



KU LEUVEN

SWOP: Ziekenhuissysteem Iteratie 3 en 4

Groep 10:

Olivier DUBOIS
Francis DUVIVIER
Thijs LOWETTE
Vincent VANKRUNKELSVEN

Begeleider:

Frédéric VOGELS

Docent:

Tom HOLVOET

21 augustus 2012

Inhoudsopgave

1	Introductie	2
2	Aangepast domeinmodel	3
2.1	Domeinmodel	3
2.2	Terminologie	4
3	Systeemfunctionaliteit	4
4	Subsystemen	7
5	Ontwerpbeslissingen	10
5.1	Algemene architectuur	10
5.2	Toegepaste ontwerppatronen	15
5.2.1	Tijdsverloop	15
5.2.2	Stockbeheer	16
5.2.3	Planning	19
5.2.4	Undo/redo-systeem	25
5.3	Toegepaste GRASP-principes	29
6	Uitbreidingen	38
6.1	Uitbreiding van overervingshiërarchie	38
6.2	Uitbreiding van de opdracht	40
7	Iteratie 4	40
7.1	Doorgevoerde aanpassingen	40
7.2	Uitbreiding ‘Besmettingsgevaar’	42
8	Tests	48
8.1	Algemene teststrategie	48
8.2	Scenario	49
8.3	Eclemma testrapport	49
9	Kritische zelfreflectie	52
10	Projectbeheer	52
11	Conclusie	52
A	Bijlagen	53

1 Introductie

Dit rapport kadert binnen het vak ‘Software-ontwerp’ waar we de opdracht kregen om in drie iteraties een ziekenhuissysteem te ontwerpen en te implementeren. Hierbij dienden we gebruik te maken van de *best practices* van objectgeoriënteerd ontwerpen, meer bepaald van de GRASP-principes, van de GoF-ontwerppatronen en van Martin Fowlers refactoring richtlijnen.

Het rapport begint met het domeinmodel dat we in de derde iteratie zelf moesten uitbreiden. Vervolgens geven we met behulp van UML een overzicht van het gehele systeem en zijn verschillende subsystemen. Ook wordt een eerste beeld gegeven van enkele systeemfunctionaliteiten.

Het software-ontwerp staat centraal in deze opdracht. Daarom overlopen we de algemene software-architectuur en bespreken we in detail de toegepaste GRASP-principes en de gebruikte ontwerppatronen. Ons ontwerp moet robuust, makkelijk aanpasbaar en uitbreidbaar zijn. We stellen enkele mogelijke uitbreidingen voor en bekijken wat er in het programma aangepast en toegevoegd moet worden. Dit verslag is bovendien uitgebreid met de **besprekking van de vierde iteratie**, waar een campus in een lockdown kan gaan. De aanpak wordt besproken als het derde puntje van deze sectie.

Na afloop van de derde iteratie ontvingen we feedback die verbeterpunten van het systeem suggereerde. De belangrijkste aanpassingen worden vervolgens beschreven.

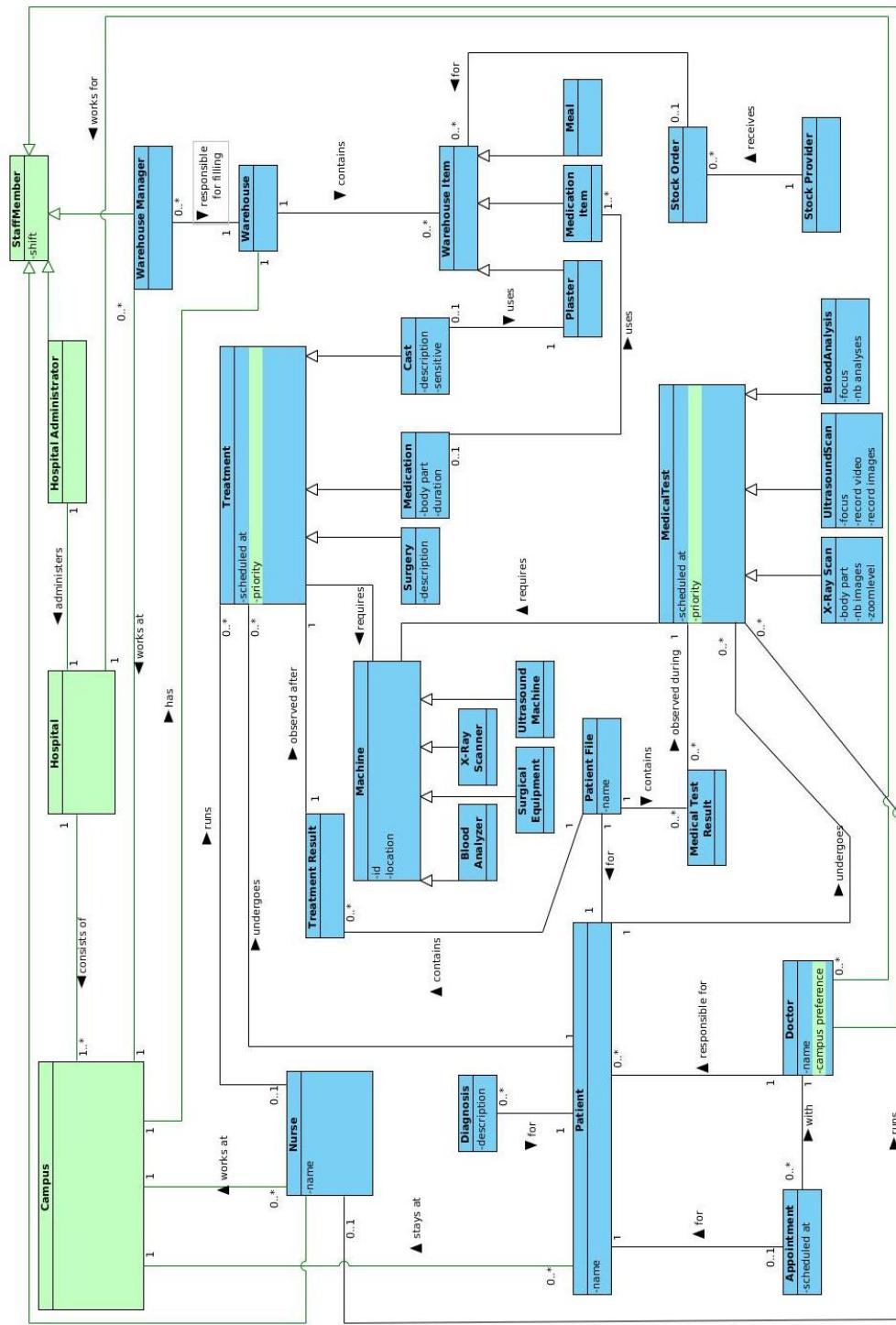
Een programma vereist een correcte werking. De juistheid van de code kan worden nagegaan met allerhande soorten tests. We verklaren onze testaanpak en bespreken ook de *code coverage* volgens de tool ‘Eclemma’. Omdat er altijd plaats is voor verbetering volgt een kritische kwaliteitsanalyse waarbij we de plus- en minpunten van de huidige code bespreken.

We eindigen het rapport met de taakverdeling binnen onze groep en een conclusie met de ervaren problemen en de getrokken lessen.

Voor we verder gaan, zouden we graag onze docent, Professor Holvoet, en onze begeleider, Dhr. Vogels, willen bedanken voor hun inzicht en advies dat ze ons tijdens dit leerrijk project hebben gegeven.

2 Aangepast domeinmodel

2.1 Domeinmodel



2.2 Terminologie

Campus

Het ziekenhuis bestaat uit 2 campussen. Elke campus heeft zijn eigen materiaal, magazijn, verpleegsters en magazijnbeheerders.

Doctor

Een dokter werkt voor het ziekenhuis en kan een voorkeur opgeven om ofwel de hele dag op eenzelfde campus te werken, ofwel gedurende de eerste 4 werkuren op de ene campus en gedurende de laatste 4 werkuren op de andere campus zijn job uit te oefenen, ofwel zich zoveel als nodig tussen de 2 campussen te verplaatsen met een maximum van 4 verplaatsingen per werkdag. Een dokter kent een prioriteit toe aan medische tests en behandelingen op basis van zijn diagnose van de betrokken patiënt.

Hospital

Het ziekenhuis waarvoor het softwaresysteem ontwikkeld wordt en waarbinnen de campussen hun informatie delen.

Hospital Administrator

Het ziekenhuis wordt georganiseerd door één enkele ziekenhuisadministrator die verantwoordelijk is voor het beheren van personeel en materiaal.

Medical Test

Een medische test heeft een prioriteit die gespecificeerd is door de dokter die de diagnose van de betrokken patiënt heeft gesteld, en wordt op basis daarvan vroeger of later gepland.

Patient

Een patiënt wordt geregistreerd bij één enkele campus maar kan op elke campus behandeld worden.

Staff Member

Een personeelslid werkt in het ziekenhuis gedurende welbepaalde werkuren en is gekend door zijn/haar naam. Er bestaan verschillende soorten ziekenhuispersoneel. Deze zijn op dit moment: *Doctor, Hospital Administrator, Nurse en Warehouse Manager*.

Treatment

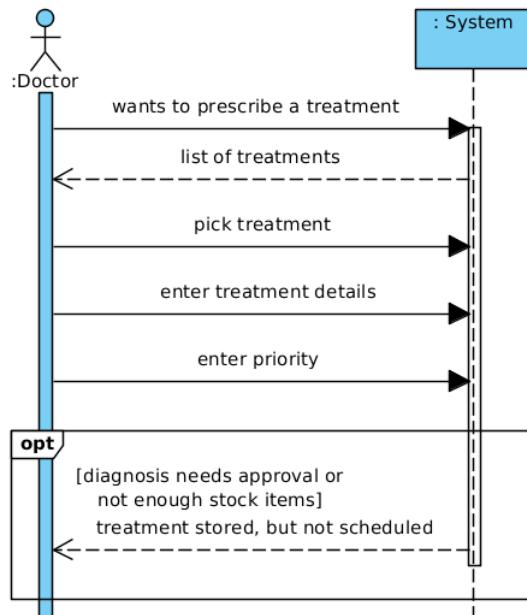
Een behandeling heeft een prioriteit die gespecificeerd is door de dokter die de diagnose van de betrokken patiënt heeft gesteld, en wordt op basis daarvan vroeger of later gepland.

3 Systeemfunctionaliteit

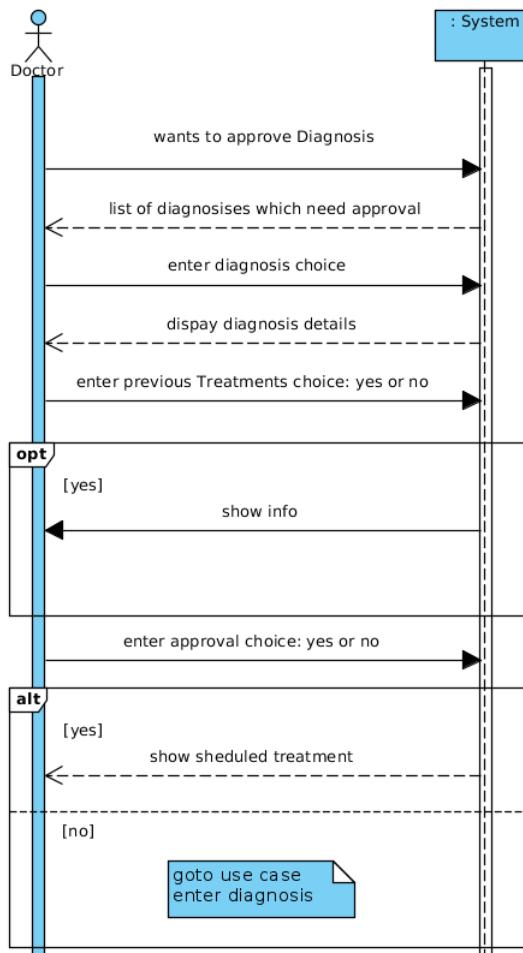
Het systeem stelt gebruikers in staat vele operaties¹ te verrichten: het plannen van afspraken tussen dokters en patiënten, het invoeren van medische testresultaten, het beheren van stock, het registreren van nieuwe dokters en verpleegsters, enzovoort.

Figuren 1, 2, 3 en 4 tonen de systeemsequentiediagrammen van deze vier belangrijke use cases.

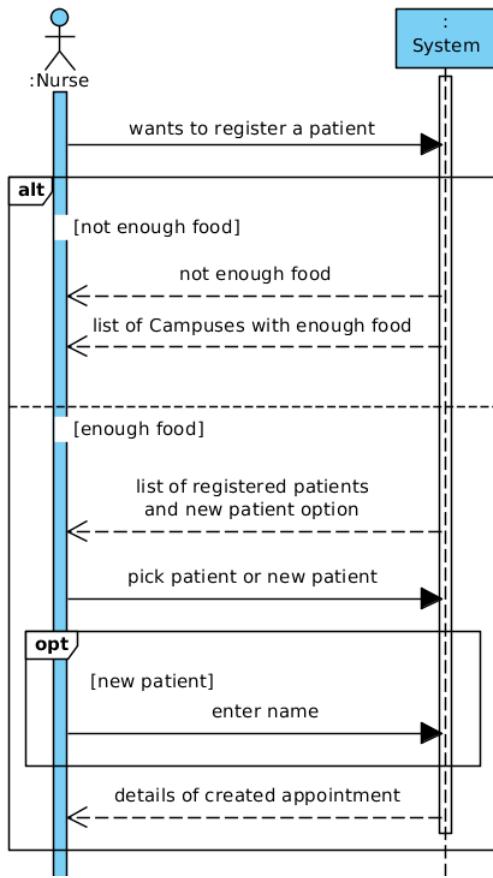
¹Voor de volledige lijst van operaties wordt verwezen naar de opgave.



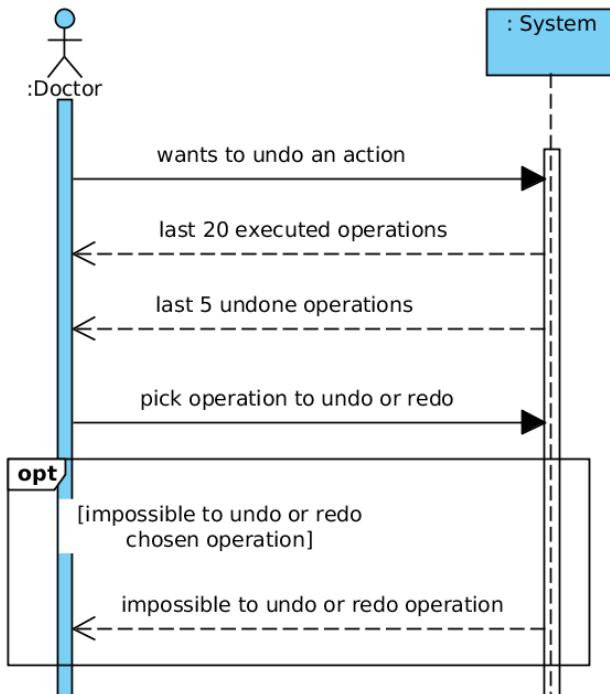
Figuur 1: System Sequence Diagram: *Prescribe Treatment*



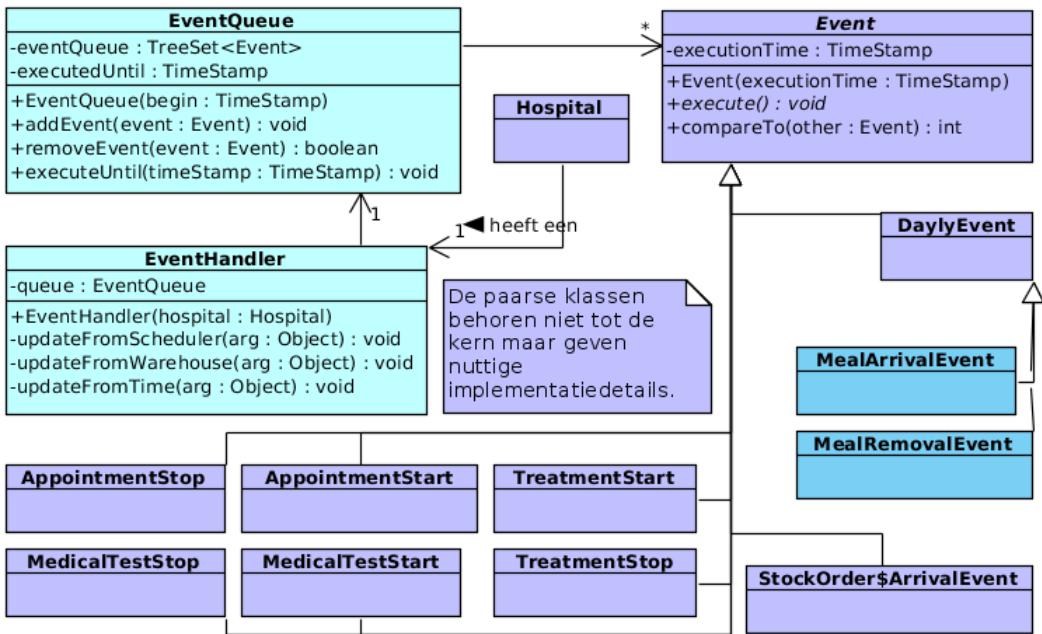
Figuur 2: System Sequence Diagram: *Approve Diagnosis*



Figuur 3: System Sequence Diagram: *Register Patient*



Figuur 4: System Sequence Diagram: *Undo Previous Action*



Figuur 5: Klassediagram: *Event*

4 Subsystemen

Het systeem bestaat uit de volgende belangrijke componenten:

- User Interface
- Modelklassen
 - Belangrijke klassen
 - Tijdsverloop
 - Stockbeheer
 - Planning
- Controllers
- Repositories

User Interface

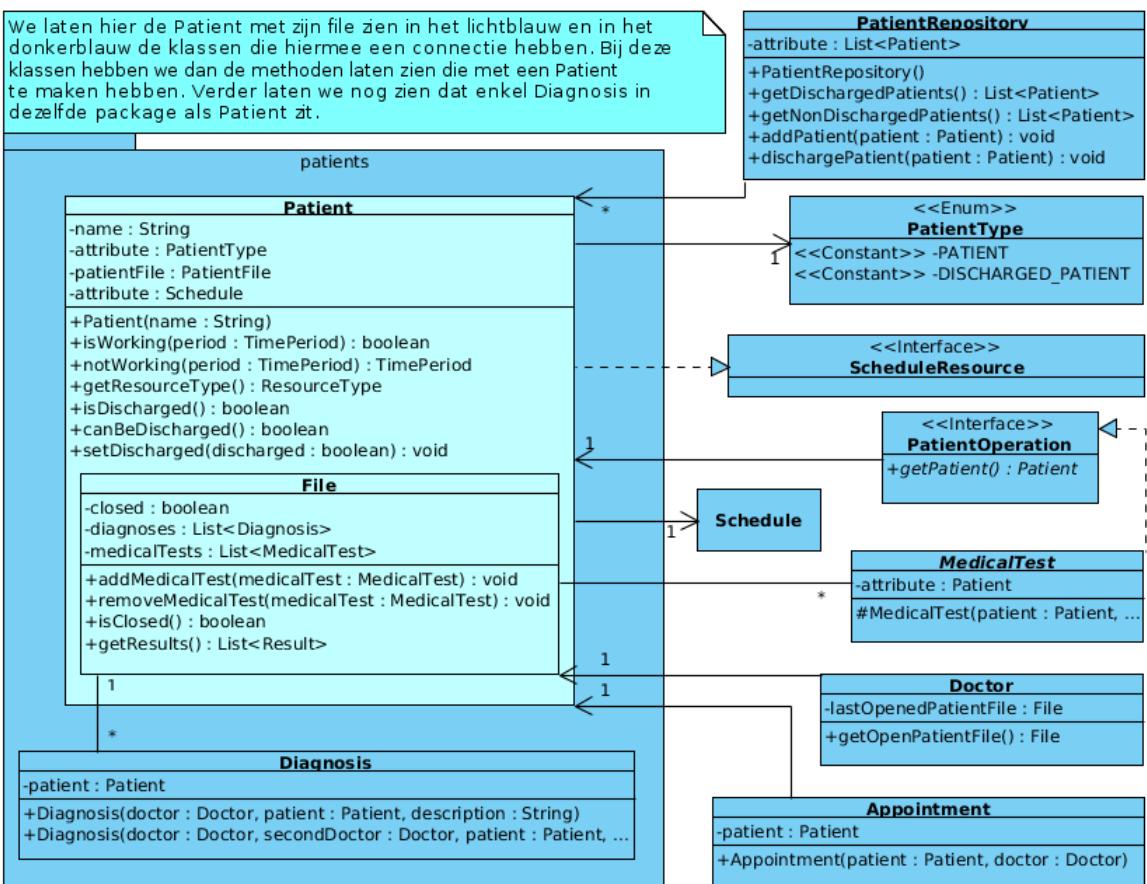
De *User Interface* stelt een ingelogde gebruiker in staat operaties te verrichten. Het communiceert enkel met de *Controllers* en staat los van de andere lagen.

Belangrijke klassen

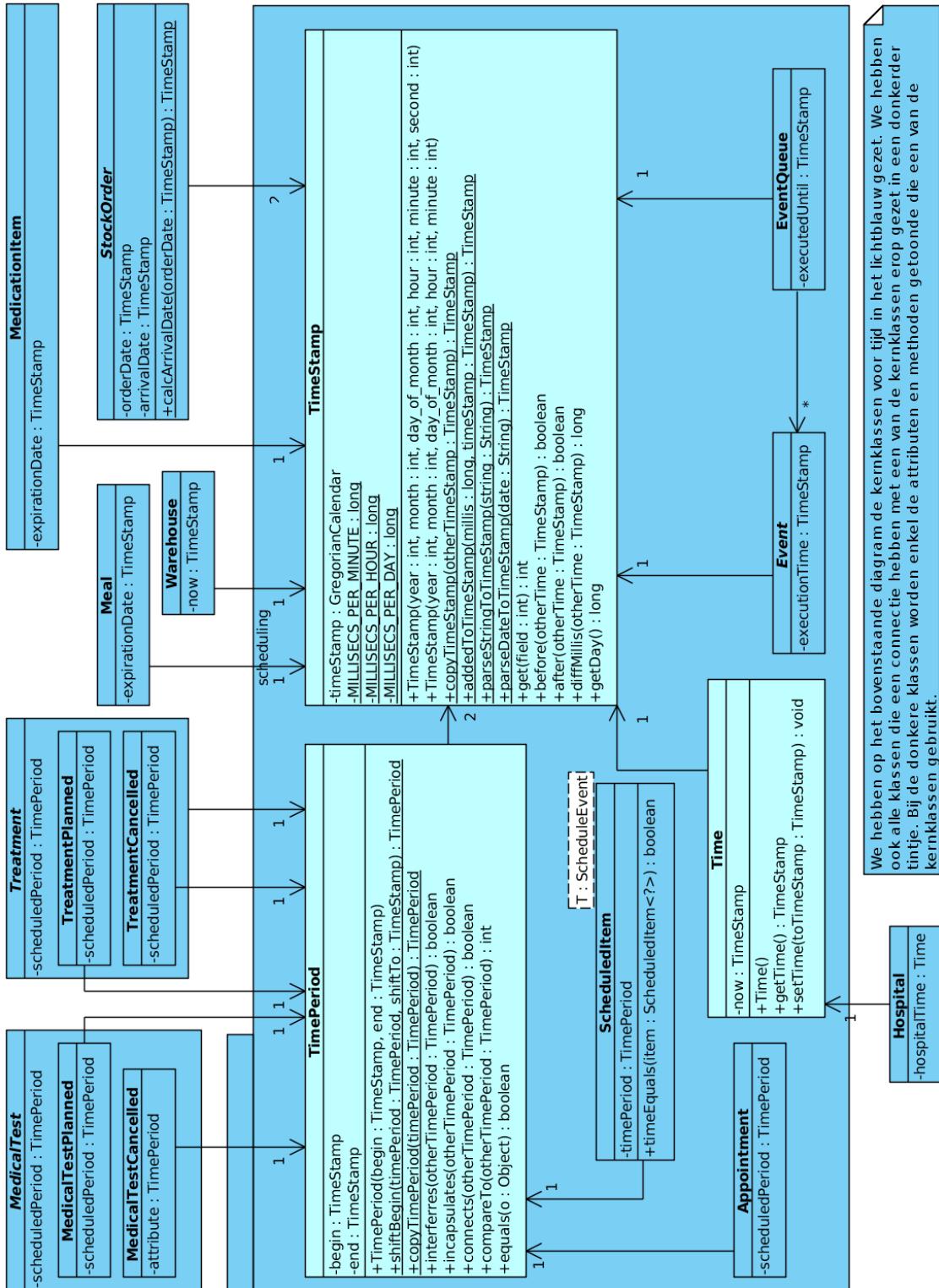
De modelklassen vormen de kern van het softwaresysteem. Zie figuur 5 en 6 voor de klassendiagrammen van twee belangrijke modelklassen, namelijk *Event* en *Patient* met *PatientFile*.

