiverede eller ej, og i vores eksempel bruger vi blot en checkbox til at slå notifikationer fra og til. Sædvanligvis ville vi have haft en variabel i en mere formaliseret erklæring, og samtidigt må observere også have notifikationer om netop ændringer i Enabled, desuagtet at notifikationer er slået fra. Det vil ofte løses ved, at vi skulle skelne mellem ændringer i status og ændringer i data.

Tæt kobling

Når vi giver os i kast med Observer Pattern, så hænger det sammen med dels muligheden for at have mange observere, men at dette pattern også giver mulighed for at opnå afkobling af forskellig grad mellem de enkelte komponenter. Det er klart, at en observer på et eller andet niveau må have et kendskab til et subject, og hvad den kan tilbyde, men en observer kan have kendskab til en konkret subject eller gennem et defineret interface, som en observer kan få gennem et mellemled, mens subject ikke vil have noget kendskab til konkrete observere. En subject vil højest kunne kende til antallet af observere, og den har en reference til den metode, den skal kalde. I hvilken klasse, metoden er defineret i, eller hvad og i hvilken sammenhæng notifikationen vil blive brugt på, eller om den overhovedet bliver brugt, har subject ikke nogen idé om.

En anden egenskab, som også har indflydelse på, hvor meget en observer kender til en subject, er selvfølgelig den information, som en observer vil være interesseret i. Vi kan enten skubbe data med i notifikationen til de enkelte observere, eller vi kan lade de enkelte observere trække den information, en observer måtte være interesseret i, fra subject. Normalt sendes der en værdi og/eller et data-objekt sammen med notifikationen, som kan antage forskellige typer af værdier afhængig af det event, som udløste notifikationen, men i refaktoreringssammenhænge kan det i praksis give mening, i hvert fald i første omgang, at lade observere trække data fra en konkret subject, fordi vores kodelinjer i forvejen mere sandsynligt vil bære præg af referencer til de objekter, de allerede interagerer med.

Med andre ord, så kan vi udnytte de eksisterende referencer i den applikation, som vi er i gang med at refaktorere, ligesom når vi definerer et interface i vores refaktorering mod MVP på baggrund af eksisterende referencer, uden at vi er nødt til at ændre markant på bestående kodelinjer. Alligevel vil vi følge op på vores observer-eksempel med en endnu højere grad af afkobling, hvis vi senere skulle have mod på at prøve kræfter med en mere afkobling. I hvert fald kan Observer Pattern i samme eksempel ses med yderligere to forskellige implementeringer, og give et indtryk af de muligheder, der ligger i dette pattern.

```
|| SettingsForm.RegisterObserver(MainFormEvent);
```

Indtil videre er afkoblingen ikke større end, at observer har en reference til en konkret MainForm (subject) i vores første eksempel, og registrerer sig selv, hvor det giver mening – dvs. i observerens form-constructor, men det kan være et andet sted hvor en observer kommer til kendskab om en subjects dannelse. RegisterObserver tager observerens metode, MainFormEvent, som et argument, og det vil være den metode, hvor observeren vil modtage events fra MainForm. Metoden vil modtage samtlige notifikationer sammen med en event-type, hvorfor metoden i praksis vil fungere som omdeler hos observeren. Herunder er f.eks. fremhævet et event, når en kalender-komponent i MainForm har ændret sig:

```
procedure TObserverForm.DateChanged;
begin
  DateEdit.Text := DateTimeToStr(MainForm.MonthCalendar.Date);
end;

procedure TObserverForm.MainFormEvent(const AEvent: TMainFormEvent);
begin
  case AEvent of
    mfEnabledChanged: EnabledChanged;
    mfSpinChanged: SpinChanged;
    mfTrackChanged: TrackChanged;
    mfColorChanged: ColorChanged;
    mfDateChanged: DateChanged;
  end;
end;
```

DateChanged-metoden består således af en kodelinje, hvor den selv henter relevant information fra MainForm, og det er den tilgang som mest sandsynligt vil læne sig op ad de kodelinjer, vi er ved at refaktorere mod MVVM. Vi ville være nødt til at lave om på både MainForm og observere, hvis Observer Pattern implementeres med en højere grad af afkobling, som vi vil se i senere eksempler. Forskellen mellem ovenstående og en metode, der skubber en værdi med notifikationen, er væsentlig, fordi referencen til en konkret MainForm kan fjernes, dvs. at observeren også kan afkobles fra subject i endnu højere grad end vi har brugt med eksplisitte referencer. Samtidig kan metoder til notifikationer gøres mere fleksible, så den værdi, der skubbes med, kan tolkes afhængig af eventet.

Begge disse tilgange er brugt i de første 2 eksempler, vi bringer med om Observer Patterns, hvor den sidste også gør brug af et interface. Der er et tredje eksempel på, hvordan observer og subject kan kobles helt fra hinanden, dvs. at hverken observer eller subject kender hinanden. I det sidste eksempel vil en subject tilbyde sin service gennem en central mellemled, og så er vi meget tæt på at have en Mediator Pattern, fordi en mediator sagtens kan implementeres med en Observer Pattern. I det mellemled kan en observer registrere sin metode, hvilket vil sige at det er mellemleddet, der håndterer registreringen og notifikationerne. Med andre ord, så kan observere også registrere sig selv før en subject overhovedet er tilgængelig. Det er en tilgang, vi vil kunne bruge i nyere projekter, når vi taler om løs kobling, men som normalt vil være svær at opnå gennem refaktore-ring.

Notifikationer og events

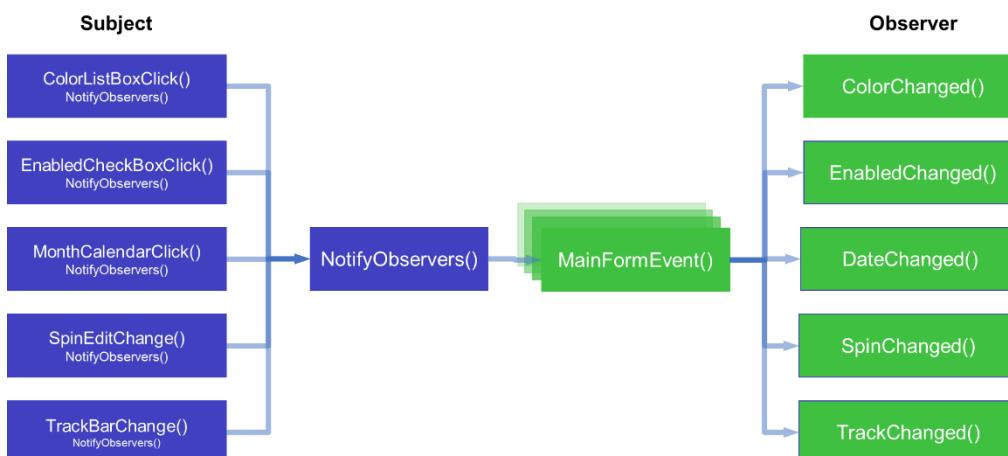
Notifikationer skal vi kunne sende og modtage på tidspunkter, hvor vi typisk har klassiske events i Delphi, som kan have interesse for en hvilken som helst observer, f.eks. MonthCalendarClick – altså at en dato har ændret sig. De typer af events, som observere også skal kende, skal selvfølgelig være defineret sammen med den metode, som observerne kan registrere.

```
type
  TMainSettingsEvent = (meDestroying, meEnabledChanged,
    meSpinChanged, meTrackChanged, meColorChanged, meDateChanged);
  TMainSettingsObserver = procedure(const AEvent: TMainSettingsEvent) of object
```

En metode behøver ikke at være mere avanceret end dette, og vi vil kunne opnå hovedmålet med MVVM, nemlig at vi kan have mange observere, der lytter til samme events. Dvs. at vi behøver ikke en notifikation ledsaget af en værdi, som observeren modtager med et event, f.eks. at en dato har ændret sig, men det betyder bare at observeren må trække data ud af en konkret subject, og vi vil da kun opnå en meget lidt grad af løs kobling. Der hviler således en stor grad af fleksibilitet i definering af notifikationer, der kan tilpasses helt individuelt og til et formål, der passer en situation.

```
procedure TSettingsForm.MonthCalendarClick(Sender: TObject);
begin
  NotifyObservers(meDateChanged);
end;
```

Det er klart, at der vil ligge en vis tilvænning ved, at events tidligere har været håndteret på kryds og tværs af units, nu skal igennem en central notificering, ligesom observere modtager events et sted og skal distribuere dem videre rundt.



Men denne koncentrering af notifikationer vil ikke bare give et medfødt overblik over de events, der rent faktisk foregår i en applikation, men også hvilke events der er

interessante for andre komponenter. Tilgangen vil også gøre det mere attraktivt at vedligeholde en kommunikation, der bliver kanaliseret centralt, med et software design pattern vi har lært at kende, fremfor direkte og konkrete referencer, der måske ligger spredt i en unit eller på tværs af flere units.

Én-til-mange observere

En stærk side ved MVVM-pattern, i forhold til MVC eller MVP, er, at der kan etableres mange relationer til en enkel komponent, men det er slet ikke sikkert, at det var det, der var interessant i refaktoreringssammenhænge. Hvis behovet er 1-1 relation, så kan MVP sagtens være tilfredsstillende. MVVM kan dog være interessant i mange situationer, fordi mange klassiske Delphi-forms er så store og komplekse, at man traditionelt også har delt dem op i mindre forms, og derefter brugt MDI, paneler, tabs, frames, eller andre kreative løsninger. I de situationer, med tætte koblinger, vil der være mange Views og en central ViewModel i forlængelse af en Model, og derfor bør Observer Pattern i sådanne sammenhænge ikke afskrives.

Knap så tæt kobling

Det andet eksempel viser et forsøg på at afkoble observer fra subject lidt mere end ved det første eksempel. Problemet er ikke så meget, at vi har en reference til en konkret subject, men at vi også er eksplisit om, at det er en bestemt komponent, der formentlig også bør være af en bestemt type, og med en bestemt property. Dvs. at observer kender til en konkret subject.

```
procedure TOwnerForm.DateChanged;
begin
  DateEdit.Text := DateTimeToStr(SettingsForm.MonthCalendar.Date);
end;
```

Den konkrete subject kan veksles til et interface, implementeret i subject, og nøjes med at referere til subject under initialisering, ved at gemme subject i en field-variabel. Alle eksplisitte referencer til en konkret subject må derfor erstattes med et kald gennem et interface, der f.eks. kunne være defineret som:

```
TMainSetting = (msEnabled, msSpinValue, msTrackPosition,
  msColorValue, msDateValue);

IMainSettings = interface['{C8F1DFE4-7EE0-4E34-A905-406E2A3B248D}']
  function GetMainSettingsEnabled: Boolean;
  function GetMainSettingsValue(const ASetting: TMainSetting): Variant;
  procedure RegisterObserver(const AObserver: TMainSettingsObserver);
  procedure UnregisterObserver(const AObserver: TMainSettingsObserver);
end;
```

Defineringen af interfacet dækker over behovet for en direkte kommunikation med subject fra observer, om end der må foretages små ændringer i kodelinjerne. Den anden fundamentale ændring kan reducere behovet for overhovedet at kommunikere med

subject væsentligt. De fleste notifikationer kan følges op med en værdi, der har betydning for hændelsen, hvilket vil sige, at hvis farven har ændret sig, så følger værdien af farven med notifikation.

```
TMainSettingsEvent = (meCreating, meDestroying, meEnabledChanged,  
    meSpinChanged, meTrackChanged, meColorChanged, meDateChanged);  
TMainSettingsObserver = procedure(const AEvent: TMainSettingsEvent;  
    const AValue: Variant) of object;
```

Dvs. at mange metoder i en observer må ændres til at bære denne ekstra værdi, men det er en meget lille pris at betale, for at minimere referencer til konkrete komponenter i en subject, som ovenikøbet skal se ud på en ganske bestemt måde. Med denne løsning vil observer være fuldstændig uviden om, hvordan en dato er blevet til, dvs. at det kan være en TMonthCalendar, TDatePicker eller noget helt tredje – det ændrer ikke noget for forholdet mellem observer og subject, fordi interfacet forbliver den samme.

```
procedure TOwnerForm.DateChanged(const AValue: TDate);  
begin  
    DateEdit.Text := DateTimeToStr(AValue);  
end;  
  
procedure TOwnerForm.MainFormEvent(const AEvent: TMainSettingsEvent;  
    const AValue: Variant);  
begin  
    case AEvent of  
        meEnabledChanged: EnabledChanged(AValue);  
        meSpinChanged: SpinChanged(AValue);  
        meTrackChanged: TrackChanged(AValue);  
        meColorChanged: ColorChanged(AValue);  
        meDateChanged: DateChanged(AValue);  
    end;  
end;
```

For subject betyder ændringer ikke de helt store, bortset fra et par getters for Enabled og de værdier, som observere kan trække fra subject, f.eks. når de skal initialisere efter en registrering. Der er valgt en tilgang, hvor værdier kan trækkes i samme metode, men det ses også at der genereres getters for hver værdi, der kan trækkes på. Det vil bero på situationen, fordi det er ikke alt der kan løses med f.eks. en Variant, men det gør vores eksempel noget nemmere. Metoden NotifyObservers() tager nu en værdi med som argument, som måtte være relevant for et event som ændring i dato osv.

```
procedure TSettingsForm.MonthCalendarClick(Sender: TObject);  
begin  
    NotifyObservers(meDateChanged, MonthCalendar.Date);  
end;
```

Selvom vi har gjort en observer mindre afhængig af en konkret subject, så er vi stadig afhængig af en enkel reference til en konkret subject, men løs kobling betyder, at ændringer i subject bør have meget lidt eller ingen betydning for en observer. Med andre ord, så kan vi udskifte vores TMonthCalendar-komponent ud med en TDatePicker og det vil ikke have nogen effekt på observere. Selvom vi stadig har en reference til en konkret subject,

så er det et yderst tilfredsstillende resultat, når vi tænker på, at vi lagde ud med et Observer Pattern, der havde en meget tæt kobling i form af konkrete referencer.

