SOLID principes

S. Single responsibility

Er zijn een aantal klassen waar deze principe wordt toegepast. Bijvoorbeeld in de MenuController zie de volgende afbeelding voor de code snippet.

```
public static MenuController getInstance()
{
    if (MenuController.menuController == null)
    {
        MenuController.menuController = new MenuController(SlideViewerFrame.getInstance());
    }

    return MenuController.menuController;
}
```

Gebruik maken van instances van andere klassen binnen in dit geval een controller is belangrijk omdat het helpt om het single responsibility principe te respecteren. Elke klasse vervult één specifieke taak niets meer, niets minder.

O. Open/closed

Er zijn een aantal klassen waar de relatie wordt verbeterd tussen in dit geval de SlideViewerComponent en Presentation. Op dit moment kunnen er nieuwe observers worden toegevoegd zonder dat de Presentation of SlideViewerComponent moeten worden gewijzigd. Het open/closed principe stelt dat klassen open moeten zijn voor uitbreiding, maar gesloten voor aanpassing.

Zoals te zien in onderstaande afbeelding, is in de presentation klasse, het mogelijk om de listeners toe te voegen, up te daten en te verwijderen.

```java
// Voeg een PresentationListener toe aan de lijst van luisteraars
public void addPresentationListener(PresentationListener presentationListener)
{
    this.presentationListeners.add(presentationListener);
}

// Verwijder een PresentationListener uit de lijst van luisteraars
public void removePresentationListener(PresentationListener presentationListener)
{
    this.presentationListeners.remove(presentationListener);
}

// Werk alle presentatie luisteraars bij met de huidige presentatie en slide
public void updatePresentationListeners()
{
    for (PresentationListener presentationListener : this.presentationListeners)
    {
        presentationListener.update(this, this.getCurrentSlide());
    }
}
```

Waarna er in de onderstaande afbeelding van de SlideViewerComponent klasse, de update plaatsvindt naar alle listeners.

```java
@Override
public void update(Presentation presentation, Slide slide)
{
    if(slide == null)
    {
        this.slide = null;
        this.slideNumber = presentation.getSlideNumber();
        this.presentationSize = presentation.getSize();
    }

    this.slide = slide;
    this.slideNumber = presentation.getSlideNumber();
    this.presentationSize = presentation.getSize();

    this.repaint();
    this.frame.setTitle(presentation.getTitle());
}
```

D. Dependency inversion

Voor de DemoPresentation klasse wordt het Facade pattern toegepast. Door een facade te introduceren kunnen verschillende delen van de applicatie eenvoudig toegang krijgen tot de presentatiefunctionaliteit zonder zich bezig te hoeven houden

met de interne werking van de gehele applicatie waardoor het aansluit aan het dependency inversion principe. Hieronder een code snippet.

```java
public class DemoPresentation extends Accessor
{
    @Override
    public void loadFile(Presentation presentation, String unusedFilename) {
        presentation.setTitle("Demo Presentation");
        Slide slide;
        slide = new Slide();
        slide.setTitle("JabberPoint");
        slide.append(1, "Het Java Presentatie Tool");
        slide.append(2, "Copyright (c) 1996-2000: Ian Darwin");
        slide.append(2, "Copyright (c) 2000-now:");
        slide.append(2, "Gert Florijn en Sylvia Stuurman");
        slide.append(4, "JabberPoint aanroepen zonder bestandsnaam");
        slide.append(4, "laat deze presentatie zien");
        slide.append(1, "Navigeren:");
        slide.append(3, "Volgende slide: PgDn of Enter");
        slide.append(3, "Vorige slide: PgUp of up-arrow");
        slide.append(3, "Stoppen: q or Q");
        presentation.append(slide);

        slide = new Slide();
        slide.setTitle("Demonstratie van levels en stijlen");
        slide.append(1, "Level 1");
        slide.append(2, "Level 2");
        slide.append(1, "Nogmaals level 1");
        slide.append(1, "Level 1 heeft stijl nummer 1");
        slide.append(2, "Level 2 heeft stijl nummer 2");
        slide.append(3, "Zo ziet level 3 er uit");
        slide.append(4, "En dit is level 4");
        presentation.append(slide);

        slide = new Slide();
        slide.setTitle("De derde slide");
        slide.append(1, "Om een nieuwe presentatie te openen,");
        slide.append(2, "gebruik File->Open uit het menu.");
        slide.append(1, " ");
        slide.append(1, "Dit is het einde van de presentatie.");
        presentation.append(slide);

        slide = new Slide();
        slide.setTitle("Bedankt voor het kijken naar deze presenatie");
        slide.append(1, "Vandaag gaan wij lekker jabberen,");
        slide.append(2, "gebruik File->Open uit het menu.");
        slide.append(1, " ");
        slide.append(1, "Jammer jammer dit is het einde");
        presentation.append(slide);
    }
```

Zoals hierboven te zien is, is er een demo presentatie klaargezet. Voor het geval dat er geen presentatie wordt ingeladen, wordt de demo presentatie laten zien. Dit past bij het dependency inversion principe, waarbij hogere modules niet afhankelijk zijn van concrete implementaties, maar van abstracties. Dankzij deze aanpak hoeft de rest van de applicatie niet te weten hoe de presentatie precies is opgebouwd.