# Mandatory Patterns Assignment

## Chosen Pattern: Memento

SWD - Forår 2020

Gruppe 6


| Deltagere | | |
|---|---|---|
| **Navn** | **AU-ID** | **Studienummer** |
| Mikkel Kousgaard Rasmussen | au610713 | 201805606 |
| Jeppe Dybdal Larsen | au616966 | 201805466 |
| Magnus Hauge Kyneb | au615049 | 201807279 |


29. Marts 2020

# Contents

# Chapter 1

# Undersøgelse af memento

## 1.1 Beskrivelse

Memento er et adfærdsmæssigt design mønster, der bliver brugt, til at give muligheden for at genoprette objekter til deres tidligere tilstande. Dette mønster tillader altså at man kan lave undo's ved hjælp af rollbacks til tidligere tilstande. Mønstret består af tre klasser; Originator, Caretaker og Memento (se figur 2.1)

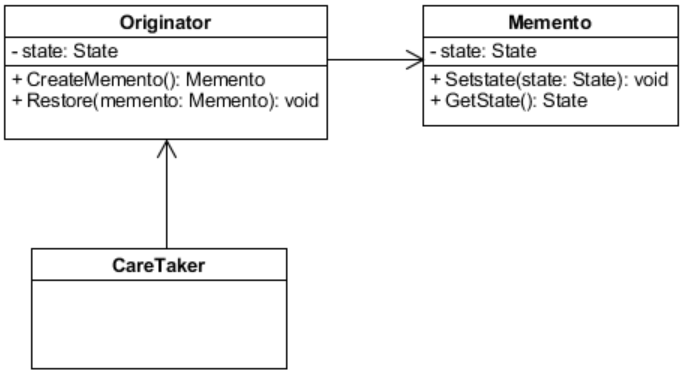


Figure 1.1: Klasse diagram: Memento design pattern

- **Originator:** Objekt som har en intern tilstand. Det er denne tilstand man gerne vil gemme/genoprette.
- **Caretaker:** Objekt der holder styr på forskellige mementos (tilstande) for `Originator`, dog uden direkte kendskab til hvad disse mementos har af data. Det er denne der holder styr på hvornår der skal kaldes hhv. `CreateMemento()` og `RestoreMemento()`.
- **Memento:** Objekt der fastholder en specifik tilstand for `Originator`. Man kan se den som et specifikt "snapshot" for `Originator`'en.

Ideen bag dette mønster er, at den interne tilstand for et objekt ønskes gemt eller genoprettet, men enkapsulationen for objektet må ikke brydes. Løsningen på dette er memonto objekter, som kun `Originator` selv kan oprette og tilgå. Caretakeren kan bede om disse memento objekter, og holde på dem (i en liste eksempelvis), hvilket gør, at den interne tilstand af objektet (`Originator`) netop er gemt eksternt. Caretakeren kan ligeledes give memento objekter tilbage til Originatoren, som derigennem kan ændre sin tilstand tilbage til den gemte tilstand i memento objektet. Det smarte ved mønstret er, at Caretakeren ikke behøves at kende Originatorens tilstand, den får blot et memento-objekt, uden at vide hvilken type den specifikt er.

Ift. SOLID-principperne, overholder memento-mønstret SRP-princippet. Da ansvaret for at gemme tilstande for Originatoren bliver opdelt i sin egen klasse, hvilket netop er Caretakeren.

Nedenfor ses et overordnet sekvensdiagram for, hvordan en Originator gemmer sin tilstand i en memento, når Caretakeren ønsker det, og hvordan en tilstand kan blive genoprettet igen.

