Software Ontwerp en Architectuur 3

6/23/2019

Patrick van Batenburg (2125605)

Inhoudsopgave

[1. Design Patterns 2](#_Toc12206682)

[1.1. Abstract Factory 2](#_Toc12206683)

[1.2. Composite pattern 3](#_Toc12206684)

[1.3. Observer pattern 4](#_Toc12206685)

[1.4. State pattern 5](#_Toc12206686)

[1.5. Strategy pattern 7](#_Toc12206687)

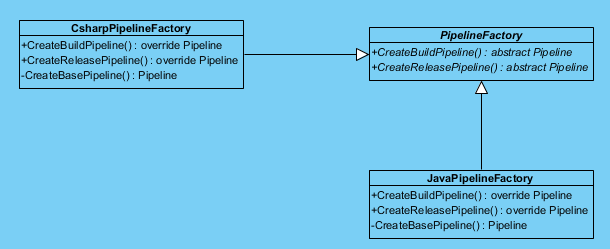
# Design Patterns

Dit project bevat in totaal 6 design patterns. Deze bestaat uit 4 behavioral patterns en 1 structural pattern. Daarnaast is er 1 creational pattern aanweizg. Sommige patterns zijn op 2 verschillende plaatsen toegepast. In dit hoofdstuk zal het gebruiken van een specifieke pattern onderbouwd en argumenteerd worden.

## Abstract Factory

Een Abstract Factory is een Creational Pattern. Het zorgt voor een interface voor het maken van een verzameling van gerelateerde of afhankelijke objecten zonder hun concrete klassen te specificeren.

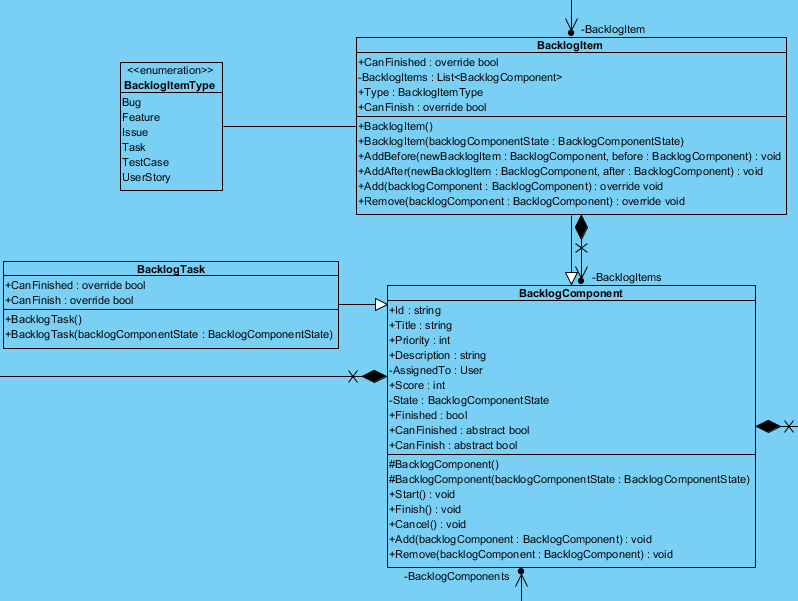
Er is hiervoor gekozen, omdat je bij Microsoft DevOPs ook kan kiezen of dat dit een ASP.NET build pipeline of een .NET Core framework build pipeline is. Een non-functional dat hierbij komt kijken gaat over de compatibility, maintainability en replaceability. Want dat gaat over de mate waarin een systeem of component met andere systemen of componenten de vereiste functies ervan kan uitvoeren, terwijl deze dezelfde hardware- of softwareomgeving deelt. De volgende non-functional gaat over de maintainability. Zoals te zien is in het diagram heeft geen associatie met andere klassen in het systeem zoals de Project klasse. Dit betekent dat het juist erg modulair is. Als ik toevallig iets wil gaan toevoegen aan de C# pipeline factory of nieuwe pipelines wilt gaan toevoegen dan heeft dit geen impact op andere componenten in het systeem. Ik zou zelfs concrete factories kunnen vervangen met een compleet andere factory zonder enige impact. Vice versa is ook hetzelfde aan de hand, bijvoorbeeld als ik de Project klasse aanpast. Deze verwacht toch een PipelineFactory in een methode om een build of release pipeline snel op te zetten. De implementatie maakt dus niet uit zo long het maar een variatie van een PipelineFactory is.