Løs kobling

Observer Pattern benyttes i situationer, hvor det er nødvendigt at objekterne ikke er så tæt koblet, som vi så det i første eksempel. Vi opnåede en mere acceptabel grad af løs kobling i det andet eksempel ved hjælp af et interface og relevante argumenter i notifikationerne, selvom observer stadig havde referencer til en konkret subject, ligesom alle erklæringer på forholdet mellem subject og observer er defineret i subjects unit. Observer Pattern kan dog også implementeres i en meget løs kobling, hvor hverken subject eller observer kender til hinanden.

Tilgangen med at indskyde et mellemled er ikke ukendt, og i den sammenhæng er der et pattern, der skiller sig ud. Mediator Pattern er bedst kendt for at tillade løs kobling imellem klasser, ved at være den eneste, der har kendskab til dele af de andre klasser. Vi vil ikke bruge Mediator Pattern i vores løsning, men ved at indskyde en klasse imellem subject og observere, som implementerer Observer Pattern og indkapsler kommunikationen mellem komponenterne i mellemleddet, så er ved meget tæt på at have Mediator Pattern uden at have det. Når vi blot er inspireret af Mediator Pattern, men ikke er det, så kan vi også tage lidt frihed og kalde vores mediator en link, hvilket har været et gennemgående begreb i mange frameworks, f.eks. TDataSourceLink, TImageLink, TContainedActionLink osv. Med andre ord, så kan vi bruge en link som mediator, som der i Delphi er stor tradition i, som udnytter Observer Pattern, der dynamisk registrerer observere og kommunikerer med dem.

```
type
  TMainSetting = (msEnabled, msSpinValue, msTrackPosition,
    msColorValue, msDateValue);
  TMainSettingsEvent = (meCreating, meDestroying, meEnabledChanged,
    meSpinChanged, meTrackChanged, meColorChanged, meDateChanged);

  IMainSettings = interface['{C8F1DFE4-7EE0-4E34-A905-406E2A3B248D}']
    function GetMainSettingsEnabled: Boolean;
    function GetMainSettingsValue(const ASetting: TMainSetting): Variant;
  end;

  TMainSettingsObserver = procedure(AEvent: TMainSettingsEvent; const AV)
  TMainSettingsObserverList = TList<TMainSettingsObserver>;

  TMainSettingsLink = class(TObject)
  private
    [weak] class var FSettingsForm: IMainSettings;
    class var FObservers: TMainSettingsObserverList;
    class destructor Destroy;
    class function GetMainSettingsEnabled: Boolean;
  public
    class procedure SetSettingsForm(const ASettingForm: IMainSettings);
    class procedure NotifyObservers(AEvent: TMainSettingsEvent; const AV)
  end;

  TOwnerSettingsLink = class(TMainSettingsLink)
  public
    class function GetMainSettingsValue(const ASetting: TMainSetting): Variant;
    class procedure RegisterObserver(AObserver: TMainSettingsObserver);
    class procedure UnregisterObserver(AObserver: TMainSettingsObserver)
  end;
```

Udsnit af koden fra filen "/Observers – 2/Main.Settings.Link.pas"

Med denne tilgang kan vi opnå en høj grad af løs kobling, fordi subject kender ikke til no-gen observers, observers kender ikke til nogen subject, og vores link kender kun et minimal interface til et subject, ligesom den kun har en metode til at notificere en observer.

Vores løsning skelner ikke imellem hvad subjekt kan se og hvad observer kan se, men vi har alligevel fremhævet at vores link kan splittes i to klasser, hvor den ene er beregnet til subject og den anden til observere. Det gør ikke nogen forskel i vores eksempel, men Observer Pattern kan også bruges i et distribueret system, hvor man begrænsrer adgangen til visse dele af et system, så man kun præsenterer et udsnit, så f.eks. observere kan fungere, og vi derfor kan beskytte de indre dele af vores system.

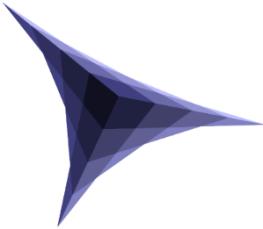
Det sidste eksempel vil nok være at gøre refaktoreringen mere kompliceret, og det vil helt sikkert give mere mening i en overordnet implementering af Observer Pattern i applikationen. I refaktorering og separeringssammenhænge vil det første eksempel være det mest nærliggende, men i MVVM-sammenhænge vil det andet eksempel være det bedste resultat. Det første eksempel opfylder målet om at have én subject og mange observere, mens det andet eksempel opfylder målet om en løsere kobling mellem subject og observere.

Observer Pattern og Architectural Patterns

Én ting er eksempler med Observer Pattern mellem komponenter eller særskilte dele af en applikation, en anden ting er hvordan Observer Pattern passer ind i Architectural Patterns, hvor en klassisk form er delt op i lag som View, ViewModel og Model. Det er klart, at det i sidste kan synes uholdbart at hvert forhold mellem View og ViewModel implementerer deres Observer Patterns, hvis en applikation består af flere hundrede af sådanne forhold. Derfor vil det også hurtigt blive indlysende, at denne del bedst løses med et såkaldt "observable framework", som de enkelte dele af en applikation bare kan tage i brug. Der vil, for de fleste udviklingsteams, hurtigt opstå et behov for en generisk implementering, og denne vil udvikle sig efterhånden som et team får erfaring med behovene for et "observable framework".

Der findes ikke et standard observable framework til Delphi, men vi behøver ikke at skabe større mystik omkring dette, fordi det kan være alt fra nogle simple skabeloner til klasser, til et færdigt sæt af klasser, der er klar til at overtage rollen som mediator for flere konkrete vertikale forhold mellem Views og ViewModels.

Vi har set mange eksempler på forms, der fungerer som "Indstillinger", som brugere kan ændre i og som mange områder af en applikation trækker på. Med refaktorering frem mod et MVVM-pattern vil sådanne forhold langsomt ændre sig til et forhold, hvor ændringer i indstillinger helt automatisk sendes ud til observere, der er interesseret i indstillinger og har betydning for deres præsentation af data i deres lille område af applikationen, hvilket betyder at Observer Pattern også kan implementeres horisontalt.



Kapitel 7

Refaktorering mod MVVM

En refaktorering mod Model-View-ViewModel (MVVM) foregår på samme måde som med separering i refaktorering mod Model-View-Presenter (MVP). Den mest markante forskel er, at vi sagtens kan definere et interface for View, som Presenter kan trække på, når vi refaktorerer mod MVP, men at vi må tage Observer Pattern i anvendelse, når vi refaktorerer mod MVVM. De to tilgange har desuden en teknisk forskel ved at MVP kun virker med en 1:1 forhold mellem View og Presenter, mens MVVM giver mulighed for mange Views til en enkel ViewModel. Det gælder også forholdene mellem ViewModel og Model.

En tredje nævneværdig udfordring bliver, når vi refaktorerer mod MVVM, er, at det er meget nemmere at definere et interface for View, som understøtter de bestående kodelinjer og referencer, som vi har flyttet over i en Presenter fra en tidligere Delphi-form. Det giver et lidt tættere kobling, men dermed ikke sagt, at vi ikke kan refaktorere mod et MVP, der kan give et meget mere afkoblet separering. Det kan også lade sig gøre at arbejde med referencer i MVVM, fordi vi ofte har mulighed for at lade observeren kende en konkret subject, men det er ikke idéen med Observer Pattern. Hændelser, ændringer osv., bør ske gennem notifikationer fra subject til observere, men bør også foregå fra observere til subject uden konkrete referencer. Observer Pattern giver MVVM et langt højere grad af mulighed for at afkoble View fra ViewModel, og ViewModel fra Model.

Det vil sige, at hovedovervejelserne for at gå videre med refaktorering mod MVVM, i sammenligning med refaktorering mod MVP, er, om der reelt er et behov for at udstille en ViewModel og Model for flere Views og ViewModels, og krav om højere grad af afkobling, eller om det er tilfredsstillende med en 1:1 forhold mellem View og Presenter, samt Presenter og Model, men hvor netop Separation of Concern er i højsæde. Der er ikke noget, der forhindrer os i, at vi benytter MVVM i visse typer af form-konstruktioner i en applikation, og MVP i andre typer af form-konstruktioner i samme applikation, fordi behovene kan være forskellige. Konstruktionerne og navnene for forms og modulerne vil derfor

spille en lang større, fordi de skal afsløre det pattern, der er brugt. Her bør genkendelighed i navnene for filerne være vigtigst.

Observer Pattern

Når vi taler om refaktorering og Observer Pattern mellem lagene i MVVM, så er det en kommunikation på form-niveau og modulerne imellem, dvs. at det ikke er specifikt rettet mod en komponent i formen eller område af formen, men skal omfatte kommunikationen for hele formen. Der er allerede data-aware-komponenter og andre linkbare komponenter, der er bygget efter Observer Pattern, og det skal bare udnyttes som de er, men det bør ske fra View gennem ViewModel, og ikke direkte fra View til Model.

Det kan selvfølgelig lade sig gøre at udvikle et komplet framework med Observer Pattern, der er indrettet til at blive brugt på tværs af en hel applikation, og implementeres lokalt gennem fælles klasser, men så bliver det svært at retfærdiggøre det som en refaktorering. Det er en opgave, der ville være oplagt at udfordre, når refaktoreringen er gennemført. Hvis vi allerede har erfaring med Observer Pattern, så er det selvfølgelig muligt at definere fælles klasser, som alligevel ville have været defineret lokalt. Disse kan indarbejdes på en måde, så de danner et fundament for en senere udvikling af en overordnet implementering af et Observer Pattern-framework.

Vi vil arbejde med et simpelt, men komplet, framework i et senere kapitel, så resultatet kan give et langt mindre aftryk på de enkelte dele af en applikation, hvor Observer Pattern er nødvendigt. Delphi har et System Messaging gennem RTL, som kunne være værd at studere, men det er en løsning, der er tættere på Publish-Subscribe Pattern end Observer Pattern. Vores eksempel vil også være mere en øvelse i Observer Pattern end et egentlig færdigt framework, men kan tjene som inspiration til de udfordringer, der er forbundet med at implementere Observer Pattern i vores MVVM-projekter.

Dette kapitel vil fokusere på at implementere Observer Pattern i de tidlige eksempler på refaktorering mod MVP, hvilket betyder at vi udnytter resultaterne de arbejder og det overblik det gav, til at skabe os det udgangspunkt, som måske er vores egentlig mål, nemlig at refaktorere mod MVVM. Vi skal bare have in mente, at vi refaktorerer fra et klassisk Delphi-projekt, hvor et nyt projekt designet efter MVVM Pattern fra start, vil se ganske anderledes ud.

Vores første eksempel er StopWatch-projektet, som bliver opdelt i et View, ViewModel og Model, men som vil få en lille tilføjelse ved, at View kan instansieres flere gange, hvilket blot skal understrege fordelen ved Observer Pattern. Vores andet eksempel er refaktorering af et eksempel, som er brugt i kapitel 6 om Observer Pattern. I det eksempel har vi snydt lidt ved at vi har ignoreret separering af Delphi-formen og udelukkende koncentreget os om implementering af et Observer Pattern. I dette kapitel vil dette projekt blive separeret i et View, ViewModel og Model, hvor Observer Pattern vil blive implementeret mellem disse lag, som så vil blive brugt til at demonstrere, hvordan samme ViewModel kan bruges forskelligt i vidt forskellige og uafhængige Views. Den gamle form med

indstillingerne, og som fungerede som subject for andre Views, er jo et View i sig selv, men efter separeringen kommer View, ViewModel, og Model endelig til deres ret. De øvrige Views vil ikke længere trække på den gamle form, men dennes ViewModel.

StopWatch MVVM

StopWatch-projektet er et lille og simpelt program, hvor en refaktorering mod MVP eller MVVM kan synes som en overkill, men ikke desto mindre, så tjener projektet sin rolle som eksempel. Vi har separeret StopWatch tidligere i en View, Presenter og Model, og i mange situationer kan denne konstruktion sagtens være tilstrækkelig. Udover muligheden for at slutte flere Views til samme ViewModel, så slutter ViewModel's rolle som "Presenter", fordi hvert View vil jo have sin egen interesse i, hvordan data præsenteres. Et View kan vise samme klokke med et format som "hh:nn", mens et andet View kan vise klokken med et format som "hh:nn:ss" eller som med analoge visere.