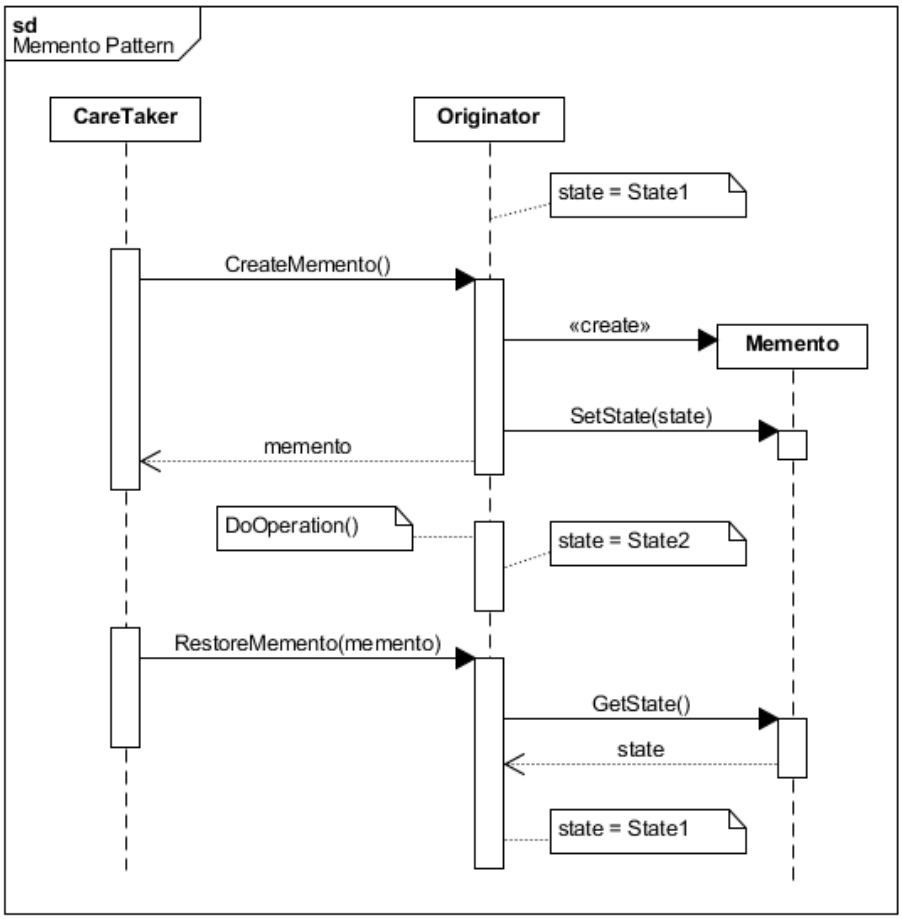


Figure 1.2: Sekvensdiagram: Memento design pattern (Save / Restore)

## 1.2 Perspektivering

Memento design mønstret tager en intern tilstand af et objekt, og pakker denne tilstand ind i en klasse for sig selv. Ansvaret for at opbevare den interne tilstand til et bestemt tidspunkt bliver altså fjernet fra objektet der har denne interne tilstand, og givet til memento objektet, hvis eneste opgave er at gemme denne tilstand til senere brug. Et andet design mønster der tager en tilsvarende tilgang til at løse problemer er Command mønstret. Her bliver en kommando gemt i en klasse for sig selv, hvormed andre klasser kan benytte denne kommando klasse, uden bekymring for hvordan kommandoen er implementeret. De to mønstre minder altså om hinanden på den måde at noget ansvar er gemt i en klasse for sig selv, og at klasserne kan sendes frem og tilbage for at blive kaldt på forskellige tidspunkter af andre klasser.

Man kan yderligere kombinere kommando mønstret med memento mønstret, hvis man eksempelvis ønsker at have en kommando der står for at lave redos.

# Chapter 2

# Design og implementering

## 2.1 Indledning

En oplagt applikation at bruge memento-pattern i, er i en tekstredigerings-app. Her skal man ved hjælp af undo-funktionen, kunne gå tilbage til sidst gemte teksttilstande. Omvendt skal man ved hjælp af redo kunne gå frem igennem de teksttilstande efter man har undo'et.

De fleste tekst-applikationer gemmer tilstande imens man skriver, uden man selv skal trykke save, hver gang man vil lave en tilstande at kunne undo/redo til.

Ved hjælp af memento-pattern kan en enkelt klasse kende til teksttilstanden indkapslet i et memento-objekt. Imens kan andre klasser bede denne om at gemme eller genoprette tilstande. I vores design ønsker vi netop dette.

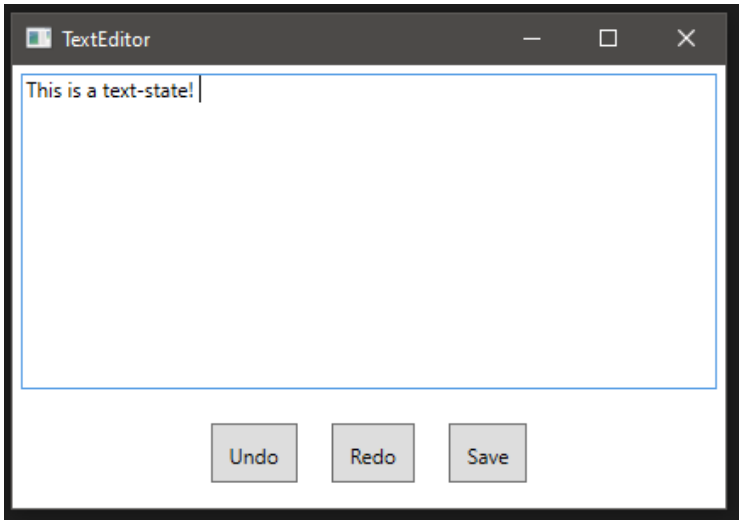


Figure 2.1: Tekst-applikation.

## 2.2 Design

Dette afsnit indeholder først en kort introduktion til design-idéen mht. memento-mønsret og herefter en mere detaljeret gennemgang af designet.

### 2.2.1 Introduktion

En TextSaver-klasse holder styr på hvornår der skal gemmes tilstande og holder på listen over tilstande. Her er det unikke ved memento igen, at TextSaver ikke kender til det den holder på.