## Composite Pattern

Een composite pattern is een structural Pattern. Het stelt objecten in boomstructuren samen om gedeeltelijke en/of hele hiërarchieën te weergeven. Daarnaast bied het de mogelijkheid oom individuele objecten en composities van objecten uniform te behandelen.

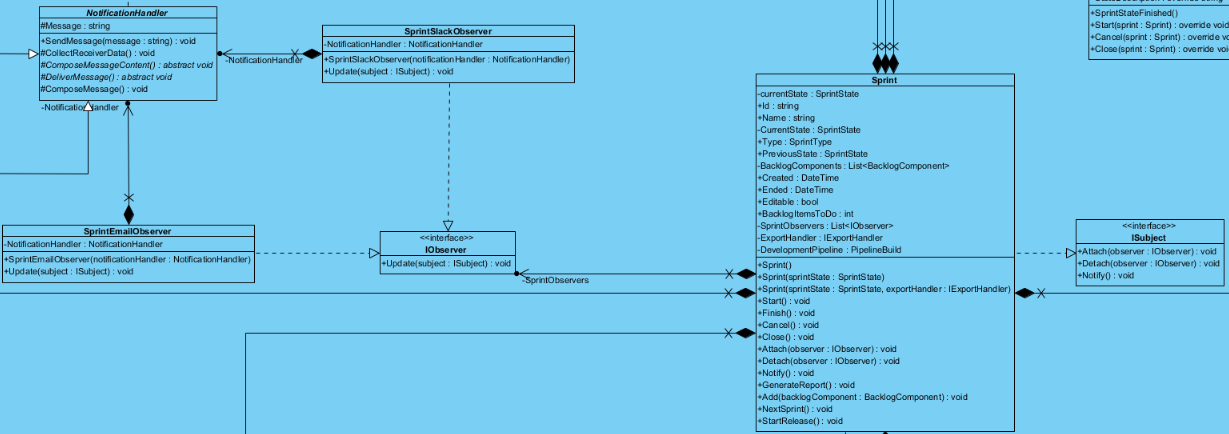
In de beschreven opdracht staat vermeld dat een sprint backlog items bevat en dat deze backlog items weer meerdere backlog items kunnen bevatten. Volgens deze beschrijving past het composite pattern hier het beste bij. Met betrekking tot de opdracht voeren BacklogTask-objecten een verzoek rechtstreeks uit en BacklogItem-objecten sturen het verzoek door naar hun onderliggende componenten, recursief naar beneden in de boomstructuur. Dit maakt klassen eenvoudiger uit te voeren, te wijzigen, te testen en opnieuw te gebruiken. Dit betekent dat de non-functional maintainability hier sterk terug in voor komt als zo wel voor usability.



## Observer Pattern

De observer pattern is een pattern dat valt onder behavioral patterns. Het definieert een een-op-veel afhankelijkheid tussen objecten, zodat wanneer een object van staat verandert, al zijn afhankelijken automatisch worden genotificeerd en bijgewerkt. Deze pattern biedt een Subject en een Observer interface aan. De enige verantwoordelijkheid van een Subject-object is om een lijst met Observer-objecten bij te houden en hen op de hoogte te brengen van wijzigingen in de status door hun updatemethode aan te roepen. De verantwoordelijkheid van Observer-objecten is om zich te registreren (en af te melden) over een Subject-object. Om op die manier hun status bij te werken wanneer ze op de hoogte worden gesteld. Dit maakt Subject-objecten en Observer-objecten lously coupled. Een Subject-object en Observer-object hebben geen expliciete kennis van elkaar. Dit betekent dat het heel herbruikbaar is en heeft dus betrekking tot de non-functional Reusability en ook nog Compatibility.

