MASTER'S THESIS

Design patterns approached from the functional paradigm within the programming language Java

Chafik, H. (Hicham)

Award date:
2020

Design patterns approached from the functional paradigm within the programming language Java

Hicham Chafik
Student:

Open Universiteit
Nederland
Tuesday 14 April, 2020

DESIGN PATTERNS APPROACHED FROM THE FUNCTIONAL PARADIGM WITHIN THE PROGRAMMING LANGUAGE JAVA

by

Hicham Chafik

Open University, faculty of Management, Science and Technology
Master's Programme in Software Engineering
to be defended publicly on tuesday 14 April, 2020 at 14:00 PM.

Student number:

Course code:	IM0101181922	
Thesis committee:	Dr. ir. Sylvia Stuurman (1st supervisor, chairman),	Open Universiteit
	Dr. ir. Harrie Passier (2nd supervisor),	Open Universiteit

INHOUDSOPGAVE

SUMMARY

Object oriented programming languages contain elements which can be considered as a shortcoming. These shortcomings can be supplemented by applying design patterns. Design patterns are known solutions for a variety of problems in the Object Oriented design Paradigm.

Design patterns can offer solutions to problems that a programming language cannot solve on its own. The goal of applying design patterns is to offer a possibility to increase the flexibility, the extendibility and the maintainability of a software-system in a situation where a programming language does not offer such possibility. Each design pattern has a specific goal, or is a solution to a specific problem. Another side effect of design patterns is the ability to improve quality attributes on a system level, like flexibility, understandability and reusability [13]. In the technical report “Eleminating OO Paterns by Java functional features“ certain OO design patterns have been rewritten by first class functions. In order to do so, the following Design patterns are applied:

- Strategy pattern.
- Template method pattern.
- Command pattern.
- Decorator pattern.
- The visitor pattern.

In this research this technical report is used as a base , while the goal of this research , is to focus merely on if the functional implementation variant of the GOF patterns is providing an improvement over to the use of the classical GOF patterns.

This research is mainly focused on a qualitative perspective. A verification is done to check if it is advisable to refactor an existing design pattern by the use of first class functions. In this scope, the current GOF patterns are being compared with the reviewed design patterns which are implemented in the technical document “Eliminating OO patterns by Java Functional Features.“

SAMENVATTING

Object georiënteerde programmeertalen bevatten elementen die als tekortkoming kunnen worden gezien. Deze tekortkomingen kunnen worden aangevuld door het inzetten van design patterns. Design patterns zijn bewezen oplossingen voor verschillende knelpunten binnen het OO-paradigma. Design patterns bieden daardoor oplossingen die een programmeertaal zelfstandig niet kan oplossen. Het doel van design patterns is om de flexibiliteit, uitbreidbaarheid en onderhoudbaarheid van een softwaresysteem te optimaliseren in de situatie waarin de programmeertaal zelf onvoldoende mogelijkheden biedt. Elke pattern heeft een bepaald doel, dan wel een probleem dat het kan oplossen. Daarnaast maken design patterns het mogelijk om de kwaliteitsattributen op systeemniveau zoals flexibiliteit, begrijpbaarheid en herbruikbaarheid te verbeteren [13].

In het technisch rapport “Eliminating OO Patterns by Java Functional Features“ zijn een aantal OO design pattern herschreven door middel van first class functions. Waarbij de volgende design pattern zijn aangehaald:

- Strategy pattern.
- Template method pattern.
- Command pattern.
- Decorator pattern.
- The visitor pattern.

In dit onderzoek vormt het technisch rapport de basis, waarbij het doel van dit onderzoek, is het op hoofdlijnen bepalen of de functionele implementatievariant van de GoF patterns daadwerkelijk een verbetering zijn ten opzicht van de klassieke GoF patterns. Het onderzoek is voornamelijk kwalitatief van aard. Hierbij is getoetst wanneer het raadzaam is om bestaande desgin patterns te refactoren door gebruik te maken van first class functions. In dit kader zijn de bestaande GoF patterns vergeleken met de herziene design patterns die in het technisch rapport “Eliminating OO Patterns by Java Functional Features” zijn uitgewerkt.

1

INTRODUCTIE

Binnen de software engineering kunnen bij de ontwikkeling van een softwaresysteem knelpunten ontstaan. Tekortkomingen binnen de programmeertaal en het paradigma waarop de programmeertaal is gebaseerd, kunnen een bijdragen leveren aan de geïntroduceerde knelpunten. Voor de verschillende knelpunten binnen het object georiënteerde (hierna: “OO“) paradigma zijn oplossingen bedacht, die zich inmiddels hebben bewezen als design oplossingen. Deze oplossingen worden ook wel “design patterns“ genoemd.

Het doel van design patterns is om de flexibiliteit, uitbreidbaarheid en onderhoudbaarheid van een softwaresysteem te optimaliseren in de situatie waarin de programmeertaal zelf onvoldoende mogelijkheden biedt. Een programmeertaal als Scala heeft bijvoorbeeld het Singelton pattern als taalelement. In dat geval biedt de programmeertaal zelf een oplossing voor het probleem.

Door de jaren heen zijn veel OO-talen uitgebreid met aspecten van functionele talen. Zo biedt de programmeertaal Java sinds versie 8 de mogelijkheid om first class functions te implementeren binnen het object georiënteerd paradigma [17]. Door de functionele aspecten die in de programmeertaal zijn geïntroduceerd, is het mogelijk om andere type implementaties van de patterns te implementeren. Deze laatstgenoemde implementatievarianten verschillen dan ook van de klassieke design patterns.

Door de functionele aspecten die in Java zijn geïntroduceerd, kunnen een aantal design patterns worden herschreven aan de hand van first class functions. Hierdoor is dus in plaats van de klassieke OO-implementatie een functionele implementatie ontstaan.

Het onderzoek zal kwalitatief van aard zijn. Hierbij zal worden getoetst wanneer het raadzaam is om bestaande desgin patterns te refactoren door gebruik te maken van first class functions. In dit kader zullen een aantal GoF patterns worden vergeleken met de herziene design patterns die herschreven zijn door middel van functionele apsecten van Java

Voor de gehele opdracht geldt dat de conclusies zo generiek mogelijk wordt uitgewerkt. Hierbij zal worden aangegeven onder welke condities de voorkeur uitgaat naar GoF patterns dan wel naar vervangende patterns waarbij gebruik is gemaakt van first class functions. Voor de praktische onderdelen wordt echter alleen rekening gehouden met de programmeertaal Java.

Hoofdstuk 2 geeft een beschrijving op welke fronten eerder onderzoek is verricht naar implementatievarianten van de bestaande GoF design patterns. Verder zal worden aangehaald hoe design patterns de kwaliteit van een design kunnen optimaliseren.

Hoofdstuk 3 geeft een beschrijving wat de aanleiding is geweest van dit onderzoek. Op basis daarvan wordt het probleem aangehaald aan de hand van een concreet voorbeeld. Naar aanleiding van het probleem zullen de hoofd- en deelvragen worden geformuleerd.

De onderzoeksresultaten zijn onderverdeeld in vier hoofdstukken waarin elke deelvraag een hoofdstuk vormt. Allereerst zal inzichtelijk worden gemaakt welke kwaliteitsaspecten de betreffende design pattern probeert te optimaliseren. Vervolgens zal in kaart worden gebracht, op basis van welke kwaliteitscriteria de implementatievarianten met elkaar kunnen worden vergeleken.

Hoofdstuk 5 geeft een beschrijving welke implementatievarianten mogelijk zijn op basis van de bestaande GoF Patterns. In Hoofdstuk 6 zal worden gekeken hoe de verschillende implementatie zich tot elkaar verhouden.

Het onderzoek zal vervolgens worden afgesloten met een conclusie voor de eerder genoemde design patterns.

2

GERELATEERD WERK

In dit onderzoek zal het technisch rapport “Eliminating OO Patterns by Java Functional Features” [1] als uitgangspunt worden genomen. In voornoemd rapport is een eerste poging gedaan om in kaart te brengen welke design patterns op een alternatieve wijze kunnen worden geïmplementeerd.

In het rapport zijn de voor- en nadelen van de klassieke en functionele design patterns bepaald door de design patterns te herschrijven aan de hand van functionele features. Daarnaast wordt een set aan vuistregels gegeven die bepalen hoe een patroon kan worden vereenvoudigd door gebruik te maken van functionele features.

In het rapport zijn de volgende vijf patterns beschreven:

- Strategy.
- Template method.
- Command.
- Decorator.
- Visitor.

Voor elke pattern zijn er drie implementatie varianten uitgewerkt, namelijk:

- **OO-implementatie:** deze representeert de standaard pattern die in het GoF boek staat beschreven.
- **enumeration implementatie:** deze representeert een vereenvoudigde variant van de OO-implementatie waarbij de logica is vervangen door constanten in de vorm van een enumeration.
- **functionele implementatie:** deze representeert een vereenvoudigde variant van de OO-implementatie waarbij de logica is vervangen door static methodes / lambda expressie.

Het onderzoek dat wordt omschreven in het technisch rapport blijkt niet het eerste onderzoek te zijn naar implementatievarianten op het gebied van design patterns. Zo hebben Hanneman en Kiczal [8] en Khalid Aljasser [2] reeds onderzoek gedaan naar andere implementatievarianten van de klassieke design patterns.

In het onderzoek van Hanneman en Kiczal is gebruik gemaakt van een aspect georiënteerde versie van de programmeertaal Java, namelijk AspectJ. Het onderzoek is tot stand gekomen doordat OO-abstractie het vaak niet mogelijk maakt om crosscuting conserns zoals logging te modulariseren. Dit probleem doet zich namelijk ook voor bij implementaties waarbij gebruik is gemaakt van design patterns. Verder wordt in het onderzoek geclaimd dat een programmeertaal invloed heeft op de implementatie van een pattern. Uit voornoemd onderzoek is naar voren gekomen dat 17 van de 23 design patterns zijn verbeterd ten opzichte van de OO-implementaties door gebruik te maken AspectJ.

In de studie van Khalid Aljasser is onderzoek gedaan naar de implementatiemogelijkheden waarbij gebruik is gemaakt van ParaAJ. ParaAJ is een uitbreiding op AspectJ. In het onderzoek is gekeken hoe de design patterns Singleton, Observer en de Decorator kunnen worden geïmplementeerd door middel van ParaAJ. Daarnaast is onderzocht hoe deze patterns zich verhouden met eerder geïmplementeerde varianten in aspect georiënteerde talen zoals AspectJ, EOS en Caesar.

2.1. KWALITEITSBORGING VAN DESIGN PATTERNS DOOR VERIFICATIE

Hanneman en Kiczal hebben in hun onderzoek niet meetbaar gemaakt op welke fronten de AspectJ implementatie een verbetering is ten opzichte van de klassieke implementatie van de design pattern. Cláudio Sant'Anna [14] gaat hier in zijn artikel wel op in. In het artikel van Cláudio Sant'Anna is een kwantitatief onderzoek verricht waarbij het artikel van Hannemmann and Kiczales als basis is genomen. In het artikel is een poging gedaan om te verifiëren of er inderdaad een verbetering is aangebracht door design patterns te benaderen door middel van het aspect georiënteerde paradigma. Hierbij is het principe seperation of concerns centraal gezet. Cláudio Sant'Anna heeft verder duidelijk proberen te maken dat belangrijke software engineering criteria zoals koppeling en cohesie getoetst moeten worden om te kunnen beoordelen of er inderdaad een verbetering heeft plaatsgevonden. Het verifiëren van de OO-eigenschappen is door middel van de volgende metrieken uitgevoerd:

- Seperation of concerns;
- Coupling;
- Cohesion;
- Size.

In het onderzoek is naar voren gekomen dat het merendeel van de GoF patterns is verbeterd om aan het principe seperation of concern van de betreffende pattern te voldoen. Er wordt echter wel de conclusie getrokken dat de implementatievariant die gebaseerd is op aspectJ niet alleen voordelen met zich mee heeft gebracht. Zo introduceren sommige patterns een hogere koppeling tussen de componenten, meer "control statements" en een toename van line of code(LOC) in de aspect georiënteerd oplossing.

Verder is uit onderzoek naar voren gekomen dat het binnen een niet strikt vakgebied zoals software engineering heel lastig blijft om algemene conclusies te trekken wanneer er inderdaad een verbetering heeft plaatsgevonden.

