SWOP - Hospitaal Iteratie 3

Groep12

Jeroen Van Gool
Ruben Lapauw
Tom De Bie
Jeroen De Coninck

Inhoudsopgave

1 Inleiding

1.1 Overzicht van het verslag

In dit verslag bespreken we het design van een uitgebreid software-systeem ontworpen voor het management van een hospitaal. Het systeem biedt momenteel ondersteuning voor een verscheidenheid aan gebruikers, zijnde dokters, verpleegsters, magazijnbeheerders en de hospitaalbeheerder. Patiënten kunnen geregistreed worden in het systeem waarna er diagnoses, medische testen en behandelingen voor hun aangemaakt kunnen worden. Verder biedt het systeem ook mogelijkheden voor het beheer van machines in het hospitaal een het plannen van afspraken tussen hospitaalpersoneel, machines en patienten.

In deel 2 van het verslag, 'Het systeem', geeft men eerst een klein overzicht van ons Hospitaalsysteem. Daarna wordt er één voor één op elk subsysteem gefocust. Voor elk subsysteem begint men eerst met een beschijving van het systeem, waarna men volgt met de bespreking van een relevante use case, waarmee men het gebruik van het systeem illustreert.

Deel 4 bevat een conclusie met een reflectie over de sterke en zwakke punten van de implementatie. Tenslotte in deel 5 zijn alle appendices verzameld, deze bevatten informatie die in het verslag hoort, maar het geïmplementeerde basissysteem niet bespreken (met uitzondering van het volledige klassendiagram natuurlijk).

1.2 Veronderstellingen

- "Een patiënt kan aan niet meer dan 10 X-ray scans per jaar onderworpen worden." Hierbij hebben we natuurlijk aangenomen dat het over de tijdspanne van een jaar gaat, en niet over een kalenderjaar. Dit is natuurlijk gemakkelijk aanpasbaar.
- Personeelsleden alsook patiënten hebben een unieke naam. Je kan echter wel een patiënt hebben met dezelfde naam als iemand van het personeel, dit leek ons logisch aangezien een personeelslid ook opgenomen kan worden in het ziekenhuis als hij of zij zelf ziek is.
- De FIFO-queue wordt nu gebruikt zoals gevraagd. Dit was geen probleem met de implementatie.
- Bij het doorspoelen van de tijd wordt als het eten op is geen eten meer gegeven aan de patiënten.
- De Stock van alle items is voldoende groot dat er niet meer items gevraagd worden dan dat het minste aantal items die in een Stock kan ziten zonder dat er bijbesteld moet worden. Met andere woorden, deze is voldoende groot dat 2 dagen na de laatste afspraak altijd voldoende items zullen zijn.
- Als de tijd doorgespoeld wordt, wordt er gevraagd om medical tests en treatments tot op de vorige dag in te vullen. We gaan niet tot de dag zelf om de nurses ook nog een kans te geven resultaten in te vullen.

2 Het systeem

2.1 Overzicht

Het hospitaal heeft verschillende subsystemen: De wereld die als oer-Object dient. Deze houdt de tijd, personen, machines en campussen bij, de campussen houden elk hun eigen voorraden bij. De personen splitsen zich op in patiënten en personeel. Het personeel kan verschillende operaties uitvoeren op het systeem, waarvan de volgende op patiënten toegepast worden: voeg diagnoses, medische testen en behandelingen toe. Medische testen en behandelingen hebben beiden een afspraak die als alle andere precondities voldaan zijn aangemaakt worden. Verder is er ook voorraad; deze voorziet maaltijden voor de patiënten en andere items voor de behandelingen. De bestellingen gebeuren hiervan automatisch. Aangekomen bestellingen worden niet automatisch verwerkt.

2.2 Gebruikers

2.2.1 Beschrijving

Het login-subsysteem is gecentreerd rond het concept van "LoginControllers". Een LoginController in ons Hospitaal-systeem is een object dat een gebruiker representeert voor de duratie dat deze gebruiker aangemeld is, het object wordt dus geïnvalideerd zodra de gebruiker zich afmeldt. Naast de huidige gebruiker te identificeren in verscheidene functies biedt een LoginController ook methoden aan om informatie op te vragen over de gebruiker of om bepaalde acties uit te voeren op informatie waartoe het gebruikers-object toegang heeft.

De klasse LoginController wordt zelf nooit geïnstantieerd: enkel subklassen worden gebruikt. Deze subklassen zijn elk gespecialiseerd in een bepaalde rol in het ziekenhuis, zo zijn er bijvoorbeeld DoctorControllers en NurseControllers die functionaliteit aanbieden specifiek aan respectievelijk dokters en verpleegsters. Natuurlijk zijn er ook LoginControllers voor elke andere personeelsrol in het hospitaal.

Het grote voordeel van deze organisatie van LoginControllers is dat we deze objecten nu ook kunnen gebruiken om ervoor te zorgen dat bepaalde acties enkel uitgevoerd worden door bevoegd personeel. Door bij deze acties een specifieke subklasse van LoginController te eisen kan dit verzekerd worden. Een geldige LoginController voor een bepaalde rol kan namelijk enkel verkregen worden via de correcte uitvoering van de aanmeld-procedure.

2.2.2 Usecases: Login, Logout

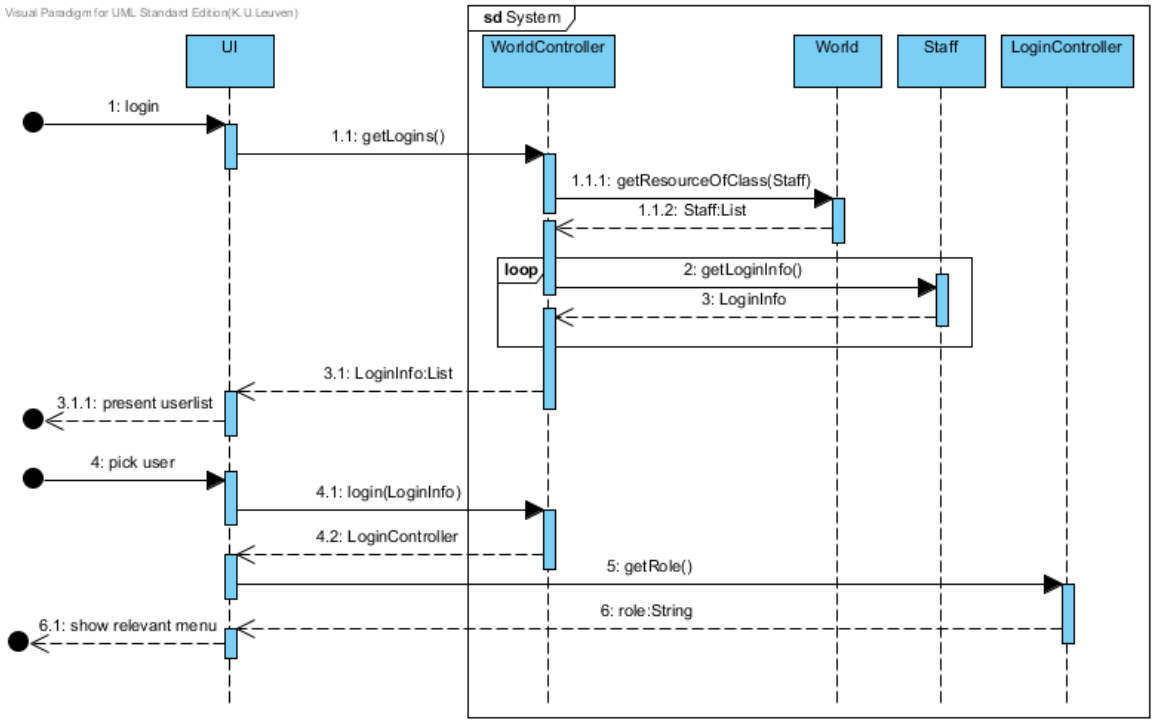
De gebruikers (het personeel) kunnen zich aanmelden op het systeem, bij het aanmelden kunnen ze kiezen op welke campus ze aanwezig zijn. Dit gebeurt aan de hand van een lijst van CampusInfo-objecten die via een WorldController opgevraagd worden aan de World. CampusInfo is een object dat enkel de informatie bevat die nodig is om een Campus te identificeren (in dit geval de naam). Met dit identificatie-object kan de `login`-methode van de WorldController de juiste campus opvragen om een CampusController voor te maken. Deze CampusController wordt doorgegeven voor gebruik door LoginControllers, zoals bv. de DoctorController of de NurseController. Na dat de gebruiker klaar is met het systeem kan men de gebruiker eenvoudig afmelden door de methode `logout()` van het LoginController-object aan te roepen. Deze invalideert het object waardoor het niet meer gebruikt kan worden.