Tijdsverloop

Bepaalde klassen van het systeem zijn tijdsafhankelijk. In figuur 7 worden deze weergegeven.



Figuur 6: Klassediagram: *Patient* met *PatientFile*



Figuur 7: Klassendiagram: *Tijdsverloop*

Stockbeheer

Stockbeheer omvat het beheren van de voorraden aan maaltijden, medicijnen en gips. Zie figuur 8 voor het bijhorende klassediagram.

Planning

Dit belangrijk en complex subsysteem staat in voor het plannen van afspraken, medische tests en behandelingen. Zie figuur 9 voor het betreffende klassediagram.

Controllers

Voor een uitleg over de werking van *Controllers* verwijzen we naar sectie 5.1. Zie figuur 10 voor het klassediagram van *Controllers*.

Repositories

De werking van *Repositories* wordt nader uitgelegd in sectie 5.1. Zie figuur 11 voor het klassendiagram van *Repositories*.

5 Ontwerpbeslissingen

5.1 Algemene architectuur

Het softwaresysteem is opgebouwd uit drie lagen: de presentatielaag, de applicatielaag en de domeinlaag. De presentatielaag is de *user interface* die de gebruiker toelaat met het systeem te communiceren, de domeinlaag is het eigenlijke programma dat de juiste berekeningen maakt en de nodige handelingen uitvoert, en de applicatielaag is de verbinding tussen de twee.

De opdeling steunt op het *Layers* patroon waarnaar Craig Larman verwijst in zijn boek ‘Applying UML and Patterns’² en past het principe van *Separation of Concerns* toe. Hierbij wordt het programma opgedeeld in aparte stukken, elk met hun eigen functionaliteit of gedrag, om de koppeling tussen de verschillende delen te verlagen en de cohesie binnen elk stuk te verhogen. Het principe van *Separation of Concerns* is dus in feite een rechtstreekse toepassing van de GRASP-principes *Low Coupling* en *High Cohesion*.

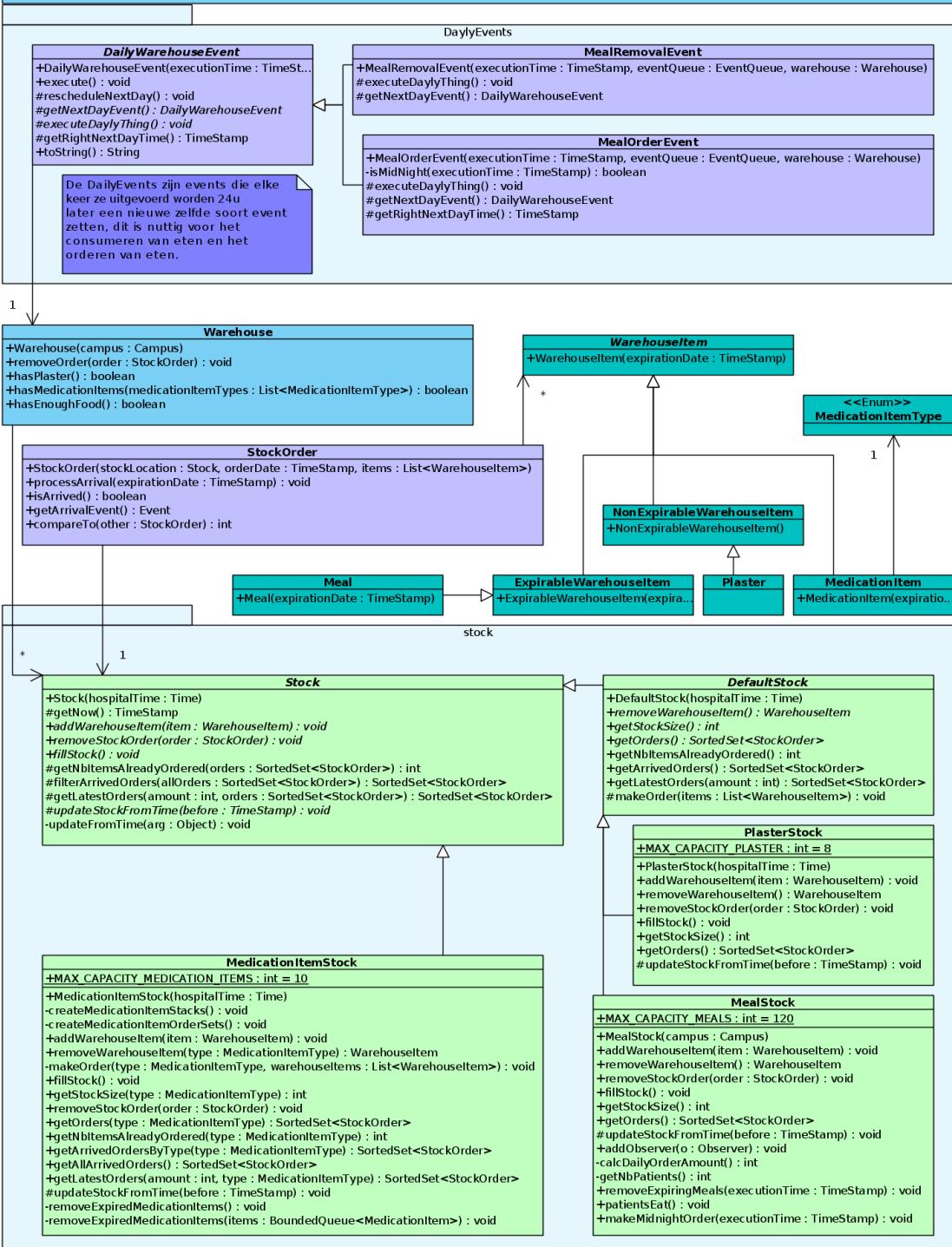
Deze twee principes vormen de bouwstenen van een goed ontworpen objectgeoriënteerd softwaresysteem. Immers, ze voorkomen dat een aanpassing in één enkel deel van het programma een verandering in heel het programma vereist en zorgen ervoor dat onderdelen van een softwaresysteem op zichzelf herbruikbaar zijn. Verder verbeteren ze de leesbaarheid van de code en bevorderen ze de onderhoudbaarheid van het programma. Ook duplicatie in de code wordt door hen geminaliseerd. Grof gesteld kan men zeggen dat alle andere GRASP-principes en zelfs alle ontwerppatronen tot lage koppeling en hoge cohesie te herleiden zijn.

Deze andere GRASP-principes of GRASP-patronen zijn terug te vinden in de verschillende lagen van de softwarearchitectuur. Zo bevat de applicatielaag een hele reeks controllerklassen. Zij zijn voorbeelden van het *Controller* patroon en kunnen bovendien als zelfbedachte klassen gezien worden als toepassingen van het *Pure Fabrication* patroon. Omdat controllers een directe koppeling tussen de user interface en de verschillende domeinklassen vermijden, zijn ze tevens voorbeelden van het *Indirection* patroon. Deze onrechtstreekse koppeling zorgt er bovendien voor dat veranderingen in de domeinklassen geen ongewenste invloed hebben op de user interface. Kortom, controllerklassen zijn voorbeelden van de GRASP-principes *Controller*, *Pure Fabrication*, *Indirection* én ook *Protected Variations*.

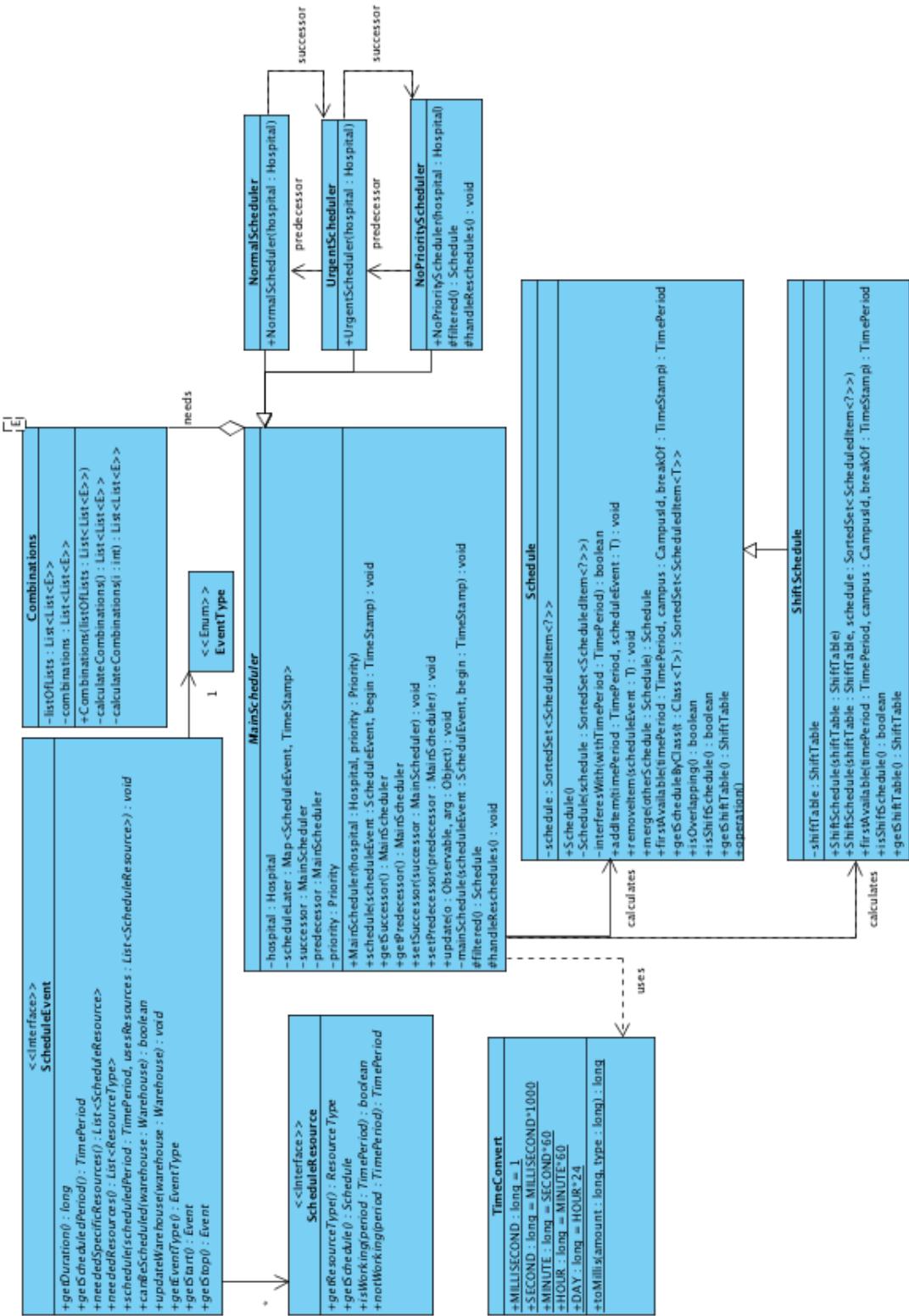
In de domeinlaag komen ook enkele GRASP-principes voor: *Information Expert*, *Creator* en *Pure Fabrication*. Voor concrete voorbeelden verwijzen we naar sectie 5.3 waar deze patronen

²Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development

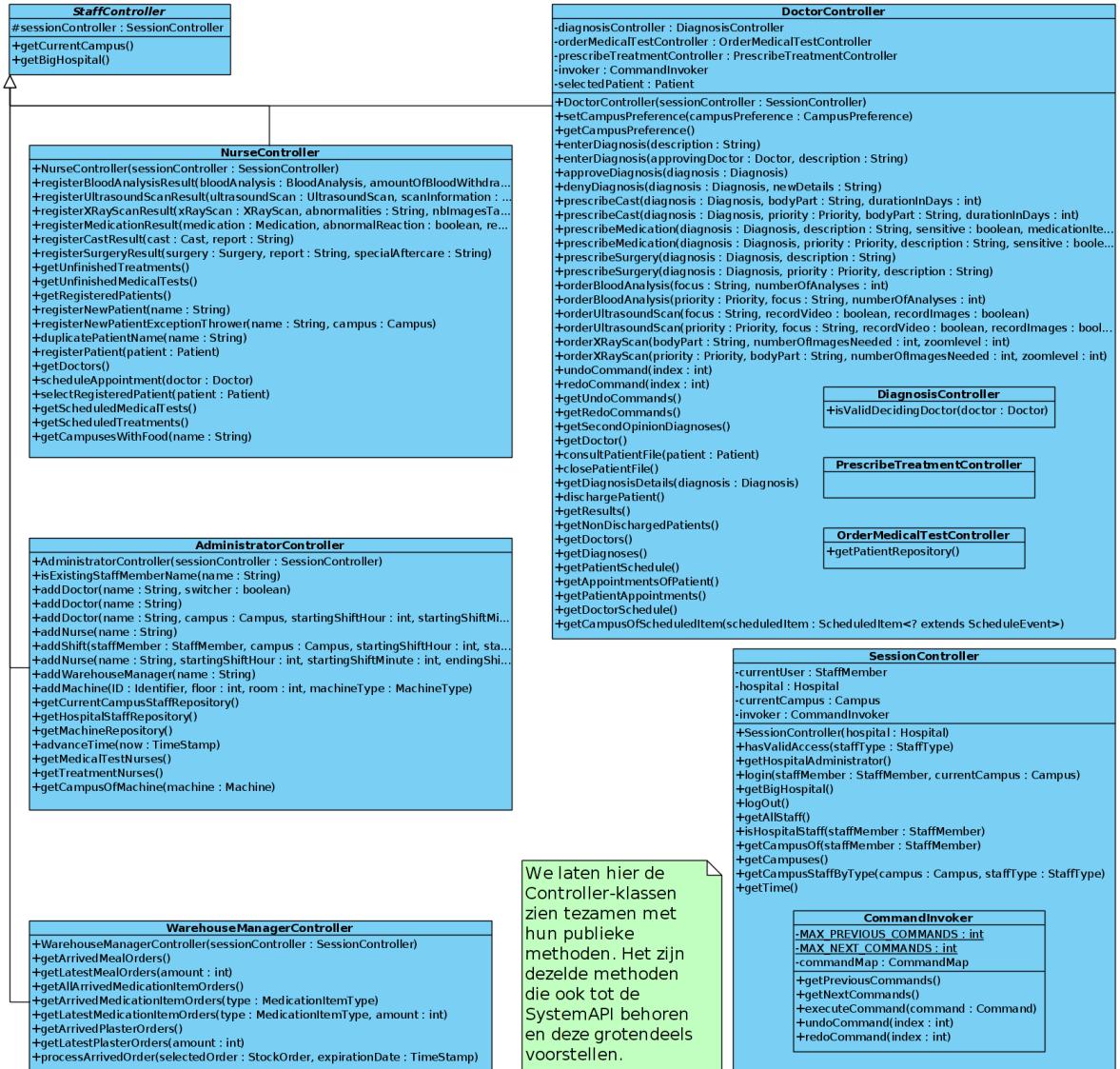
Dit klasse diagram laat alle klassen van het subsysteem Warehouse zien. Telkens zijn enkel publieke methoden en velden getoond. In Het Paars ziet we de klassen voor het bestellen en consumeren van items. In het fel groen zie je alle soorten items die in de Warehouse kunnen zitten, in het lichtblauw de Warehouse zelf en de klasse voor een bestelling en in het groen de klassen die items effectief bijhouden.



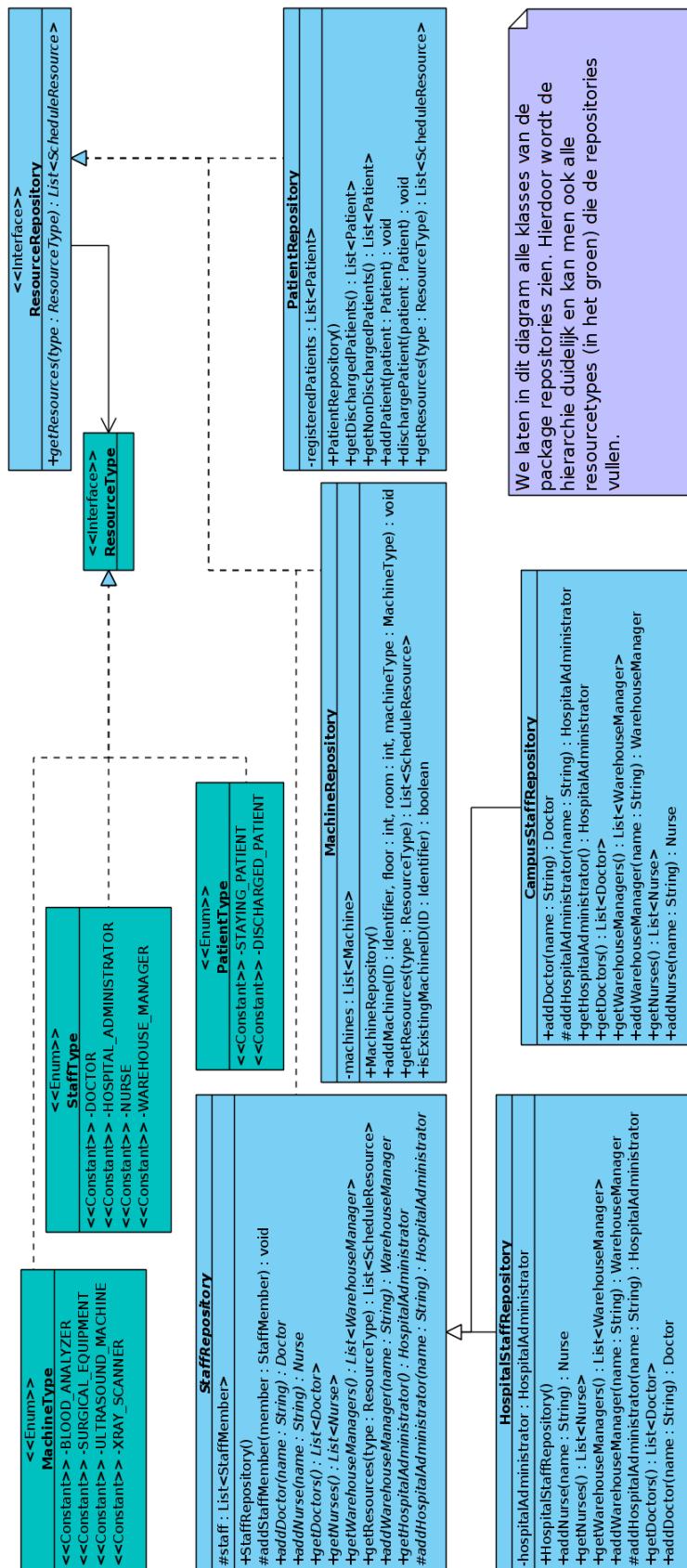
Figuur 8: Klasse diagram: Stockbeheer



Figuur 9: Klassediagram: *Planning*



Figuur 10: Klassediagram: *Controllers*



Figuur 11: Klassediagram: *Repositories*

duidelijk worden aan de hand van sequentiediagrammen voor de belangrijkste *use cases*. Het GRASP-principe *Polymorphism* wordt ook gebruikt maar dan wel veelal in ontwerppatronen.

De gelaagde architectuur van het softwaresysteem past niet alleen het principe van *Separation of Concerns* toe, maar sluit ook de communicatie van een lager gelegen laag naar een hoger gelegen laag uit zonder hetzelfde te doen voor de noodzakelijke omgekeerde richting. Dit is nogmaals een toepassing van de zo belangrijke lage koppeling. We maken hier wel een uitzondering op door de user interface rechtstreeks aan de repositoryklassen informatie te laten oproepen. *Repositories* houden unieke collecties van onder andere ziekenhuispersoneel, machines en patiënten bij. Hoewel ze ook methoden hebben om elementen toe te voegen en te verwijderen uit de collecties, mag de user interface de rechtstreekse koppeling met de repositoryklassen enkel gebruiken om informatie te lezen en vervolgens op het scherm te tonen. Hadden we deze uitzondering niet gemaakt, dan was een controllerklasse noodzakelijk geweest, die de aanvraag voor informatie delegerde naar de repositoryklassen en de juiste informatie vervolgens terugdelegerde naar de user interface.

5.2 Toegepaste ontwerppatronen

5.2.1 Tijdsverloop

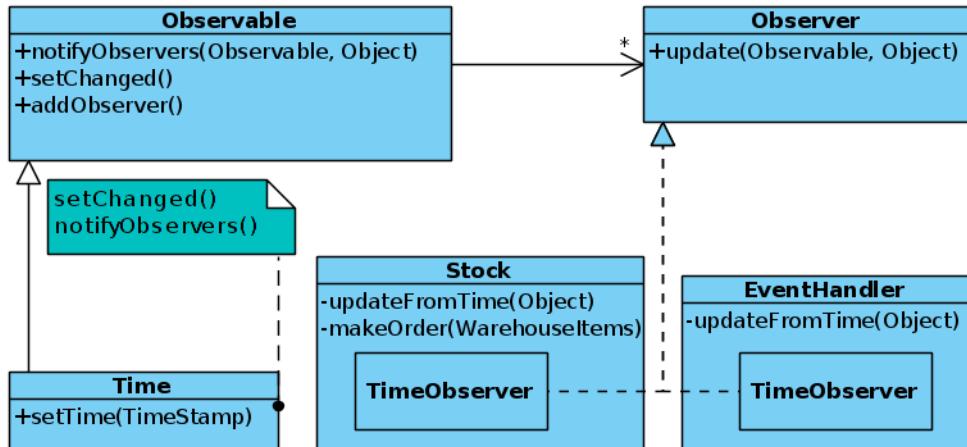
De klasse *Time* stelt in het systeem de tijd voor. Ze is geïnitialiseerd met de begintijd en is voorzien van een methode om de tijd vooruit te laten gaan. Wanneer de tijd verandert, moet dit kenbaar gemaakt worden aan een aantal klassen: de klassen *MealStock*, *MedicationItemStock* en *EventHandler*. De eerste twee representeren respectievelijk de stock van maaltijden en de stock van medicijnen. Ze worden bijgehouden in de klasse *Warehouse* die het magazijn van het ziekenhuis voorstelt, en dienen de tijd te kennen om te weten welke maaltijden en medicijnen vervallen zijn. *EventHandler* houdt dan weer een tijdslijn van toekomstige gebeurtenissen of events bij, die met het vooruitgaan van de tijd afgehandeld moeten worden.

Omdat meerdere klassen op de hoogte moeten zijn van de tijd, is het duidelijk dat we het *Observer*³ patroon kunnen toepassen. Dit patroon maakt een abstractie van enerzijds de klasse *Time* en anderzijds de klassen *MealStock*, *MedicationItemStock* en *EventHandler*. Op die manier wordt de klasse die de tijd voorstelt, en de klassen die de tijd dienen te kennen, van elkaar ontkoppeld. De klasse *Time* dient bijgevolg niet te weten wie hem gebruikt en het is gemakkelijk om nieuwe klassen die afhankelijk zijn van de tijd, toe te voegen. Voor de implementatie van het *Observer* patroon hebben we de klasse *Observable* en de interface *Observer* van Java gebruikt. Het nadeel van de standaard implementatie van Java is dat we in de methode *update(Observable o, Object arg)* het type van *o* en *arg* moeten controleren en eventueel moeten *casten*. Dit kan eenvoudig opgelost worden door een *switch statement* of een *instanceof*. Deze twee manieren zijn echter gevaarlijk. Indien er een nieuwe *Observable* subklasse bijkomt, dient een nieuwe case aan het *switch statement* of een nieuwe *instanceof* toegevoegd te worden. Dit klein detail wordt gemakkelijk over het hoofd gezien en het vergeten ervan wordt pas opgemerkt *at runtime*. Een *instanceof* werkt bovendien ook voor interfaces en geeft *true* terug indien de klasse een subklasse is van de opgelegde klasse. Dit gedrag is niet gewenst noch zonder risico's.