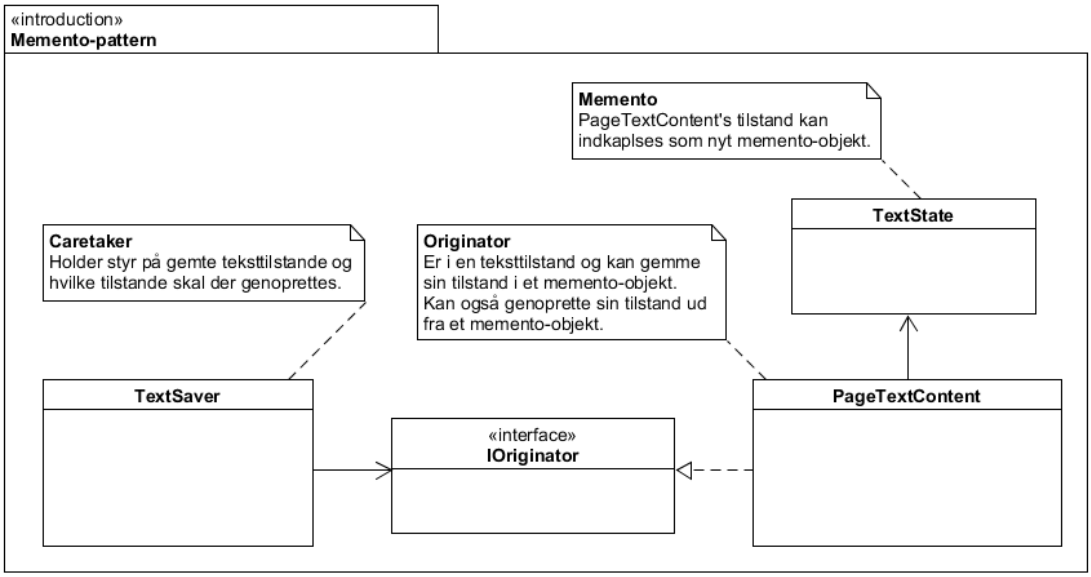


Figure 2.2: Klasse diagram: Memento-baseret tekst-applikation - introduktion.

TextSaver kan bede originator-klassen, PageTextContent, om at gemme sin (PageTextContent's) tilstand og udlevere den som et memento-objekt. PageTextContent kan også blive bedt om at genoprette sin tilstand ud fra en givet memento-objekt. Memento-objektet som holder teksttilstanden hedder i vores design TextState.

Kun PageTextContent kender til klassen TextState. TextSaver holder på et TextState som et generisk "object" uden at vide hvad det mere præcist er for et objekt. Det generiske objekt kan altid castes tilbage til at være et TextState i PageTextContent.

### 2.2.2 Overordnet design

Forneden i figur 2.3 ses det udvidede klassediagram for designet for klassebibloteket. Den indre-funktionalitet, kaldet `Memento-pattern`, består af funktionaliteten der har at gøre med memento-mønstret (hvilket netop er de klasser der blev introduceret tidligere i figur 2.2). Denne indre-funktionalitet består af hhv. en caretaker-klasse (`TextSaver`), en originator-klasse (`PageTextContent`) og en memento-klasse (`TextState`).

Udover den indre funktionalitet, er der desuden lavet interfaces til hhv. `TextSaver` og `PageTextContent`, i form af `ITextSaver` og `IPageTextContent`. Disse bruges som grænseflader imellem applikationen og mønstret.

Det skal dog nævnes, at ved at lave public get og set på den interne state i `PageTextContent`, er den ikke decideret intern længere. Dette er lidt et problem. Dog er vores argument for at gøre dette, at vi netop gerne vil linke `PageTextContent` til en tekstbok i et hovedprogram. Dette ville kun være muligt hvis vi enten udvidede en tekstboks med funktionaliteten for `PageTextContent` eller hvis vi, som vi netop har gjort, bare "simulerer" at tekstboksen og `PageTextContent` er en samlet enhed.