Om het gebruik van deze LoginControllers te illustreren zullen we de procedure voor het aanmelden eens dichter bekijken (zie figuur 1). Indien we het systeem als black box bekijken zien we dat we slechts twee API-calls nodig hebben om zich aan te melden, de eerste om een lijst van alle gebruikers in het systeem te krijgen (een LoginInfo bevat alle informatie om een gebruiker te identificeren in het systeem: de naam en de rol), de tweede voor het echte aanmelden en daarmee ook het verkrijgen van een LoginController-object. In het getoonde system sequence diagram is nog de extra stap getoond waarin de user interface de rol van de gebruiker opvraagt en daarop gebaseerd het juiste menu kiest om te tonen, meer over de UI is de vinden bij de appendices, in sectie 6.1.

Nu dat we een LoginController hebben kunnen we deze meegeven aan andere controllers die onze LoginController vervolgens kunnen gebruiken voor acties die een gebruiker in een bepaalde rol nodig hebben; bijvoorbeeld een MedicalTestController die een DoctorController vereist om medische testen te kunnen plannen.

2.2.3 Bespreking GRASP, uitbreidbaarheid en nadelen

Door de personen geabstraheerd als Schedulable bij te houden in de wereld verliest deze een hele hoop koppelingen met de personen. Er moet nu wel iedere keer gefilterd worden op klasse om een groep personen op te vragen van een bepaald type. Het voordeel is nu wel dat men een heel flexibele methode heeft om de personen te filteren: men kan bijvoorbeeld alle Personeelsleden filteren door Staff.class te gebruiken. Als men verschillende klasses opvraagt en dan de lijsten merged kan men elke combinatie van groepen maken, zonder dat de wereld moet veranderen. Deze lage koppeling zorgt voor een uitbreidbaar design, er kunnen gemakkelijk nieuwe types toegevoegd worden zoals de Warehousemanager. Het gebruik van de CampusInfo-objecten zorgt ervoor dat de gebruiker van de API nooit een echt Campus-object te zien krijgt, een hele hoop koppeling is dus vermeden hierdoor. Verder zorgt deze aanpak ervoor dat er geen ongeöorloofde toegang tot de Capmus-objecten is.



Figuur 1: Use-case: Login

Er kunnen geen meerdere verantwoordelijkheden gelegd worden bij de verschillende personen: een HospitalAdministrator kan geen docter of patiënt zijn. Hiervoor zou het design moeten aangepast worden naar een decoratorpatroon.

2.3 Input en output

2.3.1 Beschrijving

Input en output is is strikt gereguleerd. Alle output moet ofwel immutable zijn, zoals Strings en andere, ofwel read-only, zoals CampusInfo en LoginInfo, ... Voor Input moet men ook oppassen: deze moet ook immutable zijn, ofwel moet men een perfecte kopie maken van alle Objecten die ingegeven worden. Maar alleen met primitieven en Strings werken is niet uitbreidbaar en onhandig werken. Men kan bijvoorbeeld niet garanderen dat alle behandelingen hetzelfde aantal parameters hebben van hetzelfde type. De oplossing is om de parameters te abstraheren naar een Argument-object. Om de kennis van in het systeem dan ook nog af te schermen van de UI is er een vraag bij deze objecten gegeven. Hierdoor moet de UI niet de vragen stellen aan de gebruiker en kunnen gemakkelijk nieuwe behandelingen met andere parameters toegevoegd worden. Een ander gevolg is dat men direct kan controleren of de invoer mogelijk correct is.

2.3.2 Publieke Werking

De interface PublicArgument¡E¿ is de basis van de invoer. Deze heeft een vraag die aan de gebruiker moet gevraagd worden en deze kan beantwoord worden door de `setAnswer`-methode met een String. Deze wordt direct geconverteerd naar het type E. Deze Argumenten zitten in een ArgumentList, samen met objecten van de superklasse Argument¡E¿. Deze Argument¡E¿-objecten zijn voor het interne systeem om in te vullen, op het moment dat de UI de controle teruggeeft aan het systeem.

2.3.3 Interne Werking

Een generische factory verwacht als invoer om een nieuw object te maken een lijst van Arguments. Men kan een nieuwe lege lijst van Argumenten opvragen aan deze factory met de methode

`getEmptyArgumentList`. Als deze ingevuld is door de UI en door de andere code kan met de `validate`-methode gecontroleerd worden of een lijst correct is voor een bepaalde factory zonder deze factory een object te laten maken. Bij het aanroepen van de `make`-methode aangeroepen met deze lijst om een object aan te maken zal eerst automatisch deze `validate` aangeroepen worden om te controleren of de invoer wel geldig is.

2.3.4 Bespreking GRASP en uitbreidbaarheid

Deze opbouw laat toe om een grote diversiteit aan invoer uit te lezen op een veilige en generische manier. Het laat toe om de input statisch te valideren tijdens de invoer en zo duidelijker fouten terug te geven wat de cohesie ten goede komt. Het laat ook toe om informatie van het systeem te lezen zonder dat de UI deze zelf moet zoeken. Hierdoor is de koppeling tussen de UI en het systeem lager en duidelijk herkenbaar.

Men kan eenvoudig nieuwe argumenten ontwerpen en gebruiken zonder dat er iets moet veranderen aan de UI. Om informatie van andere objecten mee te geven kan men gebruik maken van het visitor-pattern en een filter.

2.4 Diagnoses

2.4.1 Beschrijving

Functionaliteit omtrent diagnoses wordt voorzien door de DiagnosisController. Interactie via dit object zorgt ervoor dat de gebruiker van de API geen weet hoeft te hebben van de interne werking van het systeem. De enige diagnose-gerelateerde functie voor eindgebruikers die niet door de DiagnoseController afgehandeld wordt zijn de `getSecondOpinions` en de `removeSecondOpinion` methoden in de DoctorController (een type LoginController, zie sectie 2.2), aangezien deze enkel gebruik maken van informatie in het DoctorController-object. Deze methoden komen verderop nog aan bod.