We hebben dit probleem aangepakt door, in elke klasse die de *Observer* interface implementeert, voor elke mogelijke *Observable* subklasse een overeenkomstige *inner* klasse te maken die *Observable* uitbreidt en *arg* aan de juiste methode doorgeeft. In de constructor van de omvattende klasse wordt deze *inner* klasse aan de *Observers* van de *Observable* subklasse toegevoegd. Op deze manier wordt de programmeur bij het toevoegen van een nieuwe *Observable* subklasse verplicht een *inner* klasse aan te maken en vervolgens als *Observer* aan de subklasse te koppelen. Het onverwacht opduiken van problemen *at runtime* wordt ook vermeden.

Het type van *o* hoeft dus niet meer gecheckt te worden. Voor het type van *arg* moet dit echter wel nog gebeuren. Aangezien *arg* in ons ontwerp slechts 1 type kan hebben, *casten* we het object naar het juiste type. Dit is uiteraard geen ideale oplossing: indien *arg* toch niet van het juiste type is, krijgen we immers pas een foutmelding *at runtime*. Een betere oplossing zou erin bestaan

³<http://www.cs.clemson.edu/~malloy/courses/patterns/observer.html>



Figuur 12: Klassediagram: *Observer (Time)*

om zelf een getypeerde *Observer* interface aan te maken die *Observer* uitbreidt en het type van *arg* vereist. Dit is dus een mogelijk verbeterpunt.

Zie figuur 12 voor het klassendiagram van het *Observer* patroon.

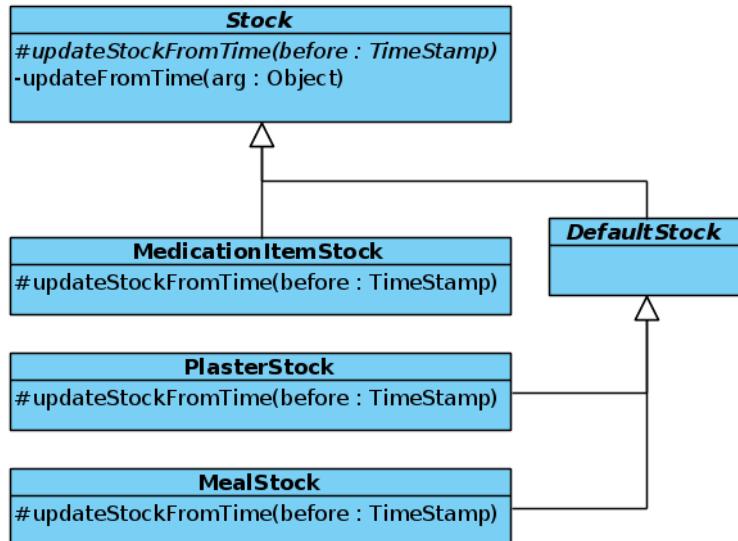
Als we het hierboven beschreven idee implementeren, dan bevatten de *Observable* subklassen *MealStock* en *MedicationItemStock* elk apart een *inner* klasse. Beide *inner* klassen zijn echter gelijkaardig en bevatten dus grotendeels dezelfde code. Om deze duplicatie te vermijden alsook om de uitbreidbaarheid van het programma te bevorderen, hebben we beslist één enkele *inner* klasse te implementeren en deze te plaatsen in de abstracte klasse *Stock*, de superklasse van *MealStock* en *MedicationItemStock*. Hierbij is gebruik gemaakt van het ontwerppatroon *Template Method*. Dit patroon definieert immers het skelet van een algoritme in een methode waarbij het sommige gedeelten aan subklassen overlaat. Anders gezegd, de gemeenschappelijke code zit in de superklasse terwijl de variërende stukken code uitgewerkt worden in de subklassen. De concrete implementatie van het *Template Method* patroon is simpel: de abstracte klasse *Stock* definieert de abstracte methode `updateStockFromTime(TimeStamp before)` die in de *inner* klasse aangeroepen wordt en die door de concrete klassen *MealStock* en *MedicationItemStock* op een verschillende manier geïmplementeerd wordt. Zie figuur 13

5.2.2 Stockbeheer

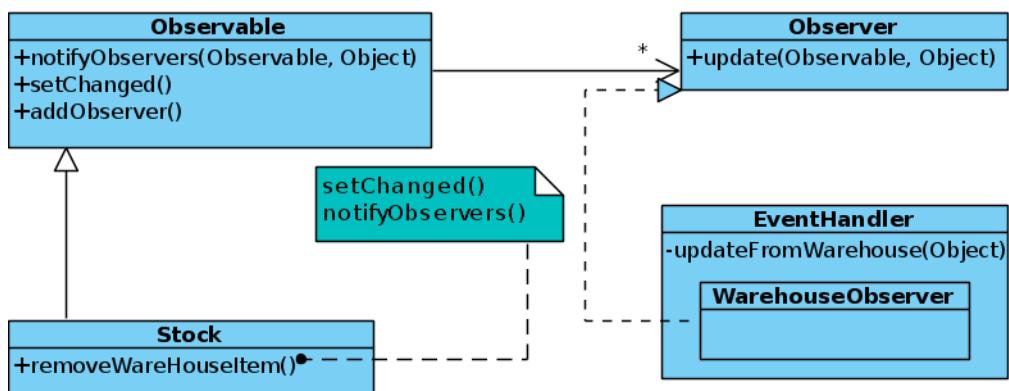
Tweede iteratie — In de tweede iteratie maakten we de onderstaande redenering.

De klasse *Warehouse*, die het magazijn van het ziekenhuis voorstelt en collecties van *Plaster*, *MedicationItem* en *Meal* bijhoudt, is op basis van de GRASP-principes *Creator* en *Information Expert* verantwoordelijk voor het aanmaken van *stock orders*. Deze dienen geplaatst en ontvangen te worden. Het plaatsen van een *order* (een bestelling) gebeurt door het toe te voegen aan een hiervoor voorziene collectie in de klasse *OrderRepository*, terwijl het ontvangen gebeurt door de bestelling door te geven aan de klasse *EventHandler* en daar het *event* dat de ontvangst van de bestelling uitvoert, aan de tijdslijn van gebeurtenissen toe te voegen. *OrderRepository* en *EventHandler* moeten dus eenzelfde bestelling van *Warehouse* doorgestuurd krijgen. We kunnen dus weer, op dezelfde manier en met dezelfde gevolgen, het *Observer* patroon toepassen. Zie figuur 14 voor het klassendiagram van *Observer* met betrekking op *Warehouse*.

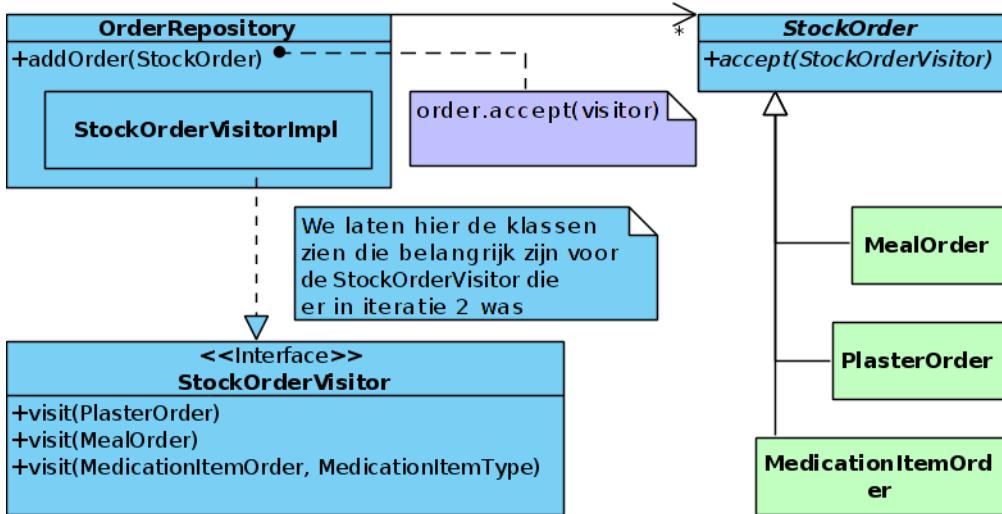
Een *order* wordt voorgesteld door de abstracte klasse *StockOrder* die drie subklassen heeft: *PlasterOrder*, *MedicationItemOrder* en *MealOrder*. In de klasse *OrderRepository* dient onderscheid gemaakt te worden tussen deze drie. Het argument *arg* van de methode `update(Observable o, Object arg)` heeft bijgevolg een van deze drie types en moet dus op drie verschillende manieren afgehandeld kunnen worden. Om geen *instanceof* of een *switch statement* tezamen met een *Enum*



Figuur 13: Klassendiagramm: *Template Method (Stock)*



Figuur 14: Klassendiagramm: *Observer (Warehouse)*



Figuur 15: Klassediagram: *Visitor (Orders)*

te gebruiken, is het *Visitor*⁴ patroon geïmplementeerd. Dit patroon laat toe om voor verschillende objecten operaties te definiëren die afhankelijk zijn van hun concrete klassen, en leent zich dus uitstekend voor de situatie. Het grote voordeel van het *Visitor* patroon over een *instanceof* of een *switch statement* met een *Enum* is dat het ‘maximale’ statische *type checking* oplegt. De programmeur kan bijgevolg *at compile time* geen operatie op een verkeerd type laten uitvoeren. De code zal immers niet compileren. Zie figuur 15 voor het klassendiagram van *Visitor* met betrekking op *Orders*.

De implementatie van het *Visitor* patroon is complex. In onze code is de *Visitor* een interface genaamd *StockOrderVisitor* met een methode *visit* die met drie verschillende argumenten kan aangeroepen worden. Het argument kan een instantie zijn van één van de subklassen. De drie subklassen van *StockOrder* zijn: *PlasterOrder*, *MedicationItemOrder* en *MealOrder*. De interface wordt geïmplementeerd door de klasse *AddingStockOrderVisitor*. Aangezien deze klasse enkel gebruikt moet worden door *OrderRepository* en dus enkel gekend moet zijn door *OrderRepository*, is ze opgenomen als *inner* klasse bij deze laatste. In de implementatie van de drie methoden wordt de *order* aan de juiste lijst toegevoegd. Het element waarop de *Visitor* toegepast wordt, is de klasse *StockOrder* die daarvoor de abstracte methode *visit(StockOrderVisitor stockOrderVisitor)* voorziet. In de concrete methode van elk van zijn subklassen, wordt de juiste *visit*-methode van *StockOrderVisitor* aangeroepen. Om alles te laten werken wordt ten slotte in de methode *addOrder(StockOrder order)* van de klasse *OrderRepository* een concreet *StockOrderVisitor* object genaamd *visitor* aangemaakt en *order.visit(visitor)* opgeroepen.

Het *Visitor* patroon wordt niet alleen gebruikt voor het toevoegen van een *order* maar ook voor het verwijderen van een *order*. Dit wordt gedaan door een nieuwe implementatie van *StockOrderVisitor* toe te voegen. Met het patroon kunnen we dus meerdere operaties op dezelfde klassen definiëren.

Derde iteratie — In de derde iteratie hebben we getracht het ontwerp van de tweede iteratie wat betreft het stockbeheer te verbeteren. Hiertoe hebben we de stukken code in de klasse *Warehouse* die betrekking hebben op het bewaren en het beheren van maaltijden, medicijnen en gips, van elkaar gescheiden en in drie nieuwe klassen geplaatst. Deze klassen zijn *MealStock*, *MedicationItemStock* en *PlasterStock* en breiden de nieuwe, abstracte klasse *Stock* uit. *Warehouse* is hierbij afgeslankt tot een klasse die een instantie van elk van deze drie klassen bijhoudt en kan doorgeven. Het voordeel van deze aanpassing is dat men bij het toevoegen van een nieuw item

⁴<http://userpages.umbc.edu/~tarr/dp/lectures/Visitor.pdf>

niet langer bestaande klassen moet aanpassen en alleen nog maar nieuwe klassen moet toevoegen.

Naast deze wijziging hebben we ook de stukken code in de klasse *OrderRepository* aangaande het plaatsen en het ontvangen van orders voor maaltijden, medicijnen en gips van elkaar gescheiden en naar de klassen *MealStock*, *MedicationItemStock* en *PlasterStock* verplaatst. Zij zijn immers *Creator* en *Information Expert* wat betreft het aanmaken van stock orders en zijn dus de aangewezen kandidaat voor het bijhouden van hun respectievelijke orders. Het gevolg hiervan is dat de klasse *OrderRepository* verwijderd kan worden en dat het niet langer nodig is om onderscheid te maken tussen een *MealOrder*, een *MedicationItemOrder* en een *PlasterOrder*. *StockOrder* volstaat als enige klasse. Het *Visitor* patroon dat in de vorige iteratie gebruikt werd, is in deze iteratie niet langer nodig. Dit laatste verlaagt niet alleen de complexiteit van de koppeling in het programma maar maakt het ook gemakkelijk om nieuwe items aan het stockbeheersysteem toe te voegen.

Wegens tijdgebrek hebben we geen verdere verbeteringen aan ons programma kunnen doorvoeren. Dit wil echter niet zeggen dat er geen zijn. Zo zouden we bijvoorbeeld twee subklassen voor de klasse *WarehouseItem* kunnen maken; de ene stelt een item dat vervalt, voor terwijl de andere een item dat nooit vervalt, voorstelt. Deze laatste maakt hierbij gebruik van een aangepaste *TimeStamp* klasse die oneindig kan voorstellen. Deze aanpassing zou er voor zorgen dat stock items die naast een vervaldatum geen andere attributen hebben, zoals *Plaster* en *Meal*, niet opgenomen hoeven te worden in de hiërarchie van magazijn items. Er dient alleen nog maar een nieuwe subklasse van *Stock* toegevoegd te worden. Verder zouden we ook de *policies* die bepalen wanneer en hoe de stock moet worden opgevuld, uit de concrete *Stock* klassen kunnen halen met behulp van bijvoorbeeld het *Strategy* patroon. Deze wijziging zou mogelijke redundante code vermijden.

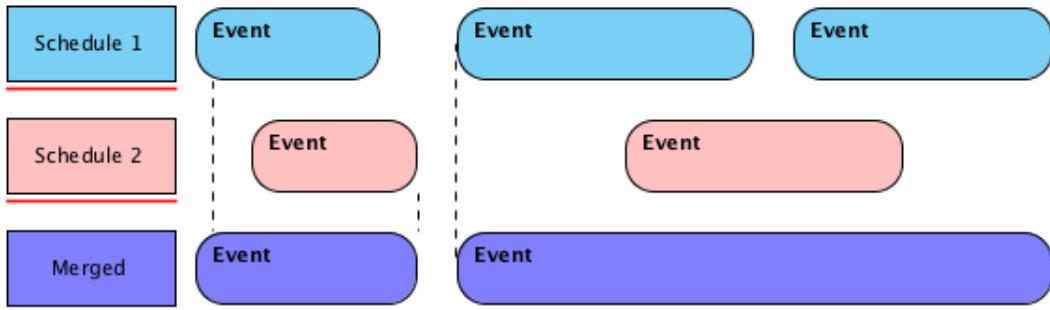
Er zijn natuurlijk ook volwaardige alternatieven voor ons ontwerp zoals bijvoorbeeld een meer dynamische aanpak waarbij men met *enums* en *hash maps* werkt. In de vorige iteratie kozen we echter voor een aanpak met ‘maximale’ statische type checking.

5.2.3 Planning

Algemeen — Er is in ons systeem ervoor gekozen het plannen te doen aan de hand van het samenvoegen van de planning van verschillende middelen. Elke *ScheduleResource* heeft een eigen *Schedule* waarin zijn planning staat, voorgesteld als een lijst van *TimePeriod* objecten. Wanneer een vrije periode moet gezocht worden in een planning na een bepaald ongeblik, wordt deze lijst afgegaan. Wanneer er een opening is tussen twee tijdsperiodes, die groot genoeg is voor de gevraagde tijdsperiode, wordt deze gepland. Voor zaken die moeten gepland worden voor één resource is dit genoeg. Er moet echter voor een combinatie van middelen gepland worden. Hiervoor dient het merge-mechanisme, waarover het hierboven al ging. Dit mechanisme kan wiskundig gezien worden als het nemen van de unie van de tijdsperiodes van twee *Schedule* objecten. Wanneer twee tijdsperiodes elkaar direct opvolgen, wordt dit in de samengevoegde planning één tijdsperiode. Op de samengevoegde planning kan hetzelfde mechanisme weer uitgevoerd worden, zo kunnen meerdere planningen samengevoegd worden.

Op Figuur 16 wordt alles rond het samenvoegen getoond. De eerste *Event* van *Schedule 1* samen met de eerste *Event* van *Schedule 2* zorgen respectievelijk voor het begin en einde van de eerste *Event* van de samengevoegde planning. De tweede en derde *Event* van *Schedule 1* zorgt voor het begin en einde van de tweede *Event* van de samengevoegde planning. De tweede *Event* van *Schedule 2* zorgt ervoor dat de opening tussen de tweede en derde *Event* van *Schedule 1* opgevuld geraakt in de samengevoegde planning.

Afspraken plannen — We moeten afspraken kunnen plannen en, wanneer het moment aangebroken is, kunnen uitvoeren. Belangrijk hierbij is dat een afspraak niet gepland kan worden als hij al gepland is, als hij bezig is of als hij gedaan is. Daarenboven kan een afspraak niet uitgevoerd worden als hij nog niet gepland is, als hij al uitgevoerd wordt en als hij gedaan is. We kunnen dus stellen dat we afhankelijk van de toestand van een afspraak al dan niet een bepaalde handeling op de afspraak mogen uitvoeren.



Figuur 16: Schema: *Scheduling - merge*

Als we een bepaalde handeling op een afspraak willen uitvoeren, moeten we dus controleren in welke toestand een afspraak zich bevindt. We zouden dit kunnen doen door de mogelijke toestanden van een afspraak in een *Enum* te definiëren en vervolgens de huidige toestand in de code te bepalen met *if-then-else statements* en de *Enum*. Dit kan echter resulteren in onleesbare en moeilijk onderhoudbare code. Immers, naarmate er meer verschillende handelingen op een afspraak uitgevoerd kunnen worden, zijn er meer *if-then-else statements* en naarmate een afspraak meer toestanden heeft, zijn de *if-then-else statements* langer. Deze *statements* bevatten bovendien vaak dubbele code en indien een nieuwe toestand toegevoegd wordt, vergeet men gemakkelijk een *if-then-else statement* aan te passen.

Om aan al deze problemen te ontsnappen, hebben we besloten het *State*⁵ patroon in te voeren. Dit patroon vermindert niet alleen *if-then-else statements* maar zorgt door het expliciet maken van de toestanden of *states* van een afspraak ook voor eenvoudige uitbreidbaarheid en verhoogde cohesie.

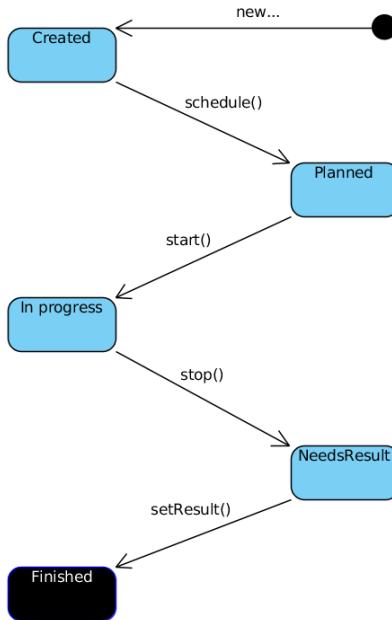
De implementatie is niet moeilijk. De klasse *Appointment* die een afspraak voorstelt, gebruikt een abstracte klasse *State* die de handelingen die op een afspraak uitgevoerd moeten kunnen worden, als methoden declareert. De klasse *State* heeft een aantal subklassen die de verschillende toestanden van een afspraak voorstellen en die uiteraard de methoden van *State* implementeren. In deze implementaties staan tevens de overgangen tussen de verschillende *states*. Het gehele werkt doordat in de methodes van de klasse *Appointment* de huidige toestand wordt opgevraagd en vervolgens hierop de overeenkomstige methode wordt aangeroepen. Merk op dat we de klasse *State* en zijn subklassen als *inner* klassen in de klasse *Appointment* hebben geëncapsuleerd omdat niemand anders kennis van deze toestanden moet hebben. In het *state* diagram staan de volgende toestanden:

- *Created*: de afspraak bestaat maar wordt niet gebruikt
- *Planned*: de afspraak is gepland
- *InProgress*: de afspraak is bezig
- *Finished*: de afspraak is gedaan

Hoewel het hier duidelijk een *State* patroon is, zou men zich kunnen afvragen waarom het geen *Strategy*⁶ patroon is. Of anders gezegd: wat is het verschil tussen het *State* patroon en het *Strategy* patroon? Het verschil ligt hem in de intentie: *State* encapsuleert toestandsafhankelijk gedrag terwijl *Strategy* algoritmes encapsuleert. Bovendien zijn bij het *State* patroon de toestandsvergangen niet transparant voor de *client*. Een toestandsverandering gebeurt namelijk in

⁵http://sourcemaking.com/design_patterns/state

⁶<http://www.oodesign.com/strategy-pattern.html>



Figuur 17: Toestandsdiagram: *Appointment*

een andere toestand. Bij het *Strategy* patroon daarentegen wordt de strategie bepaald door de *client*. De strategie wordt vastgelegd bij de creatie van de *client* en verandert niet meer.

Zie figuur 17 voor het toestandsdiagram.

Medische tests plannen — Omwille van dezelfde redenen als bij afspraken gebruiken we voor medische tests ook het *State* patroon. We hebben de volgende toestanden geïmplementeerd:

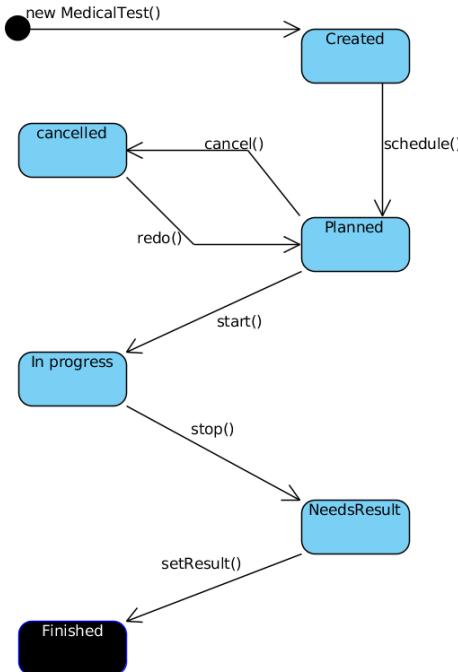
- *Created*: de medische test bestaat maar wordt niet gebruikt.
- *Planned*: de medische test is gepland.
- *Cancelled*: de medische test was gepland maar is (tijdelijk) geannuleerd.
- *InProgress*: de medische test is bezig.
- *Finished*: de medische test is gedaan.

Voor medische testen is er wel nog een bijkomende reden om het *State* patroon te gebruiken: de operatie *Order Medical Test* kan en mag enkel ongedaan gemaakt worden als de medische test nog niet bezig of gedaan is. Om dit te realiseren volstaat de methoden *execute()* en *undo()* aan de klasse *State* en zijn subklassen toe te voegen. Dit is dus een mooi voorbeeld van de uitbreidbaarheid van het *State* patroon.

Zie figuur 18 voor het toestandsdiagram.

Behandelingen plannen — De redenen om het *State* patroon te gebruiken voor behandelingen zijn dezelfde als die voor medische testen. Er zijn wel zes toestanden in plaats van vijf:

- *Created*: de behandeling bestaat maar wordt niet gebruikt.
- *Stored*: de behandeling is enkel opgeslagen in zijn diagnose.
- *Planned*: de behandeling is gepland voor al zijn resources.
- *Cancelled*: de behandeling was gepland/gestored maar is (tijdelijk) geannuleerd.
- *InProgress*: de behandeling is bezig.



Figuur 18: Toestandsdiagram: *Medical Tests*

- *Finished*: de behandeling is gedaan.

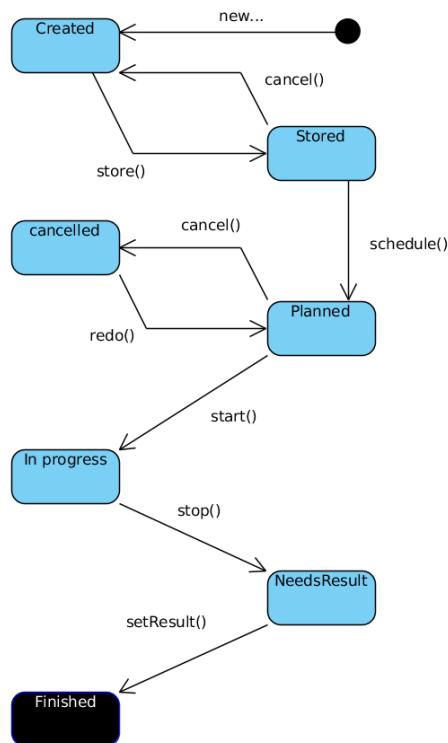
Zie figuur 19 voor het toestandsdiagram.