2.2. Design patterns en design kwaliteit

In voorgaande artikelen is niet ingegaan op de kwaliteitsaspecten die door middel van klassieke design patterns worden verbeterd. Hsueh, Chu en Chu [11] gaan hier in hun artikel wel op in. Hsueh, Chu en Chu hebben een onderzoek verricht waarbij de focus is gelegd op de invloeden die design patterns hebben op de kwaliteit van een design. Hun onderzoek is gebaseerd op het feit dat design patterns in vier onderdelen kan worden opgedeeld, namelijk:

- **pattern name:** geeft een abstracte beschrijving van de functionaliteit die vervuld wordt door de betreffende design pattern.
- **intent:** beschrijft welke doelen met het pattern bereikt kunnen worden.
- **solution:** representeert een abstracte omschrijving van de oplossing die kan worden behaald met het pattern. Daarnaast worden de elementen en de samenhang beschreven.
- **consequences:** representeert de voor- en nadelen die worden behaald door het pattern in te zetten.

Het principe dat in het artikel wordt gepresenteerd, is gebaseerd op de onderdelen *solution* en *intent* van een pattern. Er wordt namelijk gekeken of de implementaties consistent zijn aan hetgeen dat de *solution* resprenteert en de *intent* beloofd te bereiken. Met andere woorden, biedt de *solution* ook een oplossing waarbij de *intent* aangeeft wanneer het kan worden ingezet.

In het artikel wordt vanuit verschillende perspectieven naar een pattern gekeken. Een van de perspectieven is vanuit een kwaliteitsperspectief. Hierin wordt naar de *intent* van een design pattern gekeken en wordt deze opgedeeld in twee soorten requirments, namelijk:

- **Functional requirement (FR):** geeft een beschrijving welke taken een pattern zou moeten vervullen.
- **Non Functional requirement (NFR):** geeft een beschrijving welke kwaliteitsattributen (herbruikbaarheid, onderhoudbaarheid of uitbreidbaarheid) kan worden verbeterd.

Beiden requirements kunnen worden vertaald naar een daadwerkelijke structuur dat door middel van een design kan worden gepresenteerd. Hierbij wordt gekeken welke eigenschappen binnen het object georiënteerde design kunnen worden verbeterd door een relatie te leggen tussen de FR en de NRF van een klassendiagram.
Waar het in het artikel op neerkomt, is dat een design vanuit twee perspectieven kan worden benaderd. Enerzijds door een oplossing waarbij enkel de functionele requirement is uitgewerkt dat gerepresenteerd wordt door een design. Anderzijds door het tot stand brengen van een design waarbij zowel de functionele requirements en non-functional requirements in het design zijn verwerkt. In het artikel is door voornoemde onderzoekers per patterns gekeken welke kwaliteitsaspecten kunnen worden geoptimaliseerd. De twee oplossingen (FR en NRF) zijn door middel van het framework QMOOD met elkaar vergeleken om te kijken of de kwaliteitsaspecten inderdaad worden verbeterd. Tot slot is door voornoemde onderzoekers omschreven welke designeigenschappen binnen het design zijn verbeterd.

3

ONDERZOEKSVRAGEN

In dit hoofdstuk zal de aanleiding van dit onderzoek worden uiteengezet. In dit kader zal het probleem worden aangehaald aan de hand van een concreet voorbeeld. Tot slot zal naar aanleiding van het probleem en doelstelling de hoofd- en deelvragen worden geformuleerd.

3.1. AANLEIDING

OO-programmeertalen bevatten elementen die als tekortkoming kunnen worden gezien. Deze tekortkomingen kunnen worden aangevuld door het inzetten van design patterns. Design patterns bieden daardoor oplossingen die een programmeertaal zelfstandig niet kan oplossen.

Tijdens het ontwikkelen van een model kunnen er ontwerpproblemen ontstaan die een OO-programmeertaal niet zelfstandig kan oplossen. Een programmeertaal als Scala heeft bijvoorbeeld het Singelton pattern als taalelement. In dat geval biedt de programmeertaal zelf een oplossing voor het probleem. Voor de OO-programmeertalen die de ontwerpproblemen niet zelfstandig kunnen oplossen zijn elegante oplossingen voor bedacht. Design patterns bieden daardoor oplossingen die een programmeertaal niet zelfstandig kan oplossen.

Design patterns worden ingezet om non-functional requirement tegemoet te komen. Elke pattern heeft een bepaald doel, dan wel een probleem dat het kan oplossen. Daarnaast maken design patterns het mogelijk om de kwaliteitsattributen op systeemniveau zoals flexibiliteit, begrijpbaarheid en herbruikbaarheid te verbeteren [13].

Door de jaren heen krijgt een programmeertaal steeds meer functionaliteiten die de elementen van een taal aanvullen. Zo biedt de programmeertaal Java sinds versie 8 de mogelijkheid om first class functions te implementeren binnen het object georiënteerd paradigma [17]. Door functionele aspecten aan de taal toe te voegen, kunnen de tekortkomingen die door design patterns worden ondervangen, worden herzien.

Aangezien programmeertalen steeds nieuwe features erbij krijgen, ontstaat de mogelijkheid om de oorspronkelijke design patterns op een nieuwe en misschien wel een elegantere manieren te implementeren. Echter staat niet vast dat een pattern ook is te gebruiken zoals deze in 1995 door GoF zijn geïntroduceerd. De toentertijd geïntroduceerde patterns zijn gebaseerde op, de op dat moment, bestaande talen en paradigma's. De implementatievarianten die in het TR zijn beschreven, tonen aan dat het mogelijk is om de oorspronkelijke patterns op een nieuwe manier te implementeren door middel van enumeratie en functionele features die de programmeertaal te bieden heeft.

Als voorbeeld kan de observer pattern worden genomen. De intent van het pattern is als volgt gedefinieerd [4]:

> *Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*

De structuur van de design pattern die door de GoF is vrijgegeven waarmee aan de intent kan worden voldaan, is niet vaststaand. Door de jaren heen hebben de GoF patterns hun bestaansrecht wel geclaimd. Echter kan niet zomaar worden aangenomen indien van de vrijgegeven structuur wordt afgeweken, dat de aangepaste structuur nog aan de intent wordt voldaan. Doordat de programmeertaal steeds nieuwe features krijgt, ontstaat er de behoefte om de oorspronkelijke design patterns te synchroniseren met de huidige technieken en features die een programmeertaal te bieden heeft [5]. Men kan zich dus afvragen of de oorspronkelijke design patterns wel een natuurlijke oplossing bieden zoals de oplossing/pattern die men zou verwachten. Indien bewezen is dat alternatieve implementaties aan de intent voldoen, dan worden de GoF design patterns langzamerhand een synoniem voor de voorgestelde structuur.

Als voorbeeld kan de strategy pattern worden genomen. Wanneer men naar de functionaliteit en/of het doel van de pattern kijkt, dan vervult de pattern de volgende functionaliteiten:

- De strategy biedt een x aantal algoritmes waarbij het dynamisch uitwisselen van algoritmes tijdens runtime juist wordt ondersteund.
- De flexibiliteit met betrekking tot het toevoegen van een nieuwe strategy geschiedt middels het toevoegen van een klasse die de strategy interface implementeert. Hiermee wordt aan het Open Close principe voldaan.

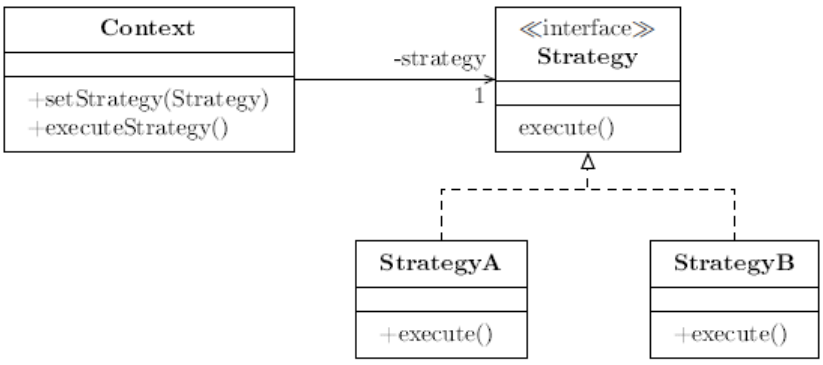
Moet men de algoritmes die toegevoegd kunnen worden wel beschouwen als klasse die toegevoegd worden? Moet men een algoritme niet als een functie beschouwen die op een bepaald moment kan worden ingezet, die tevens afhankelijk is van de situatie? Bovenstaande redeneringen zullen worden meegenomen in het vastleggen van de hoofdvraag en deelvragen.

3.2. PROBLEEM

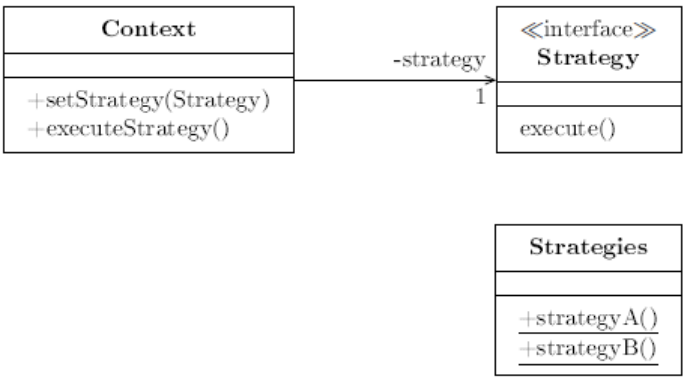
Indien we terug redeneren naar de oorsprong waaruit een design is ontstaan, komen we terecht bij de wens om een softwaresysteem te realiseren die de functionele requirements tegemoet weet te komen. Tijdens de ontwikkeling van het design, probeert men de kwaliteitsaspecten van het design te optimaliseren [9]. Door de kwaliteitsaspecten te optimaliseren, worden non-functionele requirements aan het design toegevoegd [7]. Dit komt doordat de kwaliteitsaspecten geen meerwaarde leveren voor het bereiken van het oorspronkelijke doel van de totstandbrenging van het design.

De kwaliteitsaspecten kunnen worden geoptimaliseerd, door design patterns in te zetten die bewezen oplossingen leveren voor de verschillende knelpunten binnen het object georiënteerde paradigma. Het voordeel van het introduceren van design patterns, is dat het design voorzien zal worden van de eigenschappen flexibiliteit, begrijpbaarheid en herbruikbaarheid.

Wanneer we naar de strategy pattern kijken en naar de gegeven structuur in figuur 3.1, dan komt naar voren dat de strategie het doel heeft om flexibiliteit te introduceren met betrekking tot het toevoegen en uitwisselen van verschillende algoritmes. In het TR zijn de bewezen GoF strategy patterns herschreven door middel van first class functions. In figuur 3.2 komt naar voren dat de concrete strategies vervangen zijn door functies. Doordat de gegeven structuur die door de GoF is vrijgegeven is aangepast, ontstaat de vraag of de alternatieve implementatie de oorspronkelijke kwaliteitseisen die het pattern beoogt te optimaliseren weet te vervullen.



Figuur 3.1: Strategy pattern, OO-implementatie



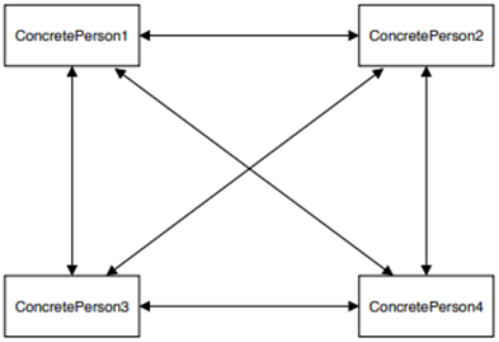
Figuur 3.2: Strategy pattern, functionele implementatie

Het tweede probleem dat zich voordoet, is dat op basis van de gegeven uitwerking niet kunt beoordelen welke implementatie in een bepaalde situatie beter is. Dit komt doordat het in het algemeen erg lastig is om door middel van metrieken de kwaliteit van een ontwerp te meten. Daarnaast is het dat de integratie van OO en FP-features artefacten introduceren, die niet door OO -metrieken noch door FP- metrieken worden ondervangen [18]. Het probleem dat zich hier voordoet, is namelijk dat de OO-metrieken op klasse-niveau fungeren en FP metrieken op functieniveau fungeren. Hierdoor is het niet mogelijk om een eerlijke vergelijking te maken tussen de verschillende softwareversies.

In het rapport “Code Quality Evaluation for theMulti-ParadigmProgramming Language Scala“ [10] is onderzoek gedaan naar de kwaliteit van de code waarbij gebruik is gemaakt van het multi-paradigma Scala. Ook hierin wordt aangegeven dat het niet zomaar mogelijk is aannames te doen dat zowel de OO-metrieken en de FP-metrieken kunnen worden gebruikt om de kwaliteit van de code te meten.

