Report iteration 3 - SWOP

Dieter Geboers, Wouter Schaekers, Stefaan Truijen
-
Professor: Tom Holvoet
Project Advisor: Marko van Dooren

2012-04-06

1 Preface

This iteration is the final one. The new assignment instructed us to update our project so that it is able to support multiple campuses on hospitals. With this new feature came a new feature for doctors as well. Doctors gained the ability to travel between campuses. They also get a say in their behaviour concerning travelling between campuses: go back and forth between all campuses, or rather have an own schedule concerning locations?

Preemptive scheduling and tasks that support multiple priority levels were not part of our assignment as we're a team of three members.

We delivered a report on the refactoring changes we planned to execute during this iteration as well as two test scenarios for our system. However, as we changed the way the system works, the test scenarios were forced to be adapted. We also thought that it would be best to create an additional testing scenario to prove that the "basics" are not the only thing we can do.

Our project advisor, Willem Penninckx, was replaced by Marko van Dooren for this iteration due to illness. We had some very constructive meetings with them both and are very thankful for the insight they gave us. Especially when we started to deviate from the core of the project by over thinking some of the design decisions.

For the first time in our SWOP history we can say with confidence that everything works like it should and there are no missing features.

Contents

1 Preface **1**

2 Introduction **3**

3 Package visible constructors **3**

4 Creational design patterns **4**
4.1 User factories and user manager 4
4.2 Machine builder and machine pool 5
4.3 Medical test factories and patient files 6
4.4 Treatment factories and diagnosis 6
4.5 Diagnosis and patient files 6
4.6 Warehouse builder and campus 7

5 Task **7**
5.1 Generic tasks versus a hierarchy of tasks 7
5.2 The state of a Task 9
5.3 The description of a Task 9

6 Warehouse **9**

7 Scheduling **10**
7.1 Description and Requirements 10
7.2 Scheduler and TaskManager 11
7.3 Schedulable and Requirable 12
7.4 LocationTimeTable and TimeTable 12
7.4.1 HospitalDate 12
7.5 Impact of having different locations 12
7.6 The location preference for doctors 13
7.6.1 Preference states 13
7.6.2 getLocationAt()-method 13

8 The updated testing scenario **13**

9 Conclusion **13**

2 Introduction

During the feedback session of iteration two, some major (asw ell as minor) issues with our code and design floated to the surface:

- MachinePool had hard coded methods to create different types machines;
- There were no system sequence diagrams available;
- The UML of the domain layer did not have any relevant methods on it;
- The implementation of Warehouse was lacking in a lot of aspects:
  - When an activity that required something from a warehouse got scheduled, there were no items removed from that warehouse;
  - The way Warehouse kept track of it's items and their maximum capacity was hardcoded;
  - The system for creating stock orders and delivering them to the warehouse wasn't implemented successfully;
  - ...
- The unit for time (milliseconds) used in the system was nowhere documented.
- Some methods and controllers required the same set of parameters and instead of an object that groups them together, they were all mentioned explicitly;
- ...

We have taken all of the above (and more) into account while implementing iteration three to make certain that our hard work would not have been in vain. In this report we'll motivate the necessity of changes we have put the project through. Also, we'll try to give some other solutions for the problems we encountered that caused us to create the system as it is. We'll then draw a conclusion and leave it at that.

As for the amount of time we spent working on this iteration in detail, we refer to `http://code.google.com/p/swop-dswx/wiki/WorkingHours` . For a high level view, here's a table:

Dieter	Stefaan	Wouter
85h	110h	110h

There has been a total of 150 hours.

3 Package visible constructors

We decided to make as many constructors as we could package visible only. Doing this denies the creation of the affected objects by any unwanted classes/people efficiently. However, the lack of friend classes and friend packages in Java made it impossible to protect every constructor of every class in such a way.
If such a class or method exists, there will be documentation stating that the method or class in question should never be used outside of the domain layer.
For example: once Machine objects get created, they are added to a MachinePool immediately. The responsibility of assigning every Machine to a MachinePool lays with the MachinePool itself.

4 Creational design patterns

The following problem presented itself to us a couple of times while developing a plausible design for the assignment: what is a *clean* way to allow users to create new objects of certain dynamically changing classes? For example: how can we make sure that the user has the ability to create a new ultra sound scanner without making the domain inconsistent or giving too much responsibility to the user interface?
An answer to this question is implementing a creational design pattern. We decided to use a builder pattern for the creation of new Machines and an abstract factory pattern for the creation of new Users, Treatments, MedicalTests.
The builder mentioned above is nothing like the StandardHospitalBuilder, ExtendedHospital-Builder, WarehouseBuilder, UserManagerBuilder,... that you may have noticed in the source code of our project. In fact, they're nothing alike. The latter builders are templates for an initialised system to avoid a bunch of code duplication across different testing units.

Without further a due, we will now briefly discuss the creational patterns mentioned in the paragraph above.

