Eindopdracht Design Patterns, Open Universiteit

Daniel S. C. Schiavini

April 14, 2019

Inhoudsopgave

Studentgegevens

Cursuscode IM0102

Opdracht Jabberpoint Inhoudsopgave

Naam Daniel S. C. Schiavini

Studentnummer 851102873

Aanpak

De eerste versie van deze opdracht is afgekeurd doordat de opdracht niet goed begrepen was. De refactoring was niet compleet doorgevoerd, maar alleen in de onderdelen die nodig waren om de inhoudsopgave toe te voegen. Deze sectie is daarom verdeeld tussen de eerste versie en de volledige refactoring.

Eerste versie

De oorspronkelijke bedoeling was om deze opdracht uit te voeren in een team met een andere student. Echter, er was geen andere student beschikbaar met een vergelijkbare tempo. Daardoor heb ik deze opdracht alleen uitgevoerd.

Voordat ik wist dat ik de opdracht alleen mocht uitvoeren, had ik al een Git repository gemaakt. Ik heb de repository zo geconfigureerd dat dit document automatisch wordt gebouwd door het gebruik van TravisCI. Zodra een nieuwe versie van het rapport naar GitHub wordt gepushed, draait een script dat het PDF-bestand maakt. Het script bouwt ook Jabberpoint, draait unit tests, en als alles gelukt is wordt een JAR-bestand gemaakt. In de master branch hangt een deployment vast dat het PDF en JAR naar GitHub pages publiceert. Deze automatisering is iets minder nodig zonder team, maar het is nog steeds wel nuttig. De CI-configuratie is beschikbaar in sectie 4.

Jabberpoint had nog geen geautomatiseerde tests en omdat ik dat belangrijk vind, heb ik van tevoren JUnit tests geschreven (zie PR#5, PR#9 en PR#10). Echter niet alle klassen waren testbaar – TravisCI draait in een Linux-container waarin de Graphic Java-klassen niet gecreëerd kunnen worden. Ik had niet genoeg tijd had om dit probleem op te lossen, daarom heb ik alleen tests geschreven voor de domein- en accessor-klassen.

Het schrijven van de tests was een goede hulpmiddel om Jabberpoint te leren kennen en om het probleemanalyse van de volgende sectie beter uit te voeren. De tests zijn hierbij erg waardevol geweest tijdens de refactoring. In vrijwel alle gevallen, als de tests sloegen werkte de applicatie via de gebruikersinterface ook.

Volledige refactoring

In de nieuwe versie van Jabberpoint heb ik de applicatie opnieuw gestructureerd. In de eerste fase van de opdracht zijn een aantal problemen geconstateerd:

- De applicatie had geen duidelijke scheiding tussen gebruikersinterface en business-logica. Om dit op te lossen is de MVC-patroon gebruikt.

- AWT-afhankelijkheden waren verspreid door de code. Hierdoor was het moeilijk om nieuwe gebruikersinterfaces te maken en om de applicatie te testen.
- Instanties werden in verschillende locaties gemaakt en vaak werd de klasse pas bruikbaar na andere methode-aanroepen. Het was bijvoorbeeld mogelijk om een `TextItem` te maken zonder tekst.
- De Java-versie was verouderd geworden.

Ten eerste is de applicatie verdeeld tussen model, view en controller (MVC patroon). Vervolgens heb ik alle onderdelen doorgenomen en steeds verbeteringen doorgevoerd, met de volgende regels in gedachte:

- Elk klasse moet precies één verantwoordelijkheid hebben.
- Elke methode moet precies één verantwoordelijkheid hebben.
- Delegatie liever dan overerving.
- Geen constructor-aanroepen buiten factories om.
- Alles wat een klasse nodig heeft om te functioneren wordt in de constructor meegegeven.

De details over het resultaat van de refactoring worden belicht in de volgende secties.

Uitleg diagrammen

De diagrammen in dit document worden gemaakt met de *yEd* applicatie. Hierin worden (tenzij anders vermeld) alleen publieke methods getoond en zijn de attributen private. De broncode van de diagrammen wordt in sectie 4 vermeld.

1 Probleemanalyse

1.1 Presentatie

Presentatie is het belangrijkste concept in Jabberpoint. Elke presentatie bestaat uit een titel en een lijst met slides. Eén van de slides wordt op een bepaald moment getoond.

In het originele ontwerp kent Jabberpoint een demo-presentatie. Deze presentatie is echter een gewone presentatie met vaste inhoud. Er zijn dus geen variaties op het concept van een presentatie.

1.1.1 Presentaties lezen en schrijven

Er zijn twee manieren om een presentatie te lezen:

- Vanuit code (demo-presentatie). In dit geval is de inhoud van de presentatie vast.
- Vanuit een bestand (XML-formaat). In dit geval heeft Jabberpoint een bestandsnaam nodig.

Er is één manier om presentaties te schrijven: naar een XML-bestand. In de toekomst kunnen meerdere output-formaten worden ondersteund.

Er is op dit moment één manier om presentaties te tonen en bedienen: met de awt-gebruikersinterface.

1.2 Slides

In de originele versie van Jabberpoint bestond maar één soort slide, maar nu krijgen we ook slides met inhoudsopgave.

Om de inhoudsopgave te analyseren kunnen we het beste de opdracht in eisen omschrijven:

- **Eis 1** : De gebruiker moet één of meer slides met inhoudsopgaven kunnen toevoegen aan een presentatie.
- **Eis 2** : Een inhoudsopgave-slide moet alle onderwerpen van de presentatie tonen.
- **Eis 3** : Het onderwerp moet slechts eenmaal worden getoond wanneer meerdere slides achter elkaar hetzelfde onderwerp hebben.
- **Eis 4** : De inhoudsopgave moet automatisch worden geüpdatet wanneer slides worden toegevoegd of verwijderd.
- **Eis 5** : De titel moet als onderwerp worden gebruikt wanneer een slide geen onderwerp heeft (**aanname**).
- **Eis 6** : Het onderwerp mag op slides worden getoond, behalve in de inhoudsopgave (optioneel).
- **Eis 7** : De onderwerpen in de inhoudsopgave mogen volgnummers krijgen (optioneel).