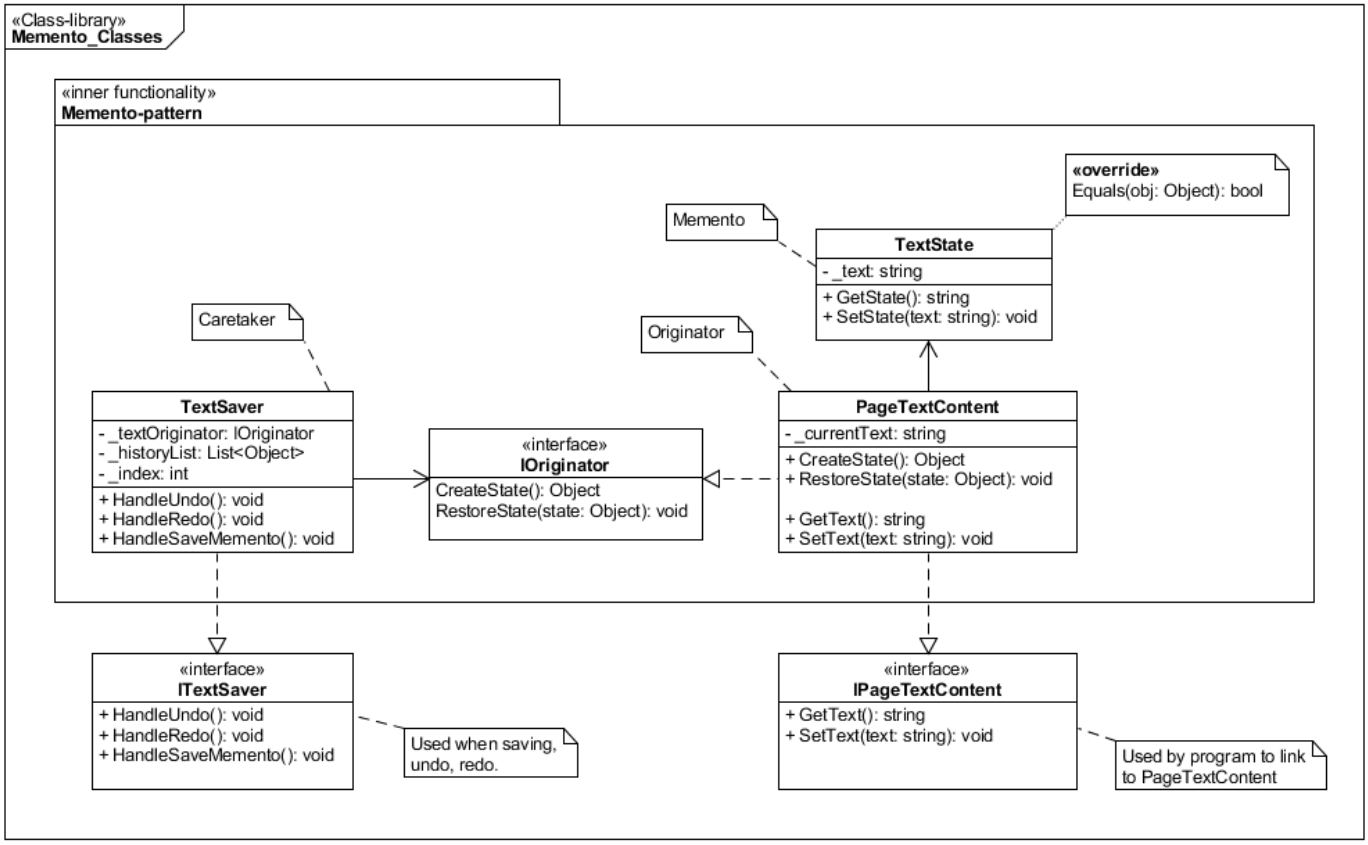


Figure 2.3: Klassediagram: Memento_Classes (klassebiblotek)

PageTextContent implementerer to forskellige interfaces; IOriginator og IPage-TextContent. Ved at dele funktionliteten op i to interfaces, overholdes SOLID ISP-prinicippet. Det undgås at TextSaver tvinges til at være afhængig af funktionerne GetText og SetText, som den ikke anvender. Omvendt kan interfacet til redigering af teksten i PageTextContent anvendes, uden at man dermed er tvunget til at være afhængig af originator-funktionalitet.

### 2.2.3 Historik: Opbevaring af memento

TextSaver opbevarer TextStates i en objekt-liste. Hver gang en TextState skal gemmes, beder TextSaver om et memento fra PageTextContent og lægger det i listen. Men hvor i listen mementoet skal lægges, kommer an på selve historikken.

For at administrere listen bruges en index-variable. Index-variablen peger som udgangspunkt på den senest gemte TextState. Hvis man ser bort fra undos og redos lægges nye saves ind bagest i listen, og index inkrementeres, så den peger derpå. Forneden ses netop dette.

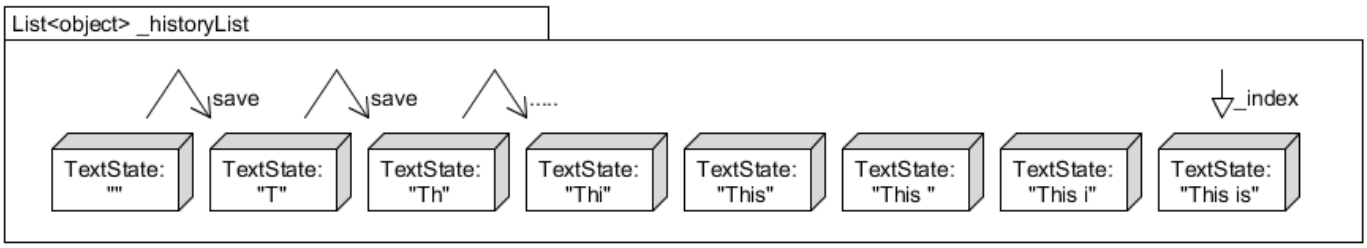


Figure 2.4: Listen i TextSaver, efter 7 saves, et ved hvert nyt tegn. Index-variablen peger på seneste memento.