4.1 User factories and user manager

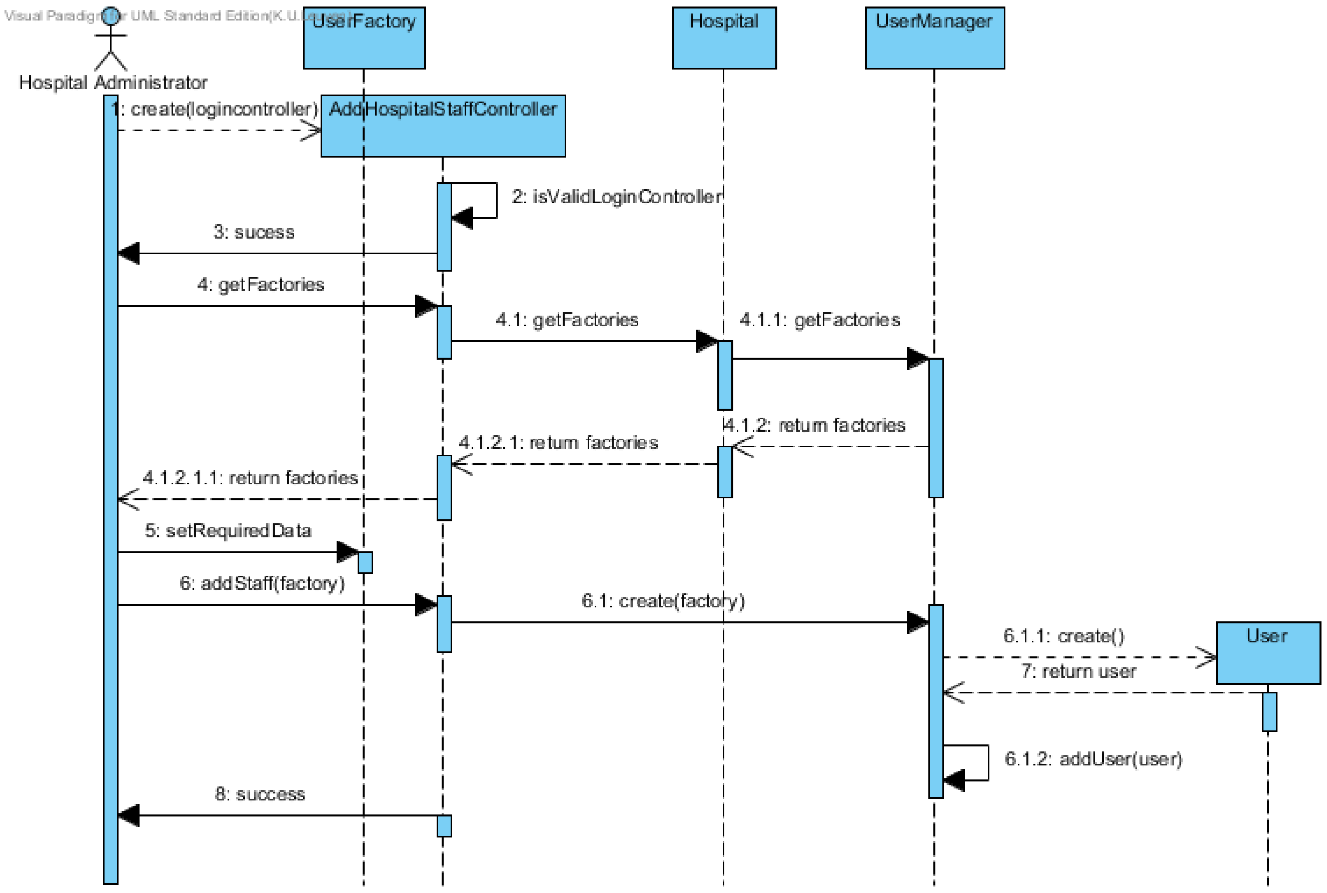

Figure 1: The sequence diagram from the add hospital use case demonstrates how user factories are to be used and how they work in the domain layer.

First, let's take a look at the sequence diagram of add new hospital staff above. We can see that the hospital administrator asks a list of all available factories, then selects the kind of factory he would like to create a new staff member from, next sets the required parameters and then lets the system create such a user.
The creation of the actual User object is the responsibility of the UserManager. This makes it easy to make certain that, when the user uses the public controller API, it's impossible to create a new User object that's not kept in a UserManager.

4.2 Machine builder and machine pool

Blood analysers, x-ray scanners and ultra sound scanners all have the same signature in their constructors. They also do not have any specific characteristics other than their types. This is the reason we chose to use a builder design pattern as a solution to the creational problem described in the beginning of this chapter.
It is also important to note that the constructor and build()-method of each builder is package visible only. This means no new MachineBuilder objects can be created without going through the MachinePool. Also the constructors of each of the machine types are package visible only. The combination of these visibility limitations means that, analogous to the case with UserFactory and UserManager, creating a new Machine or MachineBuilder object without registering it in a MachinePool is impossible.
Alas, the visibility in Java is not what it could be. We were forced to make the Machine classes themselves public as we require Machine objects in other packages. In particular, packages that contain TaskDescriptions come to mind, but more on that later.