Campussen — Het moet mogelijk zijn om logisch te plannen tussen verschillende campusen. Dit introduceert logischerwijze het probleem van het reizen. Wanneer een patient een afspraak heeft op een bepaalde campus en daarna op een andere campus, kan deze afspraak niet dadelijk volgend op de vorige gepland worden. Voortbouwend op onze werkwijze van het samenvoegen van planningen, kon dit vrij eenvoudig verwerkt worden in ons systeem. Elke planning moet voor hij gemerged wordt aangepast worden voor campus waarop we een vrij moment zijn aan het zoeken. Bij elke *Event* op de planning moet de reistijd tussen de campus van de *Event* en de gevraagde campus bijgevoegd worden aan de voorkant en aan de achterkant. De tijdsduur van de *Event* wordt dan langer, en er wordt automatisch rekening gehouden met de reistijd. De reistijd wordt opgezocht in een volledige graaf in het *Hospital* object. Dit zorgt ervoor dat er gemakkelijk uitbreidingen gedaan kunnen worden met meer dan twee campusen.

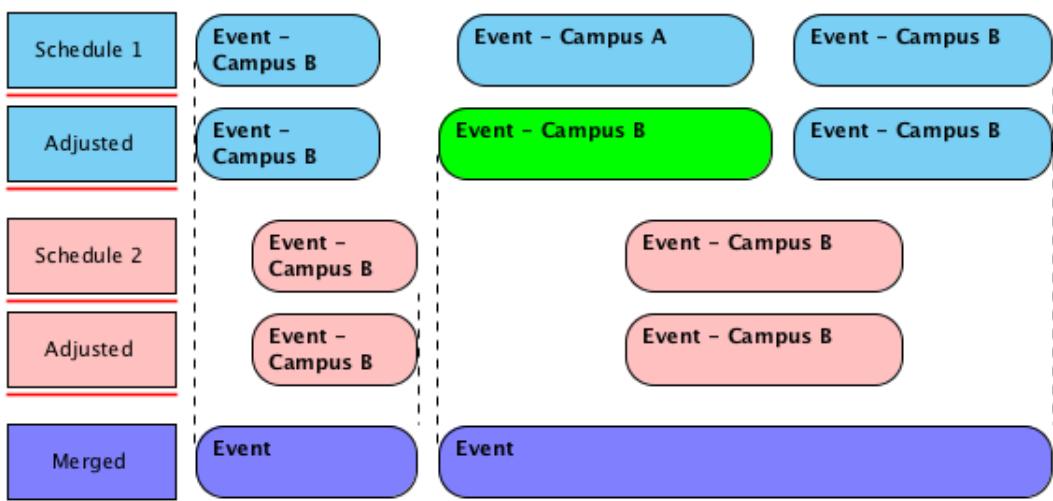
In Figuur 20 wordt onder de originele telkens de *adjusted* planning gegeven. In dit voorbeeld worden de planningen vervormt om een vrij moment te zoeken op *campus B*. De tweede *Event* van *Schedule 1* vindt plaats op *Campus A*. Door de *Event* met de juiste reistijd te verlengen langs beide kanten, wordt verzekerd dat deze *Resource* genoeg tijd heeft om te reizen (als hij/zij dit kan).

Er wordt telkens op alle campusen geprobeerd een *Event* te plannen. Lukt dit niet, bijvoorbeeld als het nodige materiaal niet op de campus aanwezig is, wordt er gewoon doorgegaan met plannen op de volgende campus. Er wordt uit alle campusen waarop kan gepland worden, de dichtsbijzijnde tijdsperiode gekozen. Op deze campus wordt dan gepland. Hierbij gebruikt het algoritme ook een afbrekingsmechanisme, waarbij er niet later wordt gezocht wanneer op een andere campus al een vroeger afspraak is gevonden.

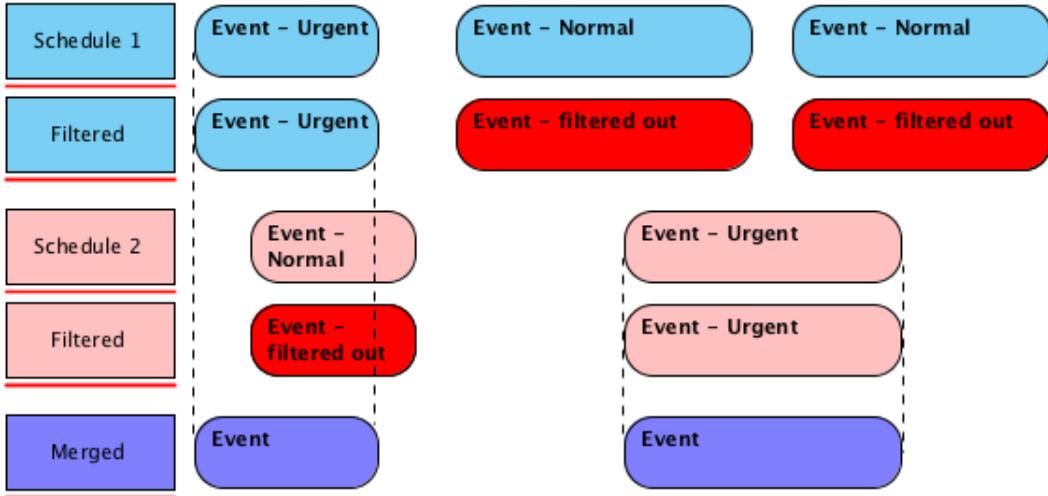
Prioriteiten — De prioriteiten werden geïmplementeerd door gebruik te maken van het *Chain of Responsibility* ontwerppatroon. Voor elke prioriteit is er een eigen planner. Deze erven over



Figuur 19: Toestandsdiagram: *Treatment*



Figuur 20: Schema: *Scheduling - campussen*

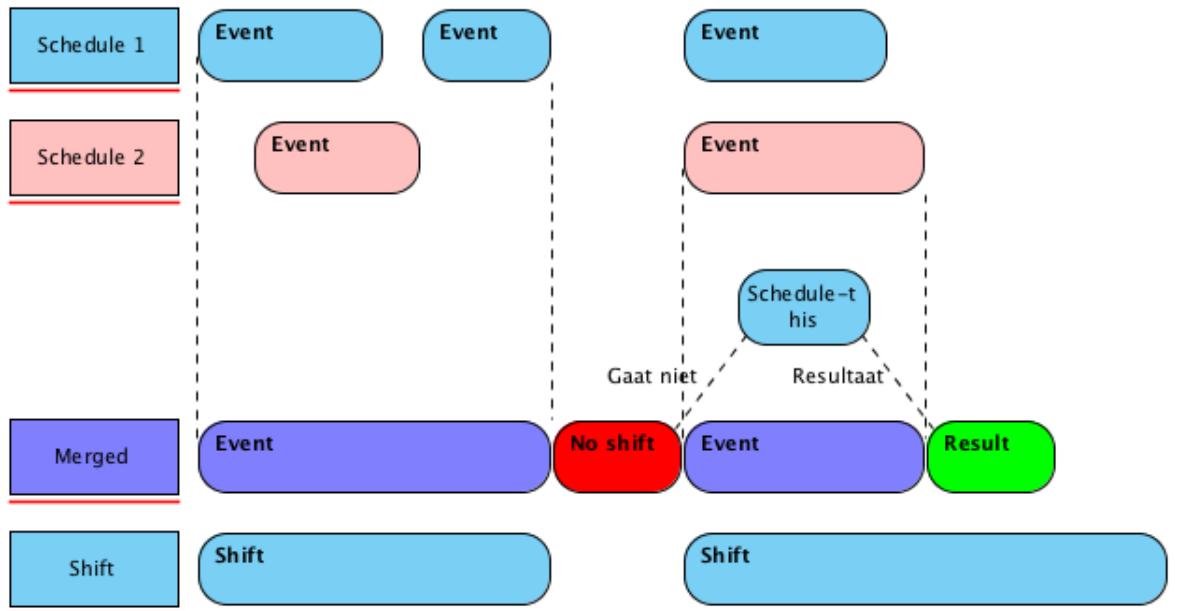


Figuur 21: Schema: *Scheduling - prioriteiten*

van *MainScheduler*. Ze geven in de constructor een *Priority* mee aan de constructor van hun superklasse. In de constructor wordt ook de volgende klasse in de keten gekozen. Het is de verantwoordelijk van de subklasse zelf om een juiste volgende *MainScheduler* aan te duiden. Dit kan eventueel ook de verantwoordelijkheid zijn van de klasse waar de eerste *Scheduler* in de keten wordt aangemaakt. Dit is in ons systeem in de klasse *Hospital*. Dit zou echter betekenen dat telkens wanneer er een nieuwe prioriteit wordt aangemaakt met een eigen scheduler, er aanpassingen zouden moeten gedaan worden in de klasse *Hospital*. Verder is er telkens een methode *handleReschedules()* en *filtered()* die het gedrag van een specifieke *Scheduler* bepalen. Wanneer deze niet worden overschreven, zal er gewoon gewerkt worden met de methode *filtered()* en/of *handleReschedules()* in de superklasse *MainScheduler*. Deze houdt rekening met de prioriteit van de specifieke *Scheduler*. De kracht van dit ontwerp is echter dat er ook planners met een speciaal gedrag kunnen toegevoegd worden, door de twee bovenstaande methodes wel te overschrijven. Er kan in ons systeem zeer gemakkelijk een prioriteit bij worden gemaakt.

Figuur 22 verduidelijkt de werking van het systeem rond prioriteiten. Wanneer er moet gescheduled worden, zal er in plaats van met de originele planningen gewerkt worden met planningen die gefiltered zijn op prioriteit. Hiervoor is een *Filter* klasse gemaakt. Wanneer er gepland wordt op de originele planning, kan er dan een inconsistentie *Schedule* ontstaan. Omdat een *Event* met hogere prioriteit een vrij moment zoekt in de gefilterde planning, kan hij in de originele planning met een andere *Event* overlappen. De planning wordt later in hetzelfde algoritme terug in een consistente staat gebracht door de methode *handleReschedules()*.

Werkuren Wanneer er een open moment gevonden is in een planning, moet in sommige gevallen nog rekening gehouden worden met de werkuren van bijvoorbeeld een *StaffMember*. Daarom krijgen deze *Resources* een *ShiftSchedule* in plaats van een *Schedule*. Dit is een subklasse van *Schedule* en overschrijft de methode *firstAvailable(TimePeriod, Campus, breakOf)*. Wanneer de planner deze methode aanroeft, zal deze in het geval van een *ShiftSchedule* automatisch rekening houden met de werkuren van de middelen. Verder moet er bij het samenvoegen van *Schedule* objecten nog rekening gehouden worden met het feit dat n van de objecten, of beide, een *ShiftSchedule* zou kunnen zijn. Er wordt dan verzekerd dat de planning die teruggegeven wordt ook een planning met werkuren is. Als beide planningen werkuren bevatten, moet de samengevoegd *ShiftSchedule* als werktabel de doorsnede van de beide werktabellen meekrijgen.



Figuur 22: Schema: *Scheduling - prioriteiten*

Hoe rekening gehouden wordt met de werkuren van *ScheduleResource* objecten, wordt duidelijk op Figuur ???. Een *Event* kan niet gepland worden buiten de doorsnede van de werkuren van de middelen met een *ShiftSchedule*. In bijhorende figuur is *Schedule 1*, *Schedule 2* of beide een *ShiftSchedule*.

Geplande afspraken, medische tests en behandelingen uitvoeren

Geplande afspraken, medische tests en behandelingen moeten effectief uitgevoerd worden. Daarom moet de klasse *Scheduler* na het plannen van een van deze gebeurtenissen de gebeurtenis ook doorgeven aan de klasse *EventHandler*. Met het oog op uitbreidbaarheid, om de koppeling tussen *Scheduler* en *EventHandler* te verlagen en omdat *EventHandler* de *Observer* interface reeds geïmplementeerd heeft, gebruiken we opnieuw het *Observer* patroon. Zie figuur 23 voor het klassendiagram.

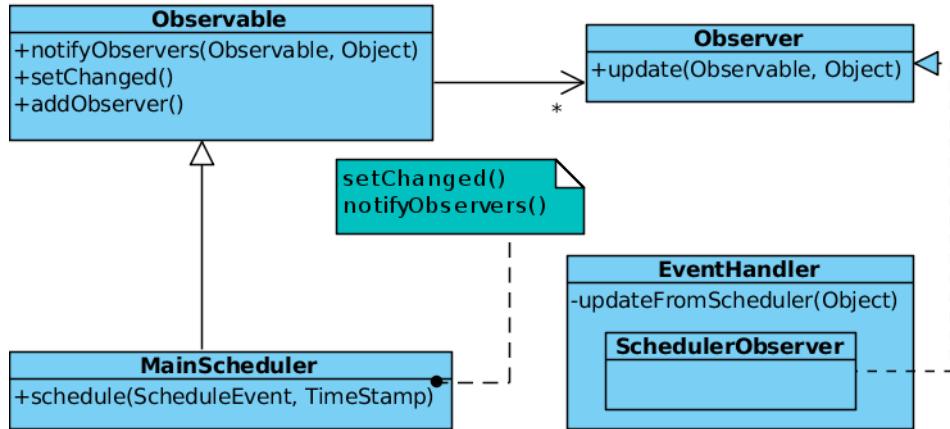
5.2.4 Undo/redo-systeem

Het systeem moet een functie voorzien die dokters toelaat hun laatste 20 acties ongedaan te maken en hun laatste 5 ongedane acties terug uit te voeren. Twee ontwerppatronen lenen zich bij uitstek voor het implementeren van een *undo/redo*-systeem. Deze zijn het *Command*⁷ patroon en het *Memento*⁸ patroon. De eerste gebruikt een object om alle informatie die nodig is om op een later tijdstip een methode aan te roepen, bij te houden. Deze informatie bestaat uit de naam van de methode, het object die de methode bevat, en de waarden van de parameters van de methode. De laatste daarentegen maakt het mogelijk om de interne toestand van een object te bewaren en op een later tijdstip het object naar deze vorige toestand te herstellen.

We hebben het *Command* patroon gekozen omdat dit op een eenvoudige en goede manier aan onze noden kan voldoen. In twee aparte lijsten kunnen enerzijds de acties die ongedaan gemaakt

⁷<http://c2.com/cgi/wiki/CommandPattern>

⁸http://en.wikipedia.org/wiki/Memento_pattern



Figuur 23: Klassediagram: *Observer (Scheduler)*

moeten kunnen worden, en anderzijds de ongedane acties die terug uitgevoerd moeten kunnen worden, opgeslagen worden. Het object dat de actie bevat die niet had mogen gebeuren, kan nu uit een van de twee lijsten gehaald worden en vervolgens gebruikt worden om de gewenste actie ongedaan of terug gedaan te maken.

Het *Memento* patroon is in onze situatie zeer moeilijk te implementeren. *Memento* maakt een momentopname van de toestand van een enkel object en bewaart dit om later het object te herstellen naar deze toestand. *Memento* kan enkel aangepaste objecten veranderen van toestand en is dus niet noodzakelijk in staat het hele ziekenhuissysteem naar een vroegere toestand terug te draaien. In ons geval zou met andere woorden een groot aantal objecten gecontroleerd moeten worden op permanente wijzigingen; en enkel indien ze niet permanent gewijzigd zijn, zouden ze overschreden mogen en moeten worden door wat in de *Memento* zit. Een andere mogelijke *Memento*-implementatie is meer functioneel gericht: elk uitgevoerd commando wordt bijgehouden en bij een undo-actie wordt de toestand hersteld tot x stappen in het verleden. Latere commando's moeten dan uitgevoerd worden op de nieuwe *Hospital* toestand. Dit kan echter problemen met zich meebrengen zoals bijvoorbeeld wanneer alles anders gepland wordt. Kortom, door de grote complexiteit die gepaard gaat met het implementeren van het *Memento* patroon, is deze geen goed alternatief voor het *Command* patroon.

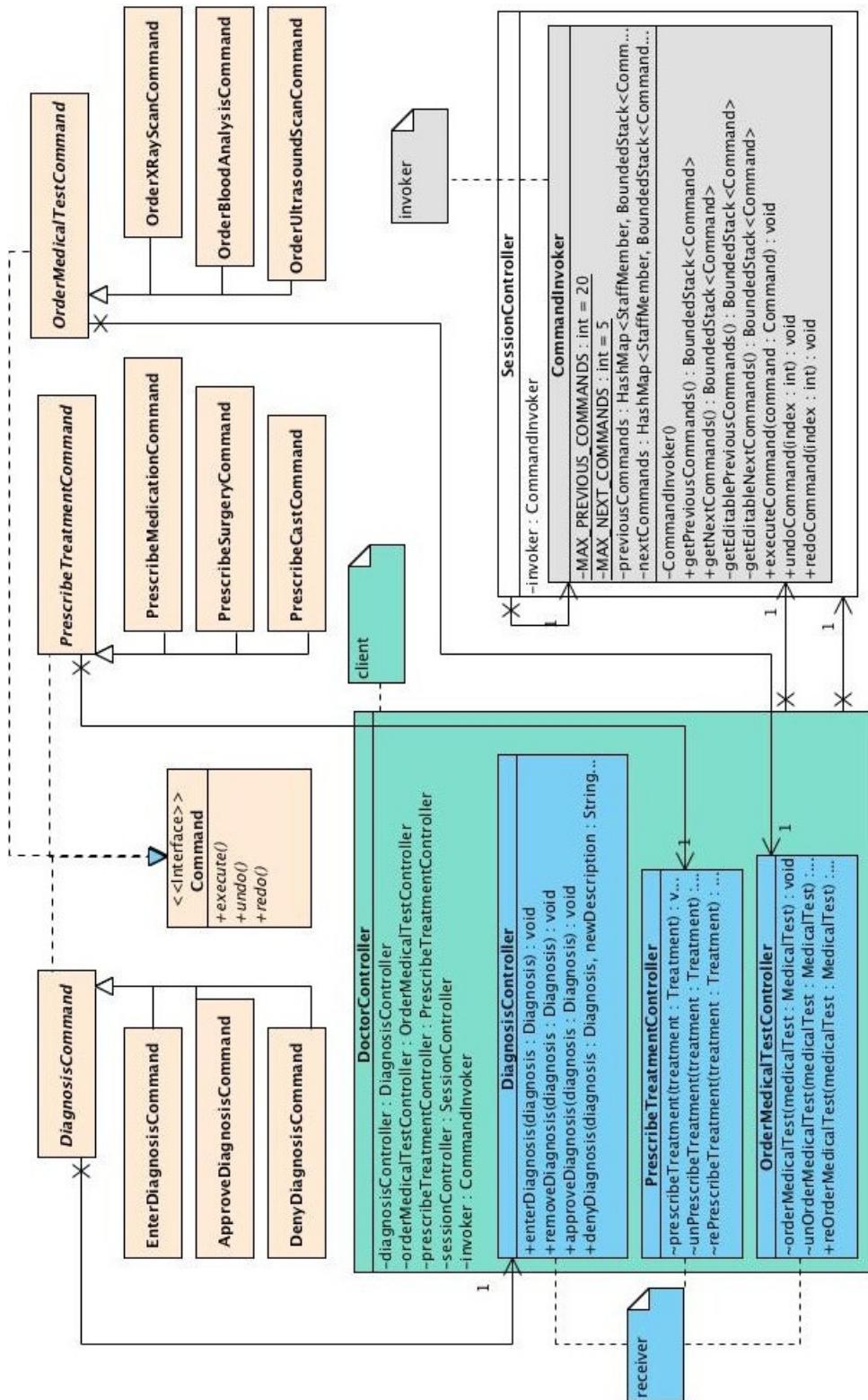
Wat de implementatie betreft, moesten de volgende vier use cases ongedaan gemaakt kunnen worden:

- *Prescribe Treatment*
- *Order Medical Test*
- *Enter Diagnosis*
- *Approve Diagnosis*

Dit hebben we gerealiseerd door drie abstracte klassen aan te maken die een interface genaamd *Command* implementeren. Deze drie klassen zijn superklassen van een heleboel andere klassen, die elk de afhandeling van een specifieke actie voor zich nemen. De exacte onderverdeling is te zien in het klassendiagram. De drie superklassen zijn:

- *PrescribeTreatmentCommand*
- *OrderMedicalTestCommand*
- *DiagnosisCommand*

Voor elke specifieke *Treatment* en *MedicalTest* is er een subklasse van respectievelijk *PrescribeTreatmentCommand* en *OrderMedicalTestCommand*. Al deze *Command* klassen implementeren



Figuur 24: Klassediagram: *Command (Undo/Redo)*

de methodes `execute()`, `undo()` en `redo()`. Deze methodes worden voorgeschreven door de interface `Command`. De `Command` klassen zullen de modellaag niet rechtstreeks aanpassen. In plaats daarvan, en omdat de controllers alle informatie omtrent de acties van een `Doctor` omvatten, zullen de commando's als `receiver` de gepaste controller meekrijgen. Er zijn drie *inner* klassen gemaakt in `DoctorController`:

- `DiagnosisController`
- `PrescribeTreatmentController`
- `OrderMedicalTestController`

Deze drie zijn geïmplementeerd als *inner* klassen omdat ze enkel aangemaakt dienen te worden bij de initialisatie van `DoctorController`. Het is beargumenteerbaar dat dit niet nodig is omdat de drie klassen onveranderbaar zijn binnen één `DoctorController`. We hebben er echter voor gekozen deze toch op te splitsen, zodat de *inner* klassen als `receiver` aan een `Command` object kunnen doorgegeven worden. Hierdoor wordt enkel de nodige functionaliteit meegegeven aan het bijhorende `Command` object en niets meer. De controller zorgt dan, als `receiver`, voor de nodige aanpassingen aan de modellaag.

In de `Command` klasse wordt een specifiek object aangemaakt dat aangepast moet worden, bijvoorbeeld een `Diagnosis`. Dit object wordt dan doorgegeven aan de controller wanneer `execute()`, `undo()` of `redo()` uitgevoerd wordt. De command klasse hoeft dus niet te weten wat er met het object gebeurt in de controller. Dit zorgt voor een zeker encapsulatie van de functionaliteit. Het object dat behandeld moet worden, kan wel best in het `Command` object worden aangemaakt, omdat de methoden `execute()`, `undo()` en `redo()` telkens op hetzelfde object moeten werken. `EnterDiagnosisCommand` moet dezelfde instantie van `Diagnosis` doorgeven aan de `DiagnosisController` bij `execute()` en `redo()`. Er wordt dus binnen één `Command` object telkens met één instantie van een object gewerkt (bijvoorbeeld van `Diagnosis`). Dit object mag dus final zijn.

De `CommandInvoker` voert enerzijds gewoon de `Command` uit, en zet anderzijds het commando in een `Map` met uitgevoerde commando's per `StaffMember`. Wanneer een `Command` wordt uitgevoerd (`execute`), komt deze automatisch in de `Map`. De `invoker` gaat vervolgens kijken in de `SessionController` wie er momenteel is ingelogd. De uitgevoerde `Command` komt in de lijst van uitgevoerde `Commands`, horende bij de ingelogde `StaffMember`. Er werd gewerkt met een `Map`, en niet met één `List`, omdat we niet willen dat een `StaffMember` een `Command` moet bijhouden, wat extra werk is, en een `Command` ongedaan kan maken, die door een andere `StaffMember` is uitgevoerd. De `CommandInvoker` is een interne klasse van `SessionController`, met een private constructor. Enkel de `SessionController` moet een `CommandInvoker` aanmaken, en die `CommandInvoker` hoort dan ook automatisch bij deze `SessionController`. `CommandInvoker` is bijgevolg final in `SessionController`. Dit wil zeggen dat alle gedane `Commands` per `StaffMember` worden bijgehouden; en dit zolang één sessie loopt. Wanneer een `StaffMember` een `Command` uitvoert, uitlogt, en terug inlogt, kan deze nog steeds de `Command` van de vorige sessie `undo`-en. Dit was in samenspraak met onze interpretatie van de opdracht. Wanneer er later meerdere ziekenhuizen aan het systeem zouden moeten worden toegevoegd, en we zeggen dat er een `SessionController` is per `Hospital`, kan er per `Hospital` ook een `Map` van gedane `Commands` per `StaffMember` bijgehouden worden. Dit draait op een positieve manier bij tot de uitbreidbaarheid van ons systeem. Het systeem om een `Command`, dat ongedaan gemaakt is, terug uit te voeren, werkt analoog aan het bovenstaande systeem om een `Command` ongedaan te maken. Er zijn echter kleine verschillen, zo worden er per `StaffMember` slechts vijf `redoable` commands bijgehouden. Het volgende scenario beschrijft één van de rand gevallen van de `Undo-/Redo Action` use case:

1. Een `Doctor` schrijft een `Treatment` voor.
2. De `Treatment` wordt gepland voor de volgende resources: `Nurse A` en `Patient A`.
3. De `Treatment` komt in de lijst van `Treatments` in `Diagnosis` van `Patient A`.
4. De `Doctor` maakt de bovenstaande actie ongedaan.