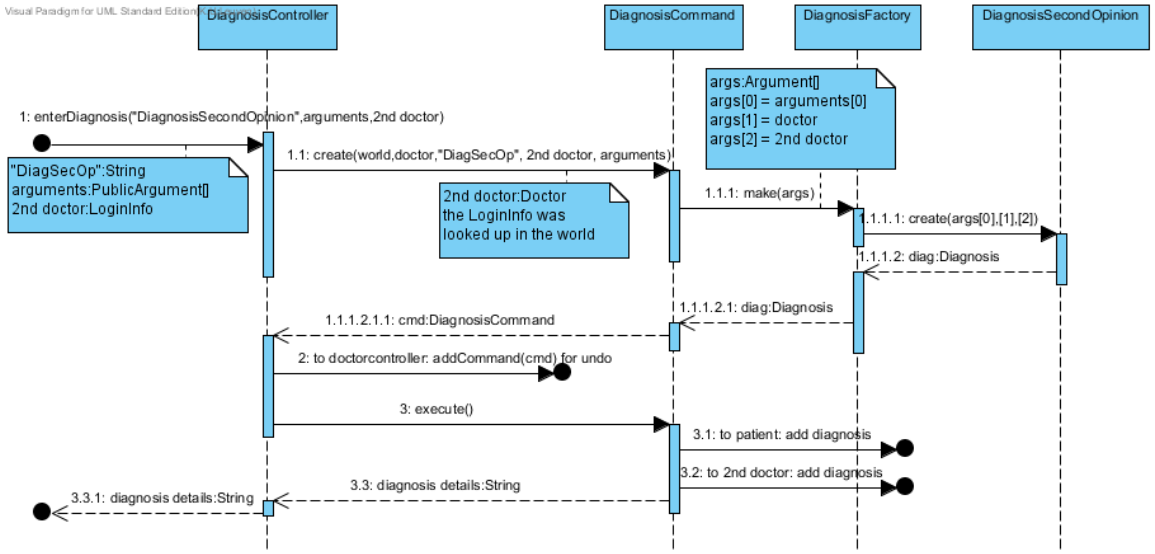
Een DiagnosisController krijgt bij constructie een geldige DoctorController en WorldController mee: deze functionaliteit is dus enkel beschikbaar voor aangemelde dokters. De WorldController geeft gecontroleerde toegang tot de benodigde andere objecten en subsystemen in de wereld. De DiagnosisController werkt met Diagnosis-objecten (en bijgevolg ook DiagnosisSecondOpinion-objecten voor diagnoses die door een andere dokter nagekeken moeten worden, deze zijn een subklasse van Diagnosis). Deze worden via de methode `enterDiagnosis` aangemaakt in de DiagnosisFactory-objecten die bestaan in de wereld.

Deze diagnoses worden bijgehouden in de patient (de patientfile is een view op een patient) waarvoor de diagnose gemaakt is, en indien een controle door een andere dokter vereist is, bij het Doctor-object van de dokter die de diagnose moet nakijken. Een diagnose kan een bijhorende Treatment bijhouden (meer informatie over Treatments is te vinden in sectie 2.6), in het geval van een DiagnosisSecondOpinion wordt deze automatisch gepland zodra de diagnose goedgekeurd wordt.

Om het ongedaan maken van acties te ondersteunen wordt elke actie via een Command-object uitgevoerd dat bij de uitvoerende dokter bijgehouden wordt. Hierop kan eenvoudigweg `undo` aangeroepen worden om een actie ongedaan te maken.

2.4.2 Usecases: EnterDiagnosis, ApproveDiagnosis

In figuur 2 zien we hoe een gegegeven DiagnosisController-object een `enterDiagnosis`-aanroep verwerkt voor diagnose met second opinion. Bij normaal gebruik wordt deze aanroep voor gegaan door aanroepen naar `getAvailableDiagnosisFactories`, `getDiagnosisArguments` en `getAvailableSecondOpinionDocto` om correcte waarden te bekomen voor de respectieve parameters. Indien ongeldige waarden gebruikt worden zal het systeem een gepaste exception geven. De naar rechts wijzende lost messages duiden op plaatsen waar verwijzingen naar het Diagnosis-object (of het geval van `addCommand` het Command-object dat de diagnose heeft aangemaakt) worden bijgehouden. Om een diagnose goed te keuren (zie figuur 3) zal het systeem een ApproveDiagnosisCommand-object aanmaken dat, wanneer uitgevoerd, de diagnose als geldig markeert en de diagnose verwijdert uit de lijst van diagnoses die nog gecontroleerd moeten worden. De diagnose zal daarop de bijhorende behandeling (indien deze bestaat) plannen. Meer informatie over behandelingen en tijdsplanning kan gevonden worden in secties 2.6 en 2.7.



Figuur 2: Interne verwerking van een `enterDiagnosis`-aanroep

In het geval dat de dokter de diagnose afkeurt zal het systeem eenvoudigweg de diagnose verwijderen uit de lijst te controleren diagnoses en een nieuwe DiagnosisWithSecondOpinion aanmaken waarbij de dokter waarvan de mening gevraagd moet worden automatisch ingevuld wordt als de dokter die de originele diagnose gemaakt heeft.

2.4.3 Bespreking GRASP

De koppeling is zo laag mogelijk gehouden door de verschillende diagnoses met een gemeenschappelijke superklasse te hebben. Deze voorziet de basistoegang tot alle diagnoses. Ook de verschillende soorten treatment zijn geabstraheerd, wat een logische stap is. Door de verbindingen enkel in een enkele richting te hebben wordt de koppeling nog verder verminderd: een diagnose weet niet tot welke patient behoort. Een doctor weet niet welke diagnoses hij allemaal gemaakt heeft...

De cohesie is ook groot, vooral in termen van de informationexpert, de informatie over diagnoses worden in de objecten zelf bijgehouden en is netjes afgeschermd van de rest van het systeem. Enkel de koppelingen, die laag zijn, kunnen de diagnose veranderen.