- **Eis 8** : Het onderwerp van de slide die na een inhoudsopgave komt, mag een speciale aanduiding krijgen (i.e. een alternatieve letterstijl) (optioneel).
- **Eis 9** : De stijl van de inhoudsopgave is vrij te bepalen.

De opdracht om een inhoudsopgave te implementeren in Jabberpoint is best interessant. Om een inhoudsopgave te tonen, is kennis nodig van de hele presentatie. Een verandering in elke slide kan invloed hebben op alle inhoudsopgaven. Deze circulaire afhankelijkheid maakt de opdracht uitdagend.

1.2.1 Commonality-variability analyse

Slides met inhoud en inhoudsopgave hebben het volgende in algemeen (*commonality*):

- Alle slides moeten een titel krijgen.
- Alle slides kunnen tekst-items hebben.

Slides met inhoud en inhoudsopgave hebben zijn in de volgende opzichten anders van elkaar (*variability*):

- Slide items worden anders gemaakt: Inhoud slides hebben een flexibele inhoud; inhoudsopgaven hebben gegenereerde inhoud.
- Inhoud slides kunnen veranderen tijdens uitvoering; alle inhoudsopgaven moeten dan automatisch worden geüpdatet (**Eis 4**).
- Inhoud slides mogen een onderwerp hebben; inhoudsopgaven niet (**Eis 6**).

1.2.2 Slide items

Er zijn twee soorten slide-items: plaatjes en tekst. Beide soorten hebben levels in algemeen, wat wordt gebruikt om de juiste stijl te kiezen. Hiernaast zijn de slide items anders van elkaar: Plaatjes hebben een filenaam en tekst-items hebben tekst.

1.3 Stijlen

Stijlen worden in Jabberpoint gebruikt om kleuren, indentatie en fonts te bepalen voor elke slide-item. In de nieuwe versie van Jabberpoint staan we vrij om aparte stijlen te implementeren afhankelijk van het soort slide. Zo kunnen we aparte stijlen definiëren voor inhoudsopgaven en voor inhoud-slides (**Eis 9**).

1.4 Gebruikersinteractie

In de originele en nieuwe versies van Jabberpoint wordt de interactie met de gebruikers niet veranderd. Het verbeteren ervan valt buiten de scope van de opdracht tot refactoring.

Jabberpoint opent automatisch de demo-presentatie of een gegeven presentatiebestand bij het opstarten. Daarna kan de gebruiker slides veranderen via het toetsenbord of via het menu.

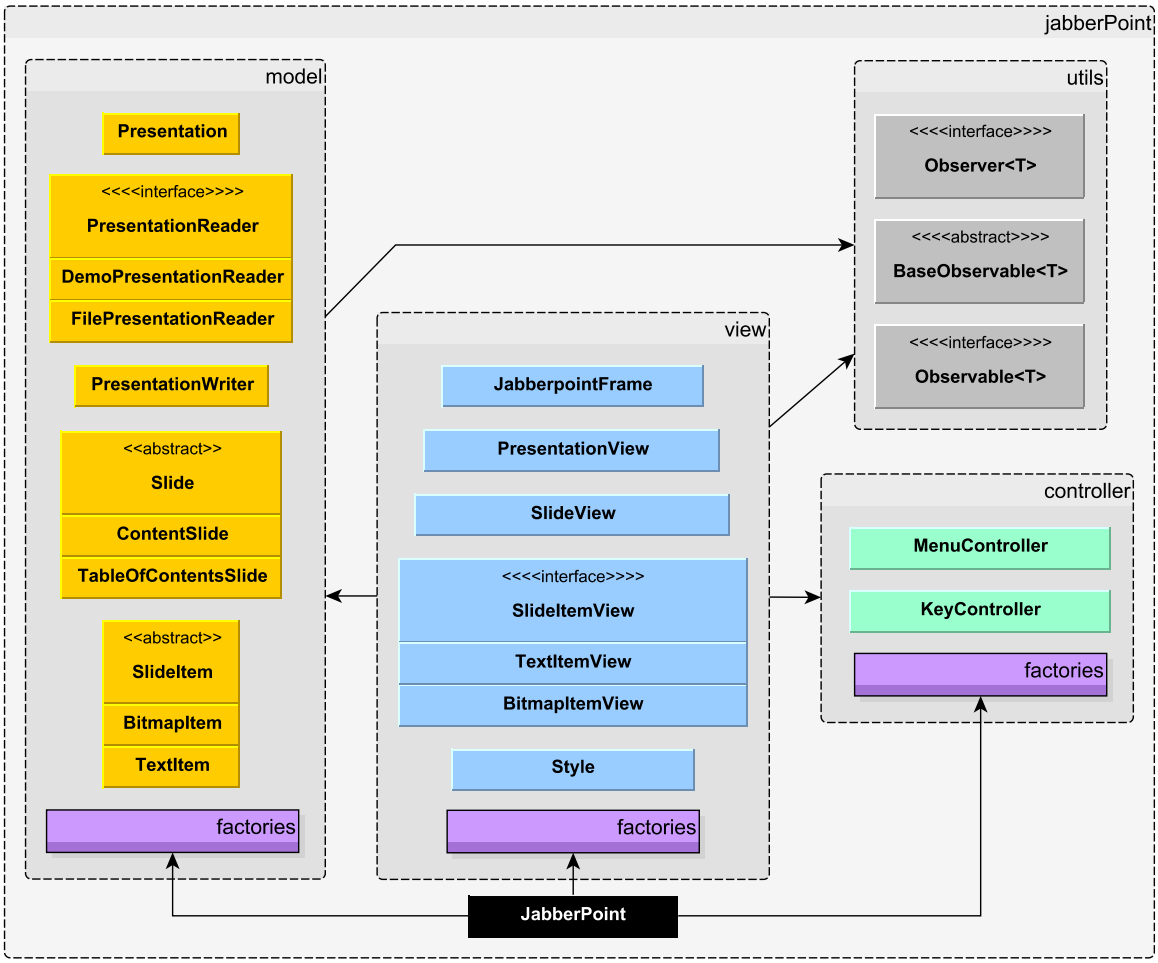
2 Ontwerp

De applicatie is nu gesplitst in meerdere Java-packages die volgen het MVC-patroon. In de onderstaande secties worden de verschillende packages belicht.