Derfor ser vi også en markant skift i ansvarsfordelingen mellem en Presenter og en ViewModel, fordi Presenter også har haft rollen for at præsentere data fra Model til View i f.eks. rette format. I MVVM vil et View foretrække at modtage data, som det er, f.eks. klokken som et TDateTime-format, og derefter selv trække på indstillinger for formatering osv. Med MVP er det derfor ikke usædvanligt at se:

```
procedure TStopWatchPresenter.UpdateDisplay(const ATimeElapsed: TDateTime);
var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.fff', ATimeElapsed);
  StopWatchView.UpdateDisplay(LTimeElapsed);
end;
```

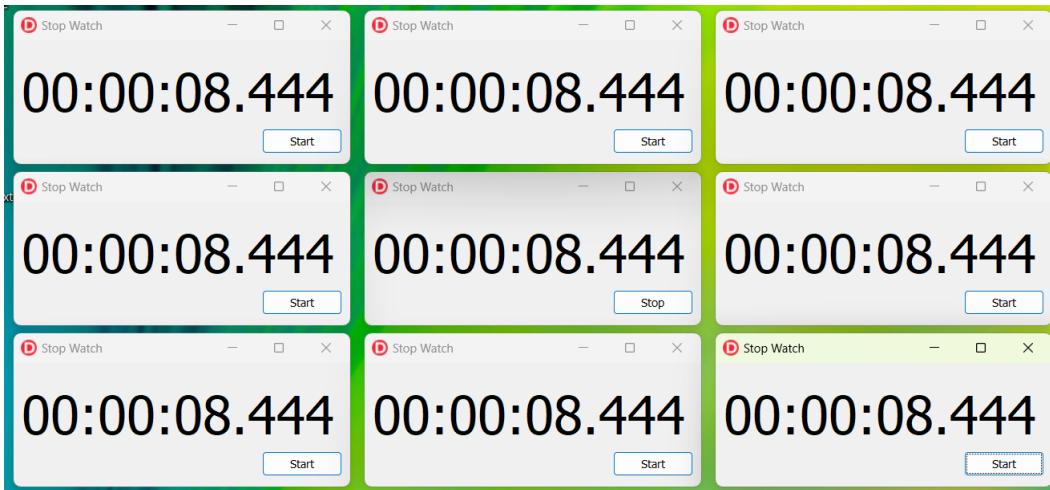
Hvorimod vi, i MVVM, straks vil sende ViewModels observere en notifikation:

```
procedure TStopWatchViewModel.UpdateDisplay(const ATimeElapsed: TDateTime);
begin
  NotifyObservers(seUpdateDisplay, ATimeElapsed);
end;
```

Af hvem og hvordan data bliver konsumeret i MVVM er ikke længere interessant, altså lagene imellem – det er derimod vigtigt, at der er en aftale mellem lagene om, at der sker notifikationer, når det forventes at det sker. Når vi taler om Presenter og ViewModel, så er det primært disse synlige forskelle, vi vil se, og hvor ViewModel mere indtager en rolle som mediator mellem View og Model. ViewModel kan indkapsle den interaktive del af Views logik, f.eks actions m.m., men det betyder ikke at al den logik skal indgå i ViewModel. ViewModel kan have den overordnede rolle i at bestemme i hvilken rækkefølge visse handlinger bør ske, udover at den naturligvis fungerer som en model for Model. View har ikke adgang til Model, så derfor funger ViewModel også som det link, som View har brug for – vi kan simpelthen også skifte Model ud, uden at View ved noget om det.

I vores StopWatch-projekt er Observer Pattern kun implementeret mellem View og ViewModel – det kan til dels fremhæve forskellen i relationer mellem View og ViewModel, samt ViewModel og Model. Relationen mellem ViewModel og Model er bygget efter et interface på samme måde som vi vil refaktorere mod MVP, men relationer mellem ViewModel og Model vil normalt også være bygge på Observer Pattern.

StopWatch-projektet har fået en tilføjelse ved, at der bliver oprettet et nyt View for hvert sekund, der er gået (indtil 9 Views er oprettet), netop for at fremhæve, at der ikke er begrænsning i antallet af Views, der kan registrere sig hos en ViewModel. De viser hver især det samme, men kører helt uafhængig af hinanden, og de kan hver især starte og stoppe den samme klokke, og de vil hver især reagere på notifikationer på samme måde.



Refaktorering fra MVP til MVVM

Vi har tidligere refaktoreret vores StopWatch-projekt fra et klassisk Delphi-projekt til et MVP-projekt, og det er præcis resultatet af dette arbejde, vi kan udnytte i en fortsættelse af refaktorering mod MVVM. I Presenter havde vi defineret et interface, som et View skulle implementere, så Presenter kunne kommunikere med samme View. Vi forudsætter således, at View kender Presenter, eller View kommer til at kende ViewModel i vores refaktorering, fordi det er de relationer, der ligger tættes på de eksisterende kodelinjer, vi har flyttet rundt.

```
IStopWatchView = interface['{0265C82F-5F42-4609-B9A0-FE221A282CFC}']
  procedure TimerStarted;
  procedure TimerStopped;
  procedure UpdateDisplay(const ATimeElapsed: string);
end;
```

Denne definition giver en fingerpeg om, hvilke notifikationer vores observere skal have. Hvordan observere skal notificeres er altid den største udfordring i implementering af Observer Pattern, fordi notifikationen skal være så kort som muligt, men skal indeholde

så meget information så muligt, så det mindsker eller fjerne observers behov for at trække flere oplysninger ud af subject efterfølgende. I vores projekt har vi blot brug for tre hændelser, hvori kun den ene hændelse indeholder en klokke. Det er ikke så kompliceret, og i større konstruktioner kan det også være nødvendigt med objekter, med nødvendig information, som ledsager en notifikation.

I StopWatch-projektet skal vi altså kunne notificere observere om at en timer er startet eller stoppet, og når det er tid til at opdatere et display. Timeren i vores projekt kører med et interval på 10 ms, så notifikationer om opdatering kommer jævnligt. I vores MVP-projekt bliver notifikationer om start eller stop af timer ikke ledsaget af nogen data, men fordi vi har brug for en generisk metode for notifikationer, så må dette argument i nogle tilfælde være "Null".

```
TStopWatchEvent = (seTimerStarted, seTimerStopped, seUpdateDisplay);
TStopWatchObserver = procedure(const AEvent: TStopWatchEvent;
                               const AValue: Variant) of object;
TStopWatchObservers = TList<TStopWatchObserver>;
```

De tre metoder i definering af interfacet bliver på den måde erstattet med tre hændelser, og vi har valgt den nemmeste tilgang til at notificere observere, nemlig en metode. De metoder skal lagres på en liste, når en observer registrerer sig. Et interface er på den måde erstattet med en række definitioner, som et View ikke skal implementere, men vi fjerner ikke selve implementeringerne af de metoder, som før var defineret af interfacet – de vil jo blive genbrugt, når View modtager notifikationer i stedet for. Den metode, som vi gerne vil registrere hos subject, og hvor vi ønsker at modtage notifikationer i, er en tilføjelse i View, som er nødvendig. I den metode ønsker vi at reagere på alle de tre hændelser, som er muligt at modtage fra ViewModel, og her kalder vi de metoder, som vi før har implementeret gennem definering af et interface, som ikke længere er i brug.

```
procedure TStopWatchView.StopWatchEvent(const AEvent: TStopWatchEvent;
                                         const AValue: Variant);
begin
  case AEvent of
    seTimerStarted: TimerStarted;
    seTimerStopped: TimerStopped;
    seUpdateDisplay: UpdateDisplay(AValue);
  end;
end;
```

Denne tilføjelse er lidt mere end hvad en Refactoring normalt vil betragte som refactoring, fordi det er en ny metode. Men det er en metode, der er nødvendig, fordi ellers har vi ikke nogen Observer Pattern, og uden dette pattern, så får vi heller ikke implementeret noget MVVM-pattern. Bortset fra, at vi registrerer vores notifikationsmetode i FormCreate og afmelder den i FormDestroy, som i øvrigt kan placeres andre steder, så er der ikke tilføjet noget andet i View. Denne udgave er således ikke meget forskelligt fra den View, som bliver brugt i vores MVP-projekt. Forskellen i definering af formen er lille, og det vil

være i implementeringsdelen, at vi bedre vil kunne genkende om et View er en del af et MVP- eller et MVVM-pattern.

View i MVP-projektet

```

unit Stopwatch.Main.View;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, V
  Vcl.StdCtrls, Vcl.ExtCtrls, Stopwatch.Main.Presenter;

type
  TStopWatchView = class(TForm, IStopWatchView)
    StopWatchDisplay: TLabel;
    StartButton: TButton;
    procedure StartButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    procedure TimerStarted;
    procedure TimerStopped;
    procedure UpdateDisplay(const ATimeElapsed: string);
  end;

```

Fra eksemplet "StopWatch - 7"

View i MVVM-projektet

```

unit Stopwatch.Main.View;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, V
  Vcl.StdCtrls, Vcl.ExtCtrls, Stopwatch.Main.ViewModel;

type
  TStopWatchView = class(TForm)
    StopWatchDisplay: TLabel;
    StartButton: TButton;
    procedure StartButtonClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  private
    procedure TimerStarted;
    procedure TimerStopped;
    procedure UpdateDisplay(const ATimeElapsed: TDateTime);
  protected
    procedure StopWatchEvent(const AEvent: TStopWatchEvent);
  end;

```

Fra eksemplet "StopWatch - MVVM"

Bortset fra metoden til notifikationerne, så har vi tilføjet FormDestroy, fordi en observer skal afmelde sig selv, hvis den har registreret sig. En subject kan selvfølgelig ikke vide, at et View pludselig er væk, så derfor må der ikke forekomme referencer til metoder, der ikke længere findes. Det kan overvejes, hvor til- og afmelding foretages, fordi FormShow og FormHide også vil være tilstrækkelig i denne situation, men det er helt op til den enkelte observer at konstruere, hvornår registreringerne foretages. Der er altså en lille detalje i initialisering i forhold til Views relation til Presenter i MVP eller ViewModel i MVVM.

View i MVP-projektet

```

procedure TStopWatchView.FormCreate(Sender: TObject);
begin
  TStopWatchPresenter.StopWatchView := Self;
end;

```

Fra eksemplet "StopWatch - 7"

View i MVVM-projektet

```

procedure TStopWatchView.FormCreate(Sender: TObject);
begin
  StopWatchViewModel.RegisterObserver(StopWatchEvent);
end;

procedure TStopWatchView.FormDestroy(Sender: TObject);
begin
  StopWatchViewModel.UnregisterObserver(StopWatchEvent);
end;

```

Fra eksemplet "StopWatch - MVVM"

Det er eksemplet på, at MVP er designet til et 1:1 forhold mellem View og Presenter, hvorimod View i vores MVVM-projekt bare vil være et View ud af mange mulige Views. Det er klart, at det er kodekonstruktioner, som vil gentage sig selv i alle de Views, vi vil tilføje i vores projekt, og det er helt op til de enkelte teams at designe en skabelon eller klasser, som f.eks. Views kan nedarve fra, der kunne "automatisere" denne del. Denne repetition af kodelinjer er dog ikke så slem, og især synligheden i initialisering med RegisterObserver er vigtige, fordi vi vil kunne genkende disse navne.

Denne konstruktion forudsætter, at ViewModel og Model er oprettet, og i dette eksempel er forudsætningen håndteret i projekt-filen. I praksis er det en tilbagevendende situation, hvor ViewModel eller Model på en eller anden måde skal være til stede, for at en observer kan registrere sig selv. En anden tilgang er anvendt i næste eksempel, som måske vil give mere mening i andre situationer.

På implementeringssektionen er der dog lidt flere kodelinjer i en MVVM-løsning, vi skal foreholde os til, men det er ikke mængden af kodelinjer, der vil komme til at fylde i metoden til notifikationerne, fordi når de grundlæggende kodekonstruktionerne først er på plads, så vokser kodelinjerne ikke væsentligt i forhold til almindelig videreudvikling af sourcekoden. Når vi ser bort fra metoden til notifikationerne, så er konstruktionen af metoderne ens for både MVP og MVVM.