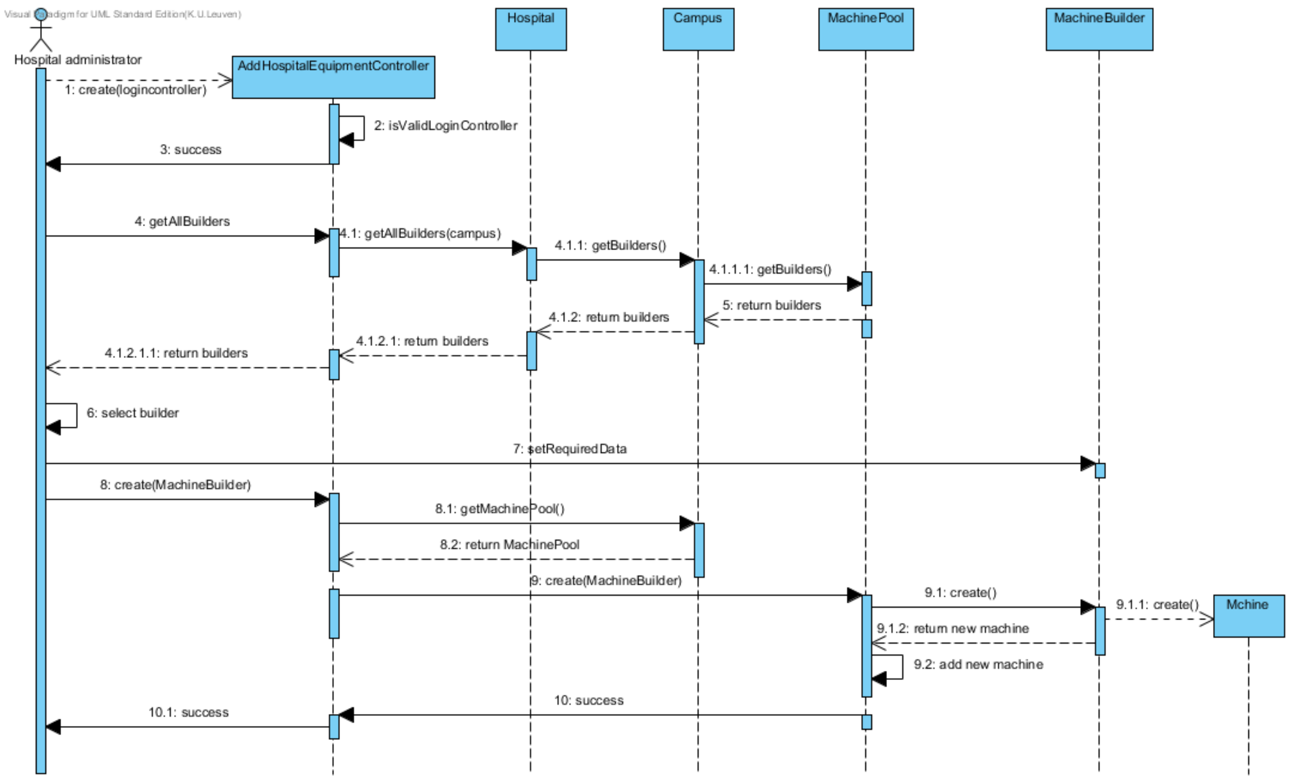

Figure 2: A sequence diagram of the add equipment use case

4.3 Medical test factories and patient files

When a doctor wants to order a new medical test for a patient, he or she will create a new OrderMedicalTestController. This controller allows the doctor in question to get all existing medical test factories. These factories are stored in the Hospital class and should be added manually at initialisation time before any use cases can be executed.
Once said factories have been acquired from the controller, the doctor can specify the required details in order for the medical test he's about to order to be exactly as he or she would like it to be. The doctor then calls upon the controller one more time, giving it the pre-set MedicalTestFactory. The controller passes this factory on to the task manager. The task manager executes the create()-method. This method generates a new MedicalTestDescription. This description is then stored in a new Task. The Task is then initialised (which means it will be stored in the patient file of the patient who the medical test is for).
We would have preferred for the create()-method to have limited visibility. Once again Java does not provide us with the possibilities to do so. Therefore we have equipped it with the `@Deprecated`-annotation so that, should someone try to call this method, they will see that it's not meant to be called. The documentation of the create()-method also informs the reader of this.
Like before, this way of things allows for a reliable domain layer to be constructed and maintained throughout the execution of the program. No MedicalTest objects will ever exist without there being a Task, as well as a PatientFile, to store them. The way Tasks are created (by first creating descriptions and then transforming them into Tasks in the TaskManager) also assures that Tasks never exist created without a TaskManager keeping track of them. Also, we would like to note that the constructor of Task is package visible within the scheduler.tasks package to aid in achieving this goal.

4.4 Treatment factories and diagnosis

Treatment factories allow doctors to create treatments that are always linked to diagnose similar to the way medical test factories allow doctors to create medical tests that are always linked to a patient file.
The doctor creates the appropriate controllers and asks them for the treatment factories available in this hospital. Just like is the case with medical test factories, these treatment factories are to be initialised at system initialisation time before any use cases can be ran.
The doctor chooses one of the given treatment factories and sets the required data. Then the doctor gives the factory back to the PrescribeTreatmentController, who then in turn, gives it to TaskManager. The TaskManager then creates a description from the factory and transforms it to a Task. The created Task gets initialised, which causes it to be linked to the diagnose it was created for.