2.1 Jabberpoint

In de package `jabberPoint` vinden we de `JabberPoint` klasse, die verantwoordelijk is om alle afhankelijkheden aan elkaar te verbinden. Hierdoor kan de hele applicatie werken zonder dat instanties buiten factories worden gemaakt (*Depedency Injection* patroon). Deze klasse heeft een `main` methode die een presentatie opent door middel van de gemaakte factories. Het diagram met de packages wordt in figuur 2 getoond.

Figuur 1: Diagram met de verschillende packages in Jabberpoint. Details zijn weggelaten voor de duidelijkheid.



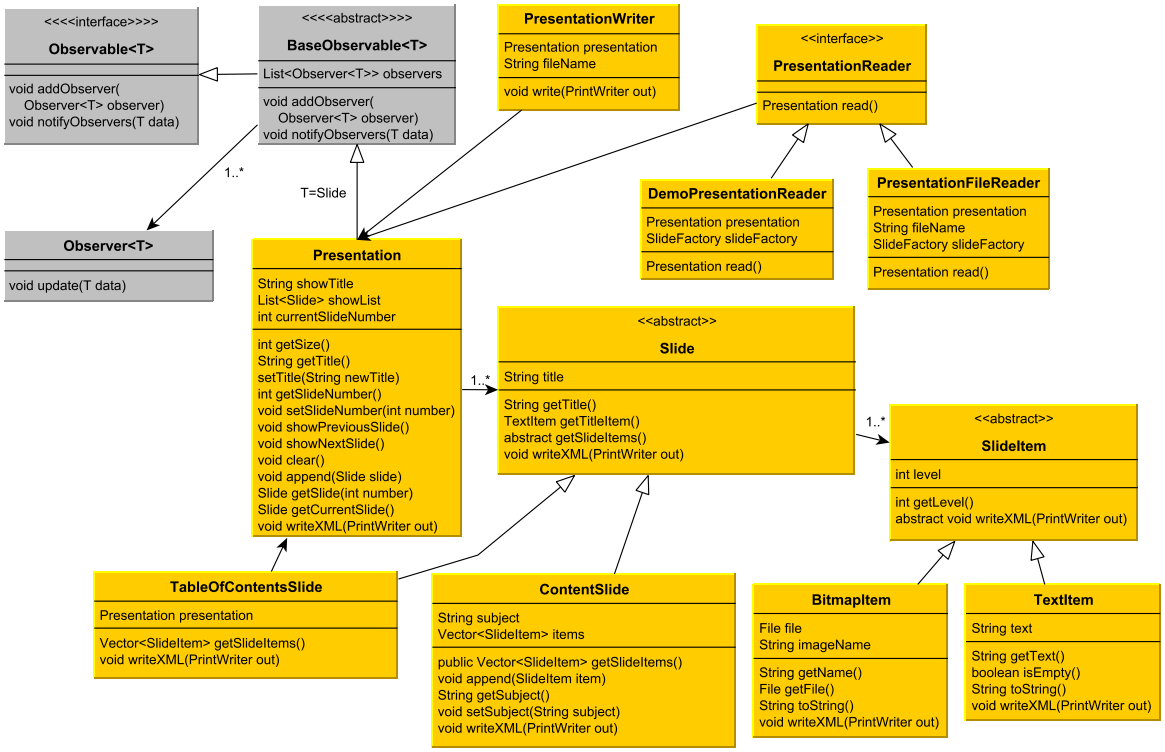
De applicatie wordt verdeeld tussen model (sectie 2.2), view (sectie 2.3) en

controller (sectie 2.4). Daarnaast hebben we een package `utils` voor klassen die niet JabberPoint-specifiek zijn (sectie 2.5).

2.2 Model

Het `jabberPoint.model` package bevat alle business-logica en is nu helemaal onafhankelijk van de gebruikersinterface. Het klassendiagram wordt in figuur 2 getoond.

Figuur 2: Klassendiagram voor de model-klassen. Factories zijn weggelaten. Referenties naar controller en view zijn beperkt gehouden voor de duidelijkheid. Het `utils` package wordt in grijs getoond.



De `Presentation`-klasse is duidelijk centraal in het systeem. Een presentatie bevat meerdere slides, en slides kunnen inhoud of inhoudsopgave bevatten. Slides kunnen meerdere slide-items bevatten, welke plaatjes of tekst mogen zijn. Samen met de stijlen is de presentatie-datastructuur compleet.

Deze data-klassen mogen niets weten van de gebruikersinterface waarin ze worden getoond. Wel zijn deze klassen verantwoordelijk om de gegevens op een generieke manieren te converteren. In dit geval was het vanzelfsprekend om een XML-formaat te gebruiken, maar dit had ook JSON kunnen zijn.

Naast de data-klassen hebben we de I/O klassen voor het lezen en schrijven van presentaties. Het lezen van presentaties kan op twee manieren gebeuren: vanuit een file of vanuit een demo-presentatie (`PresentationReader`-boom). De

presentatie kan ook naar een bestand worden geschreven (`PresentationWriter`-klasse). Deze klasse is veel simpeler geworden doordat deze geen kennis hoeft te hebben van de datastructuur.