5. De *Treatment* verliest zijn geplande tijdsperiode, en voor *Nurse A* en *Patient A* komt deze tijdsperiode vrij. De *Treatment* wordt ook uit de lijst van *Treatments* van de *Diagnosis* van *Patient A* verwijderd.
6. Wanneer de *Doctor* nu de bovenstaande actie wil *redo*-en, zal er getracht worden de *Treatment* te plannen op de oorspronkelijke periode; dus de periode die als eerste gegeven werd.
7. Wanneer dit niet lukt, zal er een *exception* gegooid worden. Deze *exception* wordt dan opgevangen in de user Interface.
8. Er kan dan gevraagd worden of de gebruiker een nieuwe periode wil plannen.

In de *CommandInvoker* worden de eigenlijke *Commands* uitgevoerd. Ze zullen dan ook in een lijst met de laatst uitgevoerde commando's gezet worden. Uit deze lijst kan dan op elk moment een commando ongedaan gemaakt worden. Als een commando ongedaan gemaakt wordt, wordt deze in een lijst gezet met commando's die terug uitgevoerd kunnen worden (*redo*). De lijst met laatst uitgevoerde commando's bestaat uit twintig elementen, de lijst met pas ongedaan gemaakte commando's bestaat uit vijf elementen. De *DoctorController* zorgt voor de communicatie tussen de *UserInterface* en de commando's van de *Doctor*. De lijsten met commando's zouden in principe ook bijgehouden kunnen worden in de *DoctorController*. Echter, door deze bij te houden in de *CommandInvoker* is er een duidelijke splitsing tussen de *Controller*-laag en de *Command* interface.

Zie figuur 24 voor het klassediagram.

5.3 Toegepaste GRASP-principes

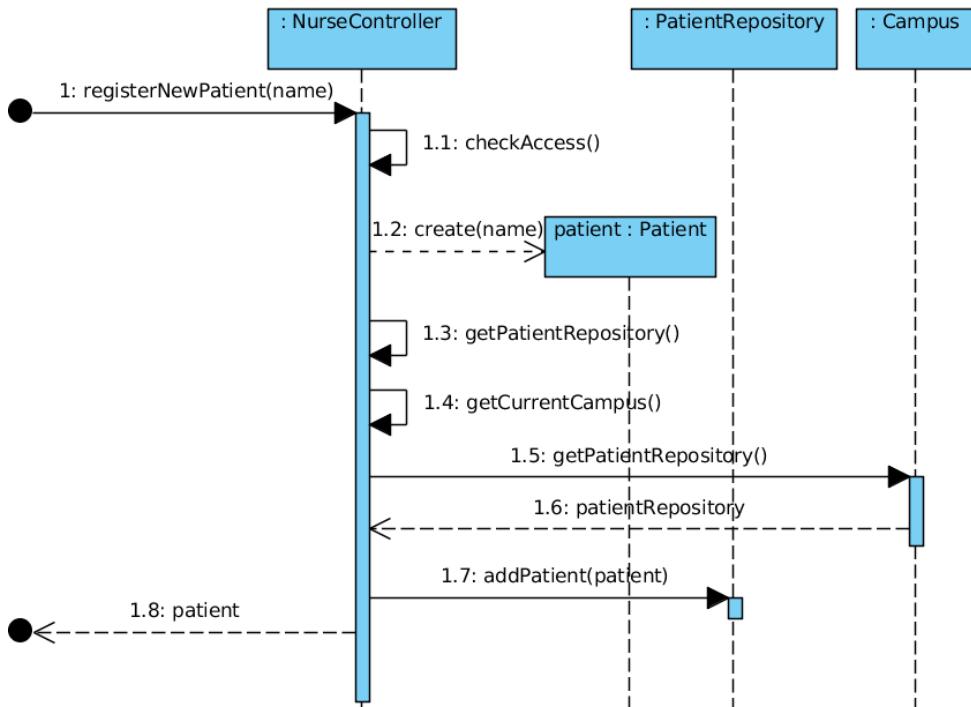
In deze sectie bespreken we voor de belangrijkste use cases de *flow* van hun sequentiediagrammen om een idee te geven van hoe verantwoordelijkheden in het systeem georganiseerd zijn.

Register Patient

In ons ontwerp hebben controllerklassen de verantwoordelijkheid om gebruikers te authenticeren en te autoriseren. Daarnaast mogen ze nieuwe instanties van klassen zoals *Nurse*, *Doctor*, *Patient* en *Machine* aanmaken. Deze klassen komen overeen met basisconcepten uit het domeinmodel en dienen enkel om informatie te bewaren. Ze hebben met andere woorden alleen *getters* en *setters* en zijn niet voorzien van enige andere functionaliteit. Door controllerklassen toe te laten instanties van deze klassen te creëren, blijft het aantal parameters dat door een controllerklasse gedelegeerd wordt, beperkt. Controllerklassen mogen buiten deze klassen natuurlijk geen nieuwe instanties van klassen creëren. Het toekennen van de bovenstaande verantwoordelijkheden aan controllerklassen is niet meer dan een keuze die weliswaar de koppeling tussen de applicatielaag en de domeinlaag versterkt maar tegelijkertijd geen invloed heeft op de koppeling tussen de presentatielaag en de domeinlaag. En het is deze laatste koppeling waar het uiteindelijk om draait. Het alternatief is controllerklassen die niets anders doen dan delegeren.

Met deze opmerking in gedachten beschrijven we nu de *flow* van de use case *Register Patient* (in het geval van een nieuwe patiënt). De methode *addPatient(String name)* van de klasse *NurseController* die op basis van het *Controller* patroon is aangemaakt, wordt aangeroepen. Eerst wordt gecontroleerd of de huidige gebruiker een *Nurse* is en of *name* wel degelijk een *String* bevat. Indien dit niet het geval is, wordt een gepaste *exception* gegooid. In het andere geval maakt *NurseController* op basis van de principes *Information Expert* en *Creator* een nieuw object van de klasse *Patient* aan en geeft het via de methode *addPatient(Patient patient)* door aan de klasse *PatientRepository*. Het object *patient* wordt ten slotte aan de lijst van registreerde patiënten toegevoegd.

Merk op dat het *Creator* patroon hier steunt op het feit dat *NurseController* de initiërende gegevens voor *Patient* bevat. Indien *NurseController* de data had gedelegeerd naar *PatientRepository*, dan was *PatientRepository Creator*; niet alleen omdat het de initiërende gegevens bevat, maar ook omdat het de objecten van de klasse *Patient* bewaart en dus nauw gebruik maakt van deze klasse. Desondanks de argumenten die in de vorige alinea's zijn aangehaald, zou men dus heel strikt kunnen zijn en kunnen stellen dat de klasse *PatientRepository* de verantwoordelijk moet



Figuur 25: Sequentiediagram: *Register Patient*

hebben om instanties van de klasse *Patient* te maken omdat het aan meer voorwaarden van het *Creator* patroon voldoet.

Zie figuur 25 voor het sequentiediagram.

Prescribe Treatment

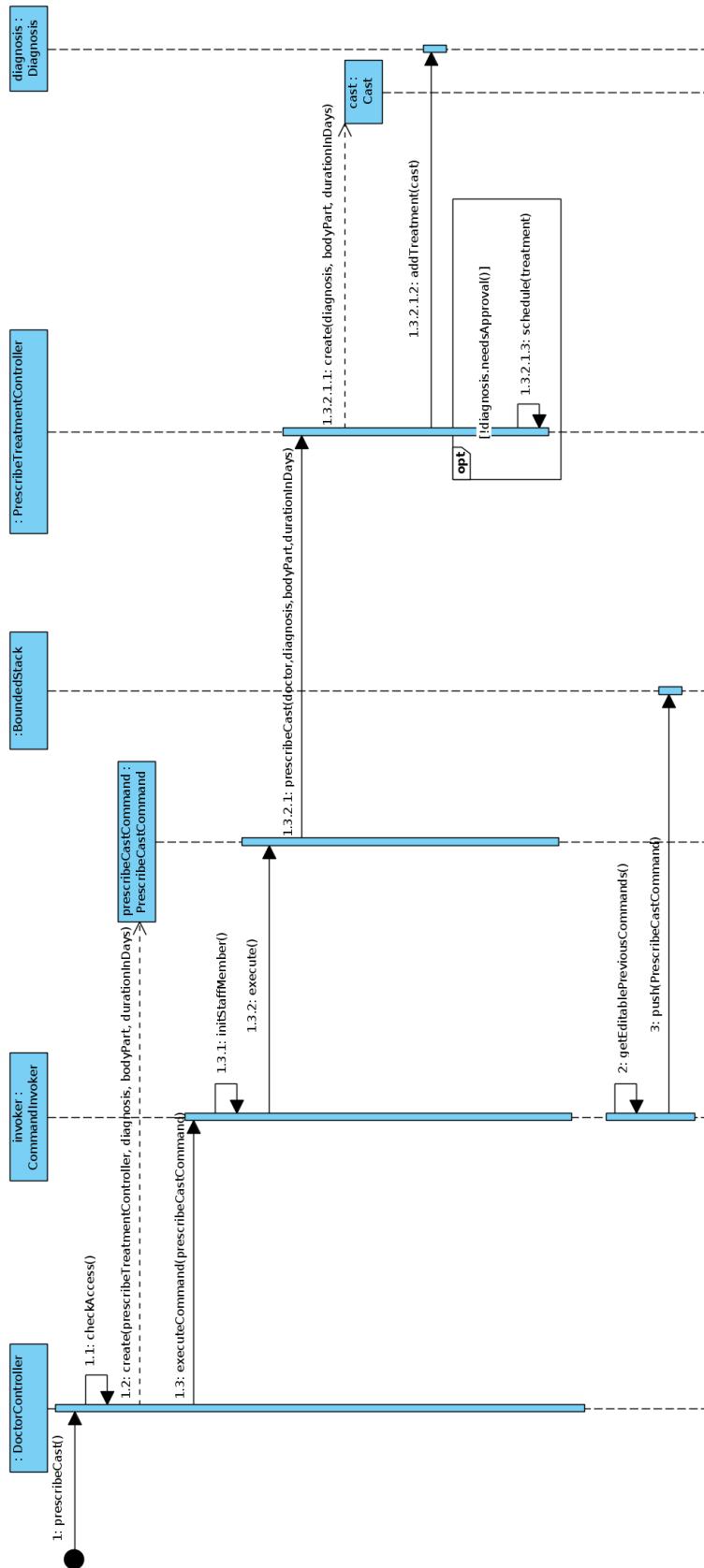
De flow van de use case *Prescribe Treatment* is minder voor de hand liggend dan die van *Register Patient* aangezien deze use case ongedaan gemaakt moet kunnen worden. Daartoe hebben we, zoals eerder aangehaald en uitgelegd, het *Command* patroon gebruikt. Voor de precieze details verwijzen we dan ook naar het desbetreffende onderdeel.

De klasse *Treatment* heeft verschillende subklassen aangezien er verschillende behandelingen mogelijk zijn. In dit voorbeeld bespreken we de use case vanuit de veronderstelling dat het *Treatment* object een object van de subklasse *Cast* is.

De methode `prescribeCast(Diagnosis diagnosis, String bodyPart, int durationInDays, Priority priority)` van de klasse *DoctorController*, die op basis van het principe *Controller* is aangemaakt, wordt aangeroepen. Eerst wordt gecontroleerd of de huidige gebruiker een *Doctor* is. Indien dit niet het geval is, wordt een gepaste *exception* gegooid. In het andere geval worden naar aanleiding van het *Command* patroon eerst een aantal noodzakelijke handelingen uitgevoerd alvorens de eigenlijke use case begint.

De methode `prescribeCast(Doctor doctor, Diagnosis diagnosis, String bodyPart, int durationInDays, Priority priority)` van de *inner* klasse *PrescribeTreatmentController* wordt opgeroepen. Eerst wordt gecontroleerd of de meegegeven argumenten aan de opgelegde voorwaarden voldoen. Mocht dit niet zo zijn, wordt een gepaste *exception* gegooid. Anders wordt op basis van de principes *Creator* en *Information Expert* een instantie van de klasse *Cast* gemaakt. *PrescribeTreatmentController* bevat immers de nodige informatie om dit te doen.

Nadat de nodige handelingen met het *Cast* object zijn uitgevoerd en indien de beschouwde *diagnosis* is goedgekeurd, wordt op bepaald moment de methode `schedule(ScheduleEvent scheduleEvent, TimeStamp begin)` van de klasse *Scheduler* opgeroepen. Deze klasse is een voorbeeld van



Figuur 26: Sequentiediagram: *Prescribe Treatment (Cast)*

het GRASP-principe *Pure Fabrication*. *Scheduler* is bedacht en ingevoerd om het ontbreken van een gepaste klasse voor het plannen van afspraken, medische tests en behandelingen op te lossen. Het maakt gebruik van twee interfaces: *ScheduleEvent* en *ScheduleResource*. De interface *ScheduleEvent* wordt geïmplementeerd door de klassen *Appointment*, *MedicalTest* en *Treatment*, terwijl *ScheduleResource* geïmplementeerd wordt door *StaffMember*, *Machine* en *Patient*. De eerste interface maakt dus een abstractie van de zaken die gepland moeten worden en de tweede van de dingen die daarvoor nodig zijn. Door op deze manier te werken kan de klasse *Scheduler* zo generisch mogelijk gemaakt worden en bijgevolg de koppeling van het systeem verlaagd worden. Dit is dus een toepassing van het *Low Coupling* principe.

De acties die in de methode *schedule(ScheduleEvent scheduleEvent, TimeStamp begin)* worden ondernomen, zijn terug te vinden in het sequentiediagram van figuur 26.

Approve Diagnosis

Net zoals de use case *Prescribe Treatment* moet ook de use case *Approve Diagnosis* ongedaan gemaakt kunnen worden. Uiteraard is hetzelfde patroon gebruikt en dus is dezelfde verwijzing hier op haar plaats. Zie figuur 27 voor het sequentiediagram.

Een diagnosis kan zowel goedgekeurd als verworpen worden. Er zijn met andere woorden twee mogelijke *flows*, die beide duidelijk weergegeven worden in de onderstaande sequentiediagrammen. Zie figuur 28 voor een sequentiediagram van de alternatieve *flow*.

Enter Medical Test Result

We bespreken de use case *Enter Medical Test Result* in het geval dat het *MedicalTest* object een instantie van de subklasse *XRayScan* is. De use case begint met het aanroepen van de methode *registerXRayScanResult(XRayScan xRayScan, String abnormalities, int nbImagesTaken, Priority priority)* van de klasse *NurseController* die een toepassing van het *Controller* patroon is. Indien alle parameters van de methode geldige waarden hebben gekregen, moet er geen *exception* gegooid worden en bevat de controllerklasse alle informatie om een object van de klasse *XRayScanResult* te creëren. Op basis van het *Creator* patroon en in tweede instantie ook op basis van het *Information Expert* patroon kennen we *NurseController* dan ook die verantwoordelijkheid toe. Het verdere verloop van de use case is logisch en is hieronder duidelijk te zien. Zie figuur 29 voor het sequentiediagram.

Undo Previous Action

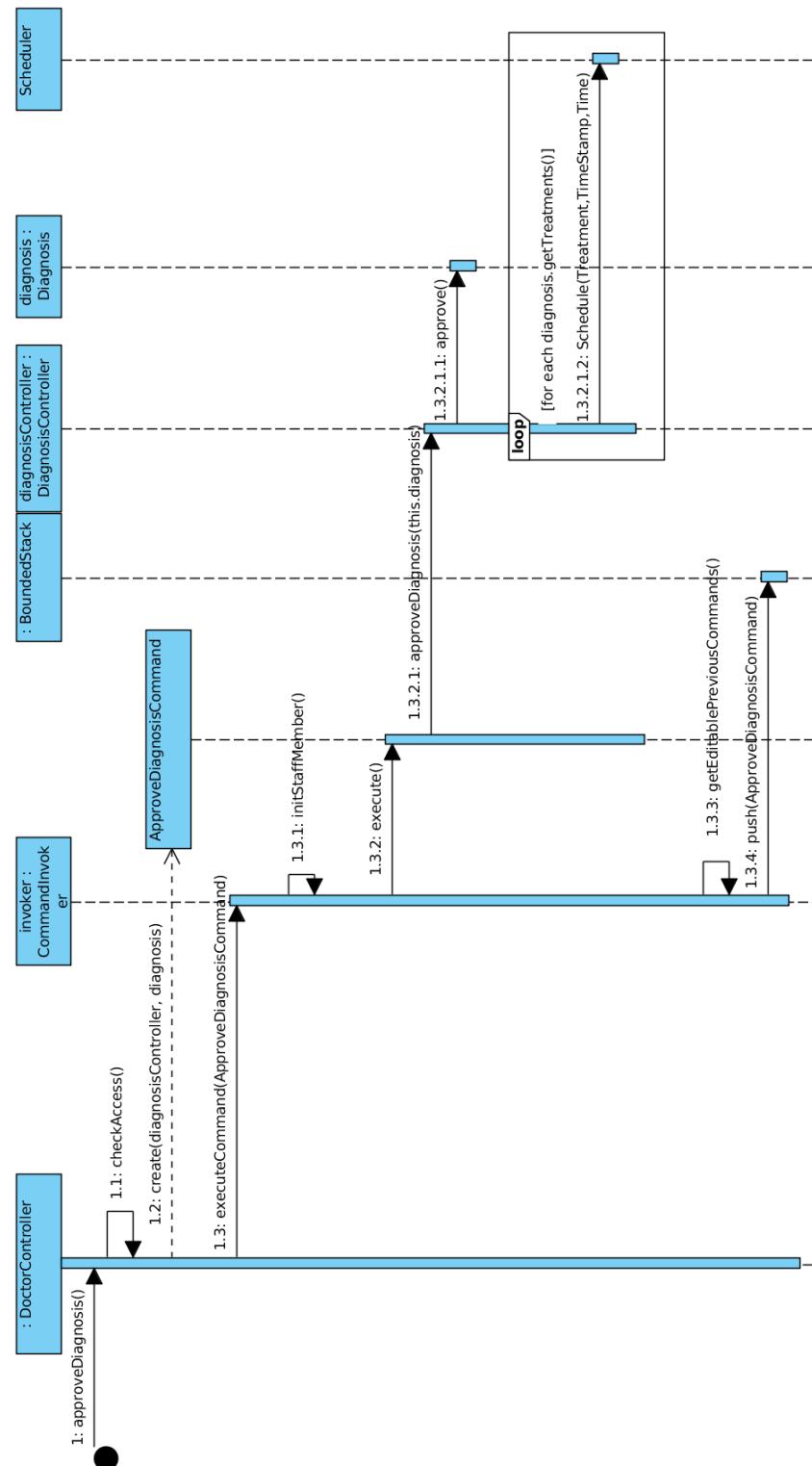
Wanneer een *Doctor* een actie uitvoert, zal deze werken via de *Command* interface. Deze interface schrijft voor dat er een *undo* methode is geïmplementeerd voor dit specifieke commando. Doordat er een interface gebruikt wordt, kunnen er steeds commando's worden bijgemaakt, die de *Command* interface implementeren. De *invoker* werkt slechts met de methodes die geïmplementeerd worden in deze interface. Dit zorgt voor *High Cohesion*. De volledige uitleg rond het *Command Design Pattern* vindt u hierboven in sectie 5.2.4. Zie figuur 30 voor het sequentiediagram.

Advance Time

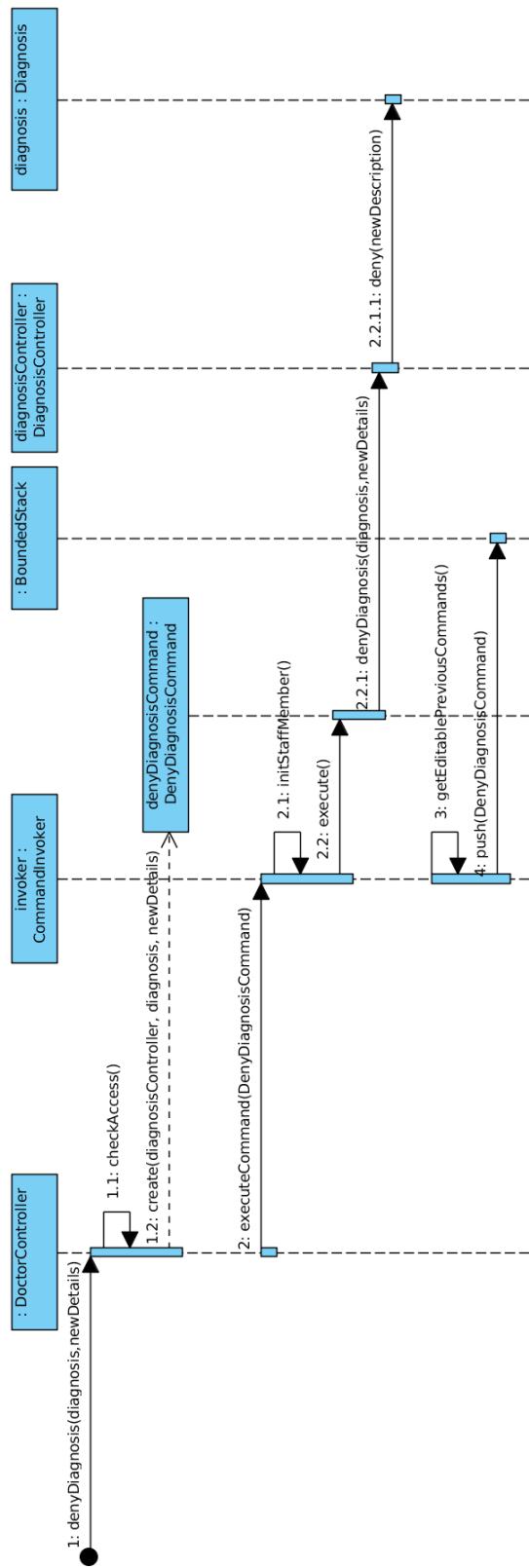
Advance Time is met voorsprong de belangrijkste use case van de tweede iteratie. Ze brengt de belangrijkste subsystemen samen en maakt naast een heleboel GRASP-principes ook gebruik van het *Observer* patroon. Dit ontwerppatroon is reeds in detail uitgelegd en wordt hier dus niet herhaald. We focussen ons daarentegen op enkele ontwerpbeslissingen die steunen op de GRASP-principes.