4.5 Diagnosis and patient files

No Diagnose objects should exist without there being a PatientFile object to store them. The sequence diagram below demonstrates how assuring this consistency happens when the controllers are used properly:

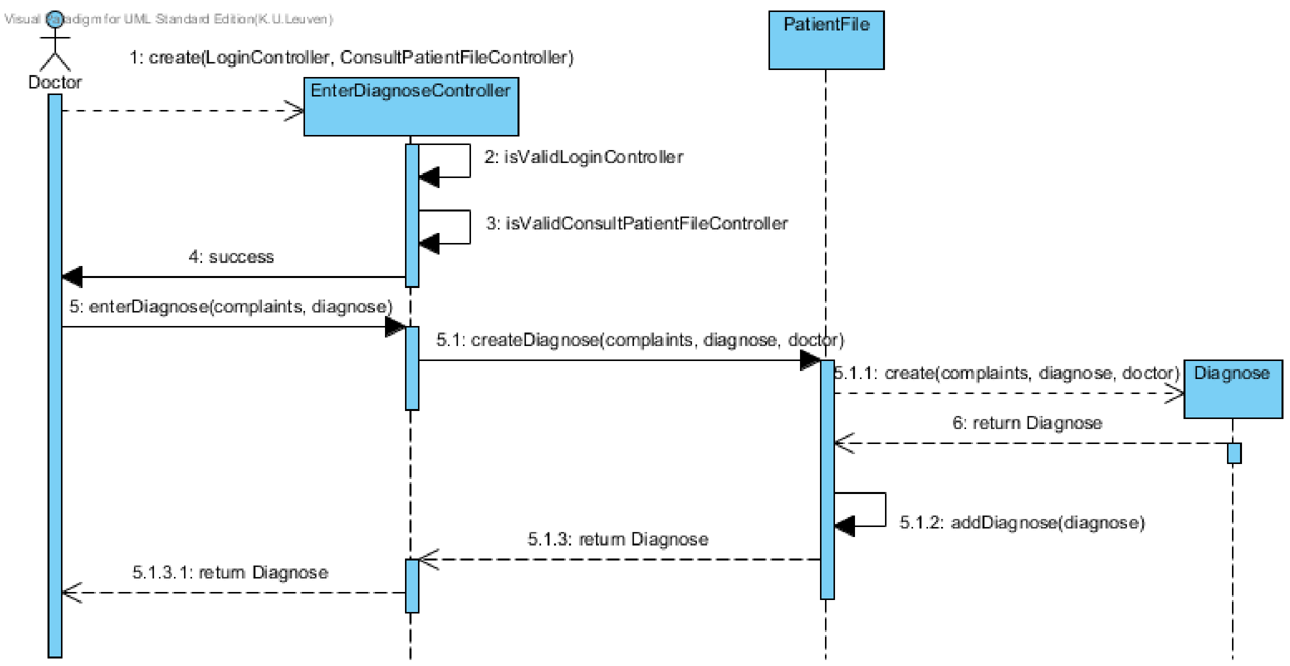
Visual Paradigm for UML Standard Edition(K.U.Leuven)
Doctor
EnterDiagnoseController
PatientFile
Diagnose
1: create(LoginController, ConsultPatientFileController)
2: isValidLoginController
3: isValidConsultPatientFileController
4: success
5: enterDiagnose(complaints, diagnose)
5.1: createDiagnose(complaints, diagnose, doctor)
5.1.1: create(complaints, diagnose, doctor)
6: return Diagnose
5.1.2: addDiagnose(diagnose)
5.1.3: return Diagnose
5.1.3.1: return Diagnose

Figure 3: The sequence diagram of the enter diagnose use case demonstrates how no diagnose can be created without a patient file storing it.

4.6 Warehouse builder and campus

Warehouses should not exist without a StockProvider (see later). Also a campus should always have a warehouse. To make this possible we use WarehouseBuilder objects at initialisation time. These warehouse builders are not design pattern builders but merely a way to assure the desired consistency mentioned above at creation time.

5 Task

Task object represent task concepts from the real world. Tasks are created from descriptions and are stored in the TaskManager. Tasks can undergo 2 transformations in their lifetimes. When they are created, they are regarded to as queued tasks. When the opportunity presents itself, the queued task becomes a scheduled task. Once the scheduled task has been executed, it becomes a finished task.

5.1 Generic tasks versus a hierarchy of tasks

Tasks come in many different forms. In the last iteration, we worked with a hierarchy of Task objects. For example: UnscheduledMedicalTestTask, ScheduledMedicalTestTask, UnscheduledTreatmentTask, ScheduledTreatmentTask, ... During a meeting with our project advisor, we came to the conclusion that such a hierarchy is not very desirable and violates the GRASP patterns. We then chose to work with a generic Task class: `Task<? extends TaskDescription>`. "Tasks from task descriptions".
The class diagram from Tasks and their descriptions can be found in figure 4 .