Een ander probleem dat zich voordoet, is wanneer design patterns worden toegepast, elke implementatie van een pattern consequenties met zich meebrengen [4]. Het is de vraag of die consequenties bij verschillende implementaties hetzelfde zijn of verschillen.

Als voorbeeld is een model aangehaald waarbij de functionele requirements gesteld zijn dat vier objecten met elkaar moet kunnen interacteren. Een mogelijke uitwerking is in figuur 3.3 weergegeven.

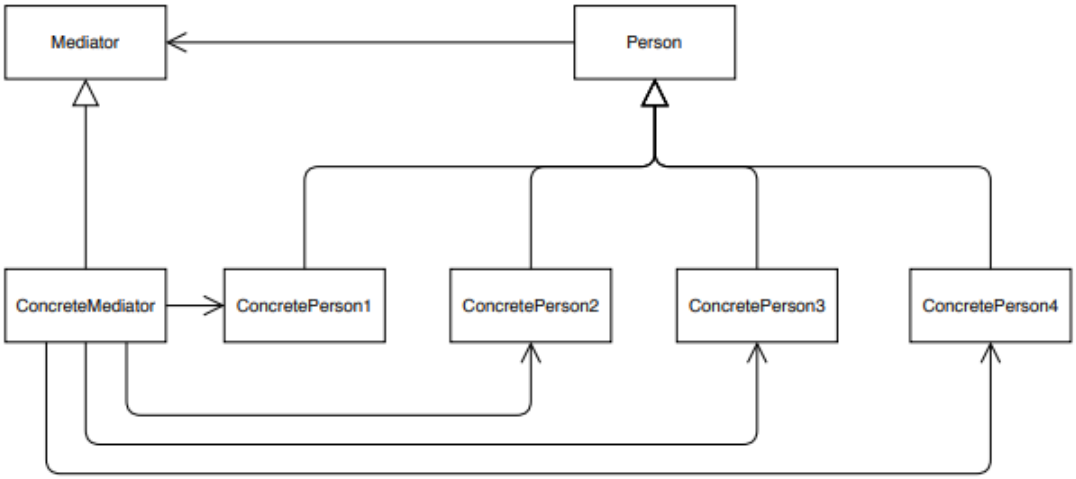


Figuur 3.3: Mediator pattern, FR

Wat in de gekozen oplossing opvalt, is dat een knelpunt wordt geïntroduceerd dat zich vertaald naar de hoeveelheid koppelingen tussen de verschillende objecten. Voor dit probleem kan de mediator pattern worden aangehaald. De intent van deze pattern is namelijk als volgt beschreven [4]:

> *Define an object that encapsulates how a set of object interacts. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.*

In figuur 3.4 is het gegeven model in figuur 3.3 gerefactored door de mediator pattern toe te passen. Waar echter rekening mee moet worden gehouden bij het optimaliseren van de cyclomatische complexiteit, cohesie en koppelingen, is dat het aantal klassen en het aantal regels code kan toenemen [13]. Uiteindelijk kan gesteld worden dat de Mediator positieve invloed heeft op het aantal koppeling dat gereduceerd wordt in het design. Echter zijn daarvoor in de plaats drie nieuwe klassen geïntroduceerd, namelijk: Mediator, ConcreteMediator en Person. Deze aspecten resulteren in de consequenties die de pattern met zich meebrengt.



Figuur 3.4: Mediator pattern, NFR

In de uitwerking van figuur 3.4 is naar voren gekomen dat elke implementatie consequenties met zich meebrengt in de vorm van voor- en nadelen. Indien de GoF implementaties worden herschreven door first class functions, ontstaat de vraag of de voor- en nadelen behouden blijven, of dat er nieuwe voor- en nadelen worden geïntroduceerd.

3.3. Doelstelling

Het doel van het onderzoek is het opstellen van criteria op basis waarvan men kan beslissen welke implementatie van een design pattern in een bepaalde situatie de voorkeur heeft. Met de opgedane kennis kan vervolgens worden beoordeeld in welke situaties de voorkeur uitgaat naar design patterns of naar first class functions dan wel een combinatie van beide paradigma's en kan een framework worden opgesteld. Daarnaast zullen de voor- en nadelen van de vereenvoudiging van de design patterns worden uiteengezet. Door hier duidelijkheid in te verschaffen, kan worden voorkomen dat een design pattern onnodig wordt herschreven aan de hand van functionele features. Hierdoor ontstaat de kans dat de totale complexiteit van het programma toeneemt. De totale complexiteit kan worden uitgedrukt door de hoeveelheid operaties die nodig zijn om een wijziging aan het programma aan te brengen. Hierbij hebben de hoeveelheid overbodige klassen en conditionele statements invloed op het uiteindelijke resultaat.

3.4. Hoofdvraag

Op basis van het eerder omschreven probleem en de doelstelling luidt de hoofdvraag van deze scriptie als volgt:

> "Wat kunnen de bijdragen van functionele aspecten van Java zijn voor de implementatie van Design patterns? "

3.5. Deelvragen

Uit de hoofdvraag kunnen de volgende deelvragen worden afgeleid:

1. Welke kwaliteitsaspecten kunnen aan de bestaande GoF patterns worden gecorreleerd?
2. Op basis van welke kwaliteitscriteria kun je de implementaties van design patterns met elkaar vergelijken?
3. Wat voor alternatieve implementaties van de bestaande design patterns zijn er mogelijk?
4. Hoe verhouden de verschillende implementaties zich tot elkaar met het oog op de kwaliteitsaspecten?
    - Welke consequenties brengt de functionele implementatie met zich mee ten opzichte van de OO-implementatie?
    - Welke doelen worden door de functionele implementatievarianten ten opzichte van de oorspronkelijke design patterns tegemoetgekomen?

4

KWALITEITSASPECTEN DESIGN PATTERNS

In dit hoofdstuk zullen de kwaliteitseigenschappen van design patterns worden aangehaald op basis van de vier onderdelen intent, toepasbaarheid, structuur en de consequenties. Hierin zal worden gekeken welke kwaliteitsaspecten aan de vier onderdelen kunnen worden gecorreleerd. Hiermee zal antwoord worden gegeven op deelvraag 1,

> *"Welke kwaliteitsaspecten kunnen aan de bestaande GoF pattern worden gecorreleerd?"*

Daarnaast zal worden gekeken hoe de kwaliteitsaspecten die aan de betreffende GoF design pattern gecorreleerd zijn, getoetst kunnen worden aan de hand van kwaliteitscriteria door middel van het SOLID principle. SOLID is een verzameling principes dat vaak gebruikt wordt binnen het OO paradigma. Wanneer functionele features gebruikt worden binnen de programmeertaal Java, dan wil dat niet zeggen dat de SOLID principes niet meer kunnen worden toegepast. In 4.4 zal per principe worden aangegeven hoe het gebruikt kan worden in zowel de OO paradigma als het FP paradigma. Hiermee zal antwoord worden gegeven op deelvraag 2,

> *"Op basis van welke kwaliteitscriteria kun je de implementaties van design patterns met elkaar vergelijken?"*

4.1. STRATEGY PATTERN

De strategy pattern valt onder de categorie "behaviour pattern" die de mogelijkheid biedt om een set aan algoritmes te definiëren. De gedefinieerde algoritmes worden afzonderlijk in aparte klassen ondergebracht waardoor het mogelijk is om de objecten zowel dynamisch als statisch uit te wisselen.

4.1.1. KWALITEITSASPECTEN STRATEGY PATTERN

Intent [4]

- *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.*
  - Door de verschillende algoritmes in te kapselen in afzonderlijke klassen, wordt de **cohesie** in het design verhoogt. Doordat het daarnaast mogelijk is om de algoritmes uit te wisselen, ontstaat er **flexibiliteit** in het design.

Toepasbaarheid [4]

- *Many related classes differ only in their behavior.*
- *You need different variants of an algorithm.*
- *An algorithm uses data that clients shouldn't know about.*
- *A class defines many behaviors, and these appear as multiple conditional statements in its operations.*

Consequenties

Voordelen [4]

- *Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse. Inheritance can help factor out common functionality of the algorithms.*
  - Verhogen flexibiliteit.
- *Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend;*
  - Verhogen cohesie en flexibiliteit.
- *Strategies eliminate conditional statements;*
  - Verlagen cyclomatische complexiteit.
- *A choice of implementations. Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.*
  - Verhogen flexibiliteit.

Nadelen [4]

- *The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one. Clients might be exposed to implementation issues. Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients;*
  - Kan een hoge mate van koppeling met zich meebrengen.
- *Communication overhead between Strategy and Context;*
  - Kan een hoge mate van koppeling met zich meebrengen.
- *Increased number of objects.*

OO-properties & Design quality

Door de strategy pattern toe te passen in het design wordt de OO-eigenschap cohesie verhoogd en de cyclomatische complexiteit verlaagd [9, 11]. Daarnaast wordt de design kwaliteit "flexibiliteit"verhoogd [9].

Wanneer we naar de strategy pattern kijken, dan is de flexibiliteit afhankelijk van de hoeveel klassen die aangepast moeten worden om een nieuwe strategy toe te voegen. Doordat er gebruik wordt gemaakt van polymorfisme, is het eenvoudig om een nieuwe klasse te introduceren dat een algoritme representeert. De flexibiliteit voor het aanbrengen van een wijziging is gedefinieerd door de hoeveelheid klassen dat moet worden aangepast om een wijziging door te voeren [12]. Voor de strategy pattern komt dat op het volgende neer:

> *Ordegrootte van (het aantal klassen dat gewijzigd is door het toevoegen van een nieuwe strategy) = O(1)*

4.2. TEMPLATE PATTERN

De template method design pattern valt onder de categorie “Behavioral patterns“ en heeft als doel om variërende delen van een algoritme te implementeren zonder de gehele structuur van de gedefinieerde algoritme aan te passen. De concrete algoritmes worden in de subklassen gedefinieerd zonder de structuur van het algoritme aan te passen.

4.2.1. KWALITEITSASPECTEN TEMPLATE METHOD

Intent [4]

- *Define the skeleton of an algorithm in an operation, deferring somesteps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.*
  - Doordat in de subklassen de concrete algoritmes worden gedefinieerd, wordt **code duplicatie** geëlimineerd en wordt de **flexibiliteit** van het design verhoogt.

Toepasbaarheid [4]

- *To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.*
- *When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.*
- *To control subclasses extensions.*

Consequenties

Voordelen [15]

- *Eliminating code duplication.*
- *You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.*
  - Verhogen cohesie.
- *You can pull the duplicate code into a superclass.*

Nadelen [15]

1. *Some clients may be limited by the provided skeleton of an algorithm.*
2. *You might violate the Liskov Substitution Principle by suppressing a default step implementation via a subclass.*
3. *Template methods tend to be harder to maintain the more steps they have*

OO-properties & Design quality

Door de Template method pattern toe te passen in het design wordt de OO-eigenschap cyclomatische complexiteit verlaagd en code duplicatie geëlimineerd [9, 15]. Daarnaast wordt de design kwaliteit "flexibiliteit" verhoogd [9, 15].

4.3. Command pattern

De Command pattern valt onder de categorie "behavioral design pattern" dat een mechanisme introduceert om de request van een "invoker" te doen scheiden van de "receiver".

4.3.1. Kwaliteitsaspecten Command pattern

Intent [4]

- *Encapsulate a request as an object, there by letting the developer to parameterize clients with different requests, queue or log requests, and support undoable operations.*
  - Door de verschillende requesten in een object in te kapselen wordt de **cohesie** in het design verhoogt. Daarnaast wordt **flexibiliteit** geïntroduceerd in het design.

Toepasbaarheid [4]

- *Parameterize objects by an action.*
- *Specify, queue, and execute requests at different times.*
- *Support undo. The Command's Execute operation can store state for reversing its effects in the command itself.*
- *Support logging changes so that they can be reapplied in case of a system crash.*
- *Structure a system around high-level operations built on primitives operations.*

Consequenties

Voordelen [4]

- *Command decouples the object that invokes the operation from the one that knows how to perform it.*
  - Verhogen flexibiliteit.
- *Commands are first-class objects. They can be manipulated and extended like any other object.*
- *You can assemble commands into a composite command. In general, composite commands are an instance of the Composite pattern.*
- *It's easy to add new Commands, because you don't have to change existing classes.*

OO-properties & Design quality

Door de command pattern toe te passen in het design worden de OO-eigenschap cyclomatische complexiteit verlaagd [9]. Daarnaast wordt de design kwaliteit "flexibiliteit" verhoogd [9].