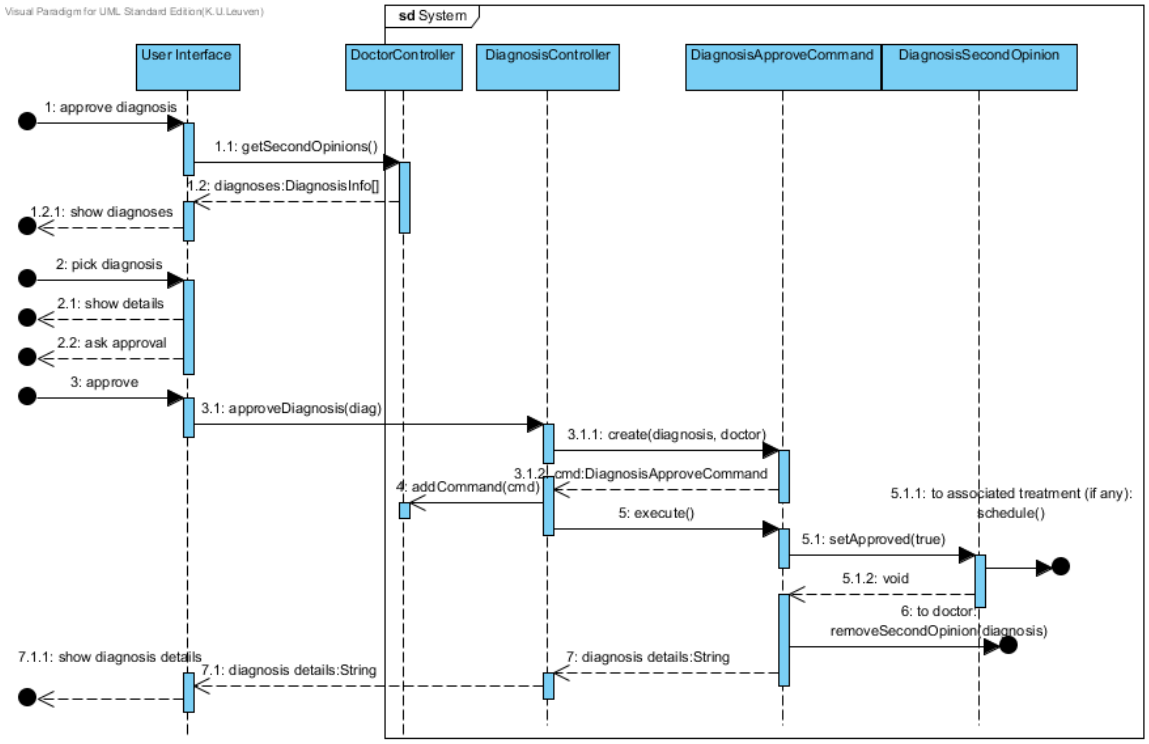
Aangezien er niet echt een duidelijke uitbreiding beschikbaar was om het design te controleren kunnen er een aantal probleemgevallen voorkomen. Een eerste aanpassing zal waarschijnlijk de abstractie van een diagnose naar een interface zijn. Nu was door de simpele uitbreiding het voldoende een uitbreidende klasse te maken. Het gebruik van de Command en de Factory met argumenten zou nog steeds flexibel genoeg moeten zijn.

2.5 Magazijn

2.5.1 Beschrijving

Hier volgt een overzicht van de belangrijkste klassen en een korte beschrijving van hun taken in ons ontwerp van het magazijn systeem:

- Stock : Deze klasse is verantwoordelijk voor het bijhouden van de voorraad van 1 soort item. De klasse Stock voorziet methodes om items te reserveren en te verwijderen uit de voorraad.
- Warehouse : Deze klasse bevat verschillende instanties van de klasse Stock, meer bepaald voor ieder soort item in het systeem n Stock. Deze klasse heeft methodes om de juiste stock te verkrijgen en om orders toe te voegen aan de bijhoorde stock.
- Items: De verschillende klassen in het package Items stellen de concrete items voor. Deze klassen zijn subklasse van de klasse Item. In deze klasse wordt bijgehouden of een item gereserveerd is en de vervaldatum van het item.



Figuur 3: Use-case: Approve diagnosis

- OrderList: De klasse Stock bevat een OrderList. Deze Orderlist is verantwoordelijk voor het bijhouden van de verschillende bestellingen voor de Stock. Order: Deze klasse stelt een bestelling voor en bevat de informatie van deze bestelling.
- OrderPlacers: De klassen in dit package zijn verantwoordelijk voor te bepalen wanneer er moet worden bijbesteld.
- LIFO-queue: Dit is een FIFO-queue geworden zoals voorzien.
- FIFO-queue: Een wrapper rond Queue¡E¿ en geabstraheerd als ItemQueue.
- ItemReservator: Deze klasse is verwijderd. En is vervangen door de ItemReservationCommand. Deze heeft een andere naam en andere verantwoordelijkheden.
- ReserveItemObserver: Deze klasse is verwijderd, een afspraak wordt direct gepland op het moment dat alle items beschikbaar zijn.
- ItemReservationCommand: Deze klasse reserveert alle nodige items in een stock. Op het moment van de afspraak komen de Items in deze klasse beschikbaar.
- ItemInfo: Dit object is een veilig object om buiten het systeem te gebruiken.
- ItemConstraint: Deze constraint wordt bij het maken van een afspraak in rekening gebracht om een goed moment van afspraak te vinden.

2.5.2 Interne Werking

In tegenstelling tot een eerdere implementatie waar afspaken pas gepland werden als we zeker waren dat alle items op voorrad waren, wordt een afspraak nu wel direct gepland. Een geschiedenis van alle afspraken wordt voor ieder type item bijgehouden en bij het plannen van de afspraak wordt een moment gezocht waar er geen tekort aan items is, noch waar er ooit in de toekomst problemen zullen komen. Het gebruik van een Item kan zich ver in de toekomst propageren en zo binnen een aantal maanden een probleem veroorzaken.

2.5.3 Bespreking GRASP

De warehouse is weer gemaakt om een zo laag mogelijke koppeling te hebben. De warehouse weet niet aan welke campus hij verbonden is, hij verzameld enkel alle verschillende stocks. De stocks weten ook niet tot welke wereld ze behoren. Dit zorgt voor een strikte afscheiding tussen de objecten en hun informatie. De lijst van orders is ook enkel gekend door de stock en bepaald welke items er besteld zijn. Het gebruik van een stockobserver verminderd ook de koppeling in stock sterk.

De twee subsystemen, het Warehouse en de Scheduler, zijn enkel met een Constraint (zie tijdsplanning, sectie 2.7 verbonden. Dit garandeerd een minimale koppeling. De constraint zelf bevat ook weinig logica en wordt door een afgescheiden algoritme afgehandeld dat apart getest kan worden.

2.5.4 Nadelen

Bij het plannen wordt geen rekening gehouden met het eventuele vervallen van de items aangezien we dit niet kunnen voorspellen. Het gebruik van een FIFO-queue en een "druk" hospitaal zorgen ervoor dat deze kans klein is.

2.6 Medische testen en behandelingen

2.6.1 Beschrijving: medische testen

De abstracte klasse MedicalTest implementeert de interface Result omdat het resultaat moet kunnen worden toegevoegd aan en opgevraagd uit de MedicalTest. Een MedicalTest implementeert ook de interface Appointable, omdat een MedicalTest een reden is om een afspraak te maken. In het systeem bestaan er op dit moment 3 verschillende subklassen van MedicalTests, namelijk XRayScan, UltraSoundScan en BloodAnalysis. Bij creatie van zo'n subklasse worden de parameters gecontroleerd en indien nodig wordt er een ArgumentConstraintException gegooid. Zo zal de UI een error kunnen melden indien er bijvoorbeeld een XRayScan met zoom 5 gemaakt zou worden.

Wegens het gebruik van preemptive-scheduling zal het plannen van een medische test voornamelijk besproken worden in de sectie Scheduling2.7 worden uitgelegd. De prioriteit van deze Appointment wordt gevraagd via een PriorityArgument.

2.6.2 Beschrijving: behandelingen

De abstracte klasse Treatment implementeert ook de interfaces Result en Appointable, om dezelfde redenen als een MedicalTest. In het systeem zijn er op dit moment 3 subklassen, namelijk Cast, Medication en Surgery. Bij het aanmaken van zo'n subklasse worden ook hier de paramenters gechecked en indien nodig wordt er een ArgumentContraintException gegooid.