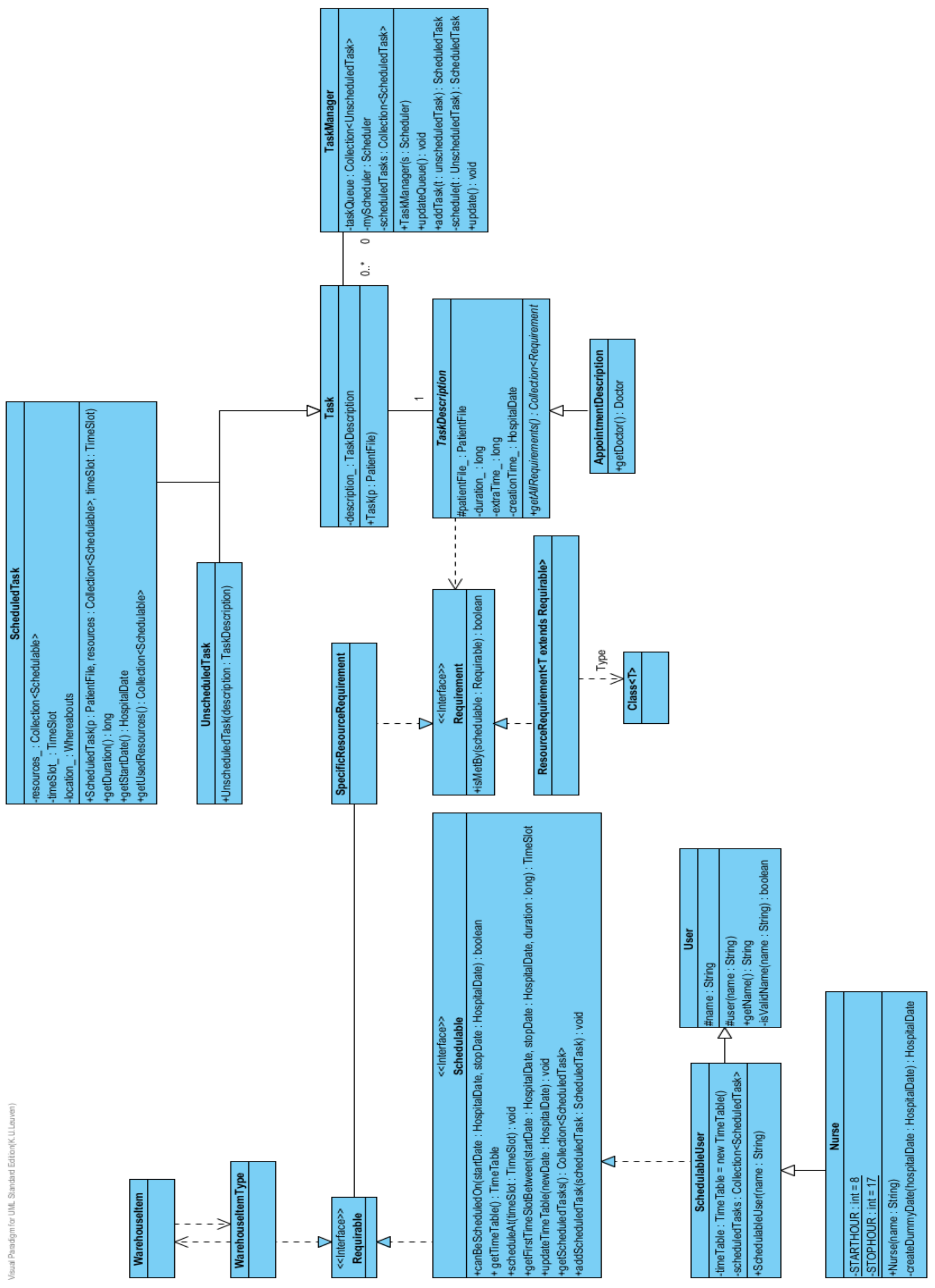
TaskManager
-taskQueue : Collection<UnscheduledTask>
-myScheduler : Scheduler
-scheduledTasks : Collection<ScheduledTask>
+TaskManager(s : Scheduler)
+updateQueue() : void
+addTask(t : unscheduledTask) : ScheduledTask
-schedule(t : UnscheduledTask) : ScheduledTask
+update() : void
0..*
0
Task
-description_ : TaskDescription
+Task(p : PatientFile)
1
TaskDescription
#patientFile_ : PatientFile
-duration_ : long
-extraTime_ : long
-creationTime_ : HospitalDate
+getAllRequirements() : Collection<Requirement
AppointmentDescription
+getDoctor() : Doctor
ScheduledTask
-resources_ : Collection<Schedulable>
-timeSlot_ : TimeSlot
-location_ : Whereabouts
+ScheduledTask(p : PatientFile, resources : Collection<Schedulable>, timeSlot : TimeSlot)
+getDuration() : long
+getStartDate() : HospitalDate
+getUsedResources() : Collection<Schedulable>
UnscheduledTask
+UnscheduledTask(description : TaskDescription)
SpecificResourceRequirement
<<Interface>>
Requirement
+isMetBy(schedulable : Requirable) : boolean
ResourceRequirement<T extends Requirable>
Type
Class<T>
<<Interface>>
Schedulable
+canBeScheduledOn(startDate : HospitalDate, stopDate : HospitalDate) : boolean
+ getTimeTable() : TimeTable
+scheduleAt(timeSlot : TimeSlot) : void
+getFirstTimeSlotBetween(startDate : HospitalDate, stopDate : HospitalDate, duration : long) : TimeSlot
+updateTimeTable(newDate : HospitalDate) : void
+getScheduledTasks() : Collection<ScheduledTask>
+addScheduledTask(scheduledTask : ScheduledTask) : void
User
#name : String
#user(name : String)
+getName() : String
-isValidName(name : String) : boolean
SchedulableUser
-timeTable : TimeTable = new TimeTable()
-scheduledTasks : Collection<ScheduledTask>
+SchedulableUser(name : String)
Nurse
-STARTHOUR : int = 8
-STOPHOUR : int = 17
+Nurse(name : String)
-createDummyDate(hospitalDate : HospitalDate) : HospitalDate
WarehouseItem
WarehouseItemType
<<Interface>>
Requirable
Visual Paradigm for UML Standard Edition(K.U.Leuven)