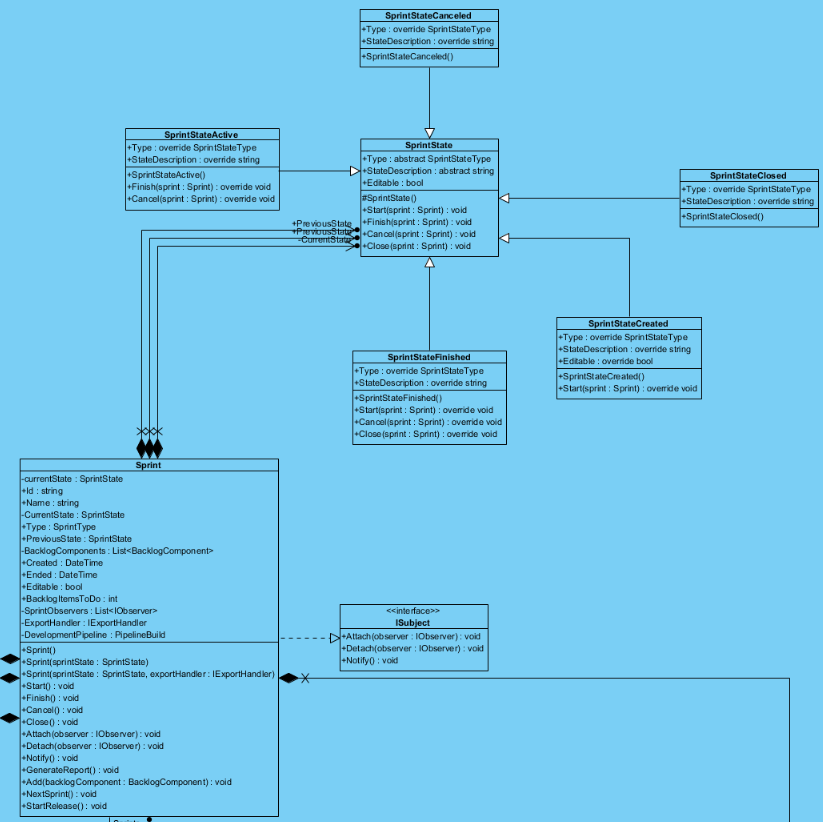
Er is voor gekozen dit patroon toe te passen op de Sprint klasse die als een subject dient. Daarnaast zijn er de SprintSlackObserver en SprintEmailObserver klassen die als de observers dienen. Dit komt doordat er in de opdracht stond dat de sprint een notificatie via email of slack kon doen als gegeven voorbeelden als de sprint bijvoorbeeld de status “afgerond” of “geannuleerd” kreeg.