4.4. SOLID PRINCIPLE

Zoals in hoofdstuk 3.2 is aangegeven, kunnen we niet zomaar aannemen dat zowel de OO-metrieken als de FP-metrieken kunnen worden gebruikt om de kwaliteit van de code te meten. Om toch over de implementatievarianten heen te redeneren en deze met elkaar te vergelijken, is er voor gekozen om de SOLID principes toe te passen. Door de SOLID principes toe te passen is het mogelijk om een ontwerp te evalueren en te toetsen of de kwaliteit van het ontwerp goed is.

Design patterns vormen een van de middelen om de kwaliteit van het design te optimaliseren. Het SOLID principle is daarentegen een acroniem dat uit een set van vijf basis principes bestaat, die het mogelijk maakt om de onderhoudbaarheid en uitbreidbaarheid van een design te optimaliseren [15]. SOLID staat namelijk voor [16]:

- Single-Responsible principle (SRP):
  - *A element should have only one reason to change.*
- Open–Closed Principle (OCP):
  - *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*
- Liskov Substitution Principle (LSP):
  - *Subtypes must be substitutable for ther base type.*
- Interface-Segregation Principle (ISP):
  - *Clients should not be forced to depend on methods that they do not use*
  - *This principle deals with the disadvantages of "fat" interfaces*
- Dependency-Inversion Principle (DIP):
  - *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *Abstractions should not depend on details. Details should depend on abstractions.*

5

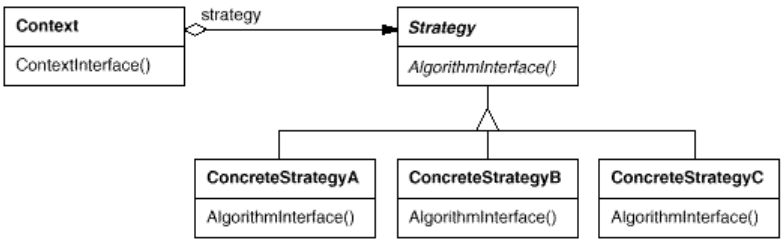
ALTERNATIEVE IMPLEMENTATIE

In dit hoofdstuk zullen verschillende implementatievarianten van de oorspronkelijke design pattern worden aangehaald. De varianten zullen later in hoofdstuk 6 worden aangehaald om deze met elkaar te vergelijken. Door de verschillende implementatievarianten aan te halen zal antwoord worden gegeven op de deelvraag 3,

> *"Wat voor alternatieve implementaties van de bestaande design patterns zijn er mogelijk?"*

5.1. STRATEGY PATTERN

De klassieke OO implementatie die door de Gang of Four is vrijgegeven zoals te zien is in figuur 5.1, bestaat uit een "Context klasse" die een associatie heeft met een abstracte klasse of interface genaamd "Strategy". In de subklassen zijn de concrete algoritmes gedefinieerd.



Figuur 5.1: OO implementatie Template method

Listing 5.1: Strategy pattern OO

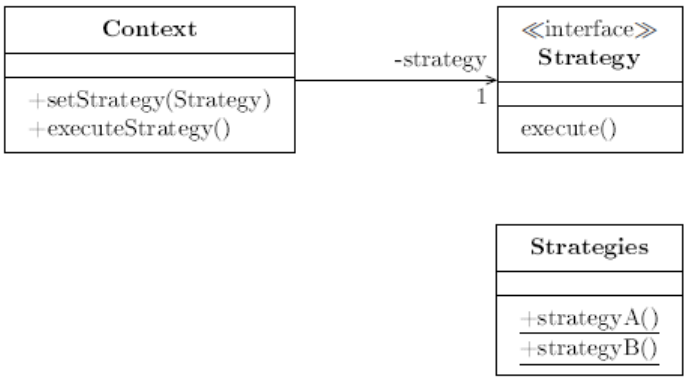
```
public class Context {
   static int number = 5;
   Strategy strategy;

   public void setStrategy(Strategy strategy) {
      this.strategy = strategy;
   }
```

```
    void execute(){
        System.out.println(strategy.execute(number));
    }
}

public interface Strategy {
    int execute(int x);
}

public class StrategyA implements Strategy {
    @Override public int execute(int x) {
        return x + 10;
    }
}

public class StrategyB implements Strategy {
    @Override public int execute(int x) {
        return x + 20;
    }
}

public class Main {
    public static void main(String arg[]) {
        Context context = new Context();
        context.setStrategy(new StrategyA());
        context.execute();

        context.setStrategy(new StrategyB());
        context.execute();
    }
}
```

5.1.1. FUNCTIONELE IMPLEMENTATIE

De OO implementatie die door de Gang of Four is vrijgegeven kan simpelweg worden vereenvoudig door gebruik te maken van first class functions waarbij de concrete strategies worden vervangen door functies. In figuur 5.2 zijn de concrete strategies vervangen door static methods / lambda expressie in de klasse “Strategies“.

Structuur functionele implementatie (FF-Strategy) [1]



Figuur 5.2: Functionele implementatie strategy pattern

Source code (FF-Strategy) [1]

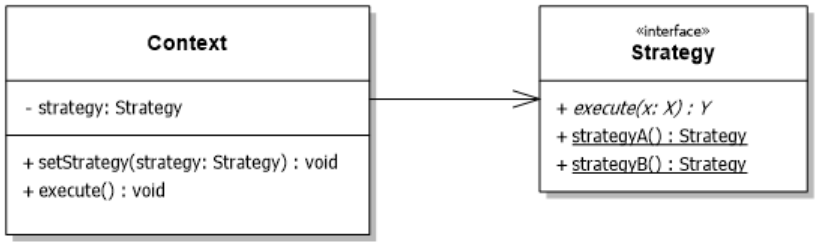
Listing 5.2: Strategy pattern

```
public interface Strategy {
   void execute ();
}

public class Strategies {
   public static void strategyA () {
      // implementation of strategy A
   }

   public static void strategyB () {
      // implementation of strategy B
   }
}

public class Context {
   private Strategy strategy ;

   public void setStrategy (Strategy strategy) {
      this.strategy = strategy ;
   }
   public void executeStrategy () {
      strategy.execute ();
   }
}
```

```
public static void main ( String [] args ) {
   Context context = new Context ();
   context.setStrategy(Strategies::strategyA);
   context.executeStrategy();
   context.setStrategy(Strategies::strategyB);
   context.executeStrategy();
   context.setStrategy(() -> System.out.println("New strategy"));
   context.executeStrategy();
}
```

In figuur 5.2 is er voor gekozen om de functionele implementatie van de Strategy pattern uit te werken door een functionele interface te definiëren waarin exact één abstracte methode is gedefinieerd. Daarnaast is een aparte “Strategies“ klasse gedefinieerd waar de strategies in worden ondergebracht. Een andere manier om de strategies te definiëren, is om de functies te verplaatsen naar de functionele interface. De Java compiler gaat er namelijk vanuit, indien in de interface exact één abstracte methode is gedefinieerd, dan wordt de interface gezien als een functionele interface [3]. Het is daarom toegestaan om in de functionele interface default en static methodes te definiëren. Daarnaast bestaat de mogelijkheid sinds Java versie 11 om private methodes te definiëren in de interface. De gegeven implementatie kan simpelweg worden vereenvoudigd zoals te zien is in figuur 5.3 door de methodes in de klasse “Strategies“ te verplaatsen naar de functionele interface “strategy“.

Structuur functionele implementatie (FF-Strategy-2)



Figuur 5.3: Functionele implementatie strategy pattern

Source code (FF-Strategy-2)

Listing 5.3: Strategy pattern

```
public interface Strategy {
   Y execute(X x);

   static Strategy strategyA() {
      //implementation of Strategy A
   }
   static Strategy strategyB() {
      //implementation of Strategy B
   }
}
```

5.1.2. HYBRIDE IMPLEMENTATIE

De functionele interface wordt gedefinieerd door exact één abstracte methode in de interface te definiëren. Hierdoor ontstaat de mogelijkheid om de gegeven uitwerking in figuur 5.1 te gebruiken als een hybride implementatie. Dat wil zeggen dat de “Strategy interface“ gebruikt kan worden als abstractielaag om concrete OO strategies te definiëren, maar het is ook mogelijk om functies te definiëren die gebaseerd zijn op dezelfde interface. Dit komt doordat het niet verplicht is om de annotatie “@FunctionalInterface“ te gebruiken om aan te geven dat een interface een functionele interface is. Het nadeel hiervan is dat de programmeur op te hoogte moet zijn dat er functies zijn gedefinieerd op basis van de interface. Indien er een abstracte method aan de interface wordt toegevoerd, dan is het geen valide oplossing meer.

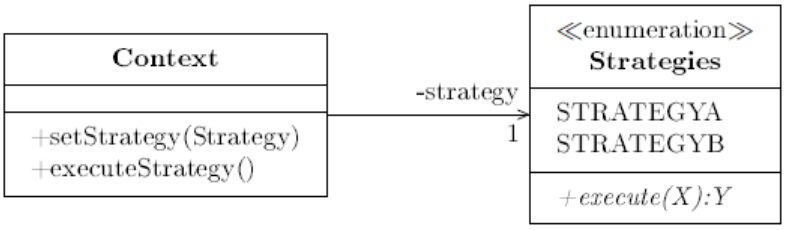
Listing 5.4: Strategy pattern hybride

```
public class Main {
   public static void main(String arg[]) {
   Context context = new Context();
   context.setStrategy(new StrategyA());
   context.execute();

   context.setStrategy(new StrategyB());
   context.execute();

   context.setStrategy((int x) -> x + 100);
   context.execute();
   }
}
```

5.1.3. ENUM IMPLEMENTATIE

De OO implementatie van de Strategy pattern kan ook herschreven worden door gebruik te maken van enumeration, waarbij de concrete strategies vervangen zijn door enum types. In figuur 5.4 zijn de algortimes gedefinieerd als enum-constanten.

Structuur functionele implementatie (Enum-Strategy) [1]



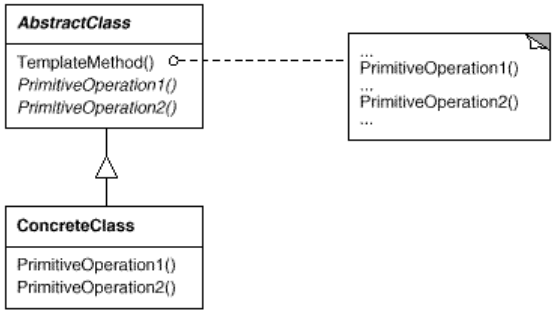
Figuur 5.4: Enum implementatie strategy pattern

Source code (Enum-Strategy) [1]

```
public enum Strategies {
   STRATEGYA {
      public Y execute (X x) {
         // implementation of strategy A
      }
   },

   STRATEGYB {
      public Y execute (X x) {
         // implementation of strategy B
      }
   };

   public abstract Y execute (X x);
}

public class Context {
   private Strategies strategy ;

   public void setStrategy ( Strategies strategy ) {
      this.strategy = strategy ;
   }

   public void executeStrategy () {
      y = strategy.execute (x);
   }
}
```

5.2. TEMPLATE METHOD

De structuur van de template method wordt beschreven door een abstracte klasse waarbij de structuur van het algoritme in de “templateMethod“ wordt vastgelegd. De concrete implementaties worden in de subklassen gedefinieerd door abstracte methodes te overschrijven. Figuur 5.5 representeert de structuur die door de Gang of Four is vrijgegeven.



