
# Herontwerp / advies

## Project Jabberpoint

23-02-2025

Versie 1.0

—

**Software Quality**
Bram & Kimmy
van Schaikweg 94
7811 KL Emmen

# Verbeterpunten

In het volgend hoofdstuk worden er een aantal verbeterpunten voorgesteld. Waar het wordt opgedeeld in wat er verandert moet worden en wat er nog mist in het systeem.

**Commentaar**

Op veel plekken in de code is er geen commentaar te vinden waar dit wel handig zou zijn. Op de plekken waar wel commentaar is geschreven is dit vaak niet beschrijvend of heeft dit geen toegevoegde waarde. De complexe en lange code stukken worden vaak beschreven met een regel aan uitleg. Dit kan voor eventuele andere softwareontwikkelaars lastig zijn om de code te onderhouden of aan te passen. Zie TextItem klasse voor een voorbeeld, van het weinige gebruik van comments terwijl er wel een hele grote functie is geschreven.

**Kwaliteit van de code**

- In het systeem zijn er verschijnen methodes, velden, parameters, variabelen en constanten waarbij de naam inconsistent en niet goed genoeg beschrijvend is. Dit kan ervoor zorgen dat er verwarring ontstaat en het lezen van de code lastig kan maken. Het is daarom verstandig om tijd te besteden aan het gebruik van passende en consistente namen. Om een voorbeeld uit het systeem te gebruiken, in MenuController wordt er gebruik gemaakt van een *mkMenuItem* methode. Daarbij staat de mk voor make, wat niet gelijk duidelijk is uit de naam. Als er dan wordt gekeken naar de createStyles methode, staat make en create eigenlijk voor dezelfde actie, dit is dus niet consistent.

- Wanneer de methodes uit de hoofdklasse wordt geïmplementeerd ontbreken er hier en daar de @Override headers, hierdoor is het niet overzichtelijk of dit een eigen gemaakte methode is of niet.

- Op veel plekken in de code, vooral in de grotere methodes wordt er slecht en inconsistent gebruik gemaakt van indentaties, hierdoor is de code lastig te lezen en wordt het begrijpen van de code ook op de proef gesteld.

**Missende attributen**

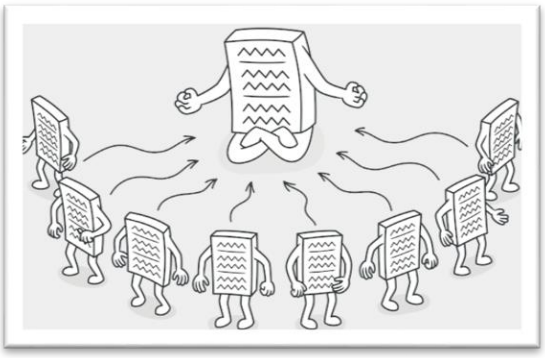
- Er wordt gebruik gemaakt van packages alleen zijn de namen onduidelijk. Waardoor de structuur wat een package kan bieden, niet optimaal wordt benut.

- Op veel plekken in de velden en methodes wordt er geen gebruik gemaakt van een access modifier, waardoor de veiligheid en concreetheid missen.
- Er mist veel data validatie, in verschillende methodes wordt er niet gekeken of het wel mogelijk is om bijvoorbeeld een SlideItem aan te maken. Dit kan simpel worden opgelost door een simpele validatie toe te voegen.
- In het systeem is er geen package voor de testen, echter zijn er geen testen aanwezig. Het maken en gebruiken van testen is belangrijk, zodat je nieuwe methodes kan testen of het werkt zoals je dit hebt gemaakt.
- Er wordt geen rekening gehouden met conventies, waardoor de code inconsistent is, ook wordt er amper gebruik gemaakt van 'this’ keyword.

# Design patterns

**Creational pattern: Singleton**

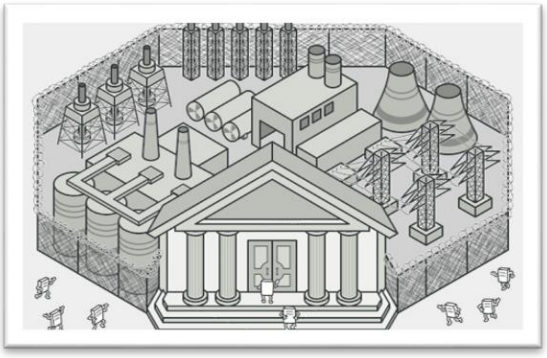
**(S - Single Responsibility Principle)**



Zowel de **MenuController** als de **keycontroller** klasse worden elk slechts één keer gebruikt binnen het programma, waardoor meerdere instanties overbodig zijn. Doormiddel van het toepassen van een **singleton** design pattern, zorgen we ervoor dat er maar één instantie wordt aangemaakt en beheerd. Dit voorkomt zowel conflicten als overbodige duplicate code en maakt het beheer hiervan eenvoudiger. Dit volgt het **Single Responsibility Principle** omdat de Singleton ervoor zorgt dat er één verantwoordelijke instantie is die het beheer van het menu afhandelt.