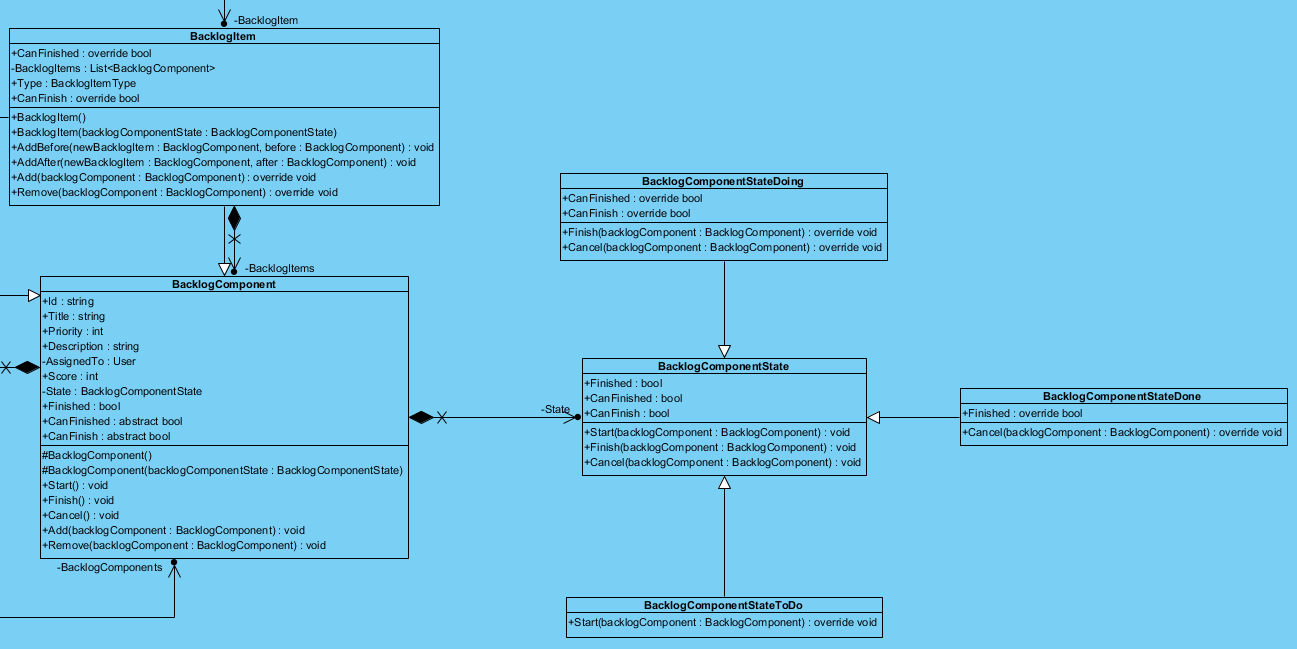
## State Pattern

Het state pattern is een behavioural pattern waarmee een object zijn gedrag kan wijzigen wanneer de interne status verandert. Het state pattern wordt gebruikt om verschillend gedrag voor hetzelfde object te encapsulate op basis van de interne status. Dit is een schonere manier voor een object om zijn runtime-gedrag te veranderen zonder conditional statements te gebruik en heeft dus betrekking tot de non-functional maintainability.

In de opdrachtbeschrijving staat dat een sprint zich in verschillende stadia kan bevinden. Daarnaast is er beschreven dat het aanpassen van een backlog item niet meer mogelijk kan zijn wanneer de sprint in een bepaalde staat is. Door conditional statements te gebruiken zou het onderhouden hiervan erg moeilijk gaan worden indien er meerdere states bij komen. Het is dus vanwege deze reden dat er voor een state pattern gekozen is



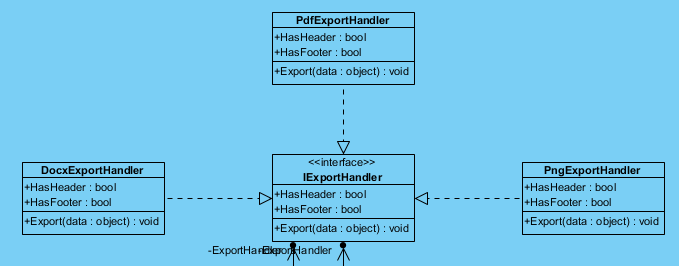
Een ander onderdeel waar de state pattern nogmaals is toegepast is bij de backlog items. Uit de opdracht was eruit te leiden dat een backlog item zich ook in verschillende stadia kan bevinden zoals Todo, Doing en Done. Om deze reden is er gekozen voor het state pattern. Dit is doordat er weer dezelfde argumenten naar voren komen zoals bij de sprint. Dat is het gebruiken van conditional statements wat het dus alleen lastiger maakt om het in de toekomst te gaan onderhouden.



## Strategy Pattern

Het Strategy pattern is een behavioral pattern dat een familie van algoritmen bevat en er een uit de pool selecteert voor gebruik tijdens runtime. De algoritmen zijn onderling uitwisselbaar, wat betekent dat ze onderling substitueerbaar zijn. Het belangrijkste idee is om objecten te maken die verschillende strategieën vertegenwoordigen. Deze objecten vormen een pool van strategieën waaruit het te gebruiken klasse kan kiezen om zijn gedrag te variëren volgens de strategie. Deze strategieën voeren dezelfde bewerking uit, hebben dezelfde taak en bieden dezelfde interface aan.