Ved undos gås baglæns i listen. Dette sker ved at index-variablen dekrementeres og PageTextContent bedes genoprette mementoet der peges på. Index-variblen ligger nu et sted i midten af listen. Nyere mementoer ligger altså stadig i listen.

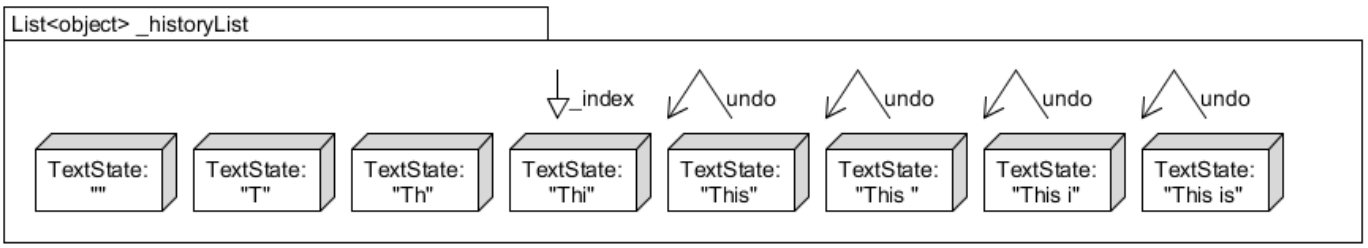


Figure 2.5: Fortsættelse af listen i TextSaver, efter 4 undos.

Redo har kun en funktion i forlængelse af undo, og går i modsatte retning. Index-variablen inkrementeres og PageTextContent bedes genoprette mementoet der peges på.

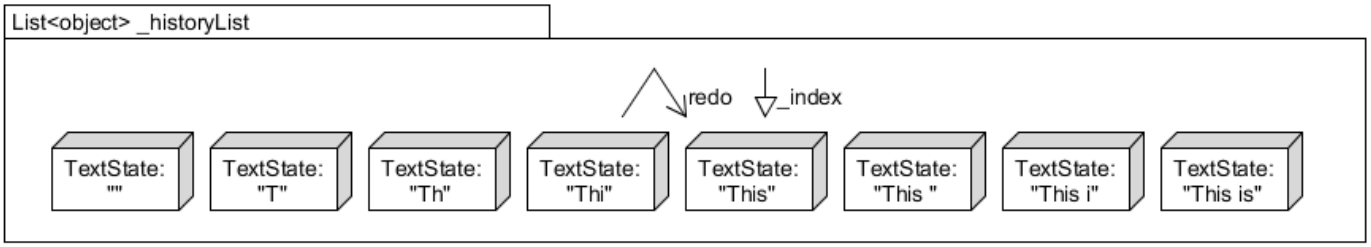


Figure 2.6: Redo i forlængelse af undos.

Efter en række undos og redos befinder man sig ofte et sted i midten af historikken. Hvis man her beslutter sig der skal gemmes en nyere version, slettes forældede "nyere" momentoer. Alt over index slettes, index inkrementeres, det nygemte memento indsættes. Denne opførsel ligner den, man ser i f.eks. Notepad++.

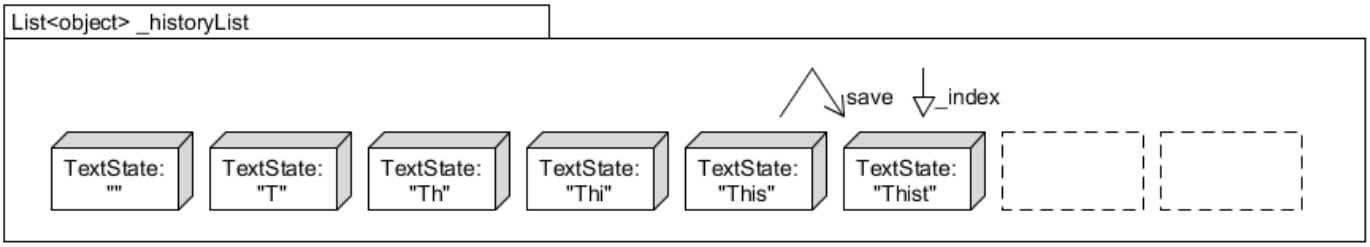


Figure 2.7: Nyt save i midten af historikken, forældede "nyere" mementoer slettes.

Med denne liste gemmes og tilgås mementoer på en simpel og effektiv måde. Denne løsning var en af flere løsninger, der blev afprøvet.

### 2.2.4 Sekvensdiagrammer

For at forstå designet, kan sekvensdiagrammerne for kørslen af `TextSaver`'s metoder ses forneden i figurene 2.8 og 2.9. Diagrammerne viser den overordnede funktionalitet for metoderne `HandleSaveMemento`, `HandleUndo` og `HandleRedo`. Det er disse metoder som selve programmet kommer til at kalde.