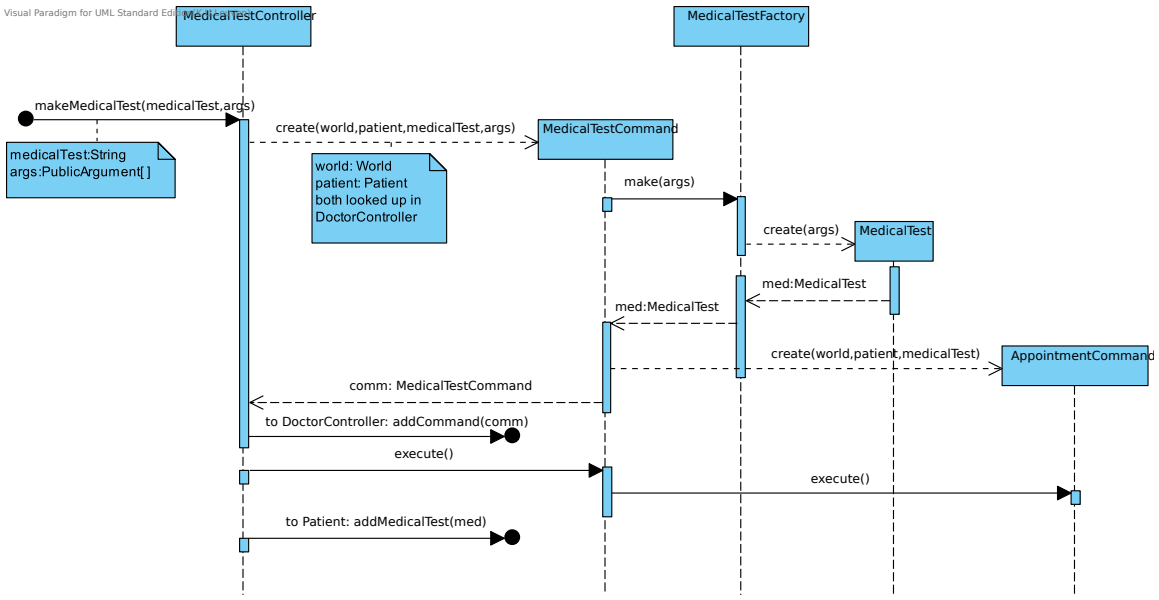
Elke subklasse heeft opnieuw een Factory, die bij creatie van de wereld in de wereld wordt aangemaakt. De TreatmentController en TreatmentResultController zorgen voor interactie tussen de UI en het systeem op gebied van Treatments. Ook het plannen van een treatment komt wordt wegens het gebruik van preemptive-scheduling verduidelijkt in de sectie Scheduling2.7.

2.6.3 Verloop usecases: OrderMedicalTest en EnterTreatment

Bij het creëren van een MedicalTest, wordt er een MedicalTestCommand gemaakt die op zijn beurt een MedicalTest laat aanmaken door de juiste MedicalTestFactory. Vervolgens maakt de MedicalTestCommand ook een AppointmentCommand aan voor het schedulen van de MedicalTest. Daarna voegt de MedicalTestController het Command toe aan de DoctorController. Vervolgens wordt het command uitgevoerd, de MedicalTest wordt toegevoegd aan de Patient. Ook wordt de test gescheduled: AppointmentCommand wordt uitgevoerd. Het maken van een behandeling gebeurt analoog.

2.6.4 Bespreking GRASP en uitbreidbaarheid

De verschillende types medische testen en behandelingen zijn heel uitbreidbaar. Er kunnen gemakkelijk andere behandelingen toegevoegd worden: Je maakt een nieuwe behandeling en zijn bijhorende factory. Het enige probleem is dat de nieuwe types misschien informatie van binnen het systeem nodig hebben, zoals diagnoses met een second opinion. In dat geval zou men een visitorpatroon moeten gebruiken om de argumenten te beantwoorden. De uitbreidbaarheid is getest geweest door



Figuur 4: Use-case: Order medical test

maar n medische test en n behandeling te implementeren zoals gezegd in teststrategy report. Dit werkte zonder problemen: alle domein informatie moest eenvoudigweg gemplementeerd worden.

De cohesie wordt sterk verhoogt door de creatie af te scheiden van het effectieve object door een factory te gebruiken. Dit zorgt ook direct dat wereld geen enkele connectie moet hebben met de behandelingen zelf, alleen met objecten die ze maken. Door dit verder te abstraheren daalt de koppeling nog verder. Het gebruik van een command schermt het volledige subsysteem af door een facade. Dit verlaagt de koppeling met externe klassen.

2.7 Tijdsplanning

2.7.1 Beschrijving

Het maken van een afspraak is een complex systeem. Het gebruik van AppointmentCommand laat toe om acties ongedaan te maken en te herdoen zonder alle informatie terug op te vragen. Dit houdt wel in dat het rekening moet houden met de veranderingen van het systeem. De ScheduleGroups laten toe om dit op een manier te doen zonder dat de command moet aangepast worden, andere flexibiliteit is dat er andere soorten groepen ook kunnen toegevoegd worden met dezelfde API. Ook het selecteren van ScheduleGroups van zowel binnen het systeem (treatment.getScheduleGroups) als buiten het systeem laten een grote flexibiliteit. Een stagair die aanwezig moet zijn op een bepaalde Surgery kan eventueel op de volgende manier toegevoegd worden: maak de afspraak ongedaan, voel een ScheduleGroup van de stagair bij en doe deze opnieuw. Hiervoor moeten enkel methoden om de afspraak op te vragen en om de stagair toe te voegen bijgemaakt worden.

Het gebruik van constraints op de groepen laat ook een grote vrijheid over. Doordat deze met een visitorpatroon werkt kan deze een hele hoop Schedulable overweg met kleine aanpassingen. Zo is de XRayConstraint voor Patient en de constraint op de werkuren voor Nurses. De TimeDelay is ook aanpasbaar een mogelijke uitbreiding is bijvoorbeeld een vertraging tot na een bepaalde datum. Dit kan gewoon door de getDelayedTimeFrame en setWorld in een interface te gieten.

Afspraken worden gemaakt voor een Appointable, dit wordt gedaan door AppointmentFactory. Deze krijgt toegang tot een aantal Constraint-objecten; deze kunnen pas valideren als aan de voorwaarde die ze opleggen (bv. niets schedulen buiten de werkuren). De AppointmentFactory zoekt een moment waarop alle constraints voldaan zijn. Ook is de eerste constraint een GetCampusConstraint voor ieder type van afspraak: deze bepaalt de campus waarop de afspraak zich zal voordoen.

Voor een afspraak tussen een dokter en een patiënt moeten bijvoorbeeld aan volgende Constraints voldaan zijn: DoctorBackToBackConstraint (DoctorPatientAppointment), Preference (van Doctor), PriorityConstraint (DoctorPatientAppointment), en WorkingHoursConstraint (Doctor). Elk van

deze constraints maken deel uit van alle voorwaarden die opgelegd zijn aan de afspraken. De PriorityConstraint zorgt dat er geen twee appointments op hetzelfde moment vallen en als er op een afspraak op dat moment staat bij een persoon dat die een lagere prioriteit heeft en dus zal verzet worden. De DoctorBackToBackConstraint zorgt ervoor dat de afspraak ofwel op het uur valt ofwel dat de afspraak direct na de vorige valt.