Zodra de tijd veranderd is en het *Observer* patroon in werking is getreden, wordt de methode *updateFromTime(Object arg)* van zowel *Stock* als *EventHandler* opgeroepen. Bij de eerste klasse gebeuren de volgende twee acties:

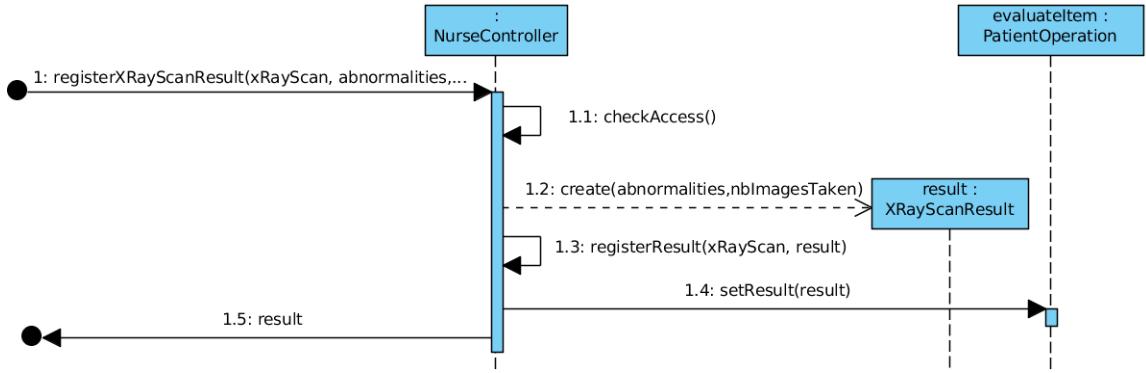
- Vervallen maaltijden en medicijnen worden verwijderd;



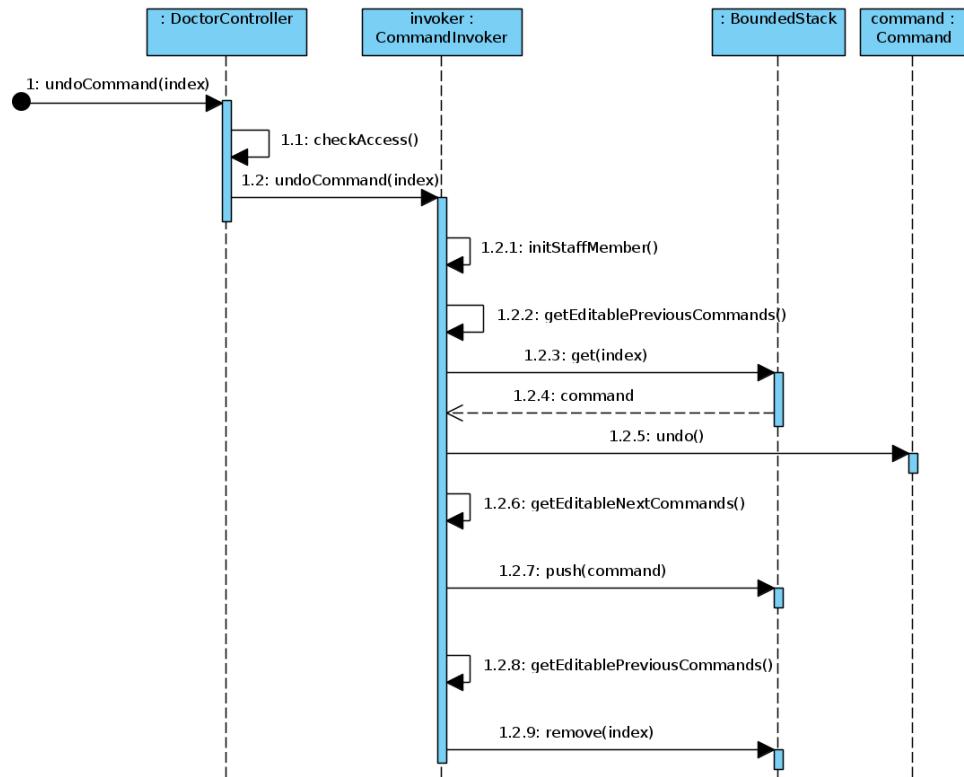
Figuur 27: Sequentiediagram: *Approve Diagnosis*



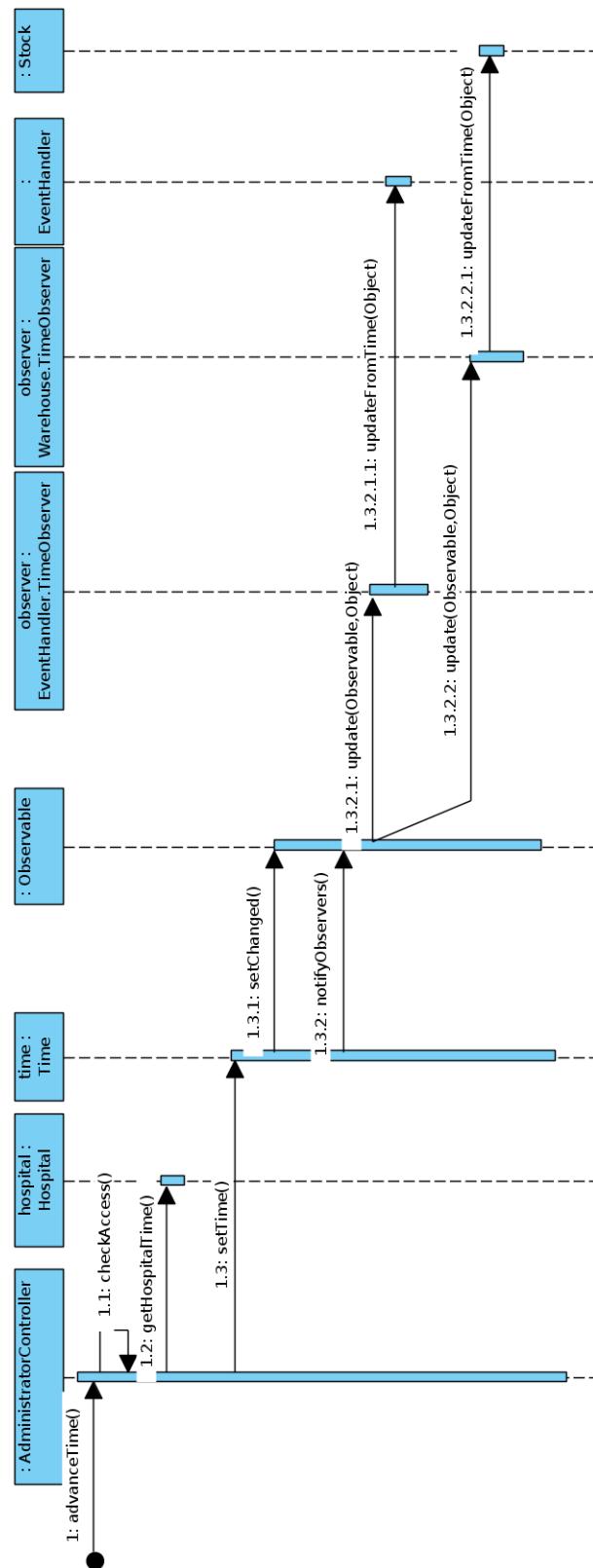
Figuur 28: Sequentiediagram: *Deny Diagnosis*



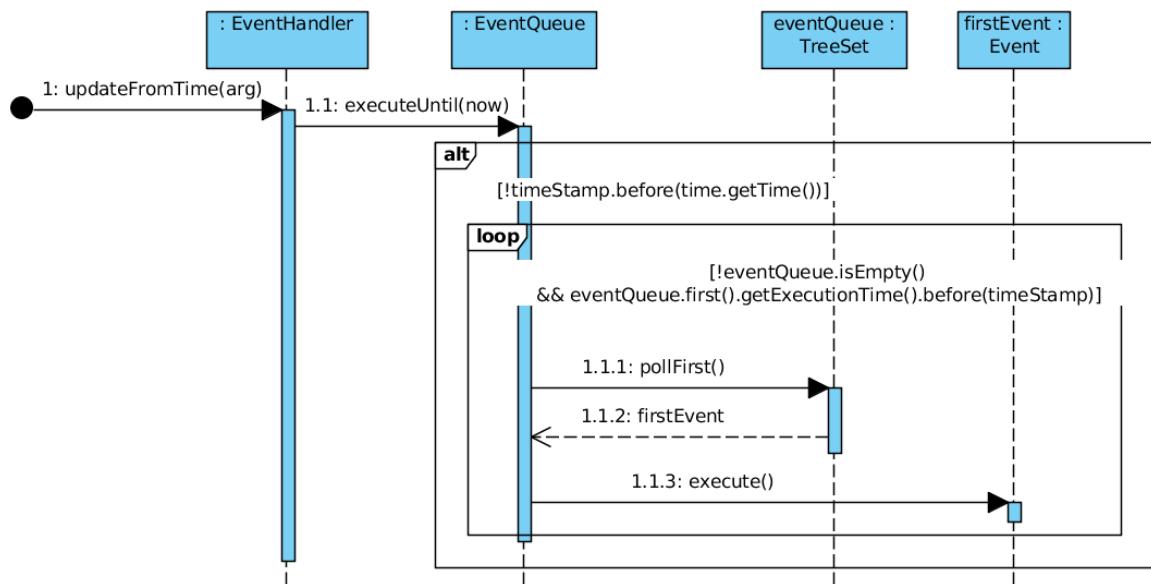
Figuur 29: Sequentiediagram: *Enter Medical Test Result (X-Ray Scan Result)*



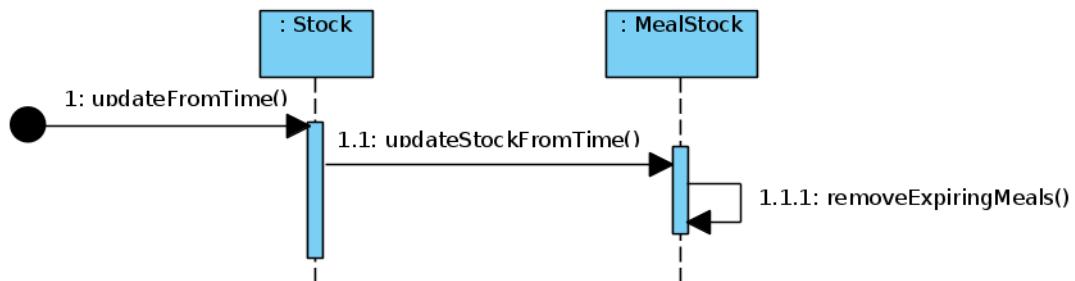
Figuur 30: Sequentiediagram: *Undo Command*



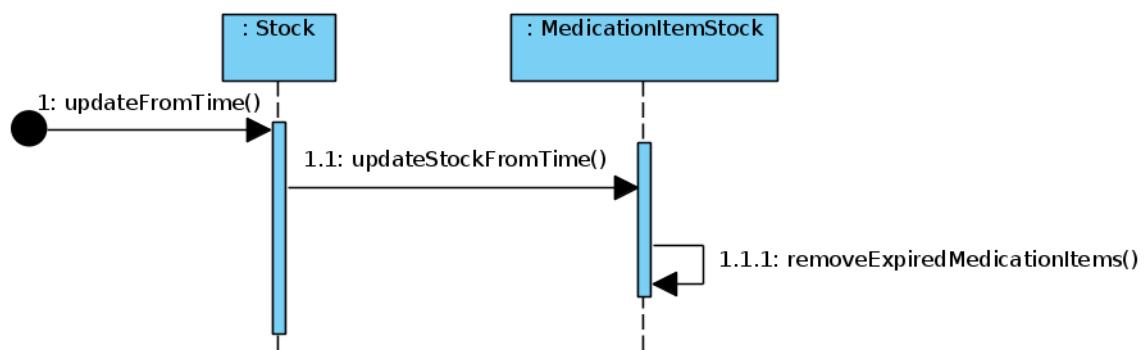
Figuur 31: Sequentiediagram: *Advance Time*



Figuur 32: Sequentiediagram: *Update from Time in Eventhandler*



Figuur 33: Sequentiediagram: *Update from Time in MealStock*



Figuur 34: Sequentiediagram: *Update from Time in MedicationItemStock*

- Indien noodzakelijk wordt een nieuw order voor medicijnen geplaatst.

De abstract klasse *Stock* geeft de eerste verantwoordelijkheid door aan de concrete klassen *MealStock* en *MedicationItemStock* omdat zij instaan voor het bewaren van objecten van respectievelijk *Meal* en *MedicationItem*. Ze zijn dus *Information Expert*. De tweede verantwoordelijk steunt op het feit dat alleen de klasse *MealStock* weet hoeveel maaltijden er nog in stock zijn en dus hoeveel nieuwe maaltijden er besteld moeten worden. *MedicationItemStock* is dus zowel *Information Expert* als *Creator*. Zie figuur 35 voor het sequentiediagram.

Bij de tweede klasse worden de gebeurtenissen op de tijdslijn één voor één uitgevoerd. *EventHandler* is de klasse die de tijdslijn bewaart, en dus *Information Expert*. Merk op dat het automatisch uitvoeren van deze events heel wat moeilijkheden met zich meebrengt. Of beter gezegd, de opgave is onduidelijk over het ontvangen van bestellingen en het afhandelen van medische tests en behandelingen. De use case *Advance Time* wordt uitgevoerd door de ziekenhuisadministrator. Het is echter de magazijnier die orders als ontvangen moet markeren en die de juiste vervaldatums moet invoeren. Een verpleegster moet dan weer de medische tests en behandelingen als uitgevoerd opgeven. Om te voorkomen dat we de use cases *Fill Stock* in *Warehouse*, *Enter Medical Test Result* en *Enter Treatment Result* niet meer zouden gebruiken, hebben we besloten de tijd bij een dergelijk event te stoppen, de juiste use case te laten uitvoeren en ten slotte de tijd weer voort te laten gaan.

Zie figuur 31, 32, 33 en 34 voor de sequentiediagrammen.

6 Uitbreidingen

6.1 Uitbreiding van overervingshiërarchie

Toevoegen van nieuwe medicijnen

Het toevoegen van een nieuw medicijn is eenvoudig. We moeten geen nieuwe methoden toevoegen. Het volstaat om de volgende zaken toe te voegen:

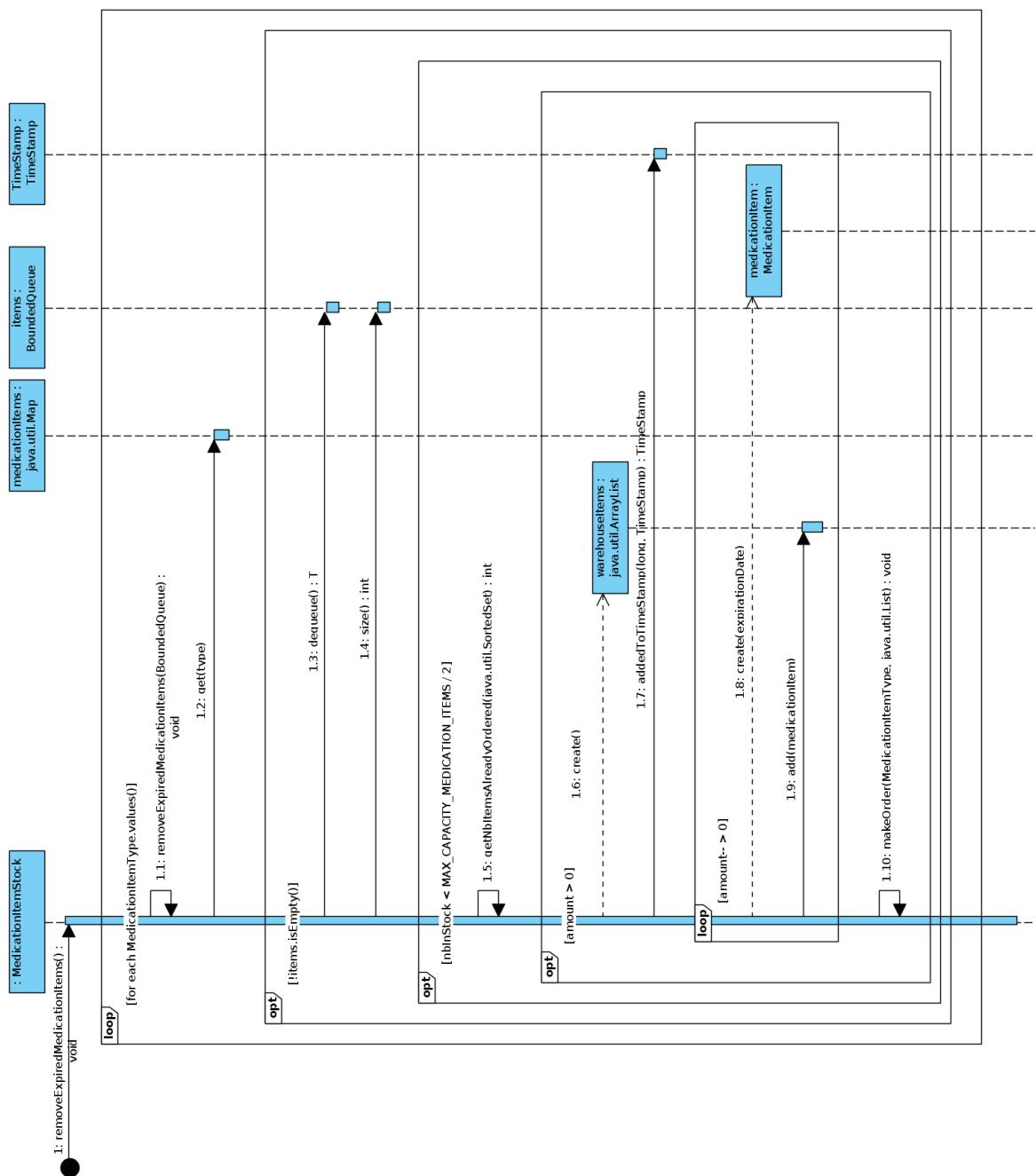
- Een nieuwe waarde in de *Enum MedicationItemType*.
- Een nieuw *key-value* paar in de *Map medicationItems* in de klasse *MedicationItemStock*.
- Een nieuw *key-value* paar in de *Map medicationItemOrders* in de klasse *MedicationItemStock*.

Toevoegen van nieuwe stock items

Tweede iteratie — Om een nieuw stock item dat geen medicijn is, toe te voegen is er meer werk vereist. Veronderstel dat we wegwerphandschoenen toevoegen. De volgende zaken dienen toegevoegd te worden:

- Een klasse *RubberGlove* met de nodige attributen en methoden.
- Een nieuwe *Stack* in de klasse *Warehouse*.
- De methoden *addRubberGlove()* en *removeRubberGlove()* in de klasse *Warehouse*.
- Een klasse *RubberGloveOrder* die overerft van *StockOrder*.
- Een nieuwe *SortedSet* met bijhorende *getter* in de klasse *OrderRepository*.
- Een nieuwe visit-methode in de interface *StockOrderVisitor* en in zijn implementatie *AddingStockOrderVisitor*.

Deze implementatie vereist veel werk omdat er rekening moet gehouden worden met het *Visitor* patroon. Het toevoegen van een nieuwe *ConcreteElement* bij het *Visitor* patroon is immers moeilijk, maar toch is het gebruik ervan te verdedigen. Het patroon verhindert, zoals eerder vermeld, het gebruik van *instanceof* en *switch statements* en zorgt voor code met ‘maximale’ statische *type checking*. Daarenboven wordt ook *casting* voorkomen. Dit zijn drie zeer belangrijke gevolgen die



Figuur 35: Sequentiediagram: *Remove Expired MedicationItems*

dan ook de doorslag gaven in onze ontwerpkeuze. Het *Visitor* patroon stelde ons bovendien in staat gemakkelijk het toevoegen en het verwijderen van *orders* te behandelen. Bovendien kan men stellen dat in een meer realistisch scenario de klassenhiërarchie zelden tot nooit verandert. Een ziekenhuis heeft een vast aantal verschillende items dat het regelmatig moet bestellen. Dit aantal verandert bijna nooit tenzij voor een nieuw medicijn. Maar deze laatste is wel eenvoudig toe te voegen.

Derde iteratie — De doorgevoerde optimalisaties die we in sectie 5.2.2 besproken hebben, zorgen ervoor dat het toevoegen van een nieuw stock item in de derde iteratie veel minder werk vereist. Enkel de volgende zaken dienen nog uitgevoerd te worden:

- Creatie van een klasse *RubberGlove* met de nodige attributen en methoden.
- Creatie van een nieuwe subklasse *RubberGloveStock* van de abstracte klasse *Stock*.
- Implementatie van de abstracte methoden van de klasse *Stock*.

Merk op dat we, desondanks de gedane wijzigingen, nog steeds geen gebruik hoeven te maken van *instanceof*, van *switch statements* of van *casting*. Ook wordt de code nog steeds gekenmerkt door ‘maximale’ statische *type checking*. Bovendien moeten in de derde iteratie geen bestaande klassen aangepast worden.

6.2 Uitbreiding van de opdracht

Undo/Redo-systeem voor Nurse

Wanneer een commando gegeven wordt via de *CommandInvoker*, wordt deze automatisch bijgehouden in de *Map* van commands per *StaffMember*. Als de acties van de verplegers ook via een *command* werken in plaats van rechtstreeks via de controller, en deze *commands* dan aangeroepen worden via de *CommandInvoker*, dan rest er voor de programmeur nog slechts het implementeren van de juiste *Command* en een *Receiver* die de nodige acties uitvoert. De *CommandInvoker* maakt het makkelijk per werknemer een lijst van gedane commands bij te houden, omdat deze heel algemeen is gehouden. De *invoker* zet in zijn map instanties van de klasse *StaffMember* en instanties die de interface *Command* implementeren. Hierdoor wordt er heel algemeen gewerkt en bestaat er toch een verband tussen de werking van het commando en het personeelslid die het uitvoert, de *invoker*.

7 Iteratie 4

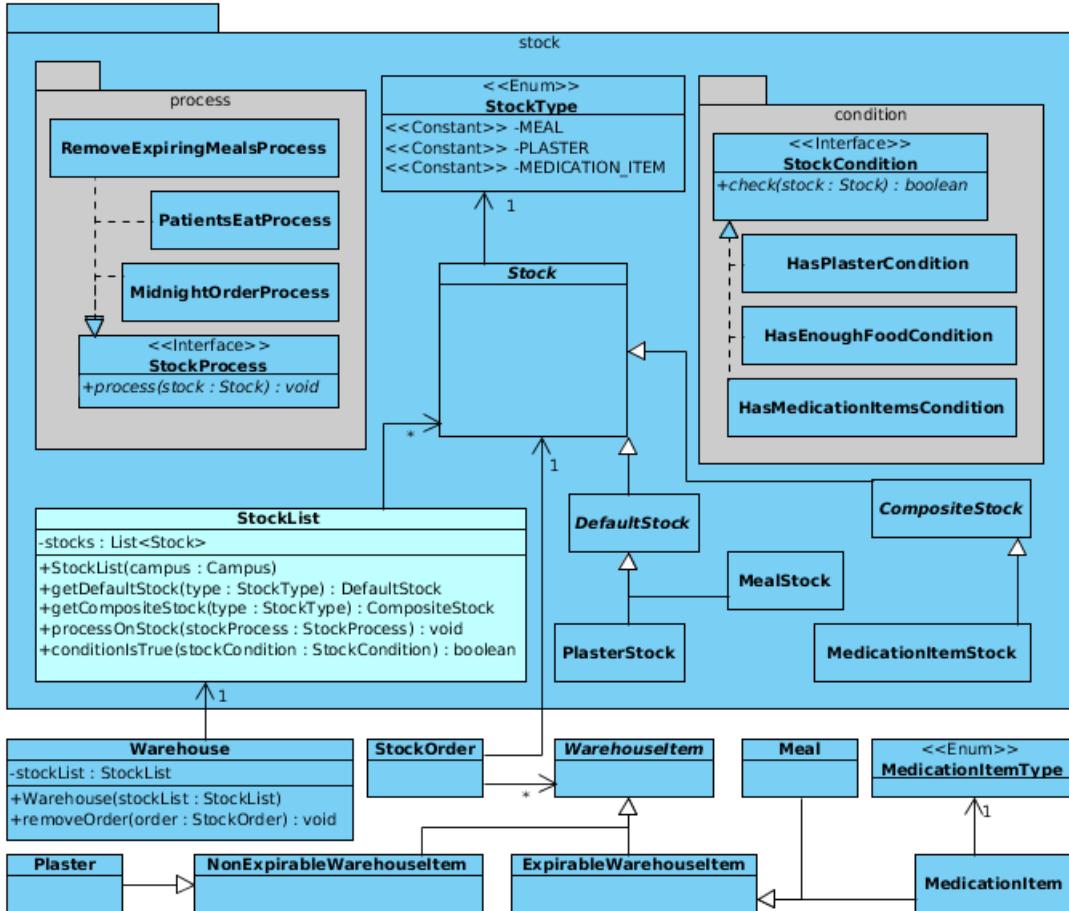
7.1 Doorvoerde aanpassingen

Op basis van de gekregen feedback zijn een reeks aanpassingen gemaakt. De belangrijkste aanpassingen worden hieronder opgesomd en besproken.

Warehouse

Om nieuwe soorten stock items aan het magazijn van het ziekenhuis te kunnen toevoegen zonder de *Warehouse* klasse te moeten wijzigen, is het ontwerp van het voorraadbeheersysteem veralgemeend. De *Warehouse* klasse heeft geen kennis meer van de verschillende soorten stocks maar werkt met een zelfgemaakte datastructuur genaamd *StockList* die alle stocks bijhoudt en toelaat een nieuwe stock toe te voegen voor een nieuw soort stock item. De stock-specifieke methodes van de klasse *Warehouse* zijn verplaatst naar de overeenkomstige *Stock* subklassen. Zij die enkel bepaalde handelingen afwerken, worden uitgevoerd door de methode *processOnStock(StockProcess stockProcess)* van de hiervoor vermelde datastructuur aan te roepen. Het type van de enige parameter van deze methode is een interface die gplementeerd wordt door de klassen *MidnightOrderProcess*, *PatientsEatProcess* en *RemoveExpiringMealsProcess*. Deze drie klassen zijn enkel verantwoordelijk voor het aanroepen van de stock-specifieke methodes: hun objecten stellen dus eigenlijk

Dit is een klassediagram gemaakt voor het illustreren van de belangrijke aanpassingen die gemaakt zijn aan de warehouse package. Enkel operaties en attributen die dit doel helpen bereiken staan er op.