Om te voorkomen dat de presentatie iets weet van de gebruikersinterface maar toch deze kan updaten, wordt het *observable*-patroon geïmplementeerd (zie sectie 2.5). De presentatie breidt `BaseObservable<Slide>` uit en kan *observers* laten weten als een nieuwe slide moet worden getoond.

Aanvullende documentatie over de klassen en hun verantwoordelijkheden zijn beschikbaar in de Javadoc bij de code (zie sectie 4).

2.2.1 Factories

Alle klassen mogen alleen door factories worden gemaakt. De factories geven alle nodige dependencies mee op het moment dat de klasse gemaakt wordt (*dependency injection* patroon). De model factories zijn gegroepeerd in het package `jabberPoint.model.factories`. Een klassendiagram van dit package wordt in figuur 3 getoond.

2.2.2 Inhoudsopgave

De oude klasse `Slide` is nu gesplitst in een abstracte klasse `Slide` en de subklasse `ContentSlide`. De nieuwe klasse `TableOfContentsSlide` genereert de inhoudsopgaven.

De grootste verschillen tussen de subklassen zijn dat de inhoudsopgave kennis moet hebben van de hele presentatie en zijn eigen inhoud moet genereren. Daardoor krijgt een `TableOfContentsSlide` een referentie naar de presentatie in de constructor.

De `Slide` klasse heeft nu een `getSlideItems` abstracte methode. Voor de normale slides zijn de slide items aanpasbaar, voor de inhoudsopgave worden de items gegenereerd.

Een screenshot van de inhoudsopgave van de demo-presentatie wordt in figuur 4 getoond.

2.2.3 Traceerbaarheid van eisen

In deze sectie geef ik onderdelen van het ontwerp aan die verantwoordelijk zijn om de eisen (zie sectie 1) te realiseren. Dit geeft dus de relatie tussen probleemanalyse en ontwerp.

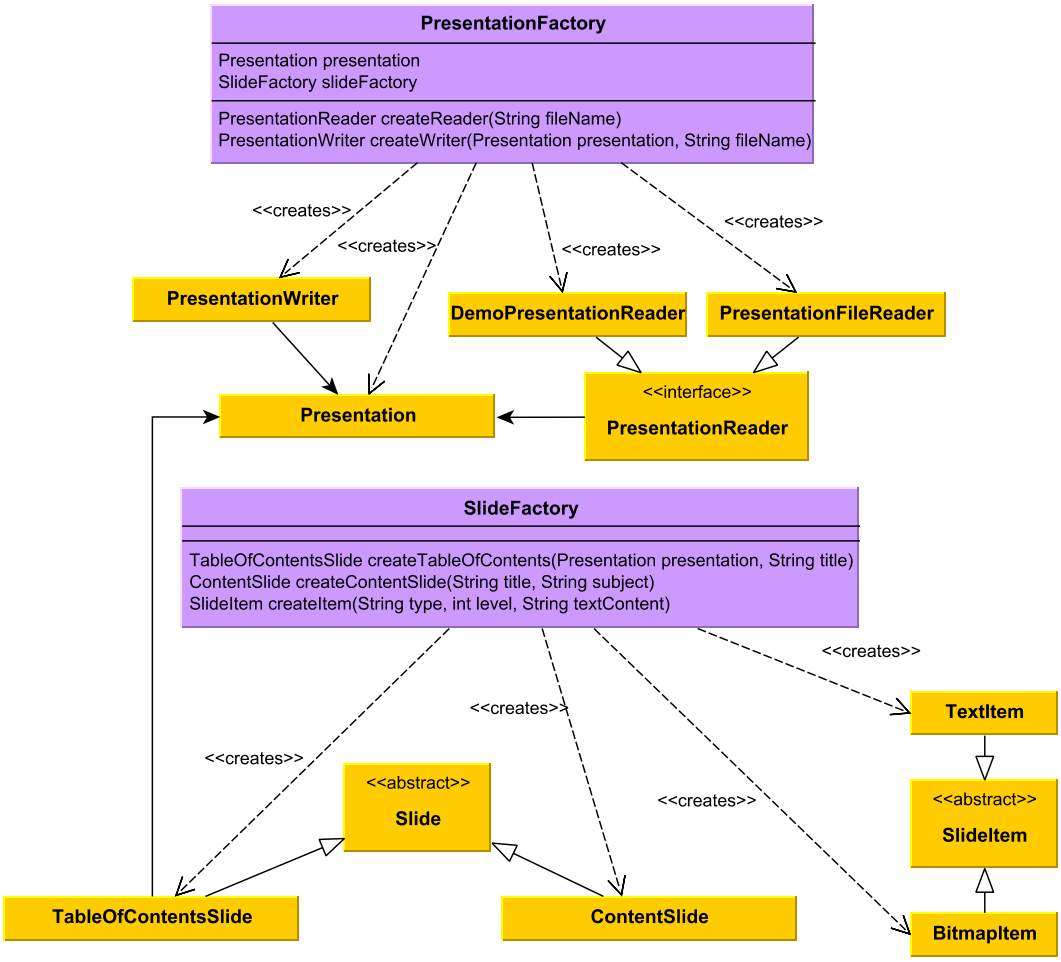
- De `Accessor`-klassen zijn verantwoordelijk voor **Eis 1**.
- De `TableOfContentsSlide` klasse is verantwoordelijk voor **Eis 2**, **Eis 3**, **Eis 4**, **Eis 5**, **Eis 7**, **Eis 8** en **Eis 9**.
- De `Slide` klasse is verantwoordelijk voor **Eis 6**.

2.3 View

Het `jabberPoint.view` package bevat alle nodige code voor de gebruikersinterface. Het klassendiagram van de view wordt in figuur 5 getoond.

Het element dat het volledige scherm opbouwt is de `JabberpointFrame`. Deze klasse bevat de twee controllers (zie sectie 2.4) en de `PresentationView`.

Figuur 3: Klassendiagram voor de model-factories. Details zijn weggelaten voor de duidelijkheid.