**Structural pattern: Façade**

**(D - Dependency Inversion Principle)**



Voor de **DemoPresentation** klasse passen we het **Facade** Pattern toe om een eenvoudige interface te bieden die de onderliggende complexiteit verbergt. Dit maakt de code eenvoudiger te begrijpen, hergebruiken en te onderhouden. 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.

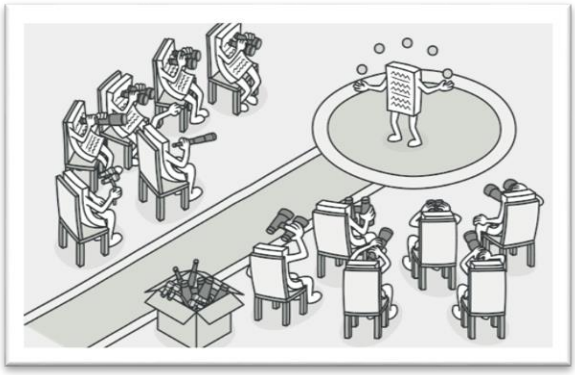
Dit sluit aan bij het **Dependency Inversion Principle (D)**:

- Hogere niveaus van de applicatie (zoals de UI) hoeven niet direct afhankelijk te zijn van de complexe interne implementatie van de **DemoPresentation**.

- In plaats daarvan interacteren deze met een abstracte en stabiele interface (de **facade**), waardoor de afhankelijkheden omgekeerd worden en de applicatie flexibeler blijft.

**Behavioral pattern: Observer**

**(O - Open/Closed Principle)**

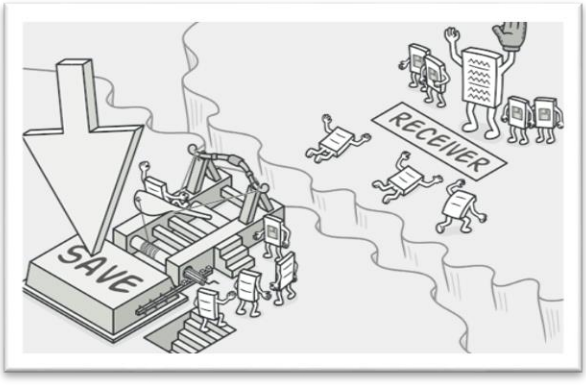


Met het **Observer** Pattern verbeteren we de relatie tussen de **SlideViewerComponent** en **Presentation**. Hierdoor kunnen beide componenten los van elkaar evolueren zonder dat er veel wijzigingen in bestaande code nodig zijn. Dit maakt het eenvoudig om nieuwe manieren toe te voegen waarop de **Presentation** en **SlideViewerComponent** met elkaar samenwerken.

Dit sluit aan bij het **Open/Closed Principle (O)**, omdat we de code uitbreidbaar maken zonder bestaande functionaliteit aan te passen. Nieuwe observers kunnen eenvoudig worden toegevoegd zonder dat de **Presentation** of **SlideViewerComponent** zelf moet worden gewijzigd.

**Behavioral pattern: Command**

**(O & S - Open/Closed & Single Responsibility Principle)**




Het **Command** Pattern zorgt voor een gestructureerde manier om gebruikersinput te verwerken. Dit doen wij doormiddel van het opsplitsen van deze acties naar afzonderlijke **command objects**. Hierdoor wordt het eenvoudiger om nieuwe invoermethoden toe te voegen zonder dat bestaande code hoeft te worden aangepast. Dit zorgt ervoor dat de bijbehorende **KeyController** klasse flexibel blijft en makkelijk uitgebreid kan worden.

Daarnaast heeft de **MenuController** klasse momenteel een complexe constructor met een onoverzichtelijke hoeveelheid herhaling en toetsende logica. Door het **Command** Pattern toe te passen en acties te scheiden in afzonderlijke commando's, wordt de code zowel modulairder als onderhoudbaarder. Hierdoor wordt niet alleen de leesbaarheid

verbeterd, maar wordt het ook eenvoudiger om functionaliteiten van de **KeyController** toe te voegen aan het menu.

Het gebruik van het **command** pattern past bij zowel het **Open/Closed Principle (O)** als het **Single Responsibility Principle (S)**:

- **O**: Nieuwe invoermethoden kunnen worden toegevoegd zonder bestaande code te wijzigen.
- **S**: Elk command object heeft één specifieke verantwoordelijkheid, wat zorgt voor een duidelijke scheiding van taken.