



---

# OGO VERSLAG

---

Yanniek Wielage | | 1597082



## Inleiding

Dit verslag heeft als doel uit te leggen welke aanpassingen er zijn gemaakt aan Jabberpoint en waarom er voor deze aanpassingen is gekozen. Dit wordt bereikt door eerst een geconstateerd probleem te formuleren, waarna de oplossing wordt beschreven. Dit verslag is geschreven tijdens de werkzaamheden, daarom is gekozen om niet alleen de oplossing te beschrijven maar ook te tonen. Elk subhoofdstuk bevat een probleem en een bijbehorende oplossing.

## Refactoring Jabberpoint

Bij het refactoren van de applicatie heb ik gebruik gemaakt de design patterns te vinden op [refactoring.guru](http://refactoring.guru) en de SOLID-principes. Beide zullen worden benoemd in de tekst waar toepasselijk.

### Probleem 1. Overbodige klasse

De AboutBox klasse is de kleinste klasse van de Jabberpoint applicatie. De klasse bevat een enkele methode, genaamd 'show()'. Zie deze klasse in figuur 1.

```
public class AboutBox {  
    1 usage  
    public static void show(Frame parent) {  
        JOptionPane.showMessageDialog(parent,  
            message: "JabberPoint is a primitive slide-show program in Java(tm). It\n" +  
                "is freely copyable as long as you keep this notice and\n" +  
                "the splash screen intact.\n" +  
                "Copyright (c) 1995-1997 by Ian F. Darwin, ian@darwinsys.com.\n" +  
                "Adapted by Gert Florijn (version 1.1) and " +  
                "Sylvia Stuurman (version 1.2 and higher) for the Open" +  
                "University of the Netherlands, 2002 -- now.\n" +  
                "Author's version available from http://www.darwinsys.com/",  
            title: "About JabberPoint",  
            JOptionPane.INFORMATION_MESSAGE  
        );  
    }  
}
```

Figuur 1 - AboutBox

AboutBox wordt eenmalig aangeroepen binnen de tekst:

```
AboutBox.show(parent);
```

AboutBox is een aparte klasse en heeft als enige verantwoordelijkheid om een popup menu te tonen. De verantwoordelijkheid voor het verwerken van menu acties hoort eigenlijk bij 'MenuController'.

Zodat de verantwoordelijkheid neer wordt gelegd bij de juiste klasse, is er gekozen om de methode te verplaatsen naar 'MenuController', te zien in figuur 2. Deze nieuwe methode wordt in plaats van AboutBox.show() kunnen worden aangeroepen wanneer nodig. In de aanpassing, is de klasse 'AboutBox' verwijderd. Op deze manier wordt de scheiding van verantwoordelijkheden in de applicatie duidelijker, zoals verwacht wordt in de 'Single Responsibility Principle' die deel uitmaakt van de SOLID-principes.

```

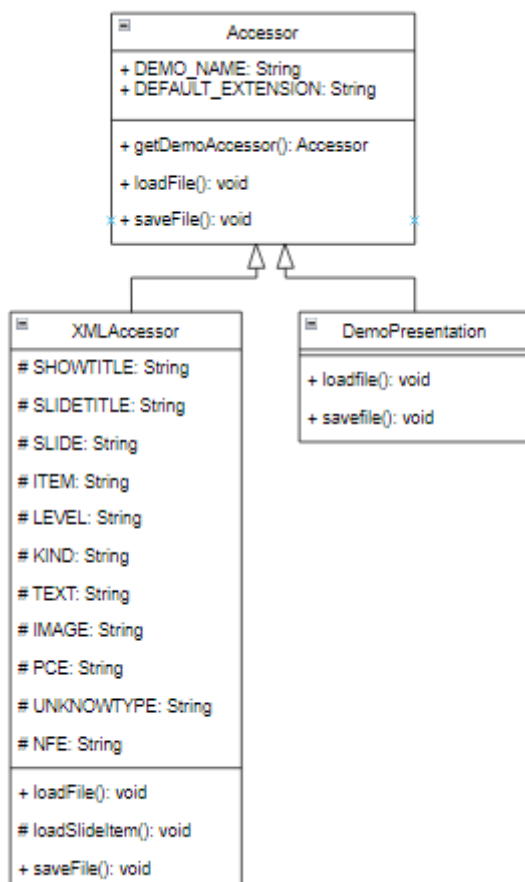
1 usage  ywielage
public void showAboutBox(Frame parent) {
    JOptionPane.showMessageDialog(parent,
        message: "JabberPoint is a primitive slide-show program in Java(tm). It\n" +
        "is freely copyable as long as you keep this notice and\n" +
        "the splash screen intact.\n" +
        "Copyright (c) 1995-1997 by Ian F. Darwin, ian@darwinsys.com.\n" +
        "Adapted by Gert Florijn (version 1.1) and " +
        "Sylvia Stuurman (version 1.2 and higher) for the Open" +
        "University of the Netherlands, 2002 -- now.\n" +
        "Author's version available from http://www.darwinsys.com/",
        title: "About JabberPoint",
        JOptionPane.INFORMATION_MESSAGE
    );
}

```

Figuur 2 – showAboutBox()

## Probleem 2. Flexibiliteit

In het klassendiagram is een drietal klassen te zien, die een relatie hebben. In dit geval is het een inheritance relatie, omdat beide 'XMLAccessor' en 'DemoPresentation' erven van 'Accessor'. Zie deze klassen in figuur 3.