Figure 4: Tasks and their descriptions

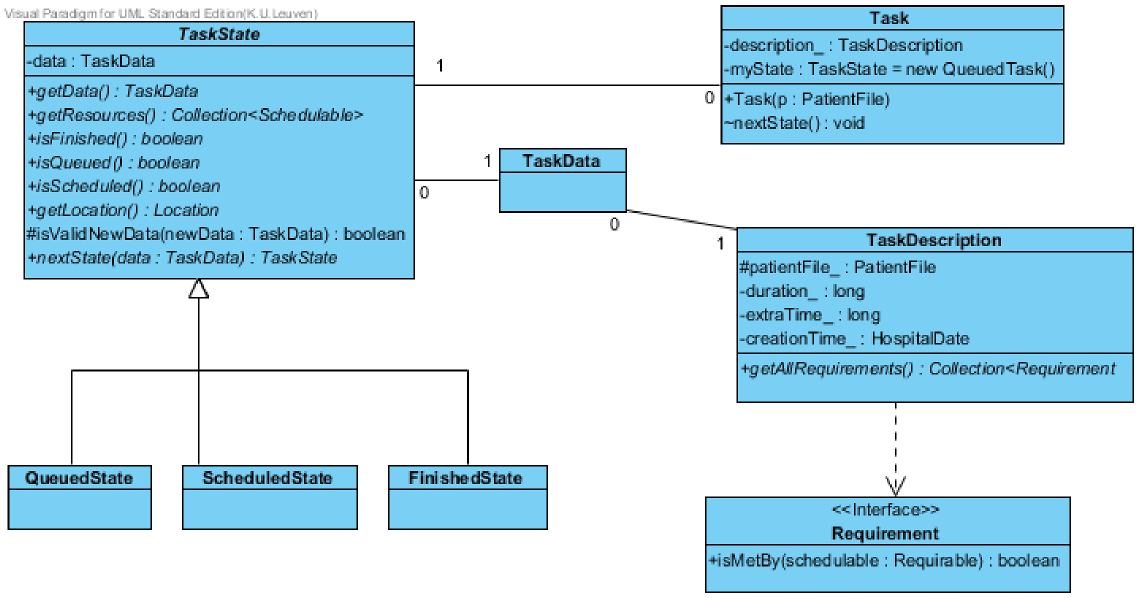
Visual Paradigm for UML Standard Edition(K.U.Leuven)
TaskState
-data : TaskData
+getData() : TaskData
+getResources() : Collection<Schedulable>
+isFinished() : boolean
+isQueued() : boolean
+isScheduled() : boolean
+getLocation() : Location
#isValidNewData(newData : TaskData) : boolean
+nextState(data : TaskData) : TaskState
Task
-description_ : TaskDescription
-myState : TaskState = new QueuedTask()
+Task(p : PatientFile)
~nextState() : void
TaskData
TaskDescription
#patientFile_ : PatientFile
-duration_ : long
-extraTime_ : long
-creationTime_ : HospitalDate
+getAllRequirements() : Collection<Requirement
<<Interface>>
Requirement
+isMetBy(schedulable : Requirable) : boolean
QueuedState
ScheduledState
FinishedState
1
0
1
0
0
1

Figure 5: Tasks and their states

5.2 The state of a Task

As described in the beginning of this chapter, tasks undergo ”transformations”. These transformations are implemented with a state pattern.
A task keeps its TaskState. The existing TaskState objects are: QueuedState, ScheduledState, FinishedState. Each state keeps a TaskData object. This object stores information relevant to the current state of the task. For example: a queued task would only keep its description while a scheduled task also keeps all resources involved in the execution of the task, where the task is to take place, etc...

5.3 The description of a Task

We define tasks by their description. Examples of these descriptions are: blood analysis, surgery, medication, appointment,...
These descriptions can be created using the controllers and the factories they can provide the user with. Once created, a description gets stored in the task data of a new queued task. A description contains requirements and conditions specific for that description. These requirements and conditions need to be met in order to schedule the queued task. (see later) You can find a diagram of these states below in figure 5 .

6 Warehouse

During this iteration we also made some large refactoring, if not implementing from scratch, from everything related to warehouses. There were a couple of significant problems, like mentioned in the introduction of this report. We managed to successfully find clean solutions for them all.
Let’s take a look at the class diagram from warehouse first:

The basic idea is that, using a WarehouseBuilder, we connect an ExpiredItemRemover and as many OrderStrategies to the warehouse as needed.