2.7.2 Bespreking GRASP en uitbreidbaarheid

Door de abstractie van behandelingen, die een soort afspraken zijn, en het effectieve afspraak-object door een interface is het subsysteem veel losser gekoppeld aan de Appointables. Dit zorgt ook voor een grotere cohesie: de informatie zit afgeschermd in een eigen object met een zeer specifiek doel. Het gebruik van een command laat toe om vaak gebruikte code te hergebruiken. Hierdoor ontstaat er een lagere koppeling in TreatmentCommand met het subsysteem en is er hogere cohesie: de code van een behandeling bij TreatmentCommand en de afspraak bij AppointmentCommand.

De code zelf heeft weinig uitbreiding nodig. Een afspraak is altijd hetzelfde formaat. De groepen zijn ook heel finaal, je zou misschien een groep kunnen toevoegen met n van de volgende vaste schedulables, maar dan zou je alle mogelijke groepen moeten hebben.

Ook de koppeling tussen de twee belangrijkste systemen is hier laag: De scheduler weet niet eens of er iets in het warehouse gedaan wordt. En behalve dat het opvragen van alle benodigde informatie is alles afgeschermd. De scheduler weet enkel dat hij constraints opvraagt van een Schedulable, en van een Appointable. De constraints hangen in een ketting: elk Constraint heeft de kans om een gekozen tijd en plaats af te wijzen, indien niet geeft hij de verantwoordelijkheid van het afwijzen door aan het volgende Constraint-object. Hierdoor kan men zeer eenvoudig meer constraints toevoegen of weghalen, men hoeft enkel de ketting langer te maken, of in te korten met de juiste Constraints zoals het nodig is.

Het algoritme is ook afgeschermd in een eigen deel binnen dit systeem. Dit laat toe om snellere algoritmes te gebruiken, zoals constraintprocessing met backjumping, ...

2.8 Administratie

2.8.1 Beschrijving

Administration gebeurt aan de hand van het aanroepen van Controllers die op hun beurt de Wereld gaan updaten. Zo zijn er de StaffController die Staff kan toevoegen, de MachineController die Machines kan toevoegen en de AdministratorController die alles afhandelt ivm het vooruitgaan van de tijd.

2.8.2 Verloop usecase: Add staff & machine

Eerst vraagt de AddStaffUI alle StaffFactories op van de StaffController en laat hij de gebruiker kiezen tussen deze Factories. Vervolgens vraagt hij de arguments op die ingevuld moeten worden. En ten slotte laat de StaffController de Factory de Staffmember aanmaken en stopt hij deze weg in de recources van de World. Indien er reeds Staff bestaat met de ingegeven naam, dan zal er een foutmelding worden meegedeeld. Om machines toe te voegen volgt men een analoge procedure aan die voor het toevoegen van een personeelslid. De meest recente toevoeging aan het systeem is het gebruik van campussen. Indien een personeelslid of een machine met een vaste plaats aangemaakt wordt, zal er vanaf heden dan ook gevraagd worden op welke campus deze zich bevinden.

3 Onbresproken Usecases

3.1 Preference

Deze usecase is eenvoudig. Een ListArgument van alle beschrijvingen van preferences worden opgevraagd en dan deze set dan de preference van de doctor. De PreferenceConstraint vraagt deze op en controleert alle mogelijke momenten met deze preference. Om een nieuwe preference te maken moet men een klasse maken die de interface Preference uitbreidt.

3.2 Undo & Redo

Iedere operatie kan in een Command gewrapped worden. Dit zorgt voor de mogelijkheid om een operatie uit te stellen en indien mogelijk ongedaan te maken en herdaan worden. Deze abstractie laat ook toe om deze operaties bij te houden om ongedaan te maken. De gedane Commands worden bijgehouden in een lijst bij de doctorcontroller: de operaties zelf maken geen deel uit van het systeem. Het gevolg is dat bij het uitloggen alle gedane operaties permanent uitgevoerd worden. Indien dit wel nodig is, kunnen de twee lijsten in de doctor zelf bijgehouden worden en dus persistent gemaakt worden.

3.3 Consult Patientfile

De open Patientfile usecase vraagt eerst alle PatientInfo objecten aan de WorldController via de DoctorController om deze dan te gebruiken om een Patient in te laden bij de Doctor.

3.4 Close Patientfile

Deze usecase zet de PatientFile op null bij de doctor.

3.5 Discharge Patient

Een dokter kan een patiënt ontslaan door simpelweg de methode `dischargePatient` van zijn DoctorController aan te roepen. Dit heeft als gevolg er aan het openstaande Patient-object gevraagd wordt of deze patiënt onstlaan kan worden. Het Patient-object controleert in zijn gegevens of nog onbehandelde diagnoses zijn of medische tests/behandelingen zonder resultaat, indien dit niet het geval is kan de patiënt ontslaan worden. De patiënt wordt zodanig gemarkeerd en alle patiënt-gerelateerde functies en methoden in de DoctorController zullen een fout genereren tot een nieuwe patiënt geopend wordt.

3.6 Register Patient

Om een bestaande patiënt op te nemen hoeft een aangemelde Nurse (die bijgevolgd toegang heeft tot een NurseController) enkel aan de wereld vragen achter de lijsten van patienten en dokters in het systeem. Uit deze lijsten kunnen dan de gepaste argumenten gekozen worden voor de /textttcheckInfunctie van de NurseController. Dit heeft als resultaat dat de patiënt als opgenomen geregistreerd wordt en er meteen ook een afspraak met de gegeven dokter gepland wordt (voor details over tijdsplanning zie sectie 2.7).

Indien de patiënt nog niet geregistreerd is in het systeem zal dit eerst gedaan worden volgens het Factory-systeem dat doorheen heel het project gebruikt is. Nadat de patiënt geregistreerd is kan men verder zoals beschreven in de eerste paragraaf.

3.7 Enter Medicaltest result

EnterResult wordt opgeroepen met argumenten: MedicalTestInfo en args (in te vullen testresults). Hierop gaat de MedicalTestController zoeken tussen alle medicaltesten van alle patiënten, welke ingevuld moet worden. Ten slotte worden de results toegevoegd. Indien er foute argumenten worden meegegeven, dan zal de gepaste foutmelding worden weergegeven.

3.8 Enter TreatmentResult

Door het gebruik van een geschiedenis in het Warehouse kunnen resultaten pas ingevuld worden wanneer de items er uit verwijderd zijn. En dit mag enkel op het juiste moment, anders kan zal de planning niet meer juist werken. Het is dus logisch dat resultaten pas uitgevoerd kunnen worden als het na de start van deze afspraak is. Anderzijds moet men tijdens het vooruitspoelen van de tijd direct de resultaten opschrijven van alle afspraken die geweest zijn. Dit zou betekenen dat men als verpleegster enkel de tijd heeft om deze in te vullen voor het einde van de afspraak. We hebben gekozen om AdvanceTime op een logischere manier te laten werken en toch niet te veel van de opgave af te wijken.