Metoden `HandleSaveMemento()` indeholder funktionaliteten for at gemme `PageTextContent`'s indre tilstand. Denne kalder `CreateState()`, hvilken returnerer et memento-objekt. Desuden tjekker metoden om det faktisk giver mening at gemme tilstanden - hvis det nyligt oprettede memento-objekt er lig det memento-objekt der ligger på den nuværende (`_index`) plads i `_historyList`, skal der ikke gemmes, derimod hvis der er sket en ændring skal den nye memento gemmes i listen og alle mementos foran `_index` skal slettes.

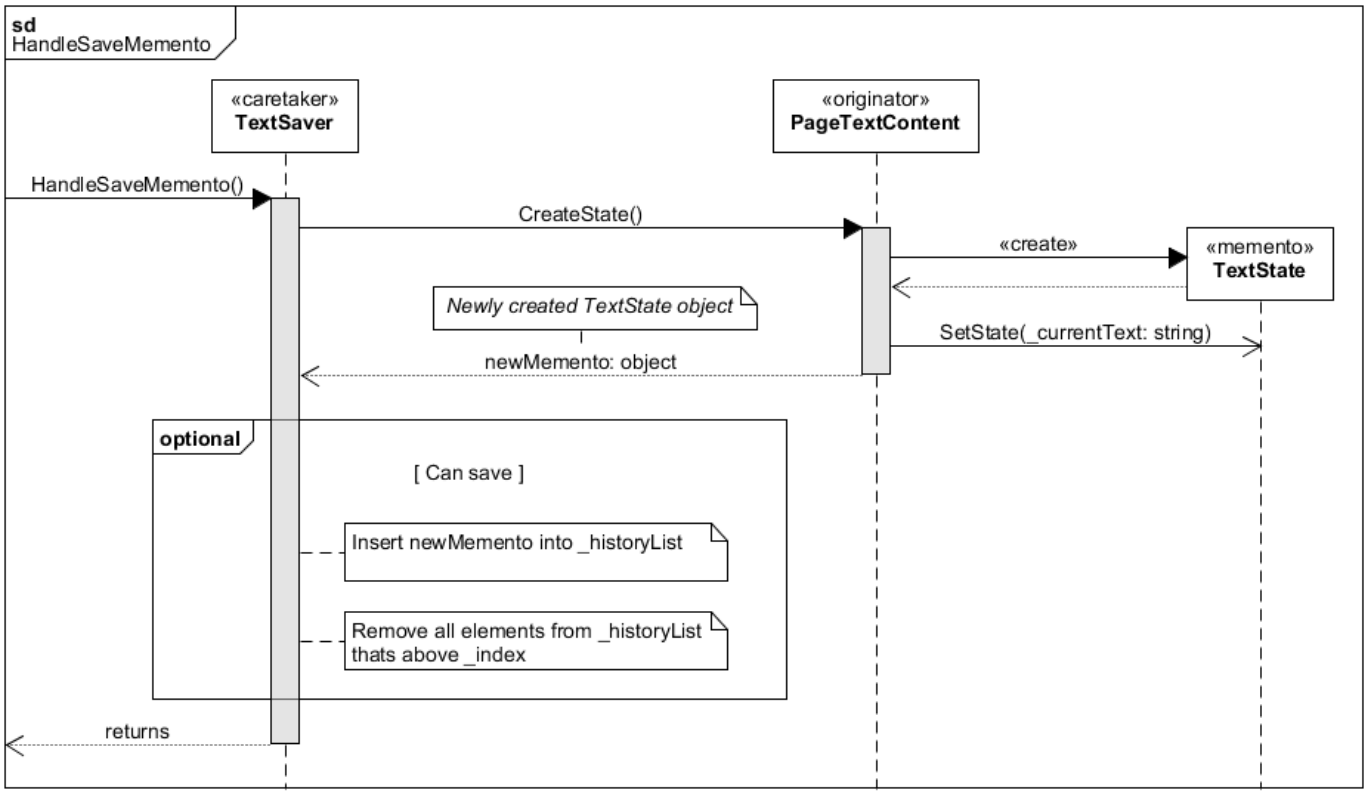


Figure 2.8: Sekvensdiagram for HandleSaveMemento()

Metoderne `HandleUndo` og `HandleRedo` er meget ens. Essensen af deres funktionalitet er at, vha. en memento, kunne "rollback" tilstanden for `PageTextContent` til en anden tilstand. Her bruges metoden `RestoreState( memento )`.