PresentationView krijgt de presentatie als referentie en is verantwoordelijk om de juiste slide te tonen. Om te weten wanneer een nieuwe slide moet worden getoond maak ik gebruik van het *observable-patroon* (zie sectie 2.5). Op het moment dat de update methode wordt aangeroepen, wordt een nieuwe SlideView gemaakt via de bijbehorende factory.

De SlideView klasse is verantwoordelijk om de slide te tekenen en maakt gebruikt van SlideItemView instanties. Deze kunnen of een BitmapItemView of een TextItemView zijn, maar de keuze voor de juiste view gebeurt in de factory (zie sectie 2.3.1)

Aanvullende documentatie over de klassen en hun verantwoordelijkheden zijn beschikbaar in de Javadoc bij de code (zie sectie 4).

Figuur 4: Inhoudsopgave van de demo-presentatie.

2.3.1 Factories

Alle klassen mogen alleen door factories worden gemaakt. De factories geven alle nodige dependencies mee op het moment dat de klasse gemaakt wordt (*dependency injection* patroon). De view factories worden gegroepeerd in het package `jabberPoint.view.factories`. Een klassendiagram van dit package wordt in figuur 6 getoond.

De `PresentationViewFactory` is verantwoordelijk om `JabberPointFrame` te maken met de juiste `PresentationView`. `SlideViewFactory` maakt de implementaties van `Slide`. `SlideItemViewFactory` maakt de implementaties van `SlideItemView`. Ten slotte maakt `StyleFactory` de juiste `Style` afhankelijk van de slide en slide-item.

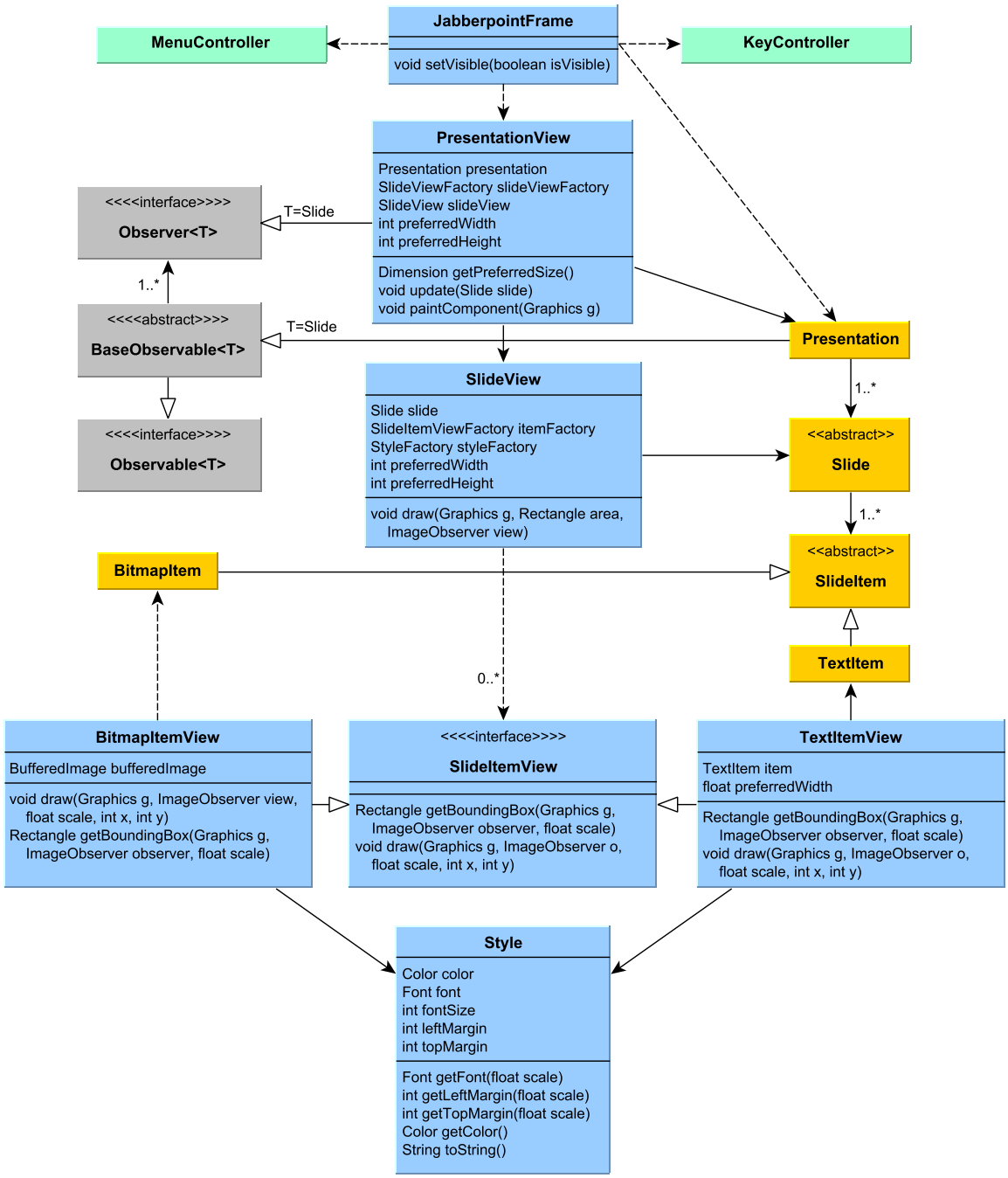
2.4 Controller

Het `jabberPoint.controller` package bevat alle nodige code voor het veranderen van de model via de gebruikersinterface. Het klassendiagram wordt in figuur 7 getoond.