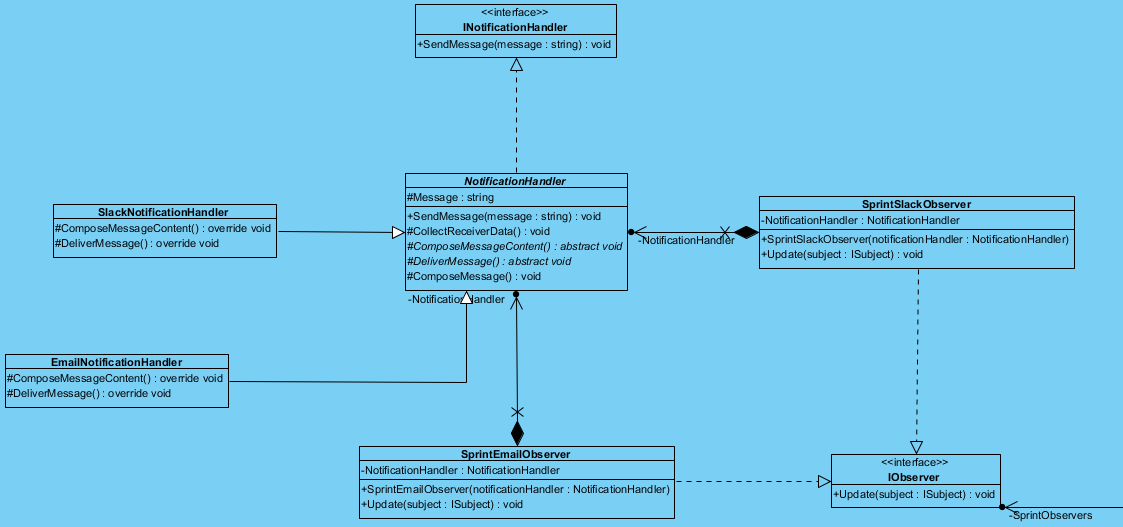
Volgens de casus moet het mogelijk zijn om een rapportage te kunnen genereren. Er was aangegeven dat er verschillende manieren zijn om rapportages te genereren. Bij het exporteren van de data naar wat voor soort formaat dan ook is dit design pattern toegepast. Het bied de mogelijkheid om het gedrag van de export te kunnen veranderen bij runtime wat het dus makkelijk maakt om van formaat naar formaat te kunnen gaan. Dit betekent dus dat de non-functional maintainability hierbij komt kijken. Er kan met gemak dingen gewijzigd worden in bestaande exportformaten of nog zelfs meerdere toegevoegd worden.



## Template Method Pattern

De Template Method Pattern valt onder een van de behavioral patterns. Het definieert het skelet van een algoritme in een methode, waarbij sommige stappen worden overgedragen naar subklassen. De Template Method laat subklassen bepaalde stappen van een algoritme herdefiniëren zonder de structuur van het algoritme te veranderen. De base class declareert dus de placeholders voor de algoritmen en de afgeleide klassen implementeren deze placeholders. Dit patroon is een voorbeeld van inversion of control omdat de code op hoog niveau niet langer bepaalt welke algoritmen moeten worden uitgevoerd. Een algoritme op lager niveau wordt in plaats daarvan geselecteerd tijdens runtime net zoals dat kan bij de strategy pattern.