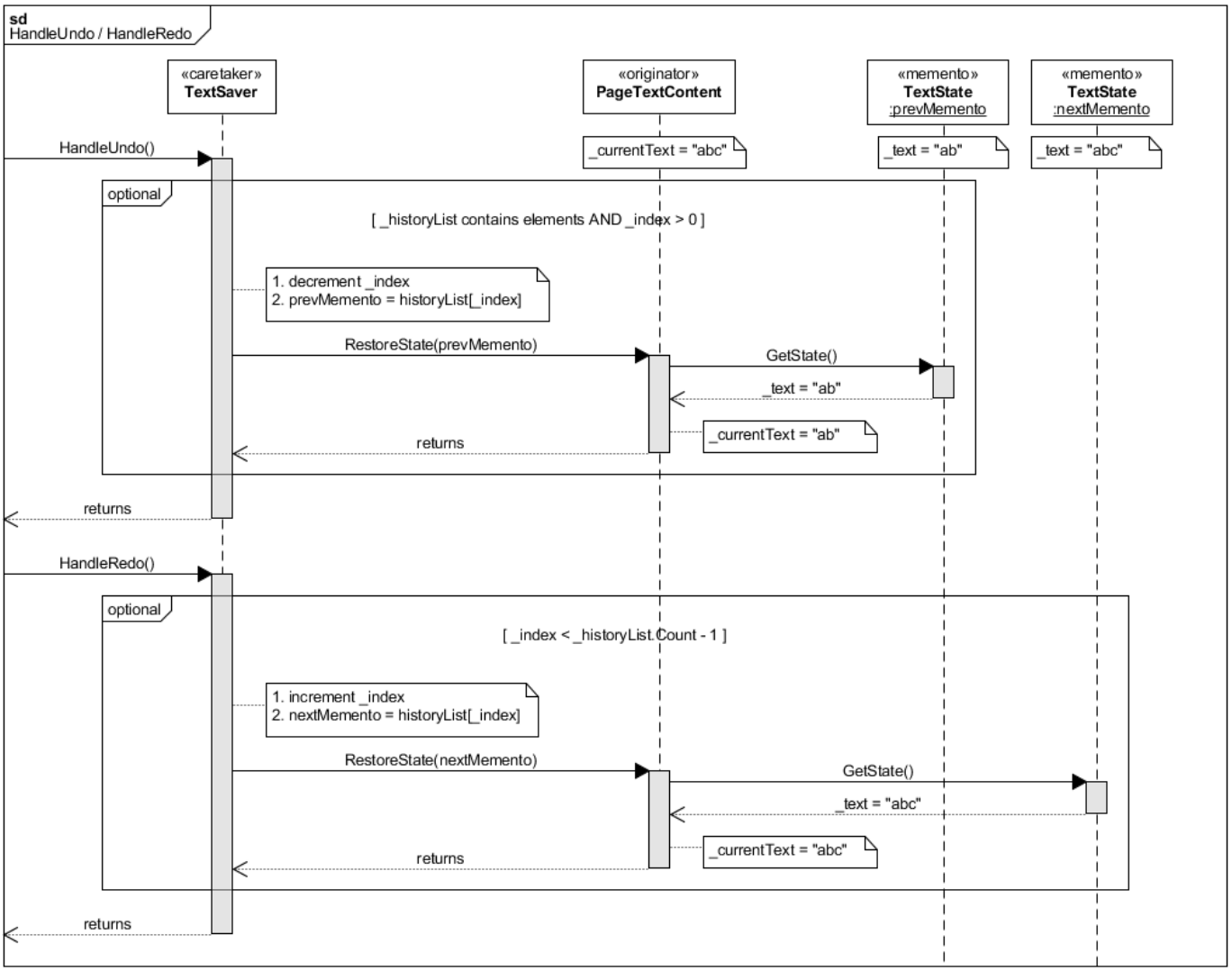


Figure 2.9: Sekvensdiagram for HandleUndo() og HandleRedo()

## 2.3 Implementering og resultat

I figur 2.10 ses klasse-diagrammet for selve texteditor-programmet. Dette program blev implementerede i WPF, da det gjorde det lettere at implementere den vindue-funktionalitet som der var brug for. Programmet er opbygget således; `MainWindow` repræsentere et vindue i WPF. Vha. xaml-kode er der blevet defineret tre knapper; en undo-knap (`UndoBut`), en redo-knap (`RedoBut`) og en save-knap `SaveBut`. Ved at trykke på disse knapper vil der blive kaldt forskellige eventhandlers i `MainWindow` (se metoderne på 2.10). Disse handlers kalder så metoderne defineret i caretaker-klassen, dog igennem dens interface `ITextSaver` (se sekvensdiagrammer i figurene 2.8 og 2.9).

Der er desuden blevet defineret en tekstbox (`EditorTxb`). Vha. eventhandleren `EditorTxb_OnTextChanged`, er denne tekstbox blevet "linket" til `PageTextContent`, da den kalder `SetText()` hver gang der sker en ændring i tekstboksens tekstfelt.