3.9 Advance Time

In de use case advance time wordt de tijd verder gezet. In ons ontwerp veranderen we de tijd in de World klasse. Een probleem is dat als de tijd veranderd, de staat van ons systeem ook moet veranderen, hiervoor maken we gebruik van het Observer patroon. De klasse die tijdsafhankelijk zijn implementeren de interface TimeObserver. En roepen de methode attachTimeObserver aan op de klasse met interface TimeSubject(in ons huidig ontwerp is dit de klasse World). De klasse World houd alle TimeObservers bij en als de Time verandert worden de observers gewaarschuwd met de methode `timeUpdate(Time t)`. Voor het ingeven van de test en behandelings resultaten overlopen we de verpleegsters en hun openstaande medical tests en treatments en als deze voor de nieuwe tijd vallen worden de gegevens ingegeven. Door de veranderingen die gebeurd zijn aan het Warehouse in de laaste iteratie was het niet logisch om results in te vullen als meteen wanneer deze verlopen. De aanpassing is gemaakt dat deze moeten ingevuld worden voor het einde van de dag en anders door de HospitalAdministrator.

3.9.1 Bespreking GRASP en uitbreidbaarheid

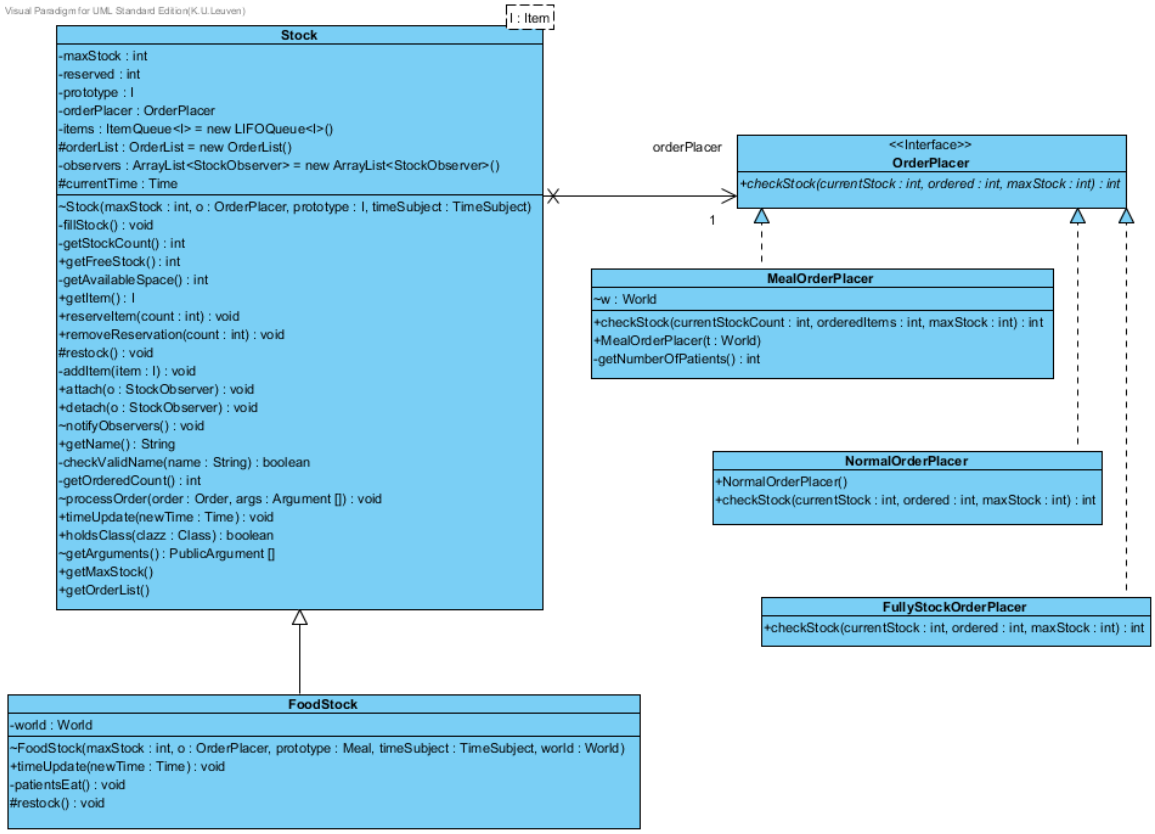
Het voordeel van de Observer is dat de koppeling laag blijft. Indien we niet met een Observer patroon zouden werken moest de controller in iedere klasse die tijdsafhankelijk is aanpassingen gaan doen, wat veel koppeling veroorzaakt. Doordat we hier met een interface TimeObserver werken is de koppeling veel beperkter. De klasse World weet niet welke specifieke klasse moeten worden aangepast. Er is enkel een lijst van observers die zelf de nodige aanpassingen doen. Dit geeft ook een goed uitbreidbaar ontwerp. Als er een nieuwe klasse wordt toegevoegd die afhankelijk is van de tijd, moeten er geen aanpassingen gebeuren in de klasse World.

3.10 Fill stock

Er zijn verschillende regels voor de verschillende categorieën van items bij te vullen in de voorraad. Zo heeft plaster een ander regel dan medication. Een oplossing hiervoor zou kunnen zijn om voor iedere verschillende regel een subklasse van Stock te maken die de juiste hoeveelheid besteld. Dit is echter niet goed uitbreidbaar. Een betere oplossing is om gebruik te maken van het Strategy patroon. We hebben de interface OrderPlacer met één methode checkStock(int currentStock, int ordered, int maxStock). Deze methode geeft het aantal te bestellen items terug. Bij het aanmaken van de klasse Stock wordt er een concrete implementatie van deze interface meegegeven. De klasse Stock kijkt iedere keer dat er een actie gebeurt op de voorraad of er nieuwe items moet besteld worden aan de hand van de methode checkStock. Voor het bijbestellen van voedsel items werkt deze methode niet, aangezien voedsel om middernacht besteld wordt. We hebben dit opgelost door een subklasse te maken van de klasse Stock, namelijk de klasse FoodStock. Deze klasse FoodStock werkt gelijkaardig aan de klasse Stock. Het grootste verschil is dat deze klasse zelf items verwijderd en bijbesteld op de tijdstippen dat ze gebruikt en besteld worden.

3.10.1 Bespreking GRASP en uitbreidbaarheid

Het voordeel van het gebruiken van het Strategy patroon is dat de Stock onafhankelijk word van het algoritme om te bepalen hoeveel er moet worden bijbesteld. Dit maakt het systeem ook meer uitbreidbaar omdat er nu makkelijk een nieuw soort items kan toegevoegt worden en 1 van de bestaande OrderPlacers kan gebruikt worden. De klasse OrderPlacer heeft ook een zeer hoge cohesie aangezien deze enkel verantwoordelijk is voor het bepalen van de te bestellen hoeveelheid. In onze oplossing voor voedsel is ons ontwerp niet zo heel goed. Een beter ontwerp zou zijn om volgens het GRASP pattern pure fabrication een klasse te maken die als verantwoordelijkheid heeft om het bestellen en verbruiken van voedsel items te leiden op de juiste tijdstippen. Hierdoor is er geen speciale subklasse van Stock nodig. Ook voor de cohesie is dit goed aangezien de klasse Stock dan enkel nog verantwoordelijk is voor het management van de voorraad en deze nieuwe klasse dan de verantwoordelijkheid voor het verbruiken en bijbestellen heeft.