Figuur 3 - Drietal klassen

'Accessor' is een abstracte klasse met als methoden:

- loadFile()
- saveFile()

Deze methoden worden met logica gevuld in beide 'XMLAccessor' en 'DemoPresentation'. loadFile() in 'DemoPresentation' is grotendeels hardcoded. Een gebruiker zou buiten het programma dit niet aan kunnen passen. Wanneer het programma moet worden uitgebreid, of wanneer de demo presentatie moet worden aangepast, moet de code worden aangepast. Dit volgt niet de 'Open/Closed' principe. Volgens deze principe moet bij uitbreiding geen code hoeven worden aangepast.

Om dit probleem op te lossen, is er een oplossing verzonnen. Bij deze oplossing wordt bepaalde logica geplaatst in 'XMLAccessor'. De volgende methode is gecreëerd:

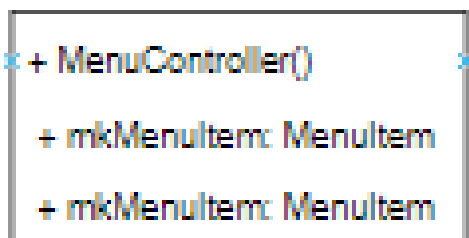
```
public void createNewPresentation(Presentation presentation, String
unusedFilename, String title, ArrayList<Slide> slides) {
    presentation.setTitle(title);
    for (Slide slide : slides) {
        presentation.append(slide);
    }
}
```

Deze methode voert dezelfde logica uit wanneer de parameters zijn gevuld. Het idee was om deze logica naar de main() methode in Jabberpoint te verplaatsen. In de main() methode wordt dan een lijst aangemaakt met data die in de presentatie moet komen, inclusief de titel van de dia's en de presentatie.

De klasse maakte al gebruik van de 'Strategy' design pattern. createNewPresentation() breidt deze klasse uit en behoudt de design pattern. Dankzij deze oplossing kan er een nieuwe presentatie worden gecreëerd die bijvoorbeeld als demo presentatie kan dienen. De presentatie kan daarna alsnog als .xml bestand worden opgeslagen, zoals van tevoren al mogelijk was.

### Probleem 3. Te veel verantwoordelijkheden

De klasse 'MenuController' bestaat uit een grote constructor waar te veel gebeurt. Wanneer alle logica in een constructor is geplaatst, volgt de code al snel niet het 'Open/Closed principe'. Deze principe gaat over de uitbreiding van de code. Aangezien alle code in de constructor is geplaatst, moet van alles worden aangepast wanneer er iets wordt toegevoegd. Daarom wordt deze constructor als een architecturaal probleem gezien. In figuur 4 wordt de huidige opbouw van de klasse getoond.



```
classDiagram
    class MenuController {
        + MenuController()
        + mkMenuItem: MenuItem
        + mkMenuItem: MenuItem
    }
```

Figuur 4 - Methoden MenuController()

Om te zorgen dat deze klasse de 'Open/Closed' principe gaat volgen, is een groot aantal van de code verplaatst naar methoden binnen dezelfde klasse. Elke methode heeft een eigen doel, waarvan bepaalde specifieker zijn dan anderen. Een voorbeeld van een nieuwe methode is de createHelpMenuItems() methode, te zien in figuur 5. Deze methode maakt een lijst van items aan die

gebruikt worden in het HELP-menu. Dit menu bevat momenteel maar één item, maar dit kan bij uitbreiding van de applicatie veranderen. Vandaar de list in plaats van een enkel MenuItem object.

```
1 usage  ywielage *
public List<MenuItem> createHelpMenuItems() {
    List<MenuItem> items = new ArrayList<>();
    items.add(createMenuItem(ABOUT, () -> menuActionController.showAboutBox(parent)));
    return items;
}
```

Figuur 5 – createHelpMenuItems()

De klasse 'MenuController' wordt omgezet naar drie klassen. Deze klassen proberen de Command design pattern en de Observer design pattern toe te voegen aan de applicatie.

Na de refactoring is de constructor vele malen kleiner en de code is overzichtelijker. Daarnaast voldoet de klasse nu aan de 'Open/Closed' principe.

De MenuController had drie verantwoordelijkheden, namelijk:

- Het aanmaken van het menu
- Het aanroepen van logica wanneer een menu item wordt gebruikt
- Het uitvoeren van de logica wanneer deze menu items worden gebruikt

Er is daarom gekozen om een twee nieuwe klassen aan te maken, de eerste heet 'MenuActionController'. Deze klasse heeft als verantwoordelijkheid om de acties uit te voeren.

De tweede klasse heet MenuInitializer en bevat de code gebruikt voor het aanmaken van het menu.

De klasse 'MenuObserver' behoudt als enige verantwoordelijkheid het beheren van de listeners. Elke

Elke klasse heeft nu een enkele verantwoordelijkheid.

#### Probleem 4: Inefficiënte commands

Wanneer een menu item wordt gebruikt, wordt een bepaald actie uitgevoerd. Deze acties zijn de logica die moet worden uitgevoerd wanneer een corresponderend menu item wordt aangeroepen. In het geval van deze locatie bestond dit uit een ActionListener opgezet in de constructor, zoals:

```
fileMenu.add(menuItem = mkMenuItem(NEW) );
menuItem.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent actionEvent) {
        presentation.clear();
        parent.repaint();
    }
});
```

Dit zorgt voor dubbele code. Dit is opgelost door gebruik te maken van de command design pattern. Zie het voorbeeld in figuur 5 voor een implementatie van deze design pattern. Op het moment dat een gebruiker klikt op de 'ABOUT'-knop, wordt de showAboutBox() methode uitgevoerd. Deze methode bevat de logica om het verwachte resultaat te bereiken.