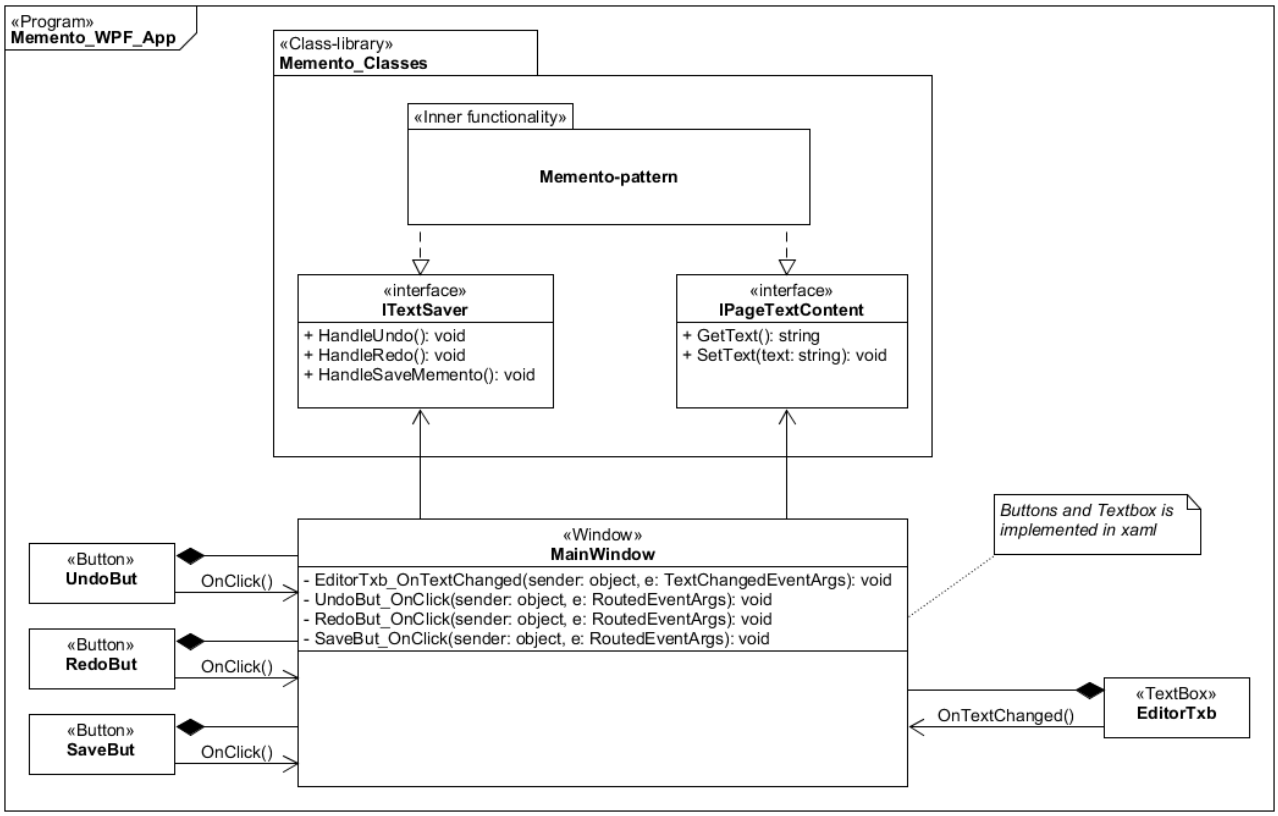


Figure 2.10: Caption

Umiddelbart er det lidt svært at vise selve program-ekserkveringen. Derfor referes der her til at man kan se præsentations-videoen for programmet.

# Chapter 3

# Konklusion

Memento design mønstret tillader at man kan gemme den interne tilstand af et objekt eksternt, uden at bryde objektets indkapsling. Ansvaret for at holde på en gemt tilstand er givet til memento klassen. Ansvaret for at oprette disse memento objekter og give dem den gemte interne tilstand, er givet til Originatoren. Ansvaret for at opbevare disse mementoer er givet til caretakeren. Denne fordeling tillader mønstret at overholde flere SOLID principper på en elegant og simpel måde. Mønstret er specielt nyttigt, når man ønsker at sin applikation skal kunne lave rollbacks til tidligere versioner af sig selv.

Når man benytter sig af dette mønster, skal man være opmærksom på, at memento objekter kræver mere hukommelse, og afhængigt af originatorens størrelse, kan et enkelt memento objekt komme til at kræve meget plads. Man skal derfor være sikker på man har hukommelse nok i sit system til at implementere mønstret. Hvis man ønsker at gemme originatorens tilstand, som en del af en anden funktion, skal man yderligere være opmærksom på, at denne funktions performance bliver forværret, da der netop kommer flere trin der skal fuldføres, før at funktionen returnerer.

Gruppen har succesfuldt gjort brug af Memento design mønstret, til at implementere en simpel tekst applikation, der kan gemme en tekststreng, og skifte mellem tidligere gemte tilstande for denne tekstreng. Dette er netop den funktionalitet der ønskes, når man vil lave rollback/undo.