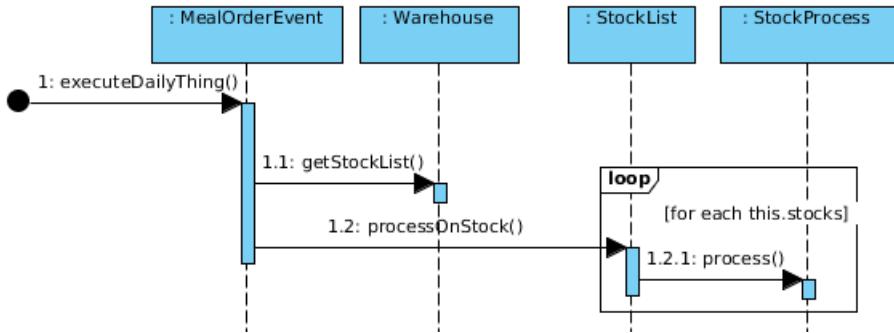
Figuur 36: Klassediagram: *Stockbeheer*

functie-oproepen voor. De stock-specifieke methodes die nagaan of bepaalde voorwaarden voldaan zijn, worden geencapsuleerd in klassen die de *Condition* interface implementeren. Deze klassen zijn *HasEnoughFoodCondition*, *HasMedicationItemsCondition* en *HasPlasterCondition* en kunnen overal waar nodig gebruikt worden. Zie figuur 36 voor het aangepaste klassediagram van stockbeheer en figuur 37 voor een voorbeeld van de *processOnStock*-methode.

Door op deze manier te werken is het toevoegen van een nieuw soort stock item sterk vereenvoudigd en minder gevoelig voor fouten. Het enige wat moet gebeuren is het toevoegen van een nieuwe waarde aan de *StockType* enumeratior en indien er stock-specifieke methoden nodig zijn, ook het aanmaken van een nieuwe *Stock* subklasse en een klasse die de *StockProcess* interface of de *Condition* interface implementeert. Buiten de *Enum StockType* dient er geen bestaande code aangepast te worden, wat de uitbreidbaarheid van het systeem ten goede komt. We kunnen dus stellen dat we ten opzichte van de derde iteratie type-informatie ingeruimd hebben voor aanpasbaarheid en uitbreidbaarheid.

StaffRepository

De klasse *StaffRepository* is ook veralgemeend. *StaffRepository* weet niet langer welke soorten personeel er zijn en heeft geen personeel-specifieke methoden meer. De klasse voorziet nu enkel



Figuur 37: Sequentiediagram: *Execute Daily Thing*

algemene methodes om een bepaald soort personeelslid toe te voegen en te verwijderen, en om een collectie van een bepaald soort personeelsleden op te vragen. Dit is gerealiseerd met behulp van de *StaffType* enumerator die de verschillende soorten personeel bijhoudt.

Het toevoegen van een nieuw soort personeelslid vraagt alleen nog het toevoegen van een nieuwe waarde aan de enumerator en niet meer het aanpassen van de klasse *StaffRepository*, wat bevorderend is voor de aanpasbaarheid en de uitbreidbaarheid van het systeem. We zouden dit laatste in principe nog verder kunnen verhogen door de *Enum* te vervangen door een eigen makkelijk uit te breiden klassenhiërarchie maar dit hebben we gezien de beperkte tijd niet meer geïmplementeerd.

User Interface

Met het oog op een strikte en volledige scheiding tussen de user interface en de domeinlaag hebben we de foutieve rechtstreekse verwijzingen naar domeinobjecten die hier en daar nog in de user interface stonden, weggewerkt met behulp van interfaces.

In plaats van interfaces kunnen ook data transfer objects gebruikt worden, die de domeinobjecten in hun geheel encapsuleren. Het gebruik van DTO's is eigenlijk een betere oplossing omdat hiervoor in tegenstelling tot het gebruik van interfaces nooit casts nodig zijn. Het toevoegen ervan vraagt echter een stuk meer aanpassingen aan de user interface. Om meer tijd aan de uitbreiding te kunnen besteden, hebben we toch voor de eerste optie gekozen.

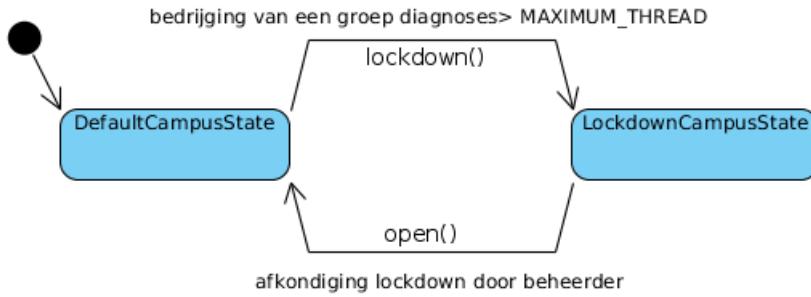
7.2 Uitbreiding ‘Besmettingsgevaar’

Voor de vierde iteratie diende het systeem te kunnen omgaan met epidemieën. Deze uitbreiding zorgt ervoor dat een campus in een *lockdown* kan gaan en er nieuwe voorwaarden gelden in het systeem. De verschillenden deelproblemen worden in de volgende punten behandeld.

Voorstelling van een campus in lockdown

We kunnen stellen dat een campus zich in twee verschillende toestanden kan bevinden, namelijk ‘open’ en ‘lockdown’. Wanneer een campus in de eerste toestand is, verloopt zijn werking normaal. Wanneer een campus in de tweede toestand verkeert, is zijn werking anders. Kortom, de werking van een campus is afhankelijk van zijn huidige toestand.

Als we een campusafhankelijke handeling willen uitvoeren, moeten we dus controleren in welke toestand de betreffende campus zich bevindt. We zouden dit kunnen doen door de mogelijke toestanden van een campus in een *Enum* te definiëren en vervolgens de huidige toestand in de code te bepalen met if-then-else statements en de *Enum*. Dit kan echter resulteren in onleesbare en moeilijk onderhoudbare code. Immers, naarmate er meer campusafhankelijke handelingen uitgevoerd kunnen worden, zijn er meer if-then-else statements en naarmate een campus meer toestanden heeft, zijn de if-then-else statements langer. Deze statements bevatten bovendien



Figuur 38: Toestandsdiagram: *Campus*

vaak dubbele code en indien een nieuwe toestand toegevoegd wordt, vergeet men gemakkelijk een if-then-else statement aan te passen.

Om aan al deze problemen te ontsnappen, hebben we besloten het *State* patroon in te voeren. Dit patroon verhindert niet alleen if-then-else statements, maar zorgt door het expliciet maken van de toestanden of states van een campus ook voor eenvoudige uitbreidbaarheid en verhoogde cohesie.

De implementatie is niet moeilijk. De klasse *Campus* die een campus voorstelt, gebruikt een interface *CampusState* die de campusafhankelijke handelingen als methoden declareert. De methoden van de interface worden geïmplementeerd door twee klassen die elk één van de verschillende toestanden van een campus voorstellen. In deze implementaties staan tevens de overgangen tussen de verschillende states. Het geheel werkt doordat in de methodes van de klasse *Campus* de huidige toestand wordt opgevraagd en vervolgens hierop de overeenkomstige methode wordt aangeroepen. Merk op dat we de interface *CampusState* en de klassen die de interface implementeren, als inner klassen in de klasse *Campus* hebben geëncapsuleerd omdat niemand anders kennis van deze toestanden moet hebben. In het *State* diagram staan de volgende toestanden:

- *Lockdown*: de campus is in lockdown
- *Open*: de campus is niet in lockdown

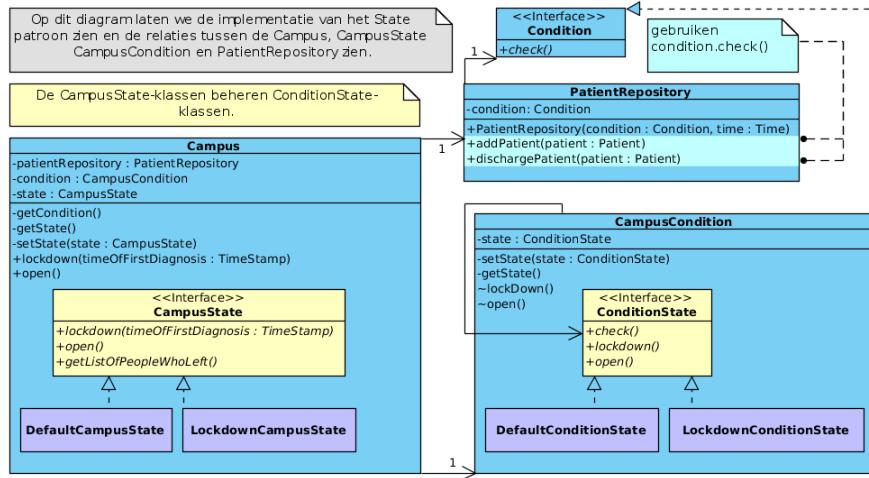
Zie figuur 38 voor het toestandsdiagram.

Werking van een campus in lockdown

Als een campus in lockdown gaat, mag het niet langer mogelijk zijn patiënten te registreren of te ontslaan. Aangezien elke campus een object heeft van de klasse *PatientRepository* waarin alle patiënten van de campus in een collectie worden bijgehouden, zou het *PatientRepository* object op één of andere manier moeten weten wanneer het wel of niet toegestaan is patiënten toe te voegen aan de collectie (= te registreren) of te verwijderen uit de collectie (= te ontslaan).

Een eenvoudige en voor de hand liggende oplossing bestaat erin informatie te verstrekken aan het *PatientRepository* object over de huidige toestand van de betreffende campus. Dit is echter in strijd met het doel van de klasse *PatientRepository* en zou bovendien een zeer sterke koppeling tussen de klassen *Campus* en *PatientRepository* creëren. De enige functie van de *PatientRepository* klasse is het beheer van de collectie patiënten. *PatientRepository* moet en mag dus geen kennis hebben van de klasse *Campus*.

Een betere oplossing is het aanmaken van een interface genaamd *Condition*. Deze interface wordt als parameter aan de constructor van de klasse *PatientRepository* toegevoegd en heeft een enkele methode, namelijk *check()*. Deze methode geeft een boolean terug die *true* is indien alle acties, of beter gezegd alle methoden, van *PatientRepository* zonder restricties mogen uitgevoerd worden, en *false* indien het toevoegen en verwijderen van patiënten niet toegestaan is. De interface



Figuur 39: Klassediagram: *CampusCondition* en *PatientRepository*

Condition wordt geïmplementeerd door de klasse *CampusCondition* die intern gebruik maakt van een State patroonimplementatie die steeds gelijk loopt met die van de klasse *Campus*. Zie figuur 39 voor het klassendiagram van de hierboven beschreven klassen en figuur 40 voor het sequentiediagram van de nieuwe werking van de *registerNewPatient*-methode.

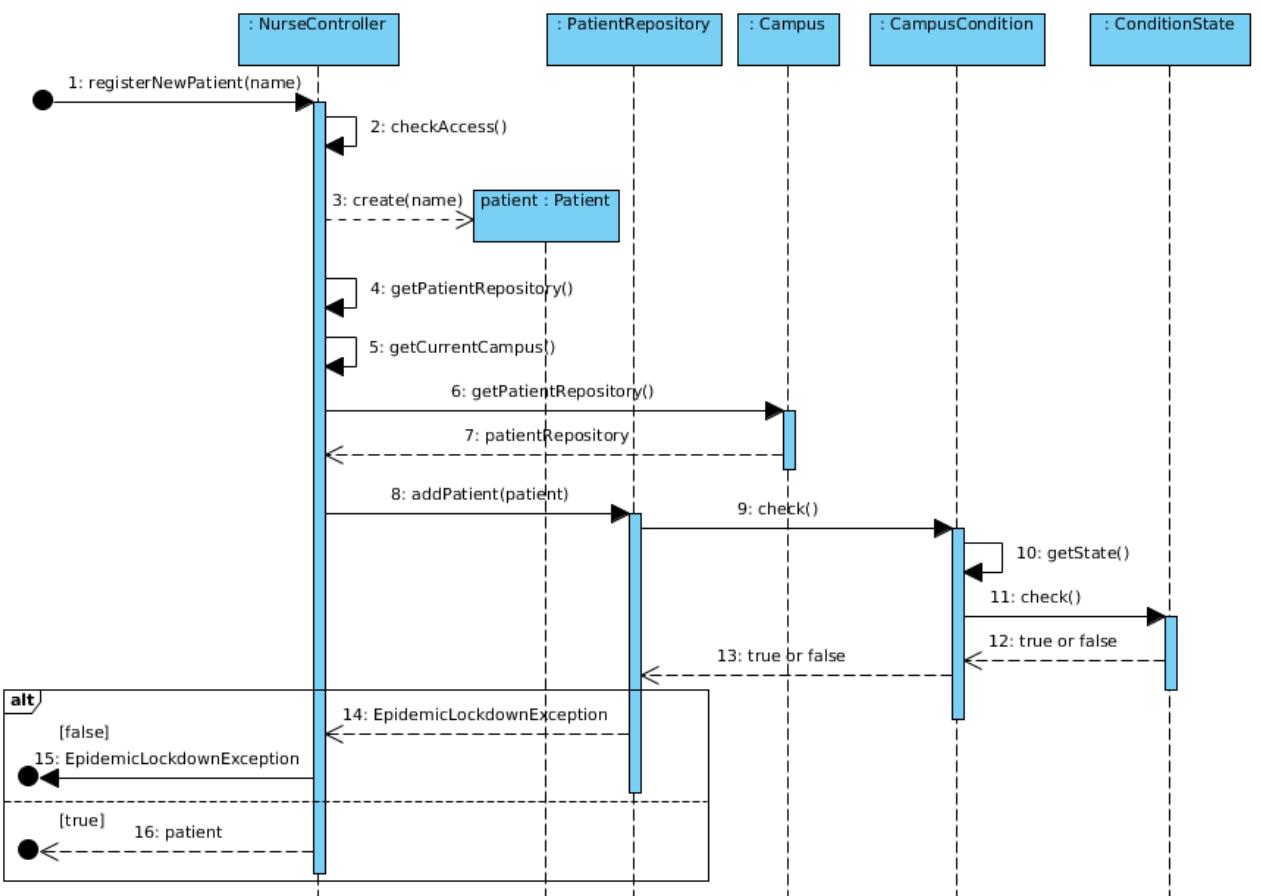
Deze manier van werken vraagt slechts een minimum aan wijzigingen in *PatientRepository*, voorkomt dat de koppeling tussen de *Campus* en de *PatientRepository* klassen wordt verhoogd, respecteert het doel van de klasse *PatientRepository* en bevordert de uitbreidbaarheid en de aanpasbaarheid van de code. Immers, er kunnen steeds andere restricties of juist geen restricties aan de methodes van *PatientRepository* worden opgelegd. Zie figuur 41 voor het aangepaste klassendiagram van *PatientRepository*.

Als een campus in lockdown gaat, moeten ook alle bestellingen voor die campus worden geannuleerd. De verschillende soorten bestellingen worden bijgehouden in de objecten van de overeenkomstige subklassen van de abstracte klasse *Stock*. Deze objecten worden op hun buurt bijgehouden in het *StockList* object van het *Warehouse* object dat bij de betreffende campus hoort.

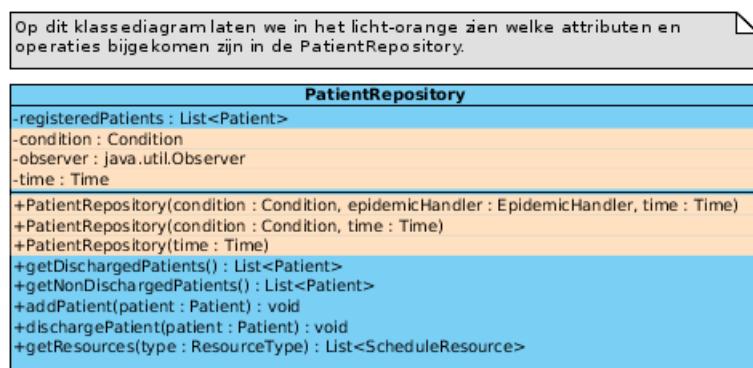
Wat de implementatie betreft laat het ontwerp toe het stukje functionaliteit toe te voegen door enkel gebruik te maken van de bestaande relaties tussen de betrokken klassen en dus het toe te voegen zonder de koppeling tussen de klassen te verhogen. Hiervoor definiëren we eerst een nieuwe methode in de abstracte klasse *Stock* en implementeren we het in de subklassen. Vervolgens voorzien we een nieuwe methode in de klasse *Warehouse* waarin de verschillende methodes van alle Stock subklassen worden aangeroepen. Tenslotte roepen we de nieuw aangemaakte methode van de klasse *Warehouse* aan in de klasse *Campus* wanneer deze laatste in lockdown gaat.

Net als bij het aanmaken van een bestelling moet ook bij het annuleren van een bestelling het overeenkomstige *Event* object dat het ontvangen van de bestelling voorstelt, aan de *EventHandler* doorgegeven worden. *EventHandler* staat in voor het beheer van de tijdslijn van gebeurtenissen en heeft in dit geval de taak de opgegeven gebeurtenis uit de tijdslijn te verwijderen. Om dit te realiseren dient alleen de code in de methode *updateFromWarehouse(Object arg)* in de klasse *EventHandler* aangepast te worden. Het *Observer* patroon uit de vorige iteraties waarbij de klasse *EventHandler* de Java-interface *Observer* implementeert en de subklassen van de abstracte klasse *Stock* de Java-klasse *Observable* uitbreiden, kan zonder enige wijzigingen hergebruikt worden. Zie figuur 42 voor de implementatie van het *Observer* patroon.

Ter herinnering worden de belangrijkste opmerkingen uit de vorige iteraties wat betreft de standaard implementatie van Java voor het *Observer* patroon herhaald. Het nadeel van deze implementatie is dat we in de methode *update(Observable o, Object arg)* het type van *o* en *arg* moeten controleren en eventueel moeten casten. Dit kan eenvoudig opgelost worden door een switch statement of een instanceof. Deze twee manieren zijn echter gevaarlijk. Indien er een

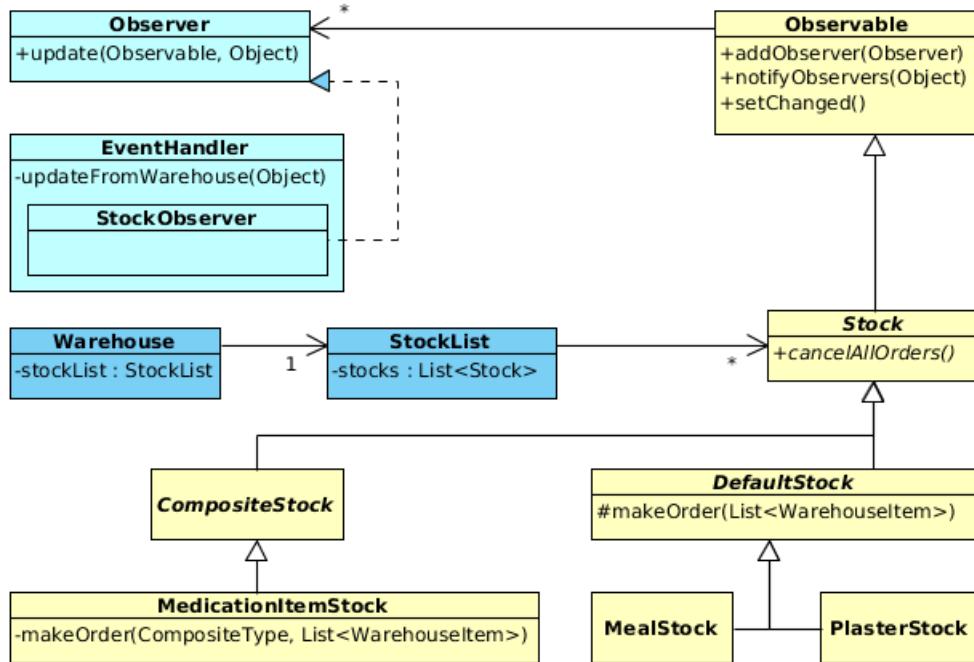


Figuur 40: Sequentiediagram: *Register New Patient*



Figuur 41: Klassediagram: *PatientRepository*

In het geel zien we hier de observables, namelijk de stocks en in het licht-groen zien we de observer ervan in de EventHandler. De warehouse staat ook op dit diagram om de relatie ervan tot de stocks duidelijk te maken.



Figuur 42: Klassediagram: *Observer (EventHandler - Warehouse)*

nieuwe *Observable* subklasse bijkomt, dient een nieuwe case aan het switch statement of een nieuwe instanceof toegevoegd te worden. Dit klein detail wordt gemakkelijk over het hoofd gezien en het vergeten ervan wordt pas opgemerkt at runtime. Een instanceof werkt bovendien ook voor interfaces en geeft true terug indien de klasse een subklasse is van de opgelegde klasse. Dit gedrag is niet gewenst noch zonder risico's.

We hebben dit probleem aangepakt door, in elke klasse die de *Observer* interface implementeert, voor elke mogelijke *Observable* subklasse een overeenkomstige inner klasse te maken die *Observable* uitbreidt en arg aan de juiste methode doorgeeft. In de constructor van de omvattende klasse wordt deze inner klasse aan de Observers van de *Observable* subklasse toegevoegd. Op deze manier wordt de programmeur bij het toevoegen van een nieuwe *Observable* subklasse verplicht een inner klasse aan te maken en vervolgens als Observer aan de subklasse te koppelen. Het onverwacht opduiken van problemen at runtime wordt ook vermeden.

Het type van o hoeft dus niet meer gecheckt te worden. Voor het type van arg moet dit echter wel nog gebeuren. Aangezien arg in ons ontwerp slechts 1 type kan hebben, casten we het object naar het juiste type. Dit is uiteraard geen ideale oplossing: indien arg toch niet van het juiste type is, krijgen we immers pas een foutmelding at runtime. Een betere oplossing zou erin bestaan om zelf een getypeerde Observer interface aan te maken die *Observer* uitbreidt en het type van arg vereist. Dit is dus nog steeds een mogelijk verbeterpunkt.

Verder moest het systeem een lijst kunnen produceren van patiënten en personeel die de campus hebben verlaten sinds de eerste diagnose die deel uitmaakt van de epidemie. In de volgende paragraaf zal de klasse *EpidemicHandler* toegelicht worden. Deze is een entiteit die boven de *Campus* staat, zonder dat deze daar teveel aan gekoppeld is. Verder zullen we zien dat deze instaat voor het beslissen van het moment waarop een bepaalde ziekte een bedreiging vormt. Hiernaast moet deze ook een *TimeStamp* bijhouden van de eerste diagnose van elke ziekte. Deze *TimeStamp* is belangrijk voor het vormen van de gevraagde lijst. Wanneer er zich een bedreiging

voordoet, wordt het tijdstip van de eerste diagnose van de desbetreffende ziekte mee doorgegeven aan de alarmfunctie (over deze alarmfunctie volgt later meer).

Wanneer de *Campus* gealarmeerd wordt en beslist dat deze in een lockdown kan gaan, wordt de *TimeStamp* van de eerste diagnose van de bedreigende ziekte als parameter meegegeven. In het *State patroon* van de *Campus* zit ook de methode *getListOfPeopleWhoLeft()*. Deze methode is enkel aanroepbaar in een *lockdown state* en werpt een uitzondering in de andere staat. De *lockdown state* bevat dus een intern veld van het tijdstip van de eerste diagnose van de epidemie. Aan de hand van dit intern veld wordt in de methode *getListOfPeopleWhoLeft()* de *PatientRepository* afgegaan en gezien welke patient nu gedischarged is en nog aanwezig was na de eerste diagnose van de epidemie. Daarnaast wordt ook de *StaffRepository* afgegaan en worden de personeelsleden toegevoegd die op de campus waren tussen het uitbreken van de epidemie en nu, maar die nu niet meer op de campus zijn. Hiervoor is een methode *wasOnCampus(CampusId id, TimeStamp start, TimeStamp stop)* toegevoegd. De klasse *StaffMember* is verantwoordelijk voor het weten of het personeelslid op de campus was tussen een zekere periode. Deze methode kijkt momenteel naar de shifts van het personeelslid. Het is echter gemakkelijk de implementatie van deze methode te veranderen en bijvoorbeeld te kijken naar het tijdsschema van de personeelsleden. Er is dus geen verandering nodig aan de methode *getListOfPeopleWhoLeft()* wanneer de interpretatie van ‘een personeelslid bevindt zich op een campus als ...’ verandert. Er kan dus besloten worden dat de koppeling hierdoor verlaagt, want de campus hoeft niets te weten van de *Shift* of de *Schedule* van het personeelslid. Deze weet immers zelf wanneer hij op een bepaalde campus is.