Figuur 5.5: OO implementatie Template method

Listing 5.5: Template method pattern OO

```
public abstract class AbstractClass {
   public final void templateMethod () {
      primitiveOperation1 ();
      primitiveOperation2 ();
   }
   protected abstract void primitiveOperation1 ();
   protected abstract void primitiveOperation2 ();
}

public class ClassA extends AbstractClass {
   protected void primitiveOperation1 () {
      // implementation
   }
   protected void primitiveOperation2 () {
      // implementation
   }
}

public class ClassB extends AbstractClass {
   protected void primitiveOperation1 () {
      // implementation
   }
   protected void primitiveOperation2 () {
      // implementation
   }
}
```

```
public static void main (String [] args) {
   AbstractClass class1 = new ConcreteClassA();
   class1.templateMethod ();
   AbstractClass class2 = new ConcreteClassB();
   class2.templateMethod ();
}
```

5.2.1. FUNCTIONELE IMPLEMENTATIE

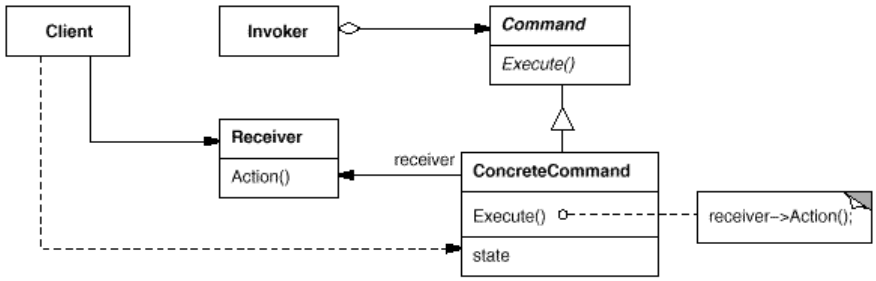
In de OO implementatie in figuur 5.5 is te zien, dat de daadwerkelijke structuur van het algoritme in één concrete methode in combinatie met meerdere abstracte methoden in de “AbstractClass“ is gedefinieerd. Echter wanneer we dit functioneel proberen op te lossen, dan kun je slechts één abstracte methode definiëren. Hierdoor voldoet de functionele implementatie niet aan de intent van het pattern, namelijk:

> *"Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure"*

Doordat de gegeven implementatie in het TR niet aan de intent voldoet, zal deze niet verder worden genomen in het beantwoorden van de hoofd- en deelvragen.

5.3. COMMAND PATTERN

De command pattern introduceert een mechanisme waarmee het mogelijk is om de zender en ontvanger van elkaar te scheiden. De daadwerkelijke operaties worden in de klasse “ConcreteCommand“ gedefinieerd. Figuur 5.6 representeert de structuur die door de Gang of Four is vrijgegeven.



Figuur 5.6: OO implementatie command pattern

Source code (Command Pattern OO) [1]

Listing 5.6: Command pattern OO

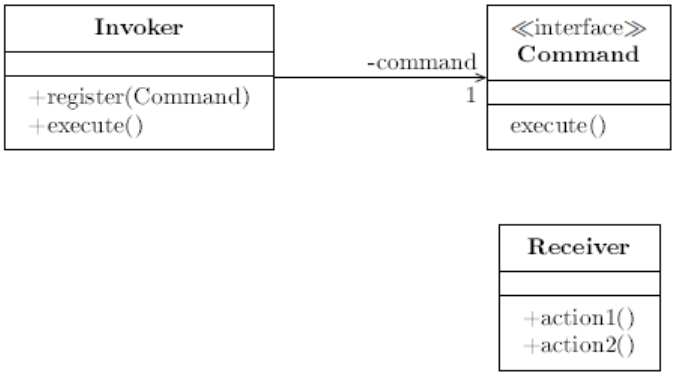
```
public class Invoker {
   private Command command ;
   public void register(Command command) {
      this.command = command ;
   }
   public void execute() {
      command.execute();
   }
}

public interface Command {
   void execute();
}

public class CommandA implements Command {
   private Receiver receiver;

   public CommandA (Receiver receiver) {
      this.receiver = receiver;
   }

   public void execute () {
      receiver . action1 ();
   }
}

public class CommandB implements Command {
   private Receiver receiver;

   public CommandB (Receiver receiver) {
      this.receiver = receiver;
   }
   public void execute() {
      receiver.action2 ();
   }
}

public class Receiver {
   public void action1() {
      // implementation
   }
   public void action2() {
      // implementation
   }
}
```

```
public static void main (String [] args) {
   Receiver receiver = new Receiver();
   Invoker invoker = new Invoker();
   invoker.register (new CommandA (receiver));
   invoker.execute();
   invoker.register (new CommandB(receiver));
   invoker.execute();
}
```

5.3.1. FUNCTIONELE IMPLEMENTATIE

De OO implementatie die door de Gang of Four is vrijgegeven kan wederom worden vereenvoudig door gebruik te maken van first class functions. In figuur 5.7 zijn de concrete "Commands" vervangen door functies. Bij aan het aanroepen van de functie wordt de daadwerkelijke action van de receiver uitgevoerd.

Structuur functionele implementatie (FF-Command) [1]



Figuur 5.7: Functionele implementatie command pattern

Source code (FF-Command) [1]

```
public interface Command {
   void execute ();
}

public class Invoker {
   private Command command ;

   public void register ( Command command ) {
      this.command = command ;
   }
   public void execute () {
      command.execute ();
   }
```

```
}

public class Receiver {
   public void action1 () {
      // implementation
   }

   public void action2 () {
      // implementation
   }
}

public static void main ( String [] args ) {
   Receiver receiver = new Receiver ();
   Invoker invoker = new Invoker ();
   invoker.register(receiver::action1);
   invoker.execute();
   invoker.register(receiver::action2);
   invoker.execute();
   invoker.register(() -> System.out.println(" extra command " ));
   invoker.execute();
   invoker.register(() -> new Receiver().action2());
   invoker.execute();
}
```

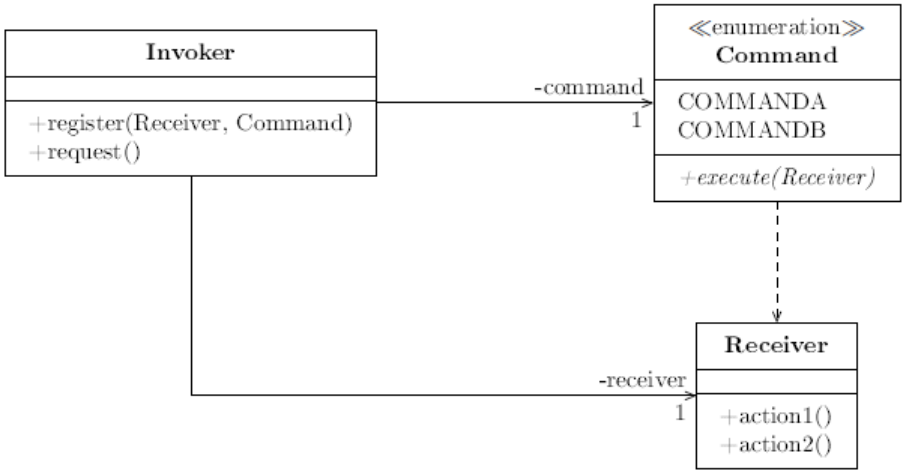
VEREENVOUDIGING FUNCTIONALE IMPLEMENTATIE COMMAND PATTERN

Ook bij de functionele implementatie van de Command pattern geldt weer dat de daadwerkelijke command in de functionele interface kunnen worden ondergebracht. Hierdoor komt de klasse “Receiver“ te vervallen.

5.3.2. ENUM IMPLEMENTATIE

Ook hier geldt weer dat de OO implementatie van de Command pattern kan worden herschreven door gebruik te maken enumeration. In figuur 5.8 zijn de concrete "Commands" gedefinieerd als enum types.

Structuur functionele implementatie (Enum-Command) [1]



Figuur 5.8: Enum implementatie strategy pattern

Source code (Enum-Command) [1]

```
public class Invoker {
   private Command command ;
   private Receiver receiver ;

   public void register (Receiver receiver , Command command) {
      this.receiver = receiver ;
      this.command = command ;
   }

   public void request () {
      command.execute ( receiver );
   }
}

public enum Command {
   COMMAND1 {
      public void execute (Receiver receiver) { receiver.action1 (); }
   },
   COMMAND2 {
      public void execute (Receiver receiver) { receiver.action2 (); }
   };

   public abstract void execute (Receiver receiver);
}
```

```
public class Receiver {
   public void action1() { // implementation }
   public void action2() { // implementation }
}
```

6

ANALYSE IMPLEMENTATIES

In dit hoofdstuk zullen verschillende implementatievarianten met elkaar worden vergeleken op het gebied van de intent, consequenties en het SOLID principe. Hiermee zal antwoord worden gegeven op deelvraag 4,

> *"Hoe verhouden de verschillende implementaties zich tot elkaar met het oog op de kwaliteitsaspecten?"*

Design patterns worden binnen een softwaresysteem toegepast om een veel voorkomend probleem op te lossen. Zo heeft elke pattern een doel op zich dat beschreven wordt door de “intent“ van het pattern. De intent van het pattern wordt ondervangen door het gegeven structuur dat door de “The Gang of Four“ is vrijgegeven. Men is er echter vrij in om van het gegeven structuur af te wijken. Men dient zich wel te realiseren of hiermee nog aan de intent van het pattern wordt voldaan. In het vorige hoofdstuk zijn enkele GoF implementatie vervangen door een functionele dan wel enumeratie implementatievarianten.

Door kritisch naar de intent van het oorspronkelijke pattern te kijken en deze te weerspiegelen met de functionele implementatie zal gekeken worden welke consequenties de functionele implementatie met zich mee brengt ten opzichte van de OO-implementatie.

Daarnaast zal worden gekeken welke consequenties door de functionele dan wel de enumeration implementatievariant worden geïntroduceerd. Hiermee zal inzichtelijk worden gemaakt welke doelen door de functionele / enumeratie implementatievarianten ten opzichte van de oorspronkelijke design patterns tegemoet worden gekomen.

Door de consequenties inzichtelijk te maken, is het mogelijk tijdens de design fase een overweging te maken naar welke implementatievariant de voorkeur uit gaat.

6.1. STRATEGY PATTERN

Intent GoF vs Intent functionele implementatie & enum

- *define a family of algorithms.*
  - **(FF-Strategy)** De strategie klassen zijn vervangen door een default functiedefintie die in de klasse “Strategies“ zijn gedefinieerd als zijnde algoritme.
  - **(Enum-Strategy)** De strategie klassen zijn vervangen door constanten die de abstracte methode implementeren.
- *encapsulate each one.*
  - **(FF-Strategy)** Elk algoritme wordt door middel van een functiedefinitie ingekapseld.
  - **(Enum-Strategy)** Elk algortime wordt binnen de enum-definitie ingekapseld.
- *make them interchangeable.*
  - **(FF-Strategy)** De algoritmes zijn zowel dynamisch als statisch uitwisselbaar door een andere functie te implementeren.
  - **(Enum-Strategy)** Ook hier zijn de algoritmes dynamisch als statisch uitwisselbaar.

Intent functionele implementatie & enum vs SOLID

- **Single responsibility principle**
  - In de OO implementatie wordt de verantwoordelijkheid van het algoritme overgedragen aan de concrete strategies waarbij de meerdere verantwoordelijkheden kunnen worden toegekend. Het is namelijk ook mogelijk om data op te slaan en deze op een later tijdstip weer te bewerken. De “FF-Strategy“ en “Enum-Strategy“ daarentegen wordt enkel de mogelijkheid geboden om de verantwoordelijkheid te dragen die aan de implementatie is toegekend. Met andere woorden, ze moeten het mogelijk maken om een set aan algoritmes in te kapselen en ze uitwisselbaar maken. Beiden implementaties leggen daarnaast de restrictie op, dat het niet mogelijk is meerdere verantwoordelijkheden te dragen. Dit komt doordat de functionele implementatie enkel de taak heeft een resultaat op te leveren op basis van de input.
- **Open closed principle**
  - De OO implementatie conformeert zich aan het open closed principle doordat het toevoegen van een algoritme geschied door het toevoegen van een subklasse. Bij de “FF-Strategy“ zijn de algoritmes ondergebracht in functiedefinities.

Strikt genomen, zou dit betekenen indien er een nieuw algoritme toegevoegd moeten worden, dat er een nieuwe klasse moet worden aangemaakt met daarin de toegevoegde algoritme in de vorm van lambda expressie. Dit zou waanzinnig zijn om dit voor elk nieuw algoritme te doen. Een nieuwe algoritme kan eenvoudig aan de Strategies klasse worden toegevoegd zonder dat de bestaande delen van het programma worden aangetast. Het principe achter “A element should have just one reason to change“ komt hiermee te vervallen, maar doordat er spaken is van functies / constanten die worden toegevoegd, voldoet het nog altijd aan het Open closed principle. Dit zelfde geldt ook voor de “Enum-Strategy“. Het open closed principle moet een uitzondering op de regel maken indien er sprake is van functies en constanten. Indien de “Enum-Strategy“ onderdeel uitmaakt van een library, dan is het niet meer mogelijk om de strategy uit te breiden met nieuwe concrete strategies. Dit kan vanuit twee perspectieven worden bekeken. Het kan een voordeel zijn indien gewenst is dat er geen nieuwe strategies kunnen worden toegevoegd. Maar in de meeste gevallen is het gewenst dat gebruikers van de library de code kunnen uitbreiden. In dat opzicht is het een nadeel.