View i MVP-projektet

```

procedure TStopWatchView.StartButtonClick(Sender: TObject)
begin
  StopwatchPresenter.StartButtonClick(Sender);
end;

procedure TStopWatchView.TimerStarted;
begin
  StartButton.Caption := 'Stop';
end;

procedure TStopWatchView.TimerStopped;
begin
  StartButton.Caption := 'Start';
end;

procedure TStopWatchView.UpdateDisplay(const ATimeElapsed:
begin
  StopwatchDisplay.Caption := ATimeElapsed;
end;

```

View i MVVM-projektet

```

procedure TStopWatchView.StartButtonClick(Sender: TObject)
begin
  StopwatchViewModel.StartButtonClick(Sender);
end;

procedure TStopWatchView.StopWatchEvent(const AEvent: TSto
  const AValue: Variant);
begin
  case AEvent of
    seTimerStarted: TimerStarted;
    seTimerStopped: TimerStopped;
    seUpdateDisplay: UpdateDisplay(AValue);
  end;
end;

procedure TStopWatchView.TimerStarted;
begin
  StartButton.Caption := 'Stop';
end;

procedure TStopWatchView.TimerStopped;
begin
  StartButton.Caption := 'Start';
end;

procedure TStopWatchView.UpdateDisplay(const ATimeElapsed:
  var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.fff', ATimeElap
  StopwatchDisplay.Caption := LTimeElapsed;
end;

```

Fra eksemplet "StopWatch - 7"

Fra eksemplet "StopWatch - MVVM"

Kodelinjerne i MVP ligger tættere på de oprindelige kodelinjer i den klassiske Delphi-form, som vi refaktorerede mod MVP, og resultatet kan derfor holdes på et simpelt plan. Derimod må vi tilføje kodelinjer i en refaktorering mod MVVM. View i MVVM har taget over ansvaret for præsentation af data, selvom det er ikke ud fra dette pattern bestemt, at det ansvar ligger i View, men fordi det er mest sandsynligt at hvert View vil have deres egen mål om præsentation af data. Der vil jo være forskel på en analog og en digital klokke (grafisk), hvorfor rå data som i TDateTime, som ikke er formateret, vil være mest nærliggende.

ViewModel

I definering af ViewModel finder vi nok de største visuelle ændringer i form af flere metoder, en liste over observere, samt nogle definitioner på den kontrakt, der vil være mellem observere og subject, dvs. mulige typer af hændelser. Vores første ViewModel har bevaret det interface, som Model har defineret, men i praksis vil Model også have implementeret Observer Pattern, men en notifikationsmetode til ViewModel.

I refaktoreringssammenhænge vil de definitioner på observers og hændelser oftest være forenelige for både ViewModel og Model, og derfor kan vi i mange tilfælde sætte lighedstegn mellem definitioner på observers og hændelser i ViewModel og Model. Dette forhold vil også blive berørt i næste eksempel.

Presenter i MVP-projektet

```
IStopWatchView = interface['{0265C82F-5F42-4609-B9A0-FE2
  procedure TimerStarted;
  procedure TimerStopped;
  procedure UpdateDisplay(const ATimeElapsed: string);
end;

TStopWatchPresenter = class(TDataModule, IStopWatchPrese
  procedure DataModuleCreate(Sender: TObject);
private
  procedure TimerStarted;
  procedure TimerStopped;
  procedure TimerEvent(Sender: TObject);
  procedure UpdateDisplay(const ATimeElapsed: TDateTime);
public
  procedure StartButtonClick(Sender: TObject);
  [unsafe] class var StopWatchView: IStopWatchView;
end;
```

Fra eksemplet "StopWatch - 7"

ViewModel i MVVM-projektet

```
TStopWatchEvent = (setTimerStarted, setTimerStopped, setUpd
TStopWatchObserver = procedure(const AEvent: TStopWatchE
  const AValue: Variant) of object;
TStopWatchObservers = TList<TStopWatchObserver>;

TStopWatchViewModel = class(TDataModule, IStopWatchPrese
  procedure DataModuleCreate(Sender: TObject);
  procedure DataModuleDestroy(Sender: TObject);
private
  FObservers: TStopWatchObservers;
  procedure NotifyObservers(const AEvent: TStopWatchEven
  procedure TimerStarted;
  procedure TimerStopped;
  procedure TimerEvent(Sender: TObject);
  procedure UpdateDisplay(const ATimeElapsed: TDateTime);
public
  procedure StartButtonClick(Sender: TObject);
  procedure RegisterObserver(const AObserver: TStopWatch
  procedure UnregisterObserver(const AObserver: TStopWat
end;
```

Fra eksemplet "StopWatch - MVVM"

De mest iøjnefaldende ændringer i ViewModel's implementationssektion er selvfølgelig kald til metoder i View, som ikke længere vil være mulige – interfacet er ikke længere tilgængeligt og alle hændelser skal sendes som notifikation. Hvis vi ser bort fra metoder til implementering af Observer Pattern, så tilføjer vi heller ikke flere metoder, så ændringerne er af teknisk art. De ændringer er til gengæld svære at retfærdiggøre som almindelig refaktorering, fordi vi ændrer fundamentalt på direkte kald af metoder til notifikation gennem et mellemled. Vi kan dog ikke komme udenom ændringerne, hvis vi ønsker at implementere MVVM-pattern, og derfor bliver det også nødvendigt med omhyggelige code-reviews og gennemgang af tests.

Presenter i MVP-projektet

```
procedure TStopWatchPresenter.StartButtonClick(Sender: TObject)
begin
  if StopWatchModel.StopWatchTimer.Enabled then
    StopWatchModel.StopTimer
  else
    StopWatchModel.StartTimer;
end;

procedure TStopWatchPresenter.TimerStarted;
begin
  StopWatchView.TimerStarted;
  UpdateDisplay(0);
end;

procedure TStopWatchPresenter.TimerStopped;
begin
  StopWatchView.TimerStopped;
  UpdateDisplay(Now - StopWatchModel.TimeStart);
end;

procedure TStopWatchPresenter.TimerEvent(Sender: TObject);
begin
  UpdateDisplay(Now - StopWatchModel.TimeStart);
end;

procedure TStopWatchPresenter.UpdateDisplay(const ATimeElapsed: TTimeElapsed);
var
  LTimeElapsed: string;
begin
  DateTimeToString(LTimeElapsed, 'hh:nn:ss.fff', ATimeElapsed);
  StopWatchView.UpdateDisplay(LTimeElapsed);
end;
```

Fra eksemplet "StopWatch - 7"

ViewModel i MVVM-projektet

```
procedure TStopWatchViewModel.StartButtonClick(Sender: TObject)
begin
  if StopWatchModel.StopWatchTimer.Enabled then
    StopWatchModel.StopTimer
  else
    StopWatchModel.StartTimer;
end;

procedure TStopWatchViewModel.TimerStarted;
begin
  NotifyObservers(seTimerStarted, Null);
  UpdateDisplay(0);
end;

procedure TStopWatchViewModel.TimerStopped;
begin
  NotifyObservers(seTimerStopped, Null);
  UpdateDisplay(Now - StopWatchModel.TimeStart);
end;

procedure TStopWatchViewModel.TimerEvent(Sender: TObject);
begin
  UpdateDisplay(Now - StopWatchModel.TimeStart);
end;

procedure TStopWatchViewModel.UpdateDisplay(const ATimeElapsed: TTimeElapsed);
begin
  NotifyObservers(seUpdateDisplay, ATimeElapsed);
end;
```

Fra eksemplet "StopWatch - MVVM"

Når vi ikke ser på de tekniske forskelle, så er det muligt at opretholde en konstruktion, der ikke afviger fra de oprindelige kodekonstruktioner, og på er det muligt at sammenholde de gamle sourcekoder med de nye, så det er muligt at lave familier code-review. Når alt er gennemgået, og det hele kører, som det altid har kørt, så er der heller ikke noget, der forhindrer os i at refaktorerer videre herfra. Vi har jo opnået en separation, og vi har implementeret Observer Pattern.

I vores "StopWatch - MVVM"-projekt har vi tilføjet en CreateViews-metode, der blot har til formål at skabe op til 9 Views af den samme form, og demonstrere, at ViewModel vil fungere med et vilkårligt antal Views, og at den ene Start/Stop-knap vil fungere på tværs af hele applikationen.

Settings - MVVM

Med en gennemgang i kapitlet om Observer Pattern fik vi stablet et eksempel op med indstillinger og en konkret implementering af observere, der blot "lyttede" til ændringer i indstillingerne, og vi kunne instantiere et variabelt antal Views. Vi havde snydt lidt, fordi formen for indstillingerne var en klassisk Delphi-form, og i næste eksempel vil denne form være splittet op i View, ViewModel og Model. Observer Pattern er implementeret mellem View og ViewModel, samt mellem ViewModel og Model.

Eksemplet tager udgangspunkt i et vindue med indstillinger – det er sådan set ligegyldigt, hvad indstillingerne skal bruges til, og hovedmålet er at vi kan ændre indstillingerne. Vi

skal forestille os, at disse indstillinger benyttes mange steder i en applikation, og det er ikke en atypisk situation, fordi vi ofte lader brugeren ændre formatering, farver osv. Vores tidligere eksempel med indstillinger er nu splittet op i et MVVM-pattern, så den form, hvor indstillingerne tidligere har været "lagret", nu også er et View i sig selv på lige fod med alle de andre Views. De andre Views kan præsentere de samme indstillinger på forskellig måde, og ignorerer andre indstillinger, ligesom et View kan give mulighed for at ændre en indstilling eller flere. Disse ændringer vil ligeledes forplante sig i Model og videre på tværs af applikationen.

Hvor vi før havde en enkel unit Main.Settings med en form og alle indstillingerne i samme unit, så har vi nu en ViewModel, der arbejder med en Model, hvor alle indstillingerne er gemt. Hvor vi før ofte var nødt til at have instantieret en sådan form med indstillingerne, behøver vi nu kun at instantiere ViewModel.

Delphi-udviklere, der igennem mange år har arbejdet med en RAD-tankegang, vil formentlig finde alle de units og klasser som et ganske betragteligt indgreb i de sourcekoder, vi skal vedligeholde, men separering sker på fil-niveau og ikke unormalt at se i andre programmeringsværktøjer, f.eks. i ASP.NET Core MVC. Vi skal forholde os til kodekonstruktioner, der kalder mellemliggende metoder, hvilket betyder noget overhead, hvilket ikke er ukendt ved MVVM. I vores separering har vi udskilt alle de værdier, hvor vi tidligere har brugt visuelle komponenter til at håndtere, og lagt de samme variabler ned i en Model, med samme typer osv. På nær implementering af Observer Pattern, så består vores Model udelukkende af nogle field-variabler, properties og setters.

Vi har brug for setters, fordi vi selvfølgelig skal sende en notifikation, når der er sket en ændring i et field-variabel. Og bortset fra dette, så ligner vores Model et ganske simpel object, der bare opbevarer noget data. Det kan forekomme i praksis, men oftest vil vi have en hel del tilhørende logik, der skal bearbejde de data, som Model håndterer, ligesom vi oftest også vil have adgang til databaser og meget andet. Model vil have al den forretningslogik, der måtte være nødvendig, for at data giver mening for applikationen. I vores eksempel er det bare simple og primitive data.

Vores Settings-projekt findes også i en MVP-udgave, og i Observer Pattern-sammenhænge giver den ikke så meget mening, men hvis vi kommer fra en refaktorering, hvor vi har brugt MVP som en overgang, så kan vi også se, hvor vi kommer fra. Det vil også fremhæve de dele af konstruktionerne, gør sourcekoden til et MVVM-pattern.

Model i MVP-projektet

```

IMainSettingPresenter = interface['{60BDB55E-B059-45FB-8
procedure EnabledChanged(const AEnabled: Boolean);
procedure ColorChanged(const AColor: TColor);
procedure DateChanged(const ADate: TDate);
procedure SpinChanged(const ASpin: LongInt);
procedure ZoomChanged(const AZoom: Integer);
end;

TSettingsModel = class(TDataModule)
procedure DataModuleCreate(Sender: TObject);
private
FEnabled: Boolean;
FColor: TColor;
FDate: TDate;
FSpin: LongInt;
FZoom: Integer;
procedure SetEnabled(const Value: Boolean);
procedure SetColor(const Value: TColor);
procedure SetDate(const Value: TDate);
procedure SetSpin(const Value: LongInt);
procedure SetZoom(const Value: Integer);
public
[unsafe] class var MainSettingPresenter: IMainSettingP
property Color: TColor read FColor write SetColor;
property Date: TDate read FDate write SetDate;
property Enabled: Boolean read FEnabled write SetEnabl
property Spin: LongInt read FSpin write SetSpin;
property Zoom: Integer read FZoom write SetZoom;
end;

```

Fra eksemplet "Settings - MVP"

Model i MVVM-projektet

```

TMainSettingsEvent = (meCreating, meDestroying, meEnable
meSpinChanged, meZoomChanged, meColorChanged, meDateCh
TMainSettingsObserver = procedure(const AEvent: TMainSet
const AValue: Variant) of object;
TMainSettingsObserverList = TList<TMainSettingsObserver>

TSettingsModel = class(TDataModule)
procedure DataModuleCreate(Sender: TObject);
procedure DataModuleDestroy(Sender: TObject);
private
FEnabled: Boolean;
FColor: TColor;
FDate: TDate;
FSpin: LongInt;
FZoom: Integer;
procedure SetEnabled(const Value: Boolean);
procedure SetColor(const Value: TColor);
procedure SetDate(const Value: TDate);
procedure SetSpin(const Value: LongInt);
procedure SetZoom(const Value: Integer);
protected
FObservers: TMainSettingsObserverList;
procedure NotifyObservers(const AEvent: TMainSettingsE
public
procedure RegisterObserver(const AObserver: TMainSetti
procedure UnregisterObserver(const AObserver: TMainSet
class procedure SettingsModelNeeded(const Owner: TComp
property Color: TColor read FColor write SetColor;
property Date: TDate read FDate write SetDate;
property Enabled: Boolean read FEnabled write SetEnabl
property Spin: LongInt read FSpin write SetSpin;
property Zoom: Integer read FZoom write SetZoom;
end;

```

Fra eksemplet "Settings - MVVM"

Model i MVVM har sin egen implementation af Observer Pattern, og det betyder at Model sagtens kan have flere ViewModels, som er registreret hos Model, ligesom flere Views kan registrere sig hos en enkel ViewModel. Der er ikke begrænsninger indlagt på denne måde, ligesom det sagtens kan lade sig gøre at få 2 Models til at arbejde sammen, og på samme måde kan vi registrere en ViewModel hos 2 Models, eller for dens sags skyld et View, der er registreret hos flere ViewModels. De enkelte observere bestemmer jo selv, hvilke oplysninger det vil gøre brug af, og hvilke oplysninger det vil ignorere. De forskellige konstellationer kan lade sig gøre i MVVM, men vil altid være fra View mod ViewModel, fra ViewModel mod Model, eller tværs igennem samme lag, som ofte ses på Model-laget.