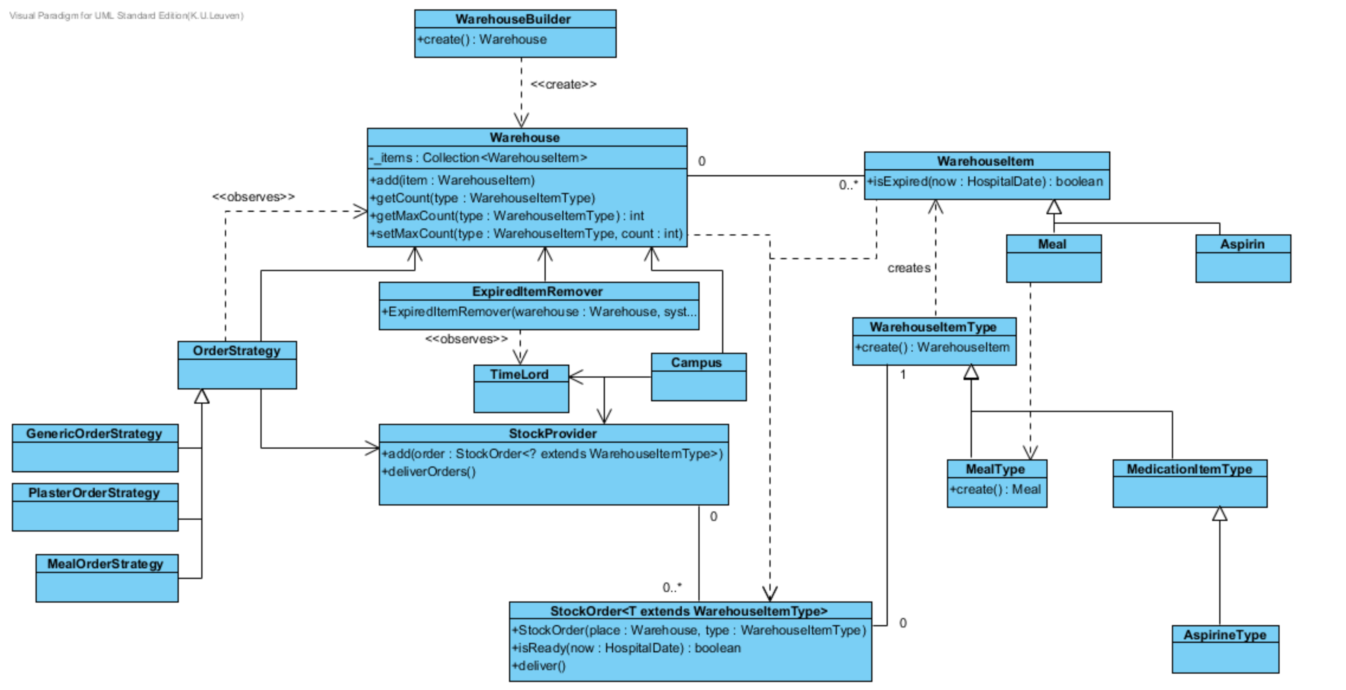
Visual Paradigm for UML Standard Edition(K.U.Leuven)
WarehouseBuilder
+create() : Warehouse
<<create>>
Warehouse
-_items : Collection<WarehouseItem>
+add(item : WarehouseItem)
+getCount(type : WarehouseItemType)
+getMaxCount(type : WarehouseItemType) : int
+setMaxCount(type : WarehouseItemType, count : int)
0
0..*
WarehouseItem
+isExpired(now : HospitalDate) : boolean
Meal
Aspirin
creates
<<observes>>
ExpiredItemRemover
+ExpiredItemRemover(warehouse : Warehouse, syst...
<<observes>>
OrderStrategy
TimeLord
Campus
WarehouseItemType
+create() : WarehouseItem
1
GenericOrderStrategy
PlasterOrderStrategy
MealOrderStrategy
StockProvider
+add(order : StockOrder<? extends WarehouseItemType>)
+deliverOrders()
0
0..*
MealType
+create() : Meal
MedicationItemType
StockOrder<T extends WarehouseItemType>
+StockOrder(place : Warehouse, type : WarehouseItemType)
+isReady(now : HospitalDate) : boolean
+deliver()
0
AspirineType

Figure 6: Class diagram from warehouse and its related classes

OrderStrategy is a class that, when the item type it's observing is running low, creates new stock orders for the warehouse automatically.
As different types require different methods of ExpiredItemRemover observes the current system time. Should the time change, the item remover will check if there are items in the warehouse that have expired due to the new system time and will remove them. This removal will, evidently, trigger order strategies to be notified.
All newly created stock orders are kept in the stock provider. The stock provider also observes the current system time. When the time changes, the stock provider will attempt to deliver all deliverable stock orders. Stock orders are deliverable if the required time to pass between order creation and the new system time is sufficient.
The stock provider is created together with the warehouse and both are stored in the campus. Working with only one stock provider per campus makes it easy to create the exact amount of new stock orders required.
The warehouse stores

7 Scheduling

The Scheduling algorithm is built in such a way so that the Scheduler doesn't know anything about the specific classes that are in the the rest of the domain layer. It only knows about abstract schedulables and requirements.

7.1 Description and Requirements

A Description is an abstract class that *describes* a possible task. Such a Description can be an Appointment, a BloodTest, ... It has all the information it needs to be able to build a task.