- **Liskov Substitution Principle**
  - Indien voldaan is aan het contract van de interface, wordt automatisch aan het Liskov Substitution Principle voldaan. Hierin verschillen de OO, FF-Strategy en Enum-Strategy verschillen niet in.
- **Interface-Segregation Principle**
  - Dit principe wordt ondervangen door het Single responsibility principle. De functies en enumerations hebben maar één taak en verantwoordelijkheid.
- **Dependency inversion principle**
  - Hier komt het eigenlijk op neer dat er tegen een interface aan geprogrammeerd moet worden. De functionele implementatie voldoet aan dit principe aangezien de functies gebruik maken van de functionele interface. De enum implementatie voldoet hier niet aan.

6.1.1. Consequenties Strategy pattern

De voordelen die in sectie 4.1.1 zijn aangehaald, blijven bij de functionele en enum implementatie behouden. Tevens blijft de flexibiliteit waarmee een nieuwe strategy kan worden toegevoegd behouden. Ook hier geldt weer een ordegrootte van 1. Tevens zijn de volgende voor- en nadelen geïntroduceerd:

Voordelen

1. **(FF / Enum)** Het probleem dat zich bij de GoF implementatie voordeed is dat het aantal klassen/objecten kan toenemen, doet zich niet meer voor doordat de strategies zijn vervangen door functies.

Nadelen

1. **(FF / Enum)** Doordat de verschillende strategies in één klassen worden gedefinieerd, bestaat de kans dat de klasse waarin alle functies zijn gedefinieerd extreem groot wordt.
2. **(FF / Enum)** Indien er in de OO variant gebruik is gemaakt van hulpfuncties, waarin complexe algoritmen zijn gebruikt, dan wordt het enorm complex om het functioneel te herschrijven.

Het eerste wat opvalt indien er gebruik wordt gemaakt van de functionele features, is dat de functionele implementatie het niet toelaat om data in de gedefinieerde strategies op te slaan. Dit betekent dat de attributen die eerder in de Strategies zijn gedefinieerd, verplaats moeten worden naar de Context. Door de first class functions word je als programmeur zijnde gedwongen om over de pattern goed na te denken hoe dit juist kan worden geïmplementeerd.

In het algemeen kan gesteld worden zodra men er voor kiest om gebruik te maken van first class functions, dat het niet meer mogelijk is om toestanden van objecten bij te houden. De GoF-implementatie laat het namelijk wel toe om data in de strategy zelf bij te houden. De consequentie die hierbij optreedt, is dat de state van de betreffende strategie in de Context moet worden bijgehouden. Dit lijkt in eerste opzicht een consequentie dat als nadeel kan worden gezien. Echter wanneer men naar de intent van de strategie pattern kijkt, dan komt naar voren dat de focus wordt gelegd op het uitwisselen van algoritmes. Doordat het niet meer mogelijk is om toestanden van objecten bij te houden in de strategie zelf, wordt de gebruiker gedwongen om zich aan de intent van het pattern te houden. Hierdoor wordt de oplossing preciezer geïmplementeerd. Hierbij wordt het bijhouden van de state en het bewerken van de state van elkaar gescheiden.

6.2. COMMAND PATTERN

Intent GoF vs Intent functionele implementatie & enum

Ook bij de Command pattern is het van belang om te kijken of de functionele en de enum implementatie voldoen aan de intent van de GoF implementatie. Hiervoor is de intent van het GoF pattern aangehaald om te verifiëren of het een valide oplossing is van de GoF implementatie.

- *The intent of the Command Design Pattern is to encapsulate a request as an object.*
  - **(FF-Command)** De request wordt door middel van een functiedefinitie ingekapseld.
  - **(Enum-Command)** De request wordt door middel van een constante in de enum gedefinieerd.
- *There by letting the developer to parameterize clients with different requests.*
  - **(FF-Command)** De request parameter kan door middel van “register method” worden toegekend.
  - **(Enum-Command)** Idem.

- *queue or log requests.*
  - **(FF-Command)** Is met de functionele implementatie niet mogelijk. Een alternatief wordt in de “support undoable operations intent“ aangehaald.
  - **(Enum-Command)** Idem.
- *and support undoable operations.*
  - **(FF-Command)** Om “Undable operations“ te ondersteunen, dienen de eerder uitgevoerde commands te worden bijgehouden. Aangezien het opslaan van data met functies niet mogelijk is, zal het met de functionele implementatie niet mogelijk zijn om aan deze intent voldoen. In 6.2.1 zal een voorstel worden gedaan hoe dit kan worden opgelost.
  - **(Enum-Command)** Idem.

Intent functionele implementatie & enum vs SOLID

- **Single responsibility principle**
  - In de OO implementatie wordt de request in de vorm van een object ingekapseld. Daarnaast krijgt de betreffende command ook de verantwoordelijkheid om een queue, log requests en undoable operations bij te houden. Hierin komt naar voren dat de command meerdere verantwoordelijkheden met zich meekrijgt. Hierdoor is het niet mogelijk om aan het “Single responsibility principle“ te voldoen. Bij de FF-Command en Enum-Command wordt een restrictie opgelegd waardoor het niet mogelijk is om data bij te houden. Hierdoor wordt gewdongen om de verantwoordelijkheden om queue, log requests en undoable operations te ondervangen elders te plaatsen.
- **Open closed principle**
  - De OO implementatie conformeert zich aan het open closed principle doordat het toevoegen van een command geschied door het toevoegen van een subklasse. Bij de FF-Command zijn de algoritmes ondergebracht in functiedefinities. Strikt genomen, zou dit beteken indien er een nieuw algoritme toegevoegd moeten worden, dat er een nieuwe klasse moet worden aangemaakt met daarin de toegevoegde algoritme in de vorm van lambda expressie. Dit zou waanzinnig zijn om dit voor elk nieuw algoritme te doen. Een nieuwe algoritme kan eenvoudig aan de Command klasse worden toegevoegd zonder dat de bestaande delen van het programma worden aangetast. Het principe achter “A element should have just one reason to change“ komt hiermee te vervallen, maar doordat er spaken is van functies / constanten, voldoet het nog altijd aan het Open closed principle. Dit zelfde geldt ook voor de Enum-Command. Het open closed principle moet een uitzondering op de regel maken indien er sprake is van functies en constanten.

Indien de “Enum-Command“ onderdeel uitmaakt van een library, dan is het niet meer mogelijk om de command uit te breiden met nieuwe concrete commands. Dit kan vanuit twee perspectieven worden bekeken. Het kan een voordeel zijn indien gewenst is dat er geen nieuwe command kunnen worden toegevoegd. Maar in de meeste gevallen is het gewenst dat gebruikers van de library de code kunnen uitbreiden. In dat opzicht is het een nadeel.

- **Liskov Substitution Principle**
  - De OO, FF-Strategy en Enum-Strategy verschillen hier niet in.
- **Interface-Segregation Principle**
  - Dit principe wordt ondervangen door het Single responsibility principle. De functies en enumerations hebben maar één taak en verantwoordelijkheid.
- **Dependency inversion principle**
  - Hier komt het eigenlijk op neer dat er tegen een interface aan geprogrammeerd moet worden. De functionele implementatie voldoet aan dit prinicipe aangezien de functies gebruik maken van de functionele interface. De enum implementatie voldoet hier niet aan.

6.2.1. Consequenties Command pattern

De voordelen die in sectie 4.3 zijn aangehaald blijven bij de functionele implementatie van toepassing. Daarbij zijn de volgende voor- en nadelen geïntroduceerd:

Voordelen

- De command objecten zijn bij de functionele implementatie overbodig waardoor de klassendiagram op basis van het aantal commands kan worden gereduceerd. Concreet betekent dat de klassendiagram kan worden gereduceerd.

Nadelen

- De functionele implementatie laat het niet toe om “Undoable operations“ te implementeren.

De mogelijkheid die ontnomen wordt om “undoable operations“ uit te voeren, lijkt in eerste instantie een nadeel. Er wordt echter niet gedefinieerd dat de Command pattern de verantwoordelijkheid moet dragen om de state van de “Invoker“ bij te houden. Doordat het juist niet meer mogelijk is om de geschiedenis van de "Invoker"dan wel de gepasseerde command bij te houden in de “Command“ zelf, wordt de programmeur gedwongen om de verantwoordelijkheid elders te plaatsen. Wanneer we naar de intent van de “Memento“ kijken, dan zien we dat dit de geschikte plek is om het probleem op te lossen.

Memento pattern

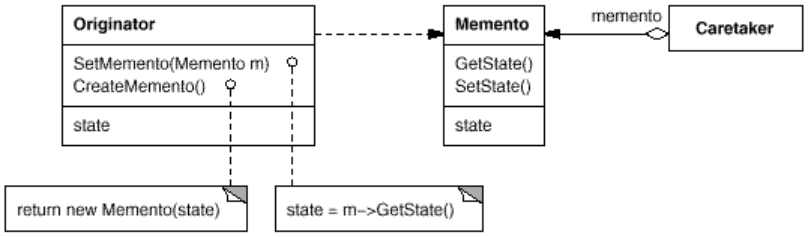
Intent [4]

- *Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.*

Toepasbaarheid [4]

- *A snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later*
- *A direct interface to obtaining the state would expose implementation details and break the object's encapsulation*

Structuur [4]



Figuur 6.1: OO implementatie memento pattern

Consequenties

Voordelen [4]

- *Preserving encapsulation boundaries. Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator. The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.*
- *It simplifies Originator. In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested. That puts all the storage management burden on Originator. Having clients manage the state they ask for simplifies Originator and keeps clients from having to notify originators when they're done.*

Nadelen [4]

- *Using mementos might be expensive. Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough.*
- *Defining narrow and wide interfaces. It may be difficult in some languages to ensure that only the originator can access the memento's state.*
- *Hidden costs in caring for mementos. A caretaker is responsible for deleting the mementos it cares for.*

7

VALIDATIE

In hoofdstuk 4 is per design pattern aangegeven welke design properties door de design patterns worden geoptimaliseerd. Wanneer we naar de Strategy en de command pattern kijken, dan komt naar voren dat de eigenschappen op klasse niveau worden geoptimaliseerd. De design properties cohesie, cyclomatische complexiteit en polymorfisme kunnen inzichtelijk worden gemaakt door OO-metrieken op het design toe te passen.

Om te kunnen toetsen welke consequenties de functionele implementatie met zich meebrengen, is er voor gekozen om het Java project “Jabberpoint“ te toetsen middels het QMOOD model. Door het QMOOD model toe te passen is het mogelijk om een kwantitatieve benadering te bieden. QMOOD biedt namelijk een vier lagen model aan waarmee het mogelijk is een relatie te leggen tussen Design quality, OO-design properties, OO-metrieken en OO design componenten. Dit biedt tevens ook mogelijk om een relatie te leggen tussen de interne en externe kwaliteiten.

Om de design properties cohesie, cyclomatische complexiteit en polymorfisme te toetsen, zijn de volgende softwareversies getoetst:

- De originele versie van Jabberpoint is als uitgangspunt genomen (versie 0.0, zonder Design patterns), getoetst middels het QMOOD model.
- De kwaliteit van de gerefactorde versie van Jabberpoint(“IM0102 Design Patterns“) is middels het QMOOD model getoetst (versie 1.0, Design patterns toegepast).
- De gerefactorde versie van Jabberpoint is middels functionele patterns herschreven (versie 2.0).

In Jabberpoint(“IM0102 Design Patterns“) is de OO-strategy en OO-command pattern vervangen door FF-strategy en FF-command pattern. Het eerste wat naar voren is gekomen nadat het design is gerefactored, is dat de factory pattern om de concrete strategies / commands te instantiëren overbodig is geworden. Daarnaast zijn de aantal klassen gereduceerd. Dit komt doordat de concrete strategies / commands zijn vervangen door functies.

Uit de resultaten die in de bijlage A en B zijn vermeld, is het volgende naar voren gekomen:

- Gemiddelde cyclomatische complexiteit(WMC) van v0.0 en v1.0 is van respectievelijk 7,00 en 10,09 omhoog gegaan.
- De koppelingsfactor(CF) van v0.0 en v1.0 is van respectievelijk 39,05% en 15,93% omlaag gegaan.
- De polymorfismeFactor(PF) van v0.0 en v1.0 is van respectievelijk 127,78% en 41,24% omlaag gegaan.
- Gemiddelde cyclomatische complexiteit(WMC) van v1.0 en v2.0 is van respectievelijk 10,09 en 7,11 omlaag gegaan.
- De koppelingsfactor(CF) van v1.0 en v2.0 is van respectievelijk 15,93% en 20,11% omhoog gegaan.
- De polymorfismeFactor(PF) van v1.0 en v2.0 is van respectievelijk 41,24% en 38,46% omhoog gegaan.