Vores eksempel indeholder ikke nogen forretningslogik, så der er nogle setters til de data, som vi har definere vores Model skal indeholde. En setter i vores MVP-projekter kalder en metode i Presenter, som er defineret af et interface, hvorimod en setter i vores MVVM-projekt vil kalde en metode i Model, der sender en notifikation til alle observere, der er registeret. Når vi ser bort fra denne detalje, og definering af et interface i MVP, og implementering af Observer Pattern i MVVM, så vil der ikke være yderligere forskel i sourcekoden.

Setter i Model i MVP-projektet

```
procedure TSettingsModel.SetColor(const Value: TColor);
begin
  if Value <> FColor then
    begin
      FColor := Value;
      MainSettingPresenter.ColorChanged(Value);
    end;
end;
```

Fra eksemplet "Settings - MVP"

Setter i Model i MVVM-projektet

```
procedure TSettingsModel.SetColor(const Value: TColor);
begin
  if Value <> FColor then
    begin
      FColor := Value;
      NotifyObservers(meColorChanged, FColor);
    end;
end;
```

Fra eksemplet "Settings - MVVM"

Det samme mønster tegner sig for forskellene mellem en Presenter i MVP og ViewModel i MVVM. Implementation af Observer Pattern fylder forholdsvis mere, og Presenter synes at virke mere overskueligt.

Presenter i MVP-projektet

```
IMainSettingView = Main.Settings.Model.IMainSettingPres
TSettingsPresenter = class(TDataModule, IMainSettingPres
  procedure DataModuleCreate(Sender: TObject);
private
  function GetColor: TColor;
  function GetDate: TDate;
  function GetEnabled: Boolean;
  function GetSpin: LongInt;
  function GetZoom: Integer;
  procedure SetColor(const Value: TColor);
  procedure SetDate(const Value: TDate);
  procedure SetEnabled(const Value: Boolean);
  procedure SetSpin(const Value: LongInt);
  procedure SetZoom(const Value: Integer);
protected
  procedure ColorChanged(const AColor: TColor);
  procedure DateChanged(const ADate: TDate);
  procedure EnabledChanged(const AEnabled: Boolean);
  procedure SpinChanged(const ASpin: LongInt);
  procedure ZoomChanged(const AZoom: Integer);
public
  [unsafe] class var MainSettingView: IMainSettingView;
  property Color: TColor read GetColor write SetColor;
  property Date: TDate read GetDate write SetDate;
  property Enabled: Boolean read GetEnabled write SetEna
  property Spin: LongInt read GetSpin write SetSpin;
  property Zoom: Integer read GetZoom write SetZoom;
end;
```

Fra eksemplet "Settings - MVP"

ViewModel i MVVM-projektet

```
TMMainSettingsEvent = Main.Settings.Model.TMainSettingsEv
TMMainSettingsObserver = Main.Settings.Model.TMainSetting
TMMainSettingsObserverList = Main.Settings.Model.TMainSet
TSettingsViewModel = class(TDataModule)
  procedure DataModuleCreate(Sender: TObject);
  procedure DataModuleDestroy(Sender: TObject);
private
  function GetColor: TColor;
  function GetDate: TDate;
  function GetEnabled: Boolean;
  function GetSpin: LongInt;
  function GetZoom: Integer;
  procedure SetColor(const Value: TColor);
  procedure SetDate(const Value: TDate);
  procedure SetEnabled(const Value: Boolean);
  procedure SetSpin(const Value: LongInt);
  procedure SetZoom(const Value: Integer);
protected
  FObservers: TMMainSettingsObserverList;
  procedure NotifyObservers(const AEvent: TMMainSettingsE
  procedure MainSettingsObserver(const AEvent: TMMainSet
  procedure ColorChanged(const AColor: TColor);
  procedure DateChanged(const ADate: TDate);
  procedure EnabledChanged(const AEnabled: Boolean);
  procedure SpinChanged(const ASpin: LongInt);
  procedure ZoomChanged(const AZoom: Integer);
public
  procedure RegisterObserver(const AObserver: TMMainSetti
  procedure UnregisterObserver(const AObserver: TMMainSet
  class procedure SettingsViewModelNeeded(const Owner: T
  property Color: TColor read GetColor write SetColor;
  property Date: TDate read GetDate write SetDate;
  property Enabled: Boolean read GetEnabled write SetEna
  property Spin: LongInt read GetSpin write SetSpin;
  property Zoom: Integer read GetZoom write SetZoom;
end;
```

Fra eksemplet "Settings - MVVM"

Hvis vi kigger nærmere på sourcekoden for ViewModel, så kan det synes omsonst at vi har metoder med én kodelinje, men vi skal bare have in mente, at i praksis vil vi have mange kodelinjer med håndtering af forskellige hændelser m.m. Ellers kan man godt være fristet til at sende notifikationer fra Model direkte videre til ViewModels observere, uden at vi behøver at håndtere notifikationerne enkeltvis i ViewModel, som vi gør i eksemplet. Derfor er det vigtigt at fremhæve, at hver notifikation skal splittes ud, så de kan

håndteres individuelt, og derefter sendes videre i systemet. Metoden ColorChanged bliver kaldt direkte fra Model i vores MVP-projekt, hvorimod samme metode først kaldes, når notifikationen er identificeret i MVVM.

Event i Presenter i MVP

```
procedure TSettingsPresenter.ColorChanged(const AColor: TColor);
begin
  MainSettingView.ColorChanged(AColor);
end;
```

Fra eksemplet "Settings - MVP"

Event i ViewModel i MVVM

```
procedure TSettingsViewModel.MainSettingsObserver(const AEvent: AE);
begin
  case AEvent of
    meEnabledChanged: EnabledChanged(AValue);
    meSpinChanged: SpinChanged(AValue);
    meZoomChanged: ZoomChanged(AValue);
    meColorChanged: ColorChanged(AValue);
    meDateChanged: DateChanged(AValue);
  end;
end;

procedure TSettingsViewModel.ColorChanged(const AColor: TColor);
begin
  NotifyObservers(meColorChanged, AColor);
end;
```

Fra eksemplet "Settings - MVVM"

På den måde kan ViewModel komme til at virke som et ligegyldigt mellemled mellem View og Model, men det skal vi ikke lade os forskrække af, fordi forholdene er anderledes, når vi refaktorerer fra en klassisk Delphi-form til MVP eller MVVM. I et rent MVVM-projekt vil vi f.eks. aldrig tilføje en OnClick-event til en knap i View, men binde denne knap til en action i ViewModel. I refaktoreringssammenhænge er det ikke sikkert, at vi har disse muligheder, uden at vi bryder principperne om refaktorering. Derfor er det mest sandsynligt, at vi ender med en såkaldt "tynd" ViewModel, som på flere måder kan binde View direkte med Model.

Vores nye View er nu en skrabet udgave af en tidligere form, hvor events fra formen kanaliseres direkte videre til en property i ViewModel. Det er ikke alle komponenter, der har adgang til en action-property, og vi kan sikkert komme langt med live-binding, men vi vil være afhængige af disse gamle events, som egentlig skulle være håndteret af ViewModel. Når brugeren vælger en farve på ColorListBox'en, så videresendes valget til ViewModel, som så blot sætter farven i Model.

View

```
procedure TSettingsView.ColorListBoxClick(Sender: TObject);
begin
  SettingsViewModel.Color := ColorListBox.Selected;
end;
```

ViewModel

```
procedure TSettingsViewModelSetColor(const Value: TColor);
begin
  SettingsModel.Color := Value;
end;
```

Model

```
procedure TSettingsModelSetColor(const Value: TColor);
begin
  if Value <> FColor then
  begin
    FColor := Value;
    NotifyObservers(meColorChanged, FColor);
  end;
end;
```

Det vil være fristende for mange at sætte farven direkte fra View og til Model, men meningen er, at View ikke bør have nogen forestilling om i hvilken format farven er lagret i Model – det kunne jo være en farvekode i tekstformat. Model vil altid en 1:1 repræsentation af den verden, den skal omfavne – dvs. farven er gemt i Model i en variabeltype, som den nu engang er, og som måske er bestemt af en datatabel, og det er der ikke nogen view, der skal have indflydelse på. Og hvis nu View har brug for en farve, som brugeren kan redigere som HTML-farvekode (hex-kode), så er det ikke Model, der skal rettes til – det er

ViewModel, der skal sørge for at View har de metoder, der skal til for, at det kan lade sig gøre. ViewModel kan f.eks. udstille en property ColorAsHex: string, og sørger for den konvertering, som View har brug for.

View	ViewModel	Model
<pre>procedure TSettingsView.ColorListBoxClick(Sender: TObject); begin SettingsViewModel.Color := ColorListBox.Selected; end;</pre>	<pre>procedure TSettingsViewModel.SetColorAsHex(const Value: string); begin SettingsModel.Color := HexToColor(Value); end;</pre>	<pre>procedure TSettingsModel.SetColor(const Value: TColor); begin if Value <> FColor then begin FColor := Value; NotifyObservers(meColorChanged, FColor); end; end;</pre>

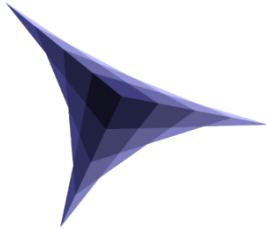
Kodelinjerne i View bliver holdt "rene", som om data som hex var den mest naturlige type, der fandtes i Model, ligesom Model får lov at arbejde med datatyper, som de nu engang er. ViewModel laver det grove arbejde, så View kan gøre sit arbejde, og Model kan fokusere på forretningslogik og dataintegritet.

Mange Views

Vi har tilføjet en række Views i vores "Settings - MVVM"-projekt, der hver især registrerer sig som observere i en ViewModel. Hvert View har deres særinteresser i de settings, som er udstillet af en ViewModel. Nogle kan redigere i værdierne, og andre viser blot en værdi.



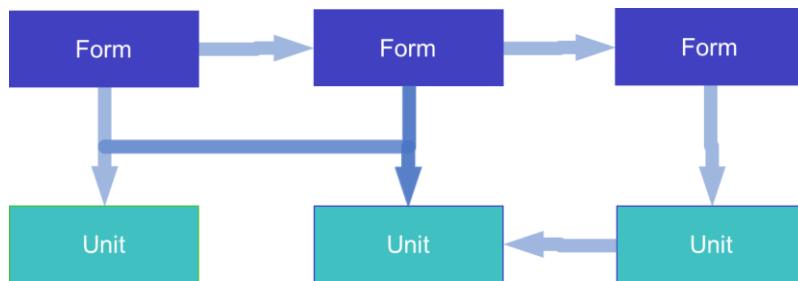
I forhold til en klassisk Delphi-form eller MVP, så er det dét MVVM kan – én Model og et mange Views.



8 Kapitel

Strukturen af et MVVM-projekt

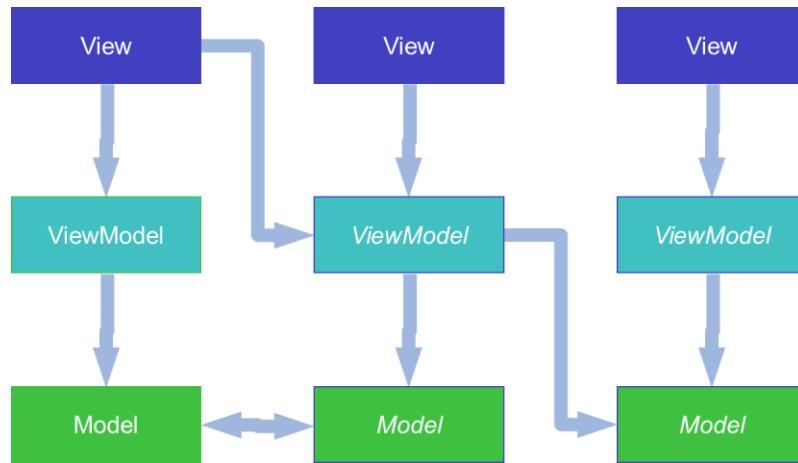
Det er ikke MVVM-projektet i sin helhed, der er struktureret i View-, ViewModel- og Model-moduler – det er det enkelte units, der er splittet op i lag som i View, ViewModel og Model. Traditionelle Delphi-forms og units skelnede ikke ved at høre til et bestemt lag, og en form kunne stort set varetage alle ansvarsområder, der hørte under denne form, vertikalt. Nedenstående diagram er naturligvis blot for at fremhæve de forhold, som er fremherskende i traditionelle Delphi-projekter, med forms der har referencer til andre forms, og at vi ofte har en MainForm, der kan have kontrol over, hvilke forms der efterfølgende oprettes og vises på skærmen.