This information is contained in the form of requirements. A Requirement is a class such as a Doctor, a WarehouseType, a Nurse, ... Requirements know what Objects are needed to forfill this Requirement. There are different Requirements, since not all Requirements are not of the same type:

- SpecificRequirement: This kind of Requirement contains one Object that needs to be fulfilled. For example a Doctor during an Appointment. This Requirement will only be met by this Doctor.
- RequirementType: This kind of Requirement contains an ObjectType. All Objects of this type will fulfill this Requirement. It is possible that there are more than one Object needed to fulfill this Requirement. This information is also stored in this Requirement. An example of this kind of Requirement are Nurses during a xray scan. Since it is not important which Nurse does the scan, only some Nurse object is enough to fulfill this Requirement.
- DiagnoseCondition: This kind of Requirement is only fulfilled if the Diagnosis of the Patient is approved.
- WarehouseItemCondition: This kind of Requirement can only be forfilled if a Warehouse has enough items of the kind of type that is stored in this condition. To check whether this condition is met, a HospitalDate and a Location is needed to check this Condition. This Requirement has a collect method, in order to delete all the items it needs from the given Warehouse.

All these Requirements have the same interface, and thus the same methods. One of the methods is *isCrucial()*. A Requirement is crucial if the Requirement needs a Machine or a SchedulableUser. This Requirement can only be met when the admin adds Machines or SchedulableUsers to the Hospital. If this Requirement is not met, the Task can never be scheduled in this Hospital with the current resources. If there are not enough resources available in the hospital to forfill this Requirement, the Task will never be able to be Scheduled and will be deleted from the system.

7.2 Scheduler and TaskManager

Due to security reasons, it is impossible to create Tasks. A user can only make Descriptions. If a Description is made, the user is obliged to add this Description to the TaskManager. The TaskManager will make a Task out of it and tries to schedule it immediately after that. The scheduling happens with a scheduling algorithm in the Scheduler class. It will try to schedule this task. It is possible that the Scheduler is not able to schedule the Task. In that case, the Scheduler will throw an exception. There are two types of exceptions. Depending on the sort of the exception, the TaskManager will decide whether it will store this Task in a queue or will throw it away.
During this process, the description will be initialised. During this initialisation, the DiagnoseCondition will start observing the Diagnose, in case the state of the Diagnose would ever change. The TaskManager will aslo has to observe the Task, so the Diagnose will notify the TaskManager indirectly.
If the Scheduler is able to schedule the Task, it will schedule all Schedulables and collect all the WarehouseItems. The state of the Task will be changed.

7.3 Schedulable and Requirable

Schedulables are Objects that can be scheduled. They have a TimeTable.
Requirables are Objects that are needed to forfill a Requirement. This is in fact just an empty interface to generalize all the Objects that are usefull for this.

7.4 LocationTimeTable and TimeTable

A TimeTable is a class that represents the individual TimeTable that each Schedulable has. It very much resembles a real life timetable in which one can keep track of ones appointments and todos. A TimeTable consists of a set of TimeSlots. Each TimeSlot in the TimeTable represents an interval in time during which the TimeTable in question is marked as being occupied. A couple of methods in TimeTable that are worth mentioning:

- invert() : TimeTable
- getUnion(TimeTable) : TimeTable
- getIntersect(TimeTable) : TimeTable
- eliminateOverlap() : void

Invert() inverts the TimeTable. This means: busy slots become free slots and the other way around. The result of this method is a new TimeTable that is the inverted one of this TimeTable. Evidently the inverted TimeTable of the inverted TimeTable of an original TimeTable, is the original TimeTable again. This method can be very useful to determine all free slots in a TimeTable.
getUnion() gets the union of two TimeTables. The union is defined analogous to the union of two collections in mathematics: its a new TimeTable that is marked as busy everywhere where either one or both of the original TimeTables is busy.
getIntersect() gets the intersection of two TimeTables. Like getUnion(), getIntersect() is defined very much like in mathematics: the intersection of two TimeTables is a new TimeTable that is marked as busy where both of the original TimeTables are busy.
eliminateOverlap() is a method that basically merges timeslots that overlap. For example: if a TimeTable contains a slot from 5AM to 8AM and another one from 7AM to 9AM, eliminateOverlap() will result the two original timeslots being reduced to only one: from 5AM to 9AM.
LocationTimeTables are exactly the same as TimeTables. The only difference is the fact that the TimeSlots are have Locations attached to it. This is needed for Schedulables that don't have a fixed location.

7.4.1 HospitalDate

Just like in the previous iteration, we use our own HospitalDate class, since there is currently no good class in java to satisfy our needs.

7.5 Impact of having different locations

The different locations had a huge impact on the scheduling algorithm. Instead of just iteration over all the resources, the algorithm had to take into account that not all resources are available everywhere. This has been eleganlty solved, by just iterating over all the locations and then trying to schedule over there. The scheduler will chose the best timeslot amongst all the possiblities.

7.6 The location preference for doctors

7.6.1 Preference states

7.6.2 getLocationAt()-method

8 The updated testing scenario

We have changed the testing scenario we handed in on the deadline of the refactoring report drastically.

9 Conclusion