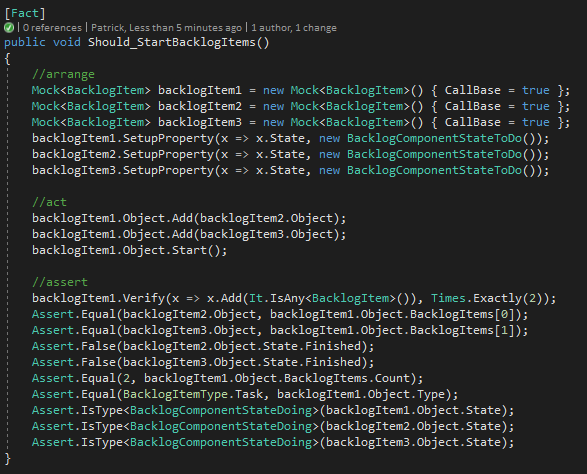
Dit design pattern is toegepast bij het notificeren van berichten. In de opdrachtbeschrijving waren er enkele voorbeelden geven om dit via email of slack te kunnen doen. De achterliggende gedachte waarom dit hier is toegepast is vanwege het gebruiken van dezelfde berichten. Het opstellen van een bericht is dus hetzelfde maar hoe deze verzonden wordt kan anders zijn.



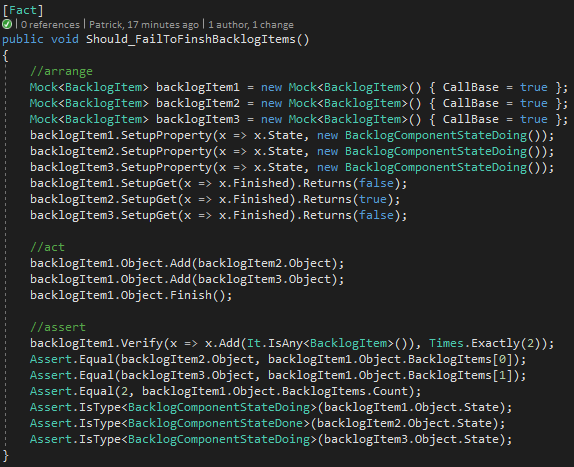
# SonarQube, testen code coverage

In het project zijn er in totaal 73 unit testen geschreven. Deze zijn geschreven met mock objecten om op die manier interfaces en abstracte klassen makkelijker te testen. Het injecteren van state is bijvoorbeeld op deze manier ook mogelijk door bijvoorbeeld de SetupProperty op het mock object aan te roepen.

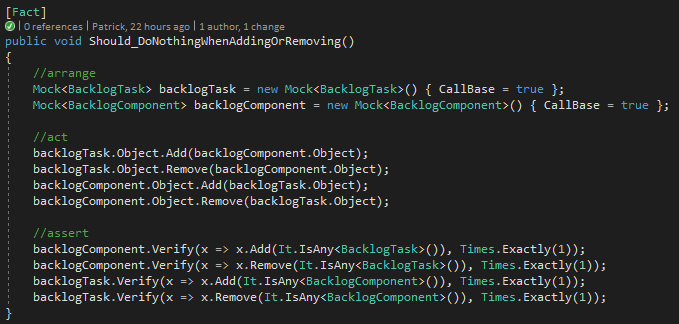
Heet veel condities zijn getest. Voorbeelden zijn het starten van een backlog item en zijn child objecten. Het andere is het annuleren van een backlog item en zijn child objecten.



Vervolgens is zijn er testen die kijken of dat iets faalt. Denk maar aan dat een blak log item alleen done kan zijn als al zijn child objecten dat ook zijn. Zoals bij deze test te zien is, zie je dat back log item 1 na deze test nog steeds op doing staat. Dit komt omdat een child object nog op doing staat en is dus nog niet af, terwijl een andere child object dat wel is en dus wel correct op done komt te staan.



Vervolgens zijn niet geïmplementeerde methoden ook getest door de Verify methode op de methode uit te oefenen. Hier is te zien dat de abracte klasse BacklogCompent zijn methode Remove en Add uitgevoerd heeft. Hetzelfde geld voor BacklogTask dat een concrete implementatie is, maar beide hebben geen lijst die bijhoudt of dat deze items toegevoegd zijn.



In deze afbeelding is te zien dat er inderdaad 73 unit testen zijn, bestaan uit faal en succes scenario’s. Daarnaast is te zien dan er 100% coverage behaald is.