I MVVM-pattern ville vi undgå referencer mellem de enkelte Views, men det vil ikke alene være svært i refaktoreringssammenhænge, det vil også være nærmest umuligt uden et værktøj uden et MVVM-framework. Den primære årsag til, at vi gerne vil undgå referencer og eksplisit instantiering af forms er selvfølgelig, at dette skaber en meget tæt kobling, men i andre udviklingsværktøjer er denne problemstilling indarbejdet i deres MVVM-framework, hvor der f.eks. kan være en action (typisk omtalt som "command"), samt måske en XAML-fil og en hel infrastruktur med data binding, samt binding-mekanisme til ViewModels, der kan instantiere et nyt View helt automatisk. Et View vil ofte ikke have nogen kodelinjer, og de enkelte komponenter vil være bundet til bestemte actions i en ViewModel, hvor et framework sørger for automatikken i bindingerne. Et View binder sig

til et ViewModel, og i mange henseender ender denne mekanisme ofte blot med at skabe 1-1 relationer, selvom MVVM-pattern kan rumme mere end det.

Et sådant framework har vi ikke i Delphi, og det vil i praksis skabe en masse udfordringer, fordi vi så gerne vil begrænse koblingerne mellem mange Views og ViewModels, men at vi jo naturligvis er nødt til selv at instantiere Views og ViewModels efter behov. Selve hændelsen fra en Form, hvor en bruger trykker på en knap, der skal åbne et nyt vindue, kan uden problemer lægges som action i en ViewModel, men vi vil bryde et princip om løs kobling, hvis denne ViewModel skal instantiere et View, som så også instantierer relevante ViewModels og herunder Models.



Som det fremgår af figuren herover, så får vi ikke en fuldkommen løs kobling, men vi ville ønske en løsere kobling ved, at det kun er øvre lag fra View, der kender konkrete ViewModels, og ViewModels, der kender konkrete Models. Views kan kende flere ViewModels, ligesom ViewModels kan kende flere Models. Vi kan også have referencer på tværs af Models, fordi forretningslogik netop kan omfatte mange områder, der skal arbejde sammen og supplere hinanden. Vi er ikke så vilde med referencer imellem ViewModels, fordi en ViewModel helst skal repræsentere en ikke-grafisk model af en eller flere Models, men dermed ikke sagt at vi ikke må – og særlig i refaktoreringssammenhænge kan dette også være nødvendigt.

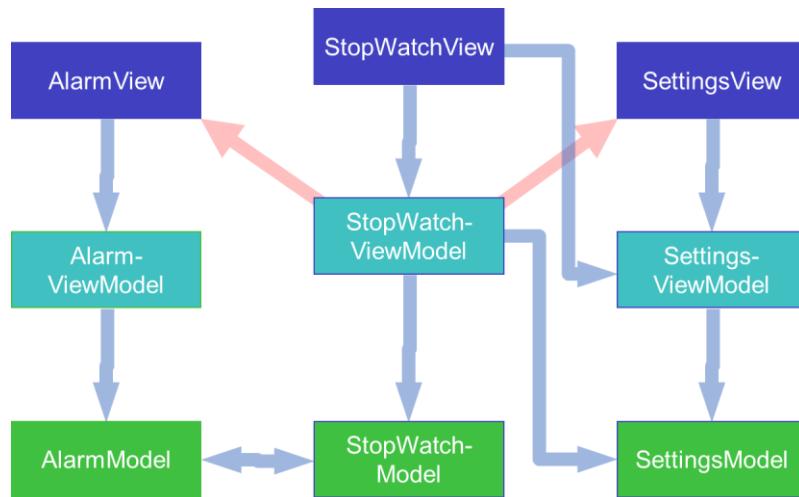
Hvis vi kunne, så ville vi helst være fri for referencer mellem Views, fordi det vil betyde en tæt kobling, og ovenikøbet en rigtig slem en af slagsen. Vi skal helst kunne skifte et View ud med et nyt, uden at det får vidtrækkende ændringer på tværs af en applikation. Og formålet med refaktorering mod MVVM vil jo netop være muligheden for, at vi til enhver tid kan finde på at udskifte et View – f.eks. hvis firmaets CMO pludselig finder på, at vi nu skal have et helt nyt look, eller at der er kommet nogle forretningsmæssige tiltag, der har indflydelse på, hvordan vi håndterer data. Det er en udfordring, når vi ikke har et framework, der understøtter MVVM og en automatisk instantiering af View og ViewModels, og et data bindingsinfrastruktur, som på anden måde end referencer vil være defineret til at være bundet sammen.

Men vi må vælge at lade være med at anskue MVVM så horisontalt, fordi Views, ViewModels og Models i praksis vil være hierarkisk forskudt i forhold til hinanden, nøjagtig som klassiske Delphi-forms, som har en MainForm, som igen har underliggende forms osv. I MVVM vil vi på samme måde have en "MainView", eller rettere en "MainViewModel", som i praksis vil stå over andre Views, hvis ViewModels igen kan stå over andre underliggende Views.

Når vi anskuer et architectural pattern som MVVM, så kan vi have en tendens til at betragte f.eks. View-laget som et horisontalt lag, men på et eller andet niveau må et View være hævet over andre Views, ligesom mange ViewModels må være hævet over nogle Views, fordi der skal tages stilling til hvilke Views og ViewModels, der skal instantieres på foranledning af f.eks. brugerens valg.

Instantiering af Views

Det er et generelt problem for udviklingsværktøjer, som ikke har et MVVM-framework integreret, at instantiering af især Views ikke kan ske uden konkrete referencer, medmindre vi er nødt til at indarbejde en større løsning med registrering af Views og indirekte referering og automatisk instantiering, når der er behov for et eller flere Views. Selv med en simpel registreringsmekanisme, så kan det blive meget svært at retfærdiggøre løsningen i forlængelse af refaktorering. Derfor vil vores umiddelbare tilgang være konkrete referencer til Views fra ViewModels, og en traditionel hierarkisk opdeling af Views.



Vi tager udgangspunkt i vores Stopwatch-projekt, som vi har udvidet med et View mere for at skabe en lidt større model, så vi kan fokusere på referencer. Ovenstående diagram viser referencer i form af de blå pile, og repræsenterer referencer, som vi "allernødigst" vil acceptere, hvorimod vi også må have referencer, som ikke er så ønskelige, men nødvendige, som er vist med de røde pile. Et gennemgående træk ved de uønskede referencer er, at de er erklæret i implementationssektionen, fordi der jo ikke nogen definitioner, vi er interesseret i – vi vil blot have noget Views instantieret, og de har afhængigheder nedad,

som de nok selv skal få initialiseret. Det er også værd at notere, at ingen unit har referencer, der springer over et lag.

Samtidig har vi et eksempel med en AlarmModel, der har en reference til en StopWatchModel, fordi den sidstnævnte model har en løbende klokke, som alarmfunktionen kan benytte sig af. Dvs. at AlarmModel registrerer en observer i StopWatchModel, når en alarm er defineret og aktiveret, som den så holder op imod aktuel klokke. Det er en simpel konstruktion, men det viser også den "kompleksitet", som vil være tilstede, hvis vi ikke er familiær med hverken MVVM og Observer Pattern. Og at det er meget nemt at rive disse modeller og referencer fra hinanden, hvis ikke disse patterns følges nøje og alle udviklere er enige om dette.

Når vi taler om instantiering, så er det fordi vi normalt ikke vil lade projekt-filen afgøre i hvilken rækkefølge og hvornår de enkelte Views bliver oprettet. Der er jo et implicit hierarki defineret i et pattern som MVVM, men at vi selvfølgelig ikke vil kunne komme udenom, at der er en MainForm, som vi kender det. Vi kan bare kalde den for MainView, når vi taler om MVVM-pattern. Det View kan vi sagtens lade projekt-filen håndtere, og den skal nok instantiere den eller de ViewModels, den måtte være afhængig af, som igen kan instantiere Models og øvrige Views om nødvendigt.

```
uses
  Vcl.Forms,
  All.StopWatch.View in 'All.StopWatch.View.pas' (StopWatchView),
  All.StopWatch.ViewModel in 'All.StopWatch.ViewModel.pas' (StopWatchViewModel)
  All.StopWatch.Model in 'All.StopWatch.Model.pas' (StopWatchModel: TDataModule)
  All.Settings.View in 'All.Settings.View.pas' (SettingsView),
  All.Settings.ViewModel in 'All.Settings.ViewModel.pas' (SettingsViewModel: T
  All.Settings.Model in 'All.Settings.Model.pas' (SettingsModel: TDataModule),
  All.Alarm.View in 'All.Alarm.View.pas' (AlarmView),
  All.Alarm.ViewModel in 'All.Alarm.ViewModel.pas' (AlarmViewModel: TDataModul
  All.Alarm.Model in 'All.Alarm.Model.pas' (AlarmModel: TDataModule);

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TStopWatchView, StopWatchView);
  Application.Run;
end.
```

Og når vi har et View som StopWatchView, som har nogle knapper, som er bundet sammen med nogle actions, der er defineret i et ViewModel, og på den måde også har gjort sig afhængig af StopWatchViewModel, så er det StopWatchView, som skal sikre at et StopWatchViewModel er instantieret. Derfor vil vi også se kodelinjer i initialisering af vores View, der markerer at vi har brug for en StopWatchViewModel.

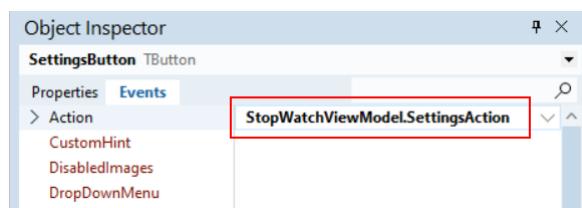
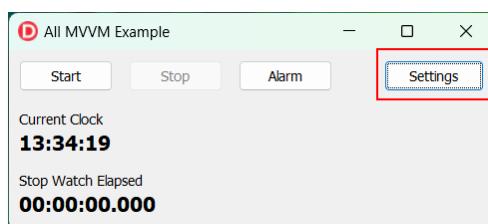
```
procedure TStopWatchView.FormCreate(Sender: TObject);
begin
  FHeightOffset := ClientHeight - ClockDisplay.Height * 2;
  FBottomOffset := ClientHeight - (StopWatchDisplay.Top + StopWatchDisplay.Hei
  TStopWatchViewModel.ViewModelNeeded(Owner));
  TSettingsViewModel.ViewModelNeeded(Owner);
end;
```

Bemærk, at initialiseringen af StopWatchView også har en afhængighed til en SettingsViewModel, fordi der skal læses indstillinger til præsentering af klokke og stopur, men det er vigtigt at adskille afhængighed fra brugerens ønske om at se et View med indstillingerne, som også har en afhængighed til en SettingsViewModel. Det er vidt forskellige egenskaber, og de skal helst gerne kunne fungere uafhængigt af hinanden.

I vores eksempel har vi løst denne afhængighed ved at definere en klasse-metode for TStopWatchViewModel, der instantierer en StopWatchViewModel, hvis denne ikke allerede er initialiseret. Dvs. at metoden kan kaldes mange gange af forskellige Views, som måtte have interesse i netop denne ViewModel. Metoden er simpel, og selvom det i de fleste tilfælde blot vil være en enkel View, der benytter den ene ViewModel, så er det en god praksis, der kan sikre, at MVVM kan skaleres.

```
class procedure TStopWatchViewModel.ViewModelNeeded(const Owner: TComponent);
begin
  if StopWatchViewModel = nil then
    StopWatchViewModel := TStopWatchViewModel.Create(Owner);
end;
```

Med andre ord, så kan bindingerne i knapperne i vores View benytte de actions, der er defineret i ViewModel, og som vi har defineret, da vi designede StopWatchView.