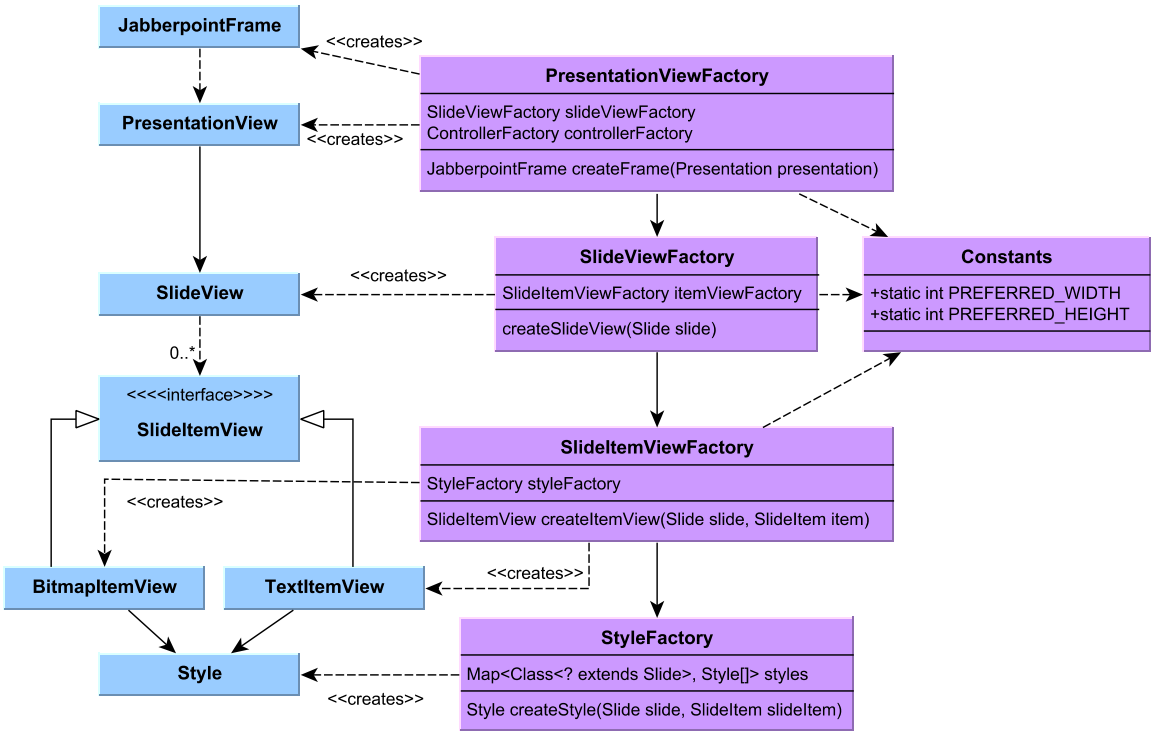
Jabberpoint kent twee manieren om de presentatie te navigeren: via het menu of via het toetsenbord. Daarom hebben we ook twee controllers: `MenuController` en `KeyController`. Beide krijgen een presentatie in de constructors zodat de controllers commando's kunnen doorgeven. `MenuController` krijgt bovendien een instance van `PresentationFactory` omdat deze presentaties moet kunnen openen. Hierin kunnen fouten optreden, en `MenuController` is niet zelf verantwoordelijk om een foutmelding te tonen. Deze worden getoond door de view via de *callback*-functie `BiConsumer onIOException`.

Aanvullende documentatie over de klassen en hun verantwoordelijkheden zijn beschikbaar in de Javadoc bij de code (zie sectie 4).

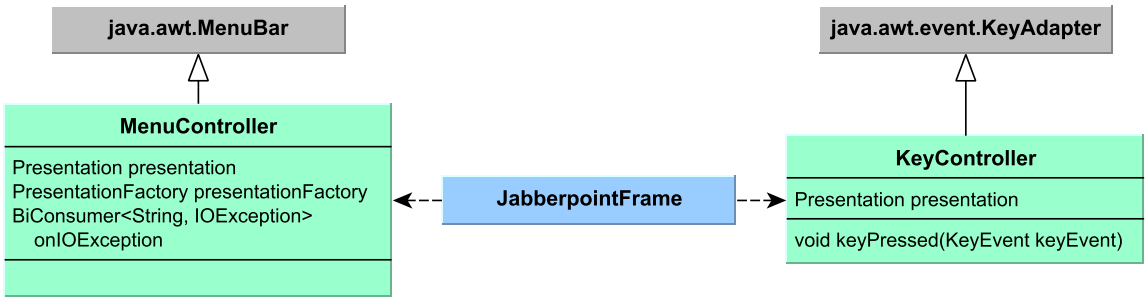
Figuur 5: Klassendiagram voor de view-klassen. Factories zijn weggelaten. Referenties naar controller en model zijn beperkt gehouden voor de duidelijkheid.



Figuur 6: Klassendiagram voor de view-factories. Details zijn weggelaten voor de duidelijkheid.



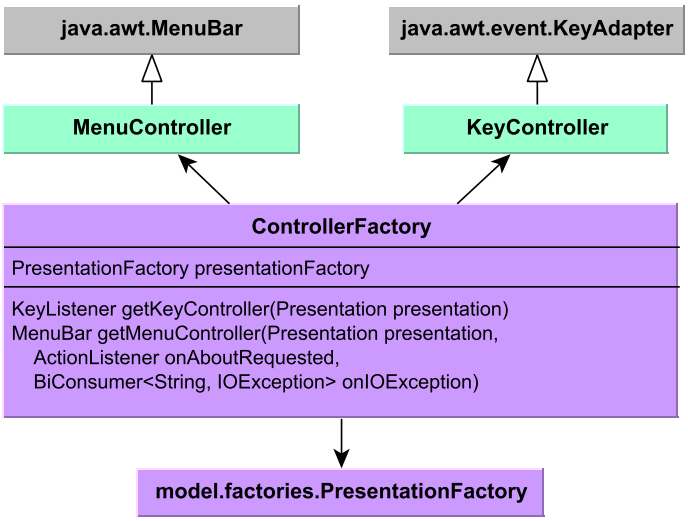
Figuur 7: Klassendiagram voor de controller-klassen. Factories zijn weggelaten. Referenties naar view en model zijn beperkt gehouden voor de duidelijkheid.