Figuur 5: Klasse diagram van het systeem om de Stock te vullen

3.11 List orders

De UI geeft alle stocknames weer door deze via de worldcontroller aan de warehouse te vragen. Na de keuze van de gebruiker voor één bepaalde stock, zal de aangemaakte WareHouseController de orders opvragen aan de stock in de warehouse. Vervolgens filtert hij de niet-gearriveerde orders eruit en ten slotte neemt hij de 20 laatste van de lijst, zoals gevraagd in de usecase.

4 Public API

We hebben ervoor gekozen om het pakket Hospital en dus ook zijn sub pakketten in de capsule SystemAPI te zetten. Aangezien de gebruikersinterface klassen buiten dit pakket in het pakket HospitalUI zitten, kan er aan de hand van de tool gecheckt worden of er geen methodes worden gebruikt die geen onderdeel zijn van onze api. Het grootste gedeelte van onze API bevind zich in het pakket Hospital.Controllers verder bevat ook het pakket Hospital.Argument een deel van de API. Er zijn nog 2 klassen buiten deze twee pakketten die een stuk van de API bevatten, namelijk de klasse DiagnosisInfo en LoginInfo. Deze twee klassen zijn gemaakt om informatie door te geven aan de API. Aangezien deze tool de encapsulatie van de capsule checkt kunnen we er van uit gaan dat mits een goede API te kiezen, we een goede afscherming hebben van het domein. Onze API bestaat voornamelijk uit de controllers. In deze controllers wordt alle invoer grondig gecontroleerd op fouten. Dit zorgt voor een goede afscherming van de gebruikersinterface.

5 Conclusie

Wat we nu gezien hebben is systeem dat zeer eenvoudig uitbreidbaar is: voor elk type object in het systeem, zij het nu personeel, machines, behandelingen of iets anders, kan men eenvoudig een nieuw soort object aanmaken door een subklasse aan te maken van de juiste hoofdklasse. Enkele

functie-implementaties en overrides later is het nieuwe object toegevoegd aan het systeem. Deze flexibiliteit komt echter ook met een kost: men moet controleren of men wel met het juiste soort object aan het werken is, en sommige functie-aanroepen zijn er iets gecompliceerder door geworden, zoals eerst het aantal en type argumenten moeten opvragen om mee te geven aan een Factory object. De voordelen wegen echter veel zwaarder door dan deze nadelen.

Ons systeem is ook zeer defensief: ongeveer alles wat fout kan lopen wordt tegengehouden door een exception. De verscheidenheid in deze exceptions zorgt ervoor dat men aan de hand van het type exception bijna direct weet wat er misgelopen is, vaak zelfs zonder de documentatie te moeten raadplegen. Voor de laatste iteratie hebben we ervoor gekozen sommige exceptions wat algemener te maken of samen te brengen onder een algemenere parent-exception. Hierdoor moeten er in de code minder verschillende exceptions afgehandeld worden en wordt het systeem eenvoudiger om mee te werken.

6 Appendices

6.1 De user interface

De UserInterface bestaat uit één MainUI, één rolUI per rol die kan inloggen en één usecaseUI per usecase. In de MainUI krijg je de optie om in te loggen en na het inloggen zal de UI van de juiste rol (bvb. Doctor, WareHouseManager, ...) gestart worden. In de rolUI krijg je alle usecaseopties die voor die bepaalde rol gelden en wordt er gefiltert op precondities. Zo zal de DocterUI enkel de optie ClosePatientFile aanbieden, als de doctor van de huidige doctorcontroller een patient file open heeft. In de usecaseUI wordt heel de usecase doorlopen: lijsten worden getoond, er worden opties aangeboden, invoer gevraagd, ...

6.2 Testverslag

6.2.1 Teststrategie

Het testen is voortgezet geweest zoals in het vorige semester. Er zijn een aantal scenarios opgesteld die de volledige API testen, in andere tests wordt een deel van de API getest. Het schedulen van afspraken is uitvoerig getest geweest. Alle constraints worden getest op uitzonderingsgevallen. Ook de AppointmentConstraintSolver werd getest.

Dit is zoals voorzien in ons testverslag van vorig semester.

6.2.2 Eclemma-verslag

Eclemma rapporteert in tegenstelling tot iteratie 2 een redelijke coverage van 70%. Dit komt omdat de code van de UI afgescheiden is geweest naar een apart project. Ook zijn ongebruikte klassen verwijderd of in gebruik genomen, met de nodige testen. De minst geteste klasses zijn nog steeds de exceptions.

Echter de beste aanduiding van de kwaliteit van onze testsuite is het feit dat doorheen het implementeren van de software we gebruik hebben kunnen maken van onze tests om fouten in het systeem te kunnen opsporen. Deze ervaring van nut uit de tests gehaald te kunnen hebben zonder onzekerheid of het systeem nu wel échtẃerkt is waarschijnlijk ook de beste indicator voor testsuite-kwaliteit.

6.3 Werkverdeling

De Bie Tom	37u	register patient, API, Scenariotest, kleine aanpassingen
De Coninck Jeroen	46u	Refactoring, Campusssen, Verslag
Lapauw Ruben	~62u	Scheduling, AddEquipment & AddStaffMembers Warehouse, Verslag
Van Gool Jeroen	40u	Diagnoses, MedicalTest & Verslag Verslag

6.4 Volledig klassendiagram

druk dit apart af gespreid over 2 paginas en voeg in in plaats van dit blad

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		0%		n/a	2	2	5	5	2	2	1	1
Hospital		87%		81%	7	24	4	32	2	11	0	1
Hospital.Argument		53%		33%	11	25	26	61	8	22	4	9
Hospital.Controllers		72%		70%	53	177	131	454	23	111	1	18
Hospital.Exception		31%		n/a	33	47	54	75	33	47	22	35
Hospital.Machine		81%		58%	11	38	12	71	2	26	0	6
Hospital.MedicalTest		83%		82%	21	117	49	286	8	75	0	8
Hospital.Patient		64%		55%	51	121	76	236	15	70	1	9
Hospital.People		65%		64%	16	55	24	93	8	41	0	6
Hospital.People.PeopleFactories		53%		42%	7	18	18	39	2	12	0	3
Hospital.Schedules		60%		62%	56	127	124	292	20	68	1	10
Hospital.Schedules.Constraints		85%		62%	10	39	8	69	0	23	0	4
Hospital.Treatments		74%		62%	25	87	48	186	11	62	0	8
Hospital.WareHouse		45%		30%	98	148	163	312	33	74	0	9
Hospital.WareHouse.ItemQueues		62%		62%	5	11	6	16	3	7	0	1
Hospital.WareHouse.Items		50%		40%	24	52	32	76	14	42	0	9
Hospital.WareHouse.OrderPlacers		64%		50%	4	9	3	18	2	7	0	3
Hospital.World		74%		68%	16	57	69	221	9	40	0	4
HospitalUI.AdminUI		0%		0%	23	23	164	164	10	10	4	4
HospitalUI.DoctorUI		0%		0%	53	53	275	275	14	14	7	7
HospitalUI.MainUI		0%		0%	23	23	88	88	7	7	2	2
HospitalUI.NurseUI		0%		0%	31	31	168	168	15	15	4	4
HospitalUI.WareHouseUI		0%		0%	15	15	72	72	6	6	3	3
Total	6,436 of 13,736	53%	515 of 999	48%	595	1,299	1,619	3,309	247	792	50	164

Figuur 6: Eclemma code coverage verslag voor de testsuite