Denne action består blot af et kald til en klasse-metode, der har en opgave om at instanciere SettingsView om nødvendigt, og efterfølgende vise den:

```
procedure TStopWatchViewModel.SettingsActionExecute(Sender: TObject);
begin
  TSettingsView.ShowSettingsView(Owner);
end;
```

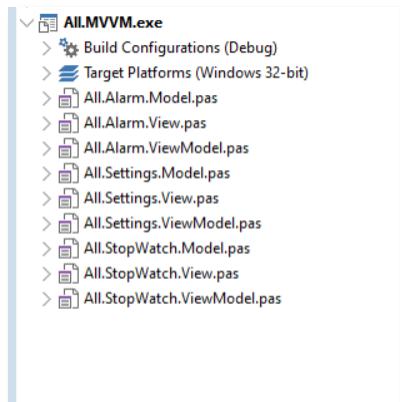
Og under TSettingsView:

```
class procedure TSettingsView.ShowSettingsView(const AOwner: TComponent);
begin
  if SettingsView = nil then
    SettingsView := TSettingsView.Create(AOwner);
  SettingsView.Show;
end;
```

På samme måde vil SettingsView sikre sig instantiering af sine afhængigheder til SettingsViewModel, når den initialiserer, og det er ikke noget som StopwatchViewModel skal bekymre sig om, selvom den er udløsende instance. Denne tilgang i instantiering af forskellige Views er god praksis i betragtning af, at vi mangler et framework, der understøtter netop dette behov i vores forsøg på at tilnærme os MVVM. Det er ikke den eneste måde at gøre det på, men i refaktoreringssammenhænge er det en tilgang, vi kan leve med. Om vi på et senere tidspunkt vil udvikle en anden og mere indirekte måde at instantiere Views på, så er det en udfordring, der er værd at udforske, selvfølgelig med sigte på at opnå en højere grad af løs kobling. Mange frameworks og løsninger sigter på en central enhed, hvor de forskellige Views, ViewModels og Models kan registrerer sig, og hvor de kan instantieres og manifesteres ved hjælp af f.eks. et mærkat eller anden form for indirekte udtryk, uden at de forskellige enheder konkret kender hinanden.

Struktuering af filerne

Når vi snakker om referencer, så vil vi heller ikke kunne komme udenom organisering af filerne, ligesom behovet vil vokse, når vi nærmest får 3 gange så mange filer end normalt pr. form. Mange projekter har allerede et eller andet mappe-struktur, som har virket naturligt, og traditionelt har vi også haft mapper som f.eks. "Library" og "Common". Den struktur skal man bare fortsætte med, og det er især mappen med forms, som vil være interessant i MVVM-sammenhænge, og i vores eksempel har vi da også taget udgangspunkt i, at samtlige filer ligger under samme mappe for at fremhæve udfordringen – jo større projekt, jo flere filer, jo større udfordring:

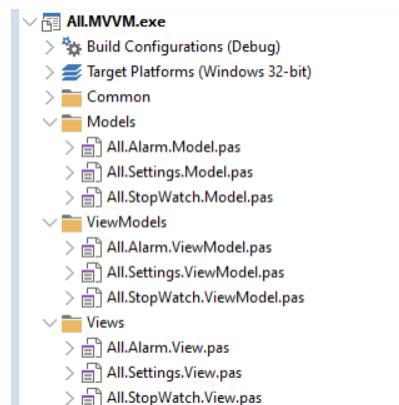
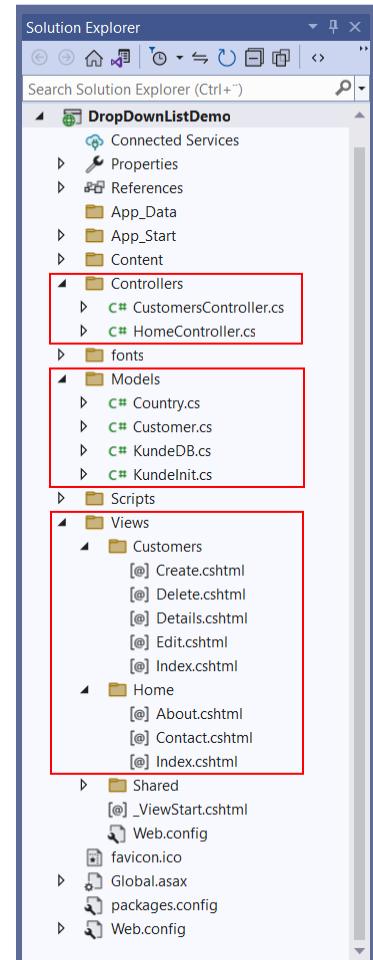


I forskellige værktøjer, der understøtter architectural patterns MVP, MVVM og MVC, har der traditionelt været et mappe-struktur, der afspejler det lag, som de enkelte filer tilhører, f.eks Views, Controllers og Models i MVC, eller Views, ViewModels og Models osv. Værktøjer, som understøtter et af disse architectural patterns har selvfølgelig nogle wizards, der kan hjælpe udvikleren med at oprette f.eks. et View og gemme tilhørende filer de rigtige steder, som udvikleren efterfølgende kan redigere i.

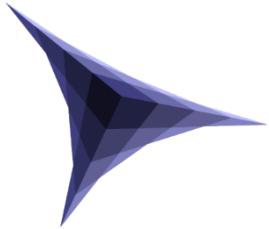
Til højre er mapperne f.eks. illustreret i en Solution Explorer fra Visual Studio fra en typisk applikation fra et kursus i ASP.NET Core MVC. Filerne er struktureret i deres respektive Views, Controllers, og Models mappe, der er let at adskille fra de øvrige mapper, fordi vi ved at værktøjet har integreret et MVC-framework. Andre værktøjer kan have en anden tilgang, men vi vil kunne genkende mapperne på det pattern, de er struktureret efter.

Delphi understøtter ikke nogen architectural pattern, så vi må selv sørge for at placere de rigtige forms og moduler i de rigtige mapper. Som vi også kan se på andre værktøjer, så er mapperne ikke begrænset til de 3 mapper, men at vi både kan have parallelle mapper og undermapper til f.eks. Views.

I vores eksempel med StopWatch-projektet har vi ikke så mange filer, men større projekter kan have et endnu større behov for at strukturerer Views i flere undermapper. For nogle værktøjer og frameworks kan der være langt flere filer at skulle håndtere, fordi filerne kan være opdelt helt ned til de enkelte klasser og funktioner, hvorimod vi i Delphi kan acceptere at holde denne opdeling af filer på traditionelle units fordelt på en form og et par modeller.



Der er ikke noget i MVVM-pattern, der dikterer at der skal være én ViewModel for hver Model, ligesom et View ikke betyder at der er et tilsvarende ViewModel. En ViewModel kan sagtens dække over en række Models, som omfatter hvert sit forretningsområde, men for et View vil det virke som om, at der blot er én underliggende Model. Med andre ord, så kan der sagtens være to Views, der præsenterer samme ViewModel på forskellige måder, som igen kan være et ViewModel over f.eks. tre forskellige Models. Det kræver disciplin i navngivning af de enkelte filer, men det er klart en fordel at der er et 1-1 forhold mellem filerne under Views, ViewModels og Models.



9

Kapitel

Generisk Observer Pattern

Med implementering af Observer Pattern i samtlige ViewModels og Models, også i Views om nødvendigt, så finder vi hurtigt ud af, at vi vælter rundt i samme kodelinjer i samtlige defineringer og implementeringer af observerbare klasser. Vi kunne definere nogle fælles klasser, som Views, ViewModels og Models kunne nedarve, men det er en ekstrem sårbar tilgang, som Delphi's IDE kan have store problemer med at håndtere, når vi har at gøre med især TForm og TModule. Men selv de simpleste implementeringer kan synes at fylde voldsomt, når det er de samme kodelinjer, der går igen og igen, ligesom defineringen kan være den samme, måske med små variationer:

```
uses
  Generics.Collections;

type
  TModelEvent = (mmDestroying);
  TModelObserver = procedure(const AEvent: TModelEvent;
    const AValue: Variant) of object;
  TModelObserverList = TList<TModelObserver>;

  TModel = class(TObject)
  protected
    FObservers: TModelObserverList;
    procedure NotifyObservers(const AEvent: TModelEvent; const AValue: Variant
public
  destructor Destroy; override;
  procedure RegisterObserver(const AObserver: TModelObserver);
  procedure UnregisterObserver(const AObserver: TModelObserver);
end;
```

Denne minimale definering af Observer Pattern fylder også i implementeringen, og hvis det er de samme kodelinjer med en generisk metode til observerne, så er det rigtig mange kodelinjer, såkaldt boilerplates, der går igen og igen.

```

  destructor TModel.Destroy;
  begin
    FreeAndNil(FObservers);
    inherited Destroy;
  end;

  procedure TModel.NotifyObservers(const AEvent: TModelEvent; const AValue: Variant);
  var
    I: Integer;
  begin
    if Assigned(FObservers) then
      for I := 0 to FObservers.Count - 1 do
        FObservers[I](AEvent, AValue);
  end;

  procedure TModel.RegisterObserver(const AObserver: TModelObserver);
  begin
    if FObservers = nil then
      FObservers := TModelObserverList.Create;
    if FObservers.IndexOf(AObserver) = -1 then
      FObservers.Add(AObserver);
  end;

  procedure TModel.UnregisterObserver(const AObserver: TModelObserver);
  begin
    if FObservers <> nil then
      FObservers.Remove(AObserver);
  end;

```

Når vi har udfordringer med at nedarve fra fælles klasser, ligesom definering af et observable interface alligevel vil koste kodelinjer i både interface-sektionen og implementeringssektionen, så kalder tilgangen på en alternativ løsning, som er til at leve med, i særdeles når vi er i gang med en større refaktorering. Vi har implementeret en link-tilgang i nogle af vores eksempler, som er vidt udbredt i data-bindingssammenhænge i mange frameworks, også i andre patterns end MVVM, men selv med denne tilgang kræver det nogle kodelinjer og initialisering, hvor den må implementeres.

Denne tilgang med links (figuren til højre) kan, med ganske få ændringer, sagtens udvikles til at håndtere et utal af observables og observere i én og samme fælles klasse, hvor vi kan slippe for at skulle implementere og initialisere i samtlige observerbare klasser igen og igen. Det vil koste lidt i performance, på samme måde som Observer Pattern allerede har en lille straf i performance, men til gengæld er tilgangen nærmest vedligeholdelsesfri.

Denne tilgang har en snert af Broker Pattern eller Mediator Pattern, og måske ikke så meget Publish-subscribe Pattern, men vi udnytter tilgangen om at bruge en mediator eller et mellemled, bare med en klasse, der har implementeret Observer Pattern. Dette mellemled kommer med samme fordel som de nævnte patterns, fordi vi kan opnå en lang højere grad af afkobling, hvis vi kan lykkes

```

uses
  Common.Model.Links;

const
  mmDestroying = $0001;

type
  TModel = class(TObject)
  private
    FModelLink: TModelLink;
  public
    constructor Create;
    destructor Destroy; override;
    property ModelLink: TModelLink read
  end;

implementation

uses
  System.SysUtils;

{ TModel }

constructor TModel.Create;
begin
  FModelLink := TModelLink.Create;
end;

destructor TModel.Destroy;
begin
  FModelLink.NotifyObservers(mmDestroy);
  FreeAndNil(FModelLink);
  inherited Destroy;
end;

```

med at afskære objekterne fra hinanden. Ved denne tilgang vil vi dog stadig skulle bruge instance-referencerne for vores Views, ViewModels og Models, eller et hvilket som helst komponent, som et observerbart objekt. Men her vil vi ikke engang være sikre på, at et aktuelt instance af en komponent rent faktisk vil kunne være i stand til at sende notifikationer, ligesom det samme observerbare objekt heller vil ikke kunne vide, om der overhovedet er observere, der lytter.

Ideen går ud på at definere en central klasse, som er udviklet til at håndtere observere alene på baggrund af en reference til et objekt – om det objekt er instantieret eller understøtter Observer Pattern er ikke en forudsætning under registrering af en observer, bortset fra at der i så tilfælde selvfølgelig ikke vil ske noget. Vi har allerede en definering af TModelLink, og denne benyttes i forlængelse af den centrale klasse.

```
type
  TObservables = TDictionay<TComponent, TModelLink>;
  TObservable = class(TObject)
  private
    class var Observables: TObservables;
  public
    class destructor Destroy;
    class procedure NotifyObservers(const Observable: TComponent; const Event: string);
    class procedure RegisterObserver(const Observable: TComponent; const Observer: TModelLink);
    class procedure UnregisterObserver(const Observable: TComponent; const Observer: TModelLink);
  end;
```

Det er en ren klasse, så der bliver aldrig lavet en instance af klassen, men består udelukkende af klassevariabler og -metoder, som vil kendetegne Observer Pattern:

```

class procedure TObservable.NotifyObservers(const Observable: TComponent; const Event: TNotifyEvent; Value: Variant);
var
  Link: TModelLink;
begin
  if Observables <> nil then
    if Observables.ContainsKey(Observable) then
      begin
        Link := Observables.Items[Observable];
        Link.NotifyObservers(Event, Value);
      end;
    end;

class procedure TObservable.RegisterObserver(const Observable: TComponent; const Observer: IObserver);
var
  Link: TModelLink;
begin
  if Observables = nil then
    Observables := TObservables.Create;
  if not Observables.ContainsKey(Observable) then
    Observables.Add(Observable, TModelLink.Create);
  Link := Observables.Items[Observable];
  Link.RegisterObserver(Observer);
end;

class procedure TObservable.UnregisterObserver(const Observable: TComponent; const Observer: IObserver);
var
  Link: TModelLink;
begin
  if Observables <> nil then
    if Observables.ContainsKey(Observable) then
      begin
        Link := Observables.Items[Observable];
        Link.UnregisterObserver(Observer);
      end;
    end;
end;

```

Det er tydeligt, at den eneste forskel, mellem vores oprindelige eksempel på Observer Pattern og en central implementering, er, at vi har flere lister over observers, der hver er knyttet til en observérbar reference i form af en object, nedarvet fra en TComponent. Det behøver ikke at være en TComponent, men vores Views, ViewModels og Models vil arve fra TComponent, og i andre situationer kan deres properties, såsom Owner, være interessante. Denne simple implementering skal afsluttes med oprydning.

```

class destructor TObservable.Destroy;
var
  Link: TModelLink;
begin
  if Observables <> nil then
    begin
      for Link in Observables.Values do
        Link.Free;
      FreeAndNil(Observables);
    end;
end;

```

Vi kan udbygge eller udvikle en anden tilgang med en central implementering af Observer Pattern, der er indrettet efter behov og applikation, men dette lille eksempel alene vil betyde en lang mindre aftryk i resten af Observer Pattern på tværs af en applikation. En simpel tilgang med både model-link og en central liste over observables kan holdes på under 100 kodelinjer, og der er en meget lille performance penalty, som måske kan hentes på andre områder.