2.4.1 Factories

Alle klassen mogen alleen door factories worden gemaakt. De factories geven alle nodige dependencies mee op het moment dat de klasse gemaakt wordt (*dependency injection* patroon). De controller factories worden gegroepeerd in het package `jabberPoint.controller.factories`. Een klassendiagram van dit package wordt in figuur 8 getoond.

Figuur 8: Klassendiagram voor de controller-factories. Details zijn weggelaten voor de duidelijkheid.



2.5 Utils

De klassen het `utils`-package zijn in een apart package geplaatst omdat ze niet specifiek voor Jabberpoint zijn. Het gaat namelijk om de implementatie van het *Observable-patroon*. De Java-klassen hiervoor hebben geen typing, waardoor casts nodig zijn. Door de toepassing van generics kan dit verbeterd worden.

Het patroon is geïmplementeerd door middel van twee interfaces en één abstracte klasse:

- `Observer<ObservedType>` is de interface voor klassen die andere klassen kunnen observeren. Deze interface bevat de method `update` om updates te krijgen.
- `Observable<ObservedType>` is de interface voor klassen die geobserveerd kunnen worden. Deze interface bevat de methoden `addObserver` en `notifyObservers`.
- `BaseObservable<ObservedType>` is een abstracte implementatie van `Observable`. Deze implementatie zorgt voor een standaard implementatie van de vereiste methoden.

Op deze manier kan de presentatie `Observable<Slide>` implementeren door `BaseObservable<Slide>` uit te breiden. View-*observers* implementeren `Observer<Slide>` om te weten als een nieuwe slide moet worden getoond. Het package wordt afgebeeld in het model-diagram (figuur 2, pagina 7).

3 Keuzen

In deze sectie gaan we een aantal vragen door die zijn opgeroepen tijdens het analyseproces.

3.1 Welke klasse is verantwoordelijk voor onderwerpen?

De inhoudsopgave moet toegang krijgen tot een lijst van onderwerpen in de presentatie. In het ontwerp krijgt `TableOfContentsSlide` een referentie naar de presentatie en is zelf verantwoordelijk om deze te lezen. En alternatief hierop is om de presentatie verantwoordelijk te houden voor de onderwerpen in de slides. `TableOfContentsSlide` zou dan een dan een onderwerpen-lijst ontvangen.

De voordelen van deze alternatief zijn:

- De inhoudsopgave hoeft niets te weten van de presentaties.
- De presentatie is verantwoordelijk om zijn eigen slides te lezen.
- De circulaire referentie wordt opgeheven.
- De lijst van onderwerpen kan tussen de verschillende inhoudsopgaven worden gedeeld.

De nadelen zijn:

- Er komt meer logica in de presentatie die alleen nodig is voor de inhoudsopgave.
- De inhoudsopgaven moeten nog steeds alle items doorlopen om te bepalen welk onderwerp het volgende is.
- De methoden die nodig zijn om de slides te lezen bestaan al in de presentatie.
- Het is conceptueel correct om een circulaire referentie te hebben, immers een inhoudsopgave moet een overzicht van de presentatie laten zien.
- De presentatie klasse wordt nog complexer.

De keuze om de lijst van onderwerpen in de inhoudsopgave te implementeren geeft dus een betere inkapseling.

3.2 Moet een inhoudsopgave een andere stijl krijgen?

De standaard stijl van de slides is niet optimaal voor een inhoudsopgave, alhoewel **Eis 9** expliciet aangeeft dat stijlen vrij te bepalen zijn. Het is met deze stijl echter niet helemaal duidelijk dat het om een lijst van onderwerpen gaat. Door een icoontje toe te voegen aan elke item en steeds dezelfde indentatie te gebruiken, wordt het duidelijker. Daarom heb ik aparte stijlen geïmplementeerd via de `StyleFactory`. Een screenshot van de inhoudsopgave met deze stijl wordt in figuur 4 getoond (pagina 10).

3.3 Moet de versiegeschiedenis in de bestanden zijn?

In de initiële versie van Jabberpoint stond bovenaan in elk bestand een geschiedenis van de versies. Met een moderne ontwikkelmethode verandert de code heel snel, de lijst wordt heel lang en niet altijd up-to-date gehouden. Bovendien gebruiken we nu een tool die specifiek gemaakt is om code-geschiedenis bij de houden, namelijk Git. Daarom heb ik besloten om deze versiegeschiedenis te verwijderen van alle files en deze alleen in Git te houden.

De auteur-melding is wel gehouden omdat deze kort is en niet zo vaak verandert. Git zou dit prima kunnen bijhouden, echter is de historie kwijt toen ik de code importeerde in een nieuwe repository. Het is dan netjes om de oorspronkelijke auteurs te benoemen.

3.4 Moet de code worden voorzien van Javadoc?

Om software echt onderhoudbaar te maken is het ontwikkelen met ontwerppatronen niet voldoende. Ik ben namelijk van mening dat de documentatie bij de code van essentieel belang is. Daarnaast is het schrijven van de bedoeling van een stukje code een belangrijk hulpmiddel om de codekwaliteit te evalueren. Daarom heb ik de hele applicatie voorzien van Javadoc op het niveau van klassen en methoden (inclusief parameters). De meeste attributen zijn overigens ook gedocumenteerd.

3.5 Is het updaten van Java verstandig?

De Jabberpoint applicatie maakte gebruik van Java 1.6. Tijdens de refactoring heb ik besloten om de Java-versie te updaten naar 1.8. Met deze versie van Java is het gebruik van lambda-functies mogelijk. Daardoor was het mogelijk om de controllers veel eenvoudiger te maken.