Overgang van een normaal werkende campus naar een campus in lockdown

We hebben reeds uitgelegd hoe een campus in lockdown wordt voorgesteld en welke acties ondernomen moeten worden als een campus in lockdown gaat. Het enige wat we nog moeten bespreken is hoe een campus te weten komt wanneer hij in lockdown moet gaan.

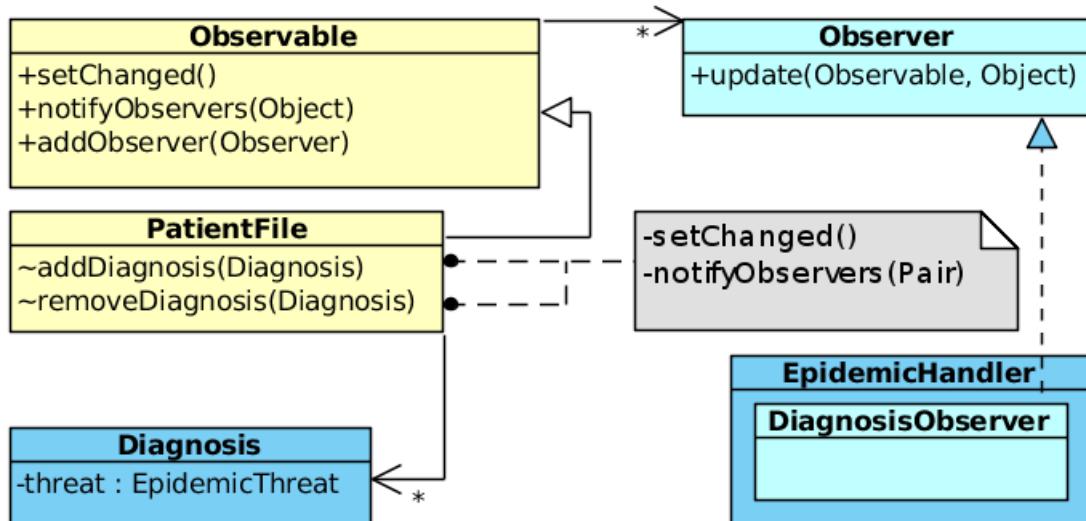
Hiertoe hebben we de klasse *EpidemicHandler* gecreëerd. Deze klasse kan gezien worden als een laag bovenop de klasse *Campus* die een bepaalde campus vertelt wanneer hij in lockdown moet gaan. *EpidemicHandler* heeft een inner klasse genaamd *DiagnosisObserver* die de Java-interface *Observer* implementeert en genotificeerd wordt wanneer een diagnose in een patiëntdossier opgeslagen wordt, of om in het termen van klassen en objecten uit te drukken wanneer een *Diagnosis* in een *PatientFile* van een *Patient* opgeslagen wordt.

Het is met andere woorden de klasse *PatientFile* die de Java-klasse *Observable* uitbreidt. Het nadeel hiervan is dat er een *PatientFile* object voor elke patiënt aangemaakt wordt en dus dat de *DiagnosisObserver* bij elk *PatientFile* object geregistreerd moet worden. We zouden in de plaats hiervan alle diagnoses in een bepaalde klasse kunnen laten aanmaken en deze klasse kunnen laten overerven van de Java-klasse *Observable*. Het voordeel hiervan is dat we de *DiagnosisObserver* bij slechts een object moet registreren. Een aangemaakte *Diagnosis* is echter niet noodzakelijk hetzelfde als een *Diagnosis* die opgeslagen wordt in een *PatientFile*. Vaak moet een aangemaakte *Diagnosis* nog extra informatie krijgen of moet het nog door een tweede dokter goedgekeurd worden. Het is omwille van dit laatste dat we voor de eerste manier van werken hebben gekozen. Zie figuur 43 voor de implementatie van het Observer patroon.

Merk op dat het Observer patroon ervoor zorgt dat de koppeling tussen de klassen *EpidemicHandler* en *PatientFile* minimaal is. Dit laat toe om, indien gewenst, aparte *EpidemicHandlers* voor verschillende epidemieën te maken. De aanpasbaarheid en de uitbreidbaarheid van het systeem worden daarmee uiteraard positief beïnvloed.

Wanneer nu een diagnose in een patiëntdossier opgeslagen wordt, wordt via het Observer patroon de methode *update/Observable o, Object arg*) van de inner klasse *DiagnosisObserver* aangeroepen. Deze methode roept op zijn beurt de methode *updateFromDiagnosis(Object arg)* van de outer klasse *EpidemicHandler* aan. *EpidemicHandler* bepaalt vervolgens de risicoklasse van de diagnose op basis van zijn beschrijving. Er zijn drie risicoklassen: ‘laag’, ‘gemiddeld’ en ‘hoog’. Deze zijn geïmplementeerd als subklassen van de klasse *EpidemicThreat*; meer bepaald als de klassen *LowEpidemicThreat*, *MidEpidemicThreat* en *HighEpidemicThreat*. Van zodra alle diagnoses die tot een bepaalde risicoklasse horen, tezamen een bepaalde risicofactor overschrijven,

In het geel zien we hier de observable, namelijk de Patientfile waarin diagnoses zitten en in het licht-groen zien we de observer ervan.



Figuur 43: Klassediagram: *Observer (EpidemicHandler - PatientFile)*

moet de betreffende campus in lockdown gaan. Dit laatste gebeurt door het aanroepen van de methode *notifyAlarm(Object arg)* van de interface *Alarm* die geïmplementeerd is door de klasse *Campus*. Het gebruik van deze interface zorgt voor abstractie tussen de klassen *Campus* en *EpidemicHandler* zodat, indien gewenst, een *Campus* object gemakkelijk door andere objecten kan ‘gealarmeerd’ worden. Zie figuren 44 en 45 voor de nieuwe werking van de *addDiagnosis*-methode, en figuur 46 voor de nieuwe package ‘epidemic’.

Lage koppeling en hoge cohesie

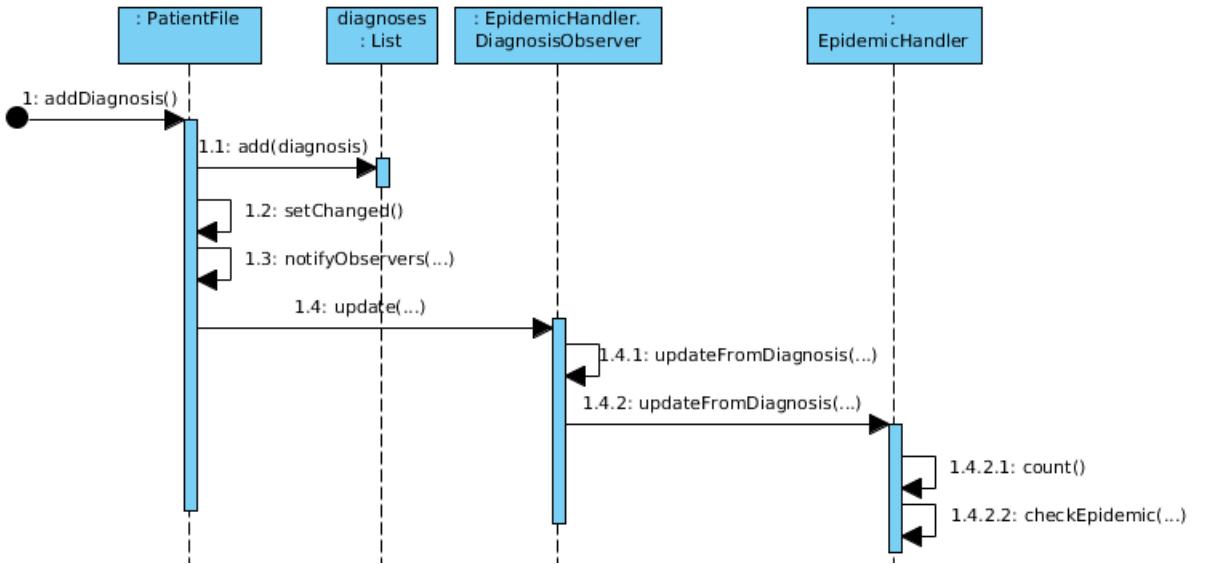
We hebben de belangrijkste aanpassingen aan de code uitgelegd en de verschillende aspecten van het ontwerp van de uitbreiding besproken. Belangrijk hierbij is dat alles rond lage koppeling en hoge cohesie draait. Zoals reeds aangehaald zorgt het *State* patroon, dat we in deze uitbreiding hebben gebruikt, voor een verhoogde cohesie zonder de koppeling te verhogen. Het *Observer* patroon daarentegen zorgt voor een verlaagde koppeling zonder de cohesie te verlagen. Uit het onderdeel ‘Werking van een campus in lockdown’ blijkt verder dat ons ontwerp uit vorige iteraties ons toeliet door alleen nog maar de GRASP-principes *Information Expert* en *Creator* te gebruiken bepaalde aanpassingen en uitbreidingen uit te voeren zonder de koppeling te verhogen of de cohesie te verlagen. Ten slotte is ook de strikte scheiding tussen de user interface, de controllers en de domeinklassen als toepassing van het principe van *Separation of Concerns* een voorbeeld van lage koppeling en hoge cohesie.

8 Tests

8.1 Algemene teststrategie

Onze teststrategie, en bij uitbreiding ook onze algemene aanpak voor de implementatie van het project, is gebaseerd op *Test Driven Development*⁹. Hierbij schrijven we eerst de tests en dan pas de code. Binnen *Test Driven Development* gebruiken we twee soorten testmethodes: black-

⁹<http://www.methodsandtools.com/archive/archive.php?id=20>



Figuur 44: Sequentiediagram: *Add Diagnosis (deel 1)*

en white-box testen. Voor meer informatie wordt verwezen naar het tussentijds verslag van de tweede iteratie.

8.2 Scenario

Door middel van uitgebreide testscenario's testen we de robuustheid van het systeem en in het bijzonder de controllers. Deze testscenario's mogen enkel de controllers gebruiken om met het systeem te spreken, wat ook voor de user interface geldt. De scenario's trachten elke functionaliteit minstens één keer te doorlopen. De scenario's kunnen gecompileerd worden met een *TestSuite* klasse die gebruikt maakt van JUnits Test Suite functionaliteit.

De volgende scenario's werden opgesteld:

Scenario 1: Patient — Dit scenario doorloopt de vele potentiële mogelijkheden die een patient kan ondergaan, o.a. het invoeren van een diagnose, het opstellen van een behandeling, het registreren van resultaten, enzovoort.

Scenario 2: Scheduling — Scenario 2 test de vele functionaliteiten waar planning te pas komt.

Scenario 3: Warehouse — Het derde scenario heeft betrekking tot stockbeheer en de bevoegdheden van magazijnbeheerders.

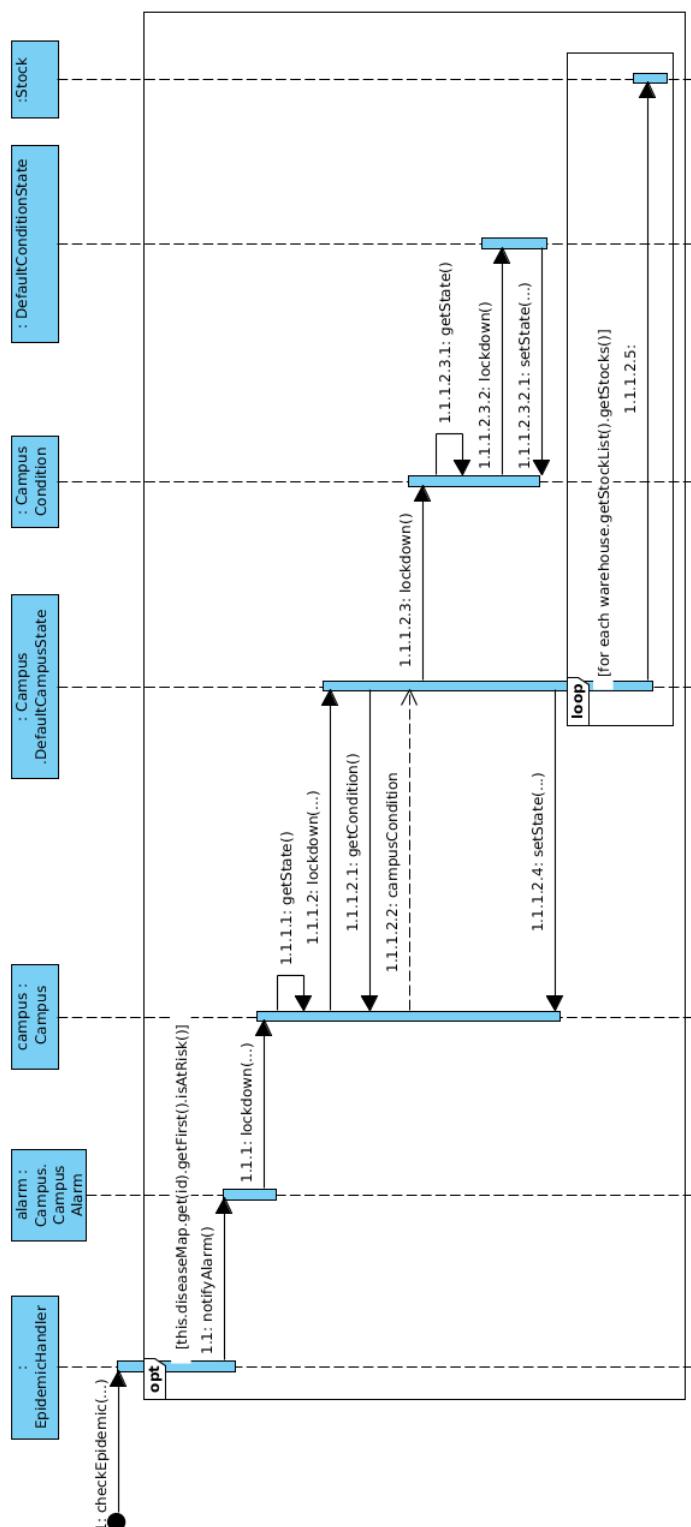
Scenario 4: Campus switching — Het vierde scenario test één van de grootste uitbreidingen van de derde iteratie, namelijk het wisselen tussen meerdere campussen.

Scenario 5: Epidemic — Het laatste scenario test de uitbreidingsfunctionaliteit van de vierde iteratie, het behandelen van epidemieën.

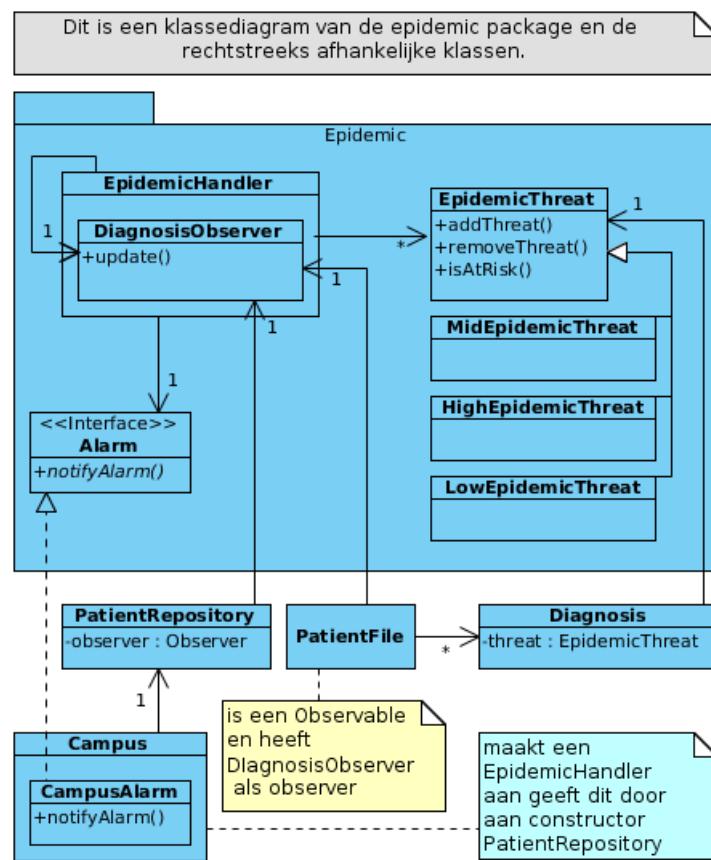
Voor verdere uitleg wordt verwezen naar het tussentijds verslag van deze iteratie.

8.3 EclEmma testrapport

EclEmma geeft je de kans te onderzoeken hoeveel procent van de code je met je testen dekt. Onze testen dekken 83.4% van de code, maar wetende dat je de testen en de *user interface* mag uitsluiten, is de *coverage* ongeveer 92.8%.



Figuur 45: Sequentiediagram: *Add Diagnosis (deel 2)*



Figuur 46: Klassediagram: *Package epidemic*

9 Kritische zelfreflectie

In de loop van dit project hebben we kennis gemaakt met de 23 ontwerppatronen en hebben we ze voor het eerst in een praktische context toegepast. Om een goed en uitgedacht ontwerp te bekomen, hebben we geprobeerd ontwerppatronen enkel te gebruiken in situaties waar ze echt nodig zijn. We hebben dus getracht een juiste ingesteldheid aan te nemen en op die manier te vermijden dat we patronen zouden gebruiken en alleen om patronen te hebben.

Terugkijkend op ons resultaat denken we dat dit aardig gelukt is. We hebben het *Observer* patroon gebruikt in situaties waar één object dezelfde informatie moet doorspelen aan meerdere objecten en niet moet weten welke objecten dit zijn. Een verlaagde koppeling is het gevolg. Het *Command* patroon hebben we ingevoerd om een undo/redo-systeem te ontwikkelen dat uitbreidbaar is zonder bestaande klassen aan te passen. Het object dat een operatie aanroept, is bovendien ontkoppeld van diegene die weet hoe de operatie uitgevoerd wordt. Het *Visitor* patroon dat we in de tweede iteratie gebruikten om *instanceof*, *switch statements* en *casting* te vermijden en om code met ‘maximale’ statische *type checking* te bekomen, is vervangen door een beter alternatief. Dit alternatief maakt gebruik van het *Template Method* patroon en biedt naast de voordelen uit de vorige iteratie ook nieuwe, bijkomende voordelen. Bovendien is ook de koppeling verlaagd. Het *State* patroon ten slotte zorgt voor overzichtelijke code en voorkomt te veel en/of te lange *if-then-else statements*; met een verhoogde cohesie tot gevolg.

Natuurlijk is ons ontwerp nog voor verbetering vatbaar. We denken hierbij vooral aan het voorraadbeheerprobleem dat we in deze iteratie reeds optimaliseerden maar dat we, zoals we eerder al aangehaald hebben, nog verder kunnen bijschaven. Desondanks dit mogelijk verbeterpunt en alle andere en kleinere verbeterpuntjes die in de loop van dit verslag vermeld zijn, zijn we overtuigd dat we, alles in rekening brengend, een degelijk project afleveren.

10 Projectbeheer

Tijdsverdeling

Iteratie 3

	Francis	Olivier	Thijs	Vincent
Code	37	45	24	51
Documentatie	4	9	4	1
UML	18	1	0	1
Testen	21	3	41	30
Verslag	0	26	11	3
Totaal	81	84	80	86

Iteratie 4

Aan de vierde iteratie hebben we een week besteed. Hierbij heeft Francis zich geconcentreerd op de code en de UML-diagrammen, heeft Olivier zich toegelegd op het ontwerp en het verslag, heeft Thijs zich gefocust op de code en de testen en heeft Vincent zich toegelegd op het ontwerp en de code.

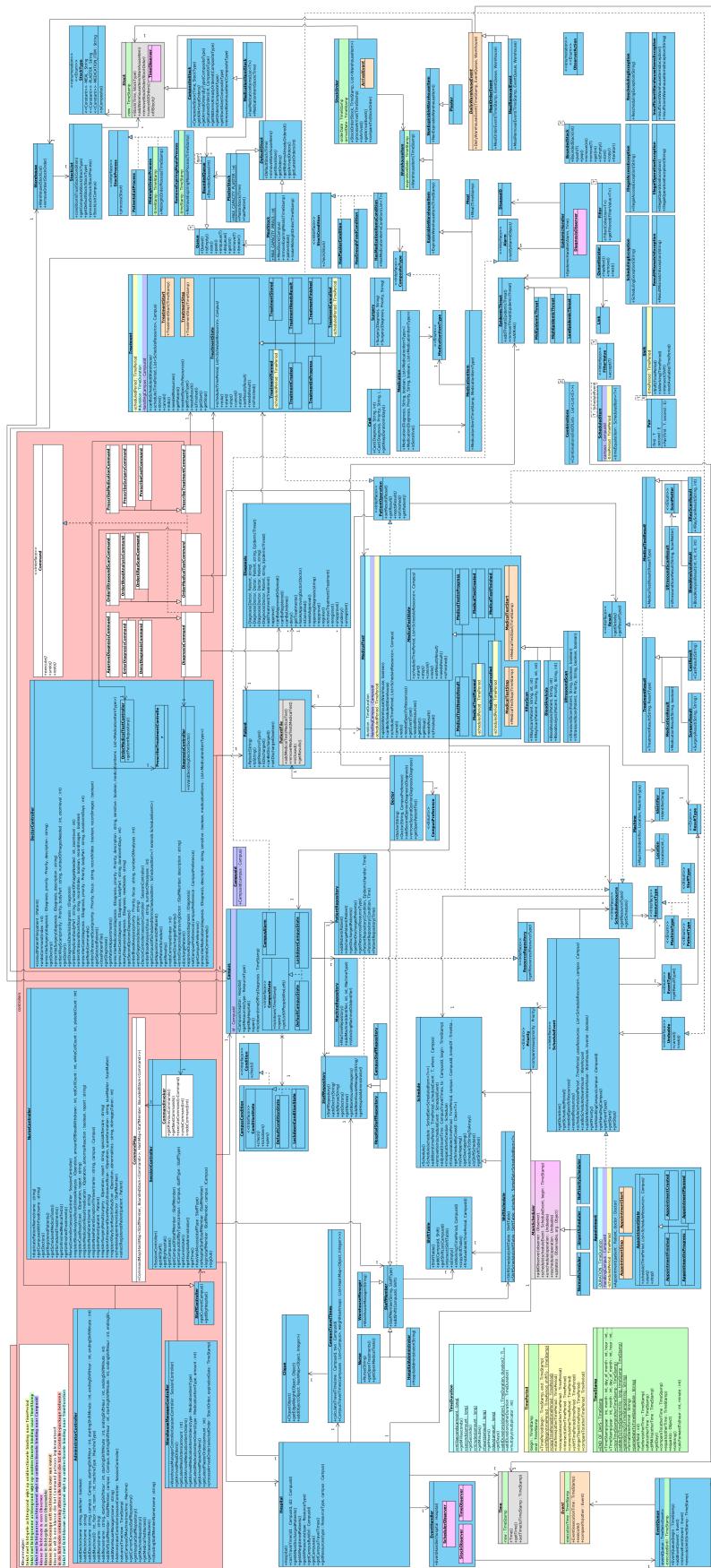
11 Conclusie

Dit project heeft ons enorm veel tijd en moeite gekost maar heeft ons ook enorm veel bijgeleerd. Door ons de kans te geven een softwaresysteem volledig te ontwerpen en te implementeren hebben we niet alleen de GRASP-principes en de GoF-ontwerppatronen leren toepassen maar hebben we ook in het algemeen meer inzicht gekregen in het objectgeoriënteerd paradigma. Bovendien hebben we in de loop van deze opdracht ook beter in groep leren werken en beter met tijds- en

werkdruk leren omgaan. Kortom, dit project was een leerrijke ervaring en heeft onze praktische vaardigheden voor het latere beroepsleven sterk verbeterd.

A Bijlagen

Volledige klassediagram



54
Figuur 47: Volledig klassendiagram