Observables

En fælles klasse kan holde styr på observerbare objekter, og observables skal stort set ikke initialisere noget eller registrere sig selv, fordi klassen kan oprette en liste på vegne af et objekt, hvis et sådant ikke findes for dennes observere. En observable har kun en opgave, og det er at sende notifikationer afsted, hvor det er behørigt som f.eks. herunder:

```
procedure TSettingsModel.SetEnabled(const Value: Boolean);
begin
  if Value <> FEnabled then
  begin
    FEnabled := Value;
    TObservable.NotifyObservers(Self, meEnabledChanged, FEnabled);
  end;
end;
```

Til sammenligning med tidligere implementeringer af NotifyObservers, så tager den nye metode et argument med, som skal bruges til at identificere den ene observable fra de andre. Hvis der er nogle observere, der har brugt denne reference, som i ovenstående eksempel vil være variablen SettingsModel, og registreret sig, så vil der være en liste knyttet til referencen. Hvis der ikke er nogen observere registreret, så bliver notifikationen simpelthen bare ikke sendt rundt.

Et andet vigtigt element ved observables, er definering af de events, som kan sendes med notifikationer til observere, og disse definitioner har indtil videre ligget samme sted som defineringen af en observable. Det vil selvfølgelig være et stort problem, at vi har definering af events liggende sammen med definering af en observable, hvis vi rent faktisk gerne vil frakoble observere fra observables, så de ikke har referencer til hinanden, og som tidligere nævnt, så er dette også muligt. Denne løsning er dog udelukkende sightet efter en rationalisering af mange Observer Pattern-implementationer, men den tegner meget tydeligt op, at vi blot mangler at separere events og en reference som SettingsModel, for at vi har opnået en fuldkommen løs kobling.

Observers

En observer skal stadig implementere en metode til at modtage events fra en observable, og en observer skal stadig registreres og afregistreres på behørige tidspunkter. I modsætning til før, så sker registreringen nu i en central klasse med en reference til det objekt, vi gerne vil observere. Det er ikke nødvendigt, at objektet er instantieret, fordi det er den samme reference i hukommelsen uanset om referencen er nil eller en pointer. Derfor vil der under alle omstændigheder blive oprettet en observer-liste til den reference, hvis der ikke er nogen liste i forvejen.

```
procedure TSettingsView.FormShow(Sender: TObject);
begin
  TObservable.RegisterObserver(SettingsViewModel, SettingsObserver);
end;
```

Der er ikke nogen forskel i metoden, der modtager notifikationer, så de vil fungere som de altid har gjort.

Det er muligt at tænke i flere klasser end den ene fælles klasse til observables, så der skelnes mellem grupper af observables. Her tænkes især på moduler i en applikation, der forgår efter at være oprettet og nedlagt, så deres referencer og liste over observere ikke ligger og fylder en fælles klasse – jo mere kompakt en liste er, jo færre opslag behøver vi at lave. Her er plads til at udvikle en mediator, der er tilpasset de forskellige behov i et system.

Observer Metode

Når vi implementerer en fælles klasse for observables som foren, så giver det også en udfordring med at definere en fælles observer-metode, der er fleksibel nok til at håndtere alle tænkelige handlinger med tilpas antal parametre. Vores observer-metode er i bund og grund en *callback*, som forventes at blive kaldt asynkront og ved hændelser, som vi selv har defineret. I Windows-miljø er det ikke usædvanligt med callbacks, og vi har alle stiftet bekendtskab med Winodws Messages på et eller andet tidspunkt, og de er jo sædvanligvis kædet sammen med formens Wndproc, hvor alle messages behandles.

```
LRESULT Wndproc(  
    HWND unnamedParam1,  
    UINT unnamedParam2,  
    WPARAM unnamedParam3,  
    LPARAM unnamedParam4  
)
```

Det er ikke usædvanligt i mange miljøer, men selve definering af parametrene har overlevet siden de tidligste udgaver af Windows, hvilket vil sige operativsystemet har formået at definere en callback, der kan fungere for mange typer af hændelser. I parametrene indgår en *handle* og en *message*, samt to data-parametre. Samme udfordring står vi også med i vores observer-metode, og hidtil har vi stillet os tilfreds med *Event* og *Value* i vores eksempler – samme som *message* og *data*. Sådan behøver en observer-metode ikke at se ud, og det selvfølgelig afhænge af behovet i en given applikation.

```
TModelObserver = procedure(const Event: Integer; const Value: Variant) of ob
```

En mere generisk observer-metode kan sagtens defineres med flere parametre, men det er klart at jo færre argumenter, der skal bringes videre, jo mere overskueligt bliver det og jo bedre performance får vi. En observer-metode kan lige så godt defineres som herunder, men det handler om at finde den balance, som passer bedst til applikationen, uden at der skal argumenteres med unødige data, eller at en callback helst ikke må trække yderligere data fra afsenderen efterfølgende, fordi der mangler argumenter.

```
TModelObserver = procedure(const Sender: TObject;  
const Event, Value: Integer; const Data: Pointer) of object;
```

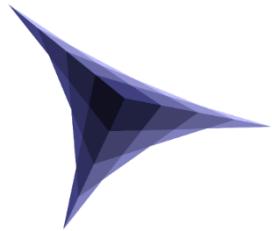
Mange Observer Pattern-implementationer bygger også på registrering af interfaces eller objekter, som kan indeholde alle nødvendige data om observer og f.eks. flere typer af OnEvent-handlere, samt et objekt, der indeholder data om en hændelse, men de kan både være komplekse og kræve mere vedligeholdelse over tid end det helt simple i callbacks. Observere som interfaces introducerer også fleksibilitet, som kan omfatte flere typer af callbacks, som en observer skal implementere, og som passer til forskellige situationer. De forskellige tilgange er værd at udforske, men komplekse løsninger kan introducere rigtig meget arbejde for en simpel observer, der blot ønsker at lytte på en given tilstand (f.eks. enabled/disabled), og derfor har callbacks altid kunnet levere de simpleste løsninger, som også er muligt at inddarbejde i en Observer Pattern-implementation, fordi det er muligt at definere dette pattern med metoder.

Hvis vi kigger på Delphi's data-aware-løsning, så vi vil hurtigt konstatere, at det også er en implementering af Observer Pattern, der er udviklet til at løse kommunikationen mellem 2 eller flere komponenter, men at det også er bygget på en tættere kobling. Oftest er det dog løst med, at de reelle komponenter arver egenskaber fra en fælles klasse, som database-komponenter er bygget over og kender, så vi f.eks. vil se TClientDataSet, TADODataSet, TSQLDataSet og lignende implementeringer alle har en stam-klasse, som data-aware-komponenterne kender – TDataSet.

Nogle af de forskellige observere i deres løsning er også bygget på en callback, selvom der nu er en datalink imellem. Datalinket har ikke altid været der, så der tidligere var en direkte registrering af "clients" mellem f.eks. en TDataSource og en TDataSet. Datalinket giver lidt større frihed, men også overhead, og callbacks er stadig det samme princip. Metoden til callback er meget simpel, idet der blot er en event (message) og en data i form af en integer. Argumentet til data kan type-castes afhængig af eventet, f.eks.:

```
procedure TFormLink.DataEvent(Event: TDataEvent; Info: NativeInt);  
begin  
  case Event of  
    deFieldChange, deRecordChange:  
      RecordChanged(TField(Info));  
  end;  
end;
```

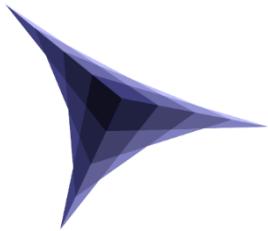
Observer Pattern er sådan set ikke nogen ny tilgang, heller ikke i Delphi-sammenhænge, og implementeres i mange forskellige afskygninger i både Delphi's egne komponent-biblioteker, og er ret så gennemgående i Windows operativsystem og forskellige frameworks.



Kapitel **10**

Architectural Patterns

Resume/konklusion og flere appendikser følger...



Appendiks A

Non-visual komponenter

Uanset om vi separerer vores projekt i henhold til et pattern eller ej, så vil disse komponenter under alle omstændigheder volde udfordringer, men vi vil alt andet lige have gjort vores arbejde med migrering noget nemmere og mere overskueligt, fordi vi vil have separeret vores forretningslogik fra handlinger og præsentationen. Listen herunder over non-visual komponenter er en hjælp til at skabe et billede af, hvor stor en udfordring en migration fra VCL til FMX kan gå hen og blive:

Komponent	VCL-units	FMX-units
TMainMenu, TPopupMenu	VclMenus	FMX.Types, FMXMenus
TActionList	VclActnList	FMXActnList
TImageList	VclImgList, VclControls	FMXImgList
TActionManager	VclPlatformDefaultStyleActnCtrls, SystemActions, VclActnList, VclActnMan	<i>Missing</i>
TPopupActionBar, TStandardColorMap, TTwilightColorMap	VclPlatformDefaultStyleActnCtrls, VclMenus, VclActnPopup	<i>Missing</i>
TCustomizeDlg	VclCustomizeDlg	<i>Missing</i>
TApplicationEvents	VclAppEvnts	<i>Missing</i>
TTrayIcon	VclExtCtrls	<i>Missing</i>
TBalloonHint	VclControls	<i>Missing</i>
TXPManifest	VclXPMan	<i>Missing</i>
TShellResources	VclShellAnimations	<i>Missing</i>
TTaskbar	SystemWinTaskbarCore, VclTaskbar	<i>Missing</i>
TJumpList	VclJumpList	<i>Missing</i>
TTimer	VclExtCtrls	FMX.Types
TCOMAdminCatalog	VclOleServer, VclCmAdmCtl	<i>Missing</i>

TDdeClientConv, TDdeClientItem, TDdeServerConv, TDdeServerItem	Vcl.DdeMan	<i>Missing</i>
TSharingContract	Vcl.ShareContract	<i>Missing</i>
TOpenDialog, TSaveDialog	Vcl.Dialogs	FMX.Types, FMX.Dialogs
TOpenPictureDialog, TSavePictureDialog, TOpenTextFileDialog, TSaveTextFileDialog	Vcl.Dialogs, Vcl.ExtDlgs	<i>Missing</i>
TFontDialog, TColorDialog, TFindDialog, TReplaceDialog, TFileOpenDialog, TFileSaveDialog, TTaskDialog	Vcl.Dialogs	<i>Missing</i>
TPrintDialog, TPrinterSetupDialog, TPageSetupDialog	Vcl.Dialogs	FMX.Types, FMX.Dialogs, FMX.Printer
TBindingsList	Vcl.Bind.DBEngExt	Fmx.Bind.DBEngExt
TWindowsStore	Vcl.WindowsStore	FMX.WindowsStore
TImageCollection	Vcl.BaseImageCollection, Vcl.ImageCollection	<i>Missing</i>
TVirtualImageList	Vcl.ImgList, Vcl.VirtualImageList	<i>Missing</i>
TGestureManager	Vcl.Controls, Vcl.Touch.GestureMgr	FMX.Types, FMX.Gestures
TFDConnection, TFDGUIxWaitCursor, TFDMSAccessService, TFDADSRestore, TFDADSUtility, TFDIBRestore, TFDIBValidate, TFDIBSecurity, TFDIBConfig, TFDIBInfo, TFDIBSDump, TFDFBNBackup, TFDFBNRestore, TFDFBTrace, TFDFBOnlineValidate, TFDSQLiteFunction, TFDSQLiteCollation, TFDSQLiteRTree, TFDSQLiteBackup, TFDSQLiteValidate, TFDSQLiteSecurity	FireDAC.VCLUI.Wait	FireDAC.FMXUI.Wait
TFDGUIxErrorDialog	FireDAC.VCLUI.Error	FireDAC.FMXUI.Error
TFDGUIxLoginDialog	FireDAC.VCLUI.Login	FireDAC.FMXUI.Login
TFDGUIxAsyncExecuteDialog	FireDAC.VCLUI.Async	FireDAC.FMXUI.Async
TFDGUIxScriptDialog	FireDAC.VCLUI.Script	FireDAC.FMXUI.Script
TAppAnalytics	Vcl.AppAnalytics	FMX.Analytics, FMX.Analytics.AppAnalytics