Echter, wat naar voren is gekomen tijdens het toetsen van de functionele implementatie, is dat de integratie van OO en FP-features artefacten introduceren, die niet door OO -metrieken noch door FP- metriken worden ondervangen [18]. Het probleem dat zich hier voordoet, is namelijk dat de OO-metrieken op klasse niveau fungeren en FP metrieken op functie niveau fungeren. Hierdoor is het niet mogelijk om een eerlijke vergelijking te maken tussen de verschillende softwareversies.

Wanneer we de functionele implementatie van de Strategy pattern benaderen, dan wordt de polymorfisme factor bepaald door de aantal functies die beschikbaar zijn gesteld als zijnde concrete strategies die in de vorm van een lambda expressie zijn uitgedrukt. De lambda expressie wordt binnen een default dan wel static method binnen de functionele interface gedefinieerd. Echter, is het niet mogelijk om de polymorfisme factor middels de OO-metriek te berekenen. Dit komt doordat de polymorfismefactor berekend wordt op basis van het aantal sub-klasse die een super-klasse heeft, aantal methodes die in de sub-klasse worden overschreven en het aantal nieuwe methodes die in de sub-klasse zijn geïntroduceerd.

7.1. CONSEQUENTIES

In Jabberpoint versie 2.0 zijn de volgende voordelen naar voren gekomen door de Strategie en Command pattern functioneel te implementeren:

Voordelen:

1. Factory pattern om de concrete strategies / commands te instantiëren is overbodig worden.
2. Aantal klassen nemen af doordat de concrete strategies / commands zijn vervangen door functies.
3. Strategies / commands kunnen niet meer verantwoordelijkheid krijgen dan gewenst is.

8

DISCUSSIE

In dit onderzoek is in eerste instantie geprobeerd om dezelfde methodiek toe te passen zoals in het artikel “A quantitative approach for evaluating the quality of design patterns“ [11]. Hierbij is de focus gelegd op de invloeden die design patterns hebben op de kwaliteit van een design. Door verschillende software versies te meten middels het QMOOD model, is het mogelijk om een uitspraak te doen welke implementatievariant nu beter scoort. Echter is tijdens het onderzoek naar voren gekomen dat het met de huidige OO- en FP-metrieken niet mogelijk is om het multi paradigma te toetsen. Dit komt doordat er artefacten door de paradigmata worden geïntroduceerd die in de literatuur nog niet zijn ondervangen. Doordat het met de huidige metrieken niet mogelijk is om het multiparadigma door te rekenen, is er voor gekozen om de implementatievarianten te toetsen middels het SOLID principe. Echter zijn dit intuïtieve waarneming geweest waarop een besluit is genomen. Een kwantitatieve onderbouwing blijft buiten beschouwing

Wanneer we naar de Strategie pattern kijken dan komt naar voren dat de OO implementatie de cohesie en polymorfisme verhoogd en de complexiteit verlaagt. Wanneer we naar de functionele implementatie kijken en het SOLID principe hanteren, dan worden de properties cohesie en polymorfisme compleet genegeerd. De vraag die hierbij gesteld kan worden, is of de design properties nog steeds stand houden in de functionele implementatie.De cyclomatische complexiteit daarentegen kan wel gesteld worden dat deze in stand blijft. De “conditional statements“ blijven bij de functionele implementatie nog altijd buiten beeld. Wanneer we naar de Design quality attributes flexibiliteit kijken, dan komt naar voren dat de flexibiliteit in beiden implementatie gelijk is gebleven.

Door een vervolgonderzoek op te zetten, kunnen de volgende punten worden aangehaald:

- Metrieken opzetten om OO-metrieken te synchroniseren met het multi paradigma.
- Op basis hiervan kan bepaald worden welke consequenties de functionele implementatie met zich mee brengen op metriekenniveau ten opzichte van de OO-implementatie.

9

CONCLUSIE

Design patterns zijn bewezen oplossingen voor verschillende knelpunten binnen het OO-paradigma. Het doel van design patterns is om de flexibiliteit, uitbreidbaarheid en onderhoudbaarheid van een softwaresysteem te optimaliseren in de situatie waarin de programmeertaal zelf onvoldoende mogelijkheden biedt. In dit onderzoek is gekeken op basis van welke criteria een keuze kan worden gemaakt naar welke implementatievariant de voorkeur gaat. Voor dit onderzoek is het technisch rapport “Eliminating OO Patterns by Java Functional Features“ als uitgangspunt genomen, waarin de Strategy, Command, Template method zijn uitgewerkt.

Doordat de gegeven structuur die door de GoF is vrijgegeven is aangepast, ontstaat de vraag of de alternatieve implementatie de oorspronkelijke kwaliteitseisen die het pattern beoogt te optimaliseren weet te vervullen. Daarnaast is het van belang om te weten of de functionele implementatie een verbetering is ten opzichte van de oorspronkelijke design pattern. Op basis hiervan is de volgende hoofdvraag geformuleerd:

> *"Wat kunnen de bijdragen van functionele aspecten van Java zijn voor de implementatie van Design patterns?"*

Elke pattern heeft een doel op zich dat opgedeeld is in 4 onderdelen, namelijk:

- **pattern name:** geeft een abstracte beschrijving van de functionaliteit die vervuld wordt door de betreffende design pattern.
- **intent:** beschrijft welke doelen met het pattern bereikt kunnen worden.
- **solution:** representeert een abstracte omschrijving van de oplossing die kan worden behaald met het pattern. Daarnaast worden de elementen en de samenhang beschreven.
- **consequences:** representeert de voor- en nadelen die worden behaald door het pattern in te zetten.

Door de intent te correleren met de design properties is het mogelijk om inzichtelijk te maken welke kwaliteitsaspecten in het design worden geïntroduceerd. Hierdoor is het mogelijk om in een later stadium te bepalen welke consequenties de functionele implementatie met zich mee brengt. Hiermee is het mogelijk om de juiste criteria op te stellen om antwoord te geven op de hoofdvraag. Daarnaast is het van belang om kritisch naar de intent van het pattern te kijken. De intent van het pattern beschrijft het doel waarvoor het ingezet kan worden en welk probleem daarmee wordt opgelost binnen het OO-paradigma. De intent van het pattern dient derhalve in de functionele implementatie stand te houden om de oorspronkelijke pattern tegemoet te komen.

In het onderzoek is naar voren gekomen dat de functionele implementatie “strategy“ en de “command pattern“ aan de oorspronkelijke intent voldoen. Bij de command pattern dient men rekening te houden dat de *“intent undoable“* operations niet in de functionele implementatie worden ondersteund. Dit probleem kan worden opgelost door het pattern te combineren met het memento pattern waarmee de verantwoordelijkheid elders wordt geplaatst. Daarnaast is naar voren gekomen indien men de functionele implementatievariant van de strategy en command pattern toepast, de implementatie preciezer aan de intent van het pattern voldoet. De functionele implementatie laat het namelijk niet toe om data in de concrete strategie op te slaan. Daarnaast komt de “factory pattern“ te vervallen waarmee de concrete “strategies“ en “commands“ bij de OO-implementatie worden gecreëerd.

Waneer we de functionele implementatie tegen het SOLID principe aanhouden, dan komt de functionele implementatie van zowel de Strategy als de Command pattern beter naar voren dan de OO implementatie. Dit komt doordat het “Single responsibility principle“ strikter wordt nageleefd.

De functionele implementatievariant van de Template method pattern voldoet niet aan de oorspronkelijke intent. Dit komt doordat de structuur van het algoritme pas bepaald wordt tijdens het creëren van de betreffende “TemplateClass“. Op basis hiervan is besloten om de functionele implementatievariant niet verder in het onderzoek mee te nemen. Het heeft namelijk geen toegevoegde waarde om deze op de verschillende fronten te toetsen indien het geen valide oplossing biedt.

De enumeration variant is een vreemde eend in de bijt. Een enum is in het leven geroepen om constanten te definiëren. Indien het voor andere doeleinden wordt gebruikt, ontstaat de kans dat dit ten kosten gaan van de leesbaarheid [3].

Om een kwantitatieve uitspraak te kunnen doen naar welke implementatievariant de voorkeur uitgaat, is het van belang om de consequenties van beiden varianten met elkaar te vergelijken. Echter is naar voren gekomen dat het met de huidige OO-metrieken niet mogelijk is om de functionele implementatievarianten te toetsen. De cohesie, complexiteit en de polymorfisme factor is niet te correleren met functionele metrieken om een evenwichtig beeld te geschetst.

Om de hoofdvraag kwantitatief te kunnen beantwoorden, is het noodzakelijk om een vervolgstudie op te zetten waarin multiparadima-metrieken worden ontwikkeld. De metrieken moet het mogelijk maken om deze in te kunnen zetten binnen het multiparadigma die betrekking hebben op programmeertaal Java.

BIBLIOGRAFIE

[1] Bijlsma A, Kok A, Passier H, Pootjes H, and Stuurman S. Eliminating OO Patterns by Java Functional Features. Technical report, Open Universiteit Nederland, 02 2019. 3, 21, 24, 27, 28, 30

[2] Khalid Aljasser. Implementing design patterns as parametric aspects using paraaj: The case of the singleton, observer, and decorator design patterns. *Computer Languages, Systems and Structures*, 45:1 – 15, 2016. 4

[3] Jeanne Boyarsky and Scott Selikoff. *OCP Oracle Certified Professional Java SE 8 Programmer 2*. John Wiley and Sons, Inc., Indianapolis, Indiana, 2016. 22, 44

[4] Gamma Erich, Helm Richard, Johnson Ralph, and Vlissides John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995. 8, 10, 11, 14, 15, 16, 17, 38

[5] Khomh F and Guéhéneuc Y. Design patterns impact on software quality: Where are the theories? pages 15–25, 2018. 8

[6] Julian Fleischer. Bridging the gap between haskell and java. Master's thesis, Universiät Berlin, 6 2013.

[7] Daniel Gross and Eric Yu. From non-functional requirements to design through patterns. *Requirements Engineering*, 6(1):18–36, Feb 2001. ISSN 1432-010X. 9

[8] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37:161–173, 2002. 4

[9] Imène Issaoui, Nadia Bouassida, and Hanêne Ben-Abdallah. Using metric-based filtering to improve design pattern detection approaches. *Innovations in Systems and Software Engineering*, 11(1):39–53, Mar 2015. ISSN 1614-5054. 9, 15, 16, 17

[10] Erik Landkroon. Code quality evaluation for the multi-paradigm programming language scala. Master's thesis, Universiteit van Amsterdam, 8 2017. 10

[11] Hsueh Nien-Lin, Chu Peng-Hua, and Chu William. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81:1430 – 1439, 2008. 5, 15, 41

[12] Abbas Rasoolzadegan. A new approach to the quantitative measurement of software flexibility. *Journal of Soft Computing and Information Technology*, 5(1), 2016. 15

[13] Hussain S, Keung J, and Khan A. The effect of gang-of-four design patterns usage on design quality attributes. pages 263–273, 2017. iii, iv, 7, 11

[14] Claudio Sant'Anna, Alessandro Garcia, Uirá Kulesza, Carlos Lucena, and Arndt Staa. Design patterns as aspects: A quantitative assessment. *Journal of the Brazilian Computer Society*, 10:49–63, 2004. 4

[15] Alexander Shvets. *Dive Into DESIGN PATTERNS*. Refactoring.Guru, 2019. 16, 18

[16] Harmeet Singh and Syed Imtiyaz Hassan. Effect of solid design principles on quality of software : An empirical assessment. 2015. 18

[17] R Warburton. *Java 8 Lambdas*. O'Reilly, 2014. 1, 7

[18] Bart Zuilhof, Rinse van Hees, and Clemens Grelck. Code quality metrics for the functional side of the object-oriented language c sharp. *CEUR-WS*, 2510, 2019. 10, 40

BIJLAGE A

.1. METRIEKEN JABBERPOINT

Project	AHF	AIF	CF	MHF	MIF	PF
Jabberpoint uitgangspunt	85,53%	76,76%	39,05%	5,00%	11,11%	127,78%
Jabberpoint IM0102 Design patterns	94,28%	69,58%	15,93%	6,23%	40,59%	41,24%
Jabberpoint Functionel Design patterns	81,14%	70,17%	20,11%	6,12%	40,72%	38,46%