3.6 Moet lezen en schrijven in een apart package komen?

Tijdens het splitsen van Jabberpoint in MVC-packages rees de vraag: waar horen de I/O klassen bij? In beschrijvingen van MVC wordt de model vaak beschreven als "*business logic-related data*". Daarom vind ik dat de `Reader` en `Writer` klassen in ook in dit package horen, zeker gezien hoe eenvoudig de applicatie is.

Er zijn nog twee opties mogelijk die ik net zo passend vind:

- Een apart package onder JabberPoint, bijvoorbeeld `jabberPoint.io`;
- Een sub-package van model, bijvoorbeeld `jabberPoint.model.io`.

3.7 Moeten overal interfaces worden gedefinieerd?

Een aantal concrete klassen in het ontwerp worden gebruikt door veel andere klassen. Bijvoorbeeld de `Presentation` klasse heeft veel referenties. Voor alle concrete klassen zouden we een interface kunnen definiëren met één implementatie. Als resultaat zouden de concrete klassen alleen in de factories voorkomen.

Echter heb ik ervoor gekozen om geen interfaces te definiëren. Het is namelijk simpel in Java om een klasse naar een interface te converteren. Elke klasse kan

altijd een interface worden zonder de gebruikers van de klasse te veranderen. Zo houden we het ontwerp kort en bondig zonder verlies van flexibiliteit.

3.8 Waarom aparte view-klassen voor slide items?

In het ontwerp zien we dat de SlideView werkt met de abstracte Slide klasse. Daarom is geen ContentSlideView of TableOfContentsSlideView nodig. De slides moeten namelijk op dezelfde manier worden getoond, alleen de inhoud is anders.

Echter voor de slide-items was een dergelijke constructie niet mogelijk. Er is dus een interface SlideItemView met implementaties TextItemView en BitmapItemView. De slide-items zijn zodanig anders van elkaar dat het niet mogelijk is om op een generieke manier te tekenen.

3.9 Waarom geen abstracte factory?

In het cursus-tekstboek worden allerlei soorten factories besproken. "Factory" wordt echter niet benoemdt in de lisjt met patronen (zie chapter 20, bijvoorbeeld page 348). De meeste factories in het ontwerp passen daarom niet in de genoemde factory patronen.

Natuurlijk hebben de factories *factory methods*. Presentation is als *singleton* geïmplementeerd, maar dit wordt geabstraheerd door de factory. Style instanties worden ook één keer gemaakt, maar in verschillende versies: dit lijkt op *object pool*, echter ook dit wordt geabstraheerd. Ik vind namelijk belangrijk dat de rest van het systeem niet hoeft te weten hoe de instanties gemanaged moeten worden.

Een andere optie zou zijn om factories abstract te maken of een interface te definiëren zodat er meerdere varianten van gemaakt kunnen worden. Hier geldt echter hetzelfde argument als in sectie 3.7: het zou ingewikkelder worden dan noodzakelijk.

In het geval van StyleFactory is er wel wat te zeggen voor twee implementaties: één voor inhoud-slides en één voor inhoudsopgave-slides. Maar dan moet ergens de beslissing worden genomen over welke factory gebruikt wordt. Hiervoor zie ik de volgende opties:

- Een interface voor StyleFactory en een klasse om de factories te maken (feitelijk een StyleFactoryFactory). Er bestaan dan 4 klassen/interfaces in plaats van één.
- De gebruiker van de klasse kiest welke factory gebruikt wordt. De StyleFactory is dan niet meer verantwoordelijk voor deze beslissing.
- Een abstracte StyleFactory kent de verschillende implementaties en kan de juiste kiezen (bijvoorbeeld in een static methode). Hierdoor ontstaat een circulaire referentie.

Deze klasse is nog redelijk simpel is en biedt voldoende flexibiliteit. Daarom heb ik gekozen voor de oplossing getoond in figuur 5 (pagina 11).

3.10 Is het gebruik van constanten een goed idee?

Het `jabberPoint.view.factories` package bevat de klasse `Constants`. Hierin bevinden zich de standaard breedte en hoogte van het scherm. Deze klasse is gemaakt om code-duplicatie te voorkomen tussen de view-klassen. De constante-waarden zijn opgenomen als `public final static int` attributen en worden dus niet opgehaald via een methode.

Een alternatieve optie zou zijn om de constanten *singleton* te maken en de attributen via methoden beschikbaar te maken. Zo zouden in de toekomst verschillende implementaties kunnen worden gemaakt. Echter worden deze constanten alleen als *preferred* waarden opgenomen in de views. De views schalen vervolgens naar de grootte van het venster. Het is dan ook niet realistisch te verwachten dat deze verschillende versies moeten krijgen. Bovendien worden deze waarden alleen in de factories gebruikt (zie figuur 6 op pagina 12).

4 Sourcecode

In deze sectie geef ik links naar de verschillende resultaten van de opdracht. Alle links betreffen zowel Jabberpoint als dit rapport. De broncode is vanzelfsprekend in combinatie met het ontwerp-documentatie in dit document.

- Jabberpoint code: github.com/DanielSchiavini/design-patterns-assignment-/tree/master/Jabberpoint
  De werking ervan wordt uitgelegd in sectie 2.
- LaTeX code: github.com/DanielSchiavini/design-patterns-assignment/tree-/master/Report
- CI-configuratie: github.com/DanielSchiavini/design-patterns-assignment-/blob/master/.travis.yml
- Diagrammen: github.com/DanielSchiavini/design-patterns-assignment/tree-/master/Report/Diagrams
- Aanpassingen op Jabberpoint: github.com/DanielSchiavini/design-patterns-assignment/compare/v0.0.1...master.
- Refactoring: github.com/DanielSchiavini/design-patterns-assignment/compare/v1.0.0...v2.0.0
- Gegenereerde bestanden: github.com/DanielSchiavini/design-patterns-assignment/tree/gh-pages
- Laatste build-rapport van TravisCI: travis-ci.com/DanielSchiavini/design-patterns-assignment/
- Rapporten van TravisCI: travis-ci.com/DanielSchiavini/design-patterns-assignment/builds