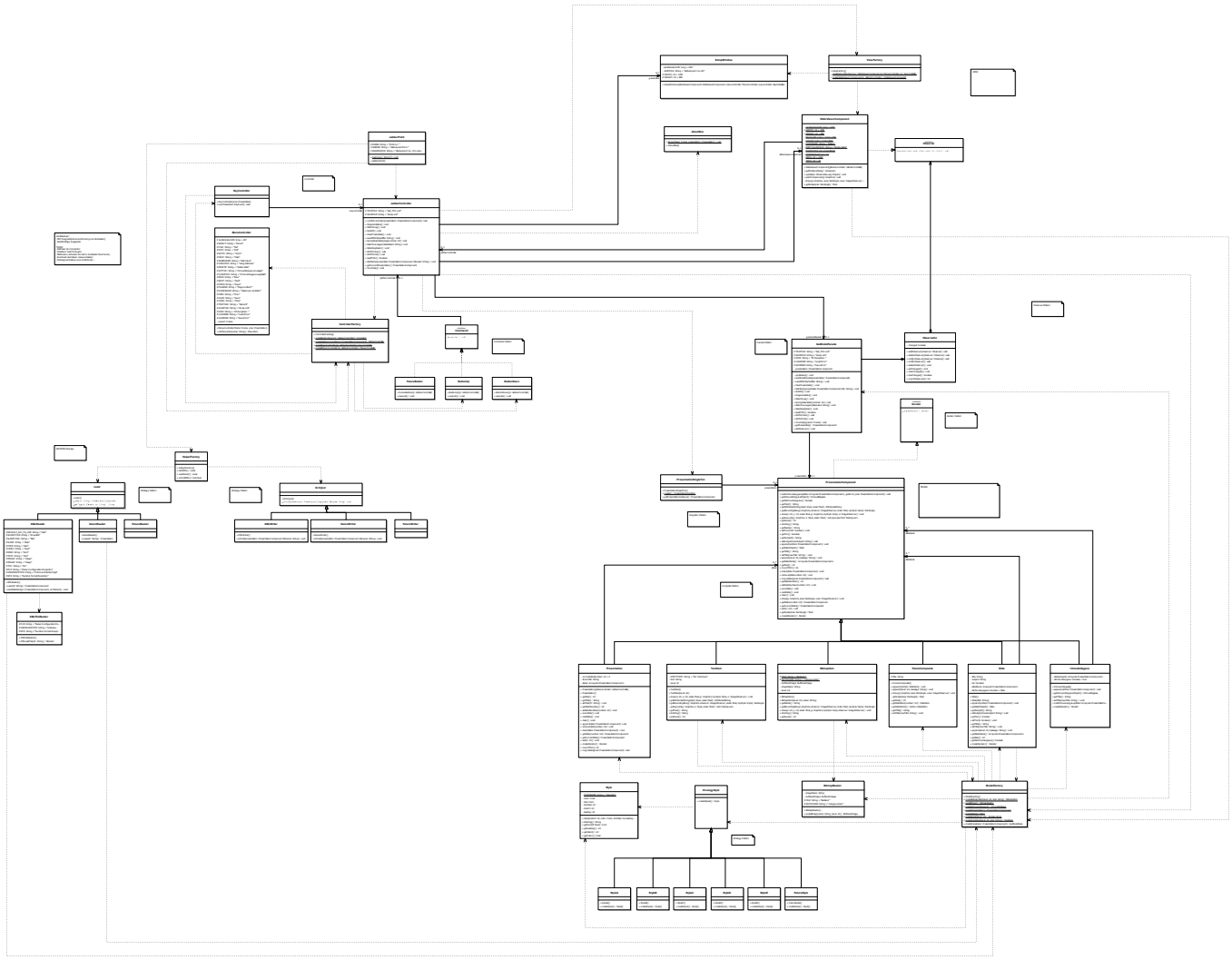
Project Jabberpoint uitgangspunt						
Class	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
AboutBox	1	1	1	0	2	1
Accessor	5	1	3	2	4	2
BitmapItem	4	2	2	0	15	6
DemoPresentation	4	2	2	0	10	2
JabberPoint	6	1	1	0	9	2
KeyController	2	2	1	0	6	6
MenuController	5	3	1	0	11	10
Presentation	9	1	3	0	21	19
Slide	7	1	1	0	20	11
SlideItem	5	1	3	2	5	3
SlideViewerComponent	3	4	2	0	21	6
SlideViewerFrame	5	6	1	0	17	3
Style	6	1	2	0	7	6
TextItem	4	2	1	0	34	15
XMLAccessor	8	2	2	0	42	13

Project IM0102 Design Patterns						
Class	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
Controller.ButtonDOWN	3	1	1	0	3	2
Controller.ButtonUP	3	1	1	0	3	2
Controller.Command	n/a	n/a	n/a	n/a	1	n/a
Controller.JabberController	13	1	1	0	37	16
Controller.JabberPoint	8	1	1	0	10	2
Controller.KeyController	5	2	1	0	7	6
Controller.MenuController	4	3	1	0	11	14
Factories.ControllerFactory	8	1	5	0	10	5
Factories.HelperFactory	7	1	3	0	3	3
Factories.ModelFactory	20	1	9	0	16	14
Factories.PresentationSingleton	3	1	1	0	3	3
Factories.ViewFactory	5	1	2	0	4	2
Helpers.BitmapReader	2	1	1	0	4	1
Helpers.DemoReader	4	2	1	0	8	1
Helpers.DemoWriter	2	2	1	0	1	1
Helpers.Lezer	6	1	2	2	5	1
Helpers.ReadStrategy	n/a	n/a	n/a	n/a	11	n/a
Helpers.ReaderStrategyDataHider	3	1	1	0	8	1
Helpers.Schrijver	5	1	1	2	1	0
Helpers.SlideIterator	n/a	n/a	n/a	n/a	1	n/a
Helpers.WriteStrategy	n/a	n/a	n/a	n/a	0	n/a
Helpers.XMLFileReader	1	1	1	0	9	1
Helpers.XMLReader	5	2	1	0	26	10
Helpers.XMLWriter	5	2	1	0	18	6
Model.BitmapItem	5	2	3	0	15	7
Model.GuiModelFacade	7	2	2	0	47	30
Model.InhoudsOpgave	2	2	4	0	19	13
Model.Presentation	3	2	2	0	38	31
Model.PresentationComponent	22	1	36	5	38	37
Model.Slide	2	2	6	0	20	15
Model.StrategyStyle	8	1	1	5	1	0
Model.Style	10	1	4	0	8	6
Model.StyleA	3	2	1	0	2	1
Model.StyleB	3	2	1	0	2	1
Model.StyleC	3	2	1	0	2	1
Model.StyleD	3	2	1	0	2	1
Model.StyleE	3	2	1	0	2	1
Model.TextItem	5	2	2	0	36	16
View.AboutBox	1	1	1	0	2	1
View.SetupWindow	5	6	0	0	11	2
View.SlideViewerComponent	9	4	2	0	35	9

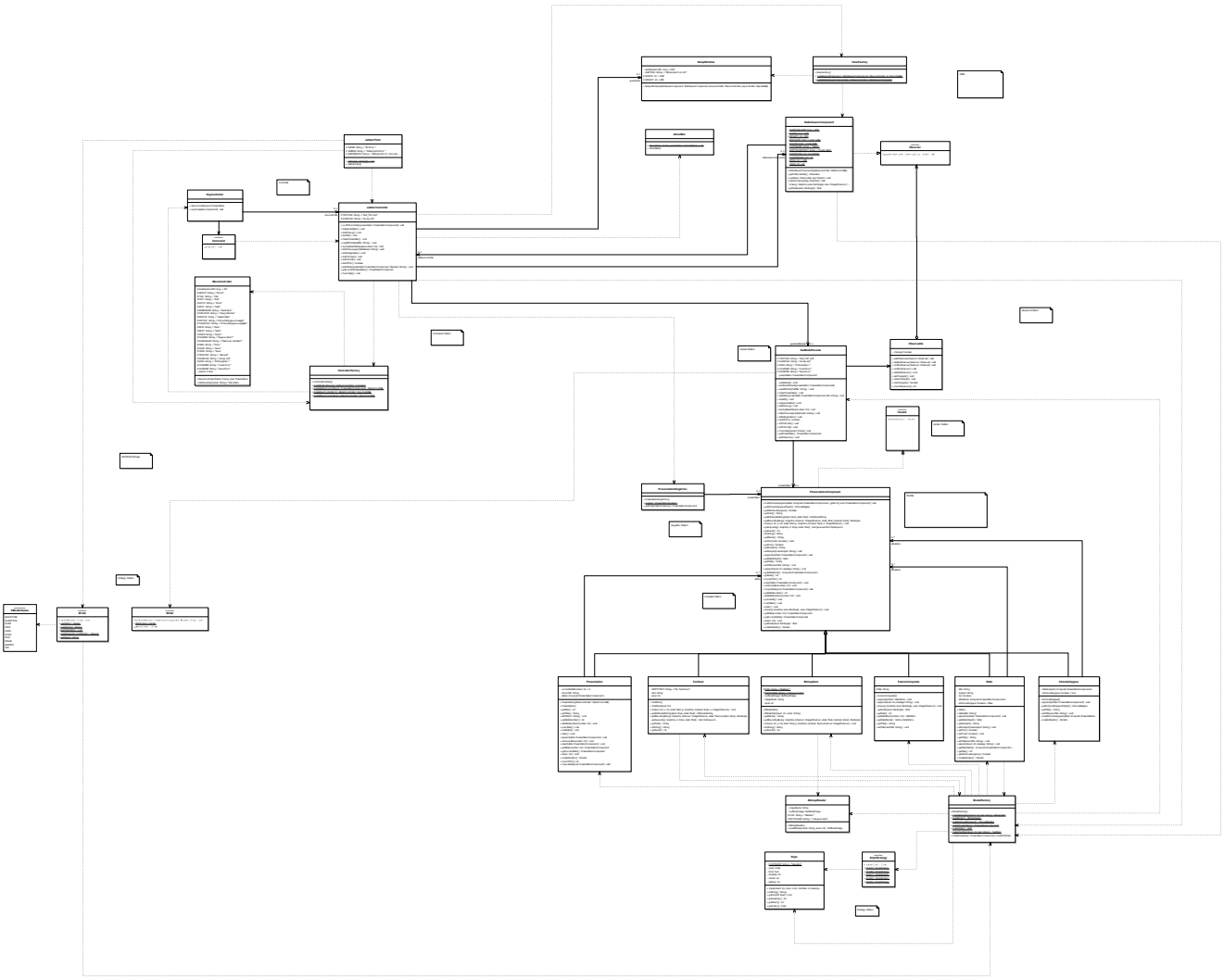
Project Jabberpoint Functional Design patterns						
Class	**CBO**	**DIT**	**LCOM**	**NOC**	**RFC**	**WMC**
Controller.Command	n/a	n/a	n/a	n/a	1	n/a
Controller.JabberController	11	1	1	0	37	16
Controller.JabberPoint	6	1	1	0	8	2
Controller.KeyController	5	2	1	0	6	7
Controller.MenuController	4	3	1	0	11	14
Factories.ControllerFactory	5	1	3	0	6	3
Factories.ModelFactory	12	1	9	0	21	14
Factories.PresentationSingleton	3	1	1	0	3	3
Factories.ViewFactory	5	1	2	0	4	2
Helpers.BitmapReader	2	1	1	0	4	1
Helpers.DemoWriter	2	2	1	0	1	1
Helpers.Reader	n/a	n/a	n/a	n/a	37	n/a
Helpers.Schrijver	2	1	1	1	1	0
Helpers.SlideIterator	n/a	n/a	n/a	n/a	1	n/a
Helpers.Writer	n/a	n/a	n/a	n/a	20	n/a
Helpers.XMLAttributes	1	n/a	1	n/a	2	2
Model.BitmapItem	5	2	3	0	15	7
Model.GuiModelFacade	6	2	2	0	47	30
Model.InhoudsOpgave	2	2	4	0	19	13
Model.Presentation	3	2	2	0	38	31
Model.PresentationComponent	18	1	36	5	38	37
Model.Slide	2	2	6	0	20	15
Model.Style	5	1	4	0	8	6
Model.StyleStrategy	n/a	n/a	n/a	n/a	7	n/a
Model.TextItem	5	2	2	0	36	16
View.AboutBox	1	1	1	0	2	1
View.SetupWindow	5	6	0	0	11	2

BIJLAGE B

.2. JABBERPOINT IM0102 DESIGN PATTERNS



.3. JABBERPOINT FUNCTIONELE IMPLEMENTATIE



BIJLAGE C

.4. SOURCE CODE

Jabberpoint IM0102 Design pattertns
https://github.com/HichamOUNL/EvertVerduin-HichamChafik

Jabberpoint functionele implementatie
https://github.com/HichamOUNL/Replacing-design-patterns