Taskmanager: Iteratie 1

Kwinten Missiaen, Steven Thuriot, Koen Van den dries, Bart Vangeneugden

Methodologieën voor Ontwerp van Programmatuur

Tom Holvoet
Mario Cruz Torres

Academiejaar: 2009 - 2010

Taskmanager: Iteratie 1

Kwinten Missiaen, Steven Thuriot, Koen Van den dries, Bart Vangeneugden

Methodologieën voor Ontwerp van Programmatuur

Tom Holvoet
Mario Cruz Torres

Academiejaar: 2009 - 2010

Contents

1 Introduction

This document serves as a documentation instrument for the first Iteration of the MOP Team Assignment.
In the following chapters, the reader will get a general overview of the outer- and inner workings of the application, accompanied by several diagrams. At first, we will discuss the view layer. The user will be explained how the user interacts with the system. Delving deeper, we lay out how packages were used to ensure safe, decoupled and thought-through class-to-class communication. Once this is covered, we explain in detail just which classes have which responsibility, and why. Also our testing approach is explained in short.
We conclude with our team organization, planning and a short self-evaluation.

2 System Operations

2.1 Task Management

The system revolves around Tasks. There are many different entities to keep in mind such as Users, Resources or Projects. But all of these things somehow have to do with Tasks itself.

2.1.1 Creation of tasks

When creating a Task, the User asked to give details about the task at hand. The user is expected to enter a short description, the start time of the task, the deadline and the average duration of the task. Also, the user is presented with a list of resources and other tasks already in the system. The user then selects a few dependencies for the task he's creating and resources he wishes to allocate.

When creating a task, the user has to make sure he does not violate the business rule when creating this Task. This means the user enters a duration, start date and end date. The start date must come before the end date and the difference between end- and start date has to be larger or equal then the duration. Also the systems tests if the entered description is an empty string. If this happens, an error is shown.

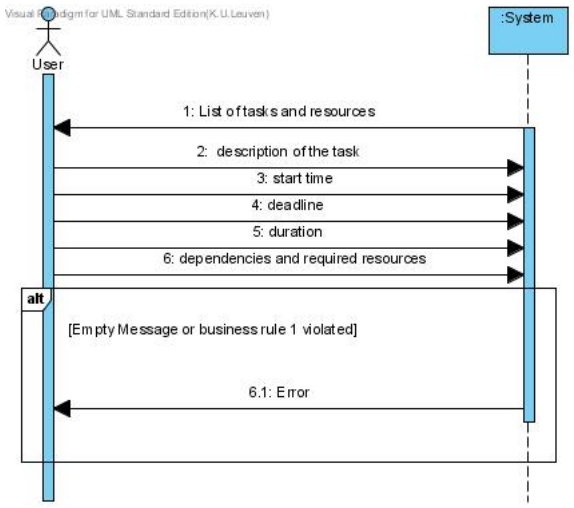

Figure 1: System Sequence Diagram describing the creation of a task

2.1.2 Removing Tasks

When a task is to be removed. It first has to check how that will affect other entities in the system. Is a Task still required by other Tasks in a dependency? If this is the case. The user will receive an error message and is asked how he wants to proceed: Cancel the operation or delete all the dependent tasks.

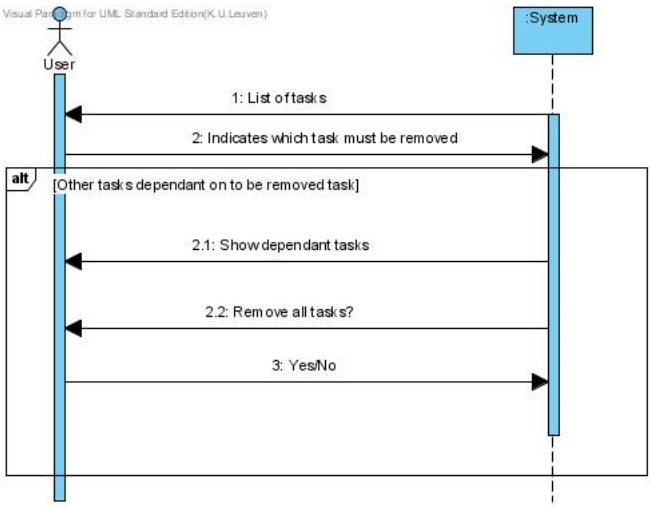

Figure 2: System Sequence Diagram describing the removal of a task

2.1.3 Modifying Tasks

When modifying tasks, the system has to make sure the user follows all the rules described above. This is because the user has the option to change all of the schedule variables, dependencies and required resources.

This means that the Business Rule has to be tested, as well as the Empty Description rule. Checking is also done on the task dependencies. For instance, it should not be possible to create a task A, dependent on task B. And modify task B to be dependent on task A. This would create a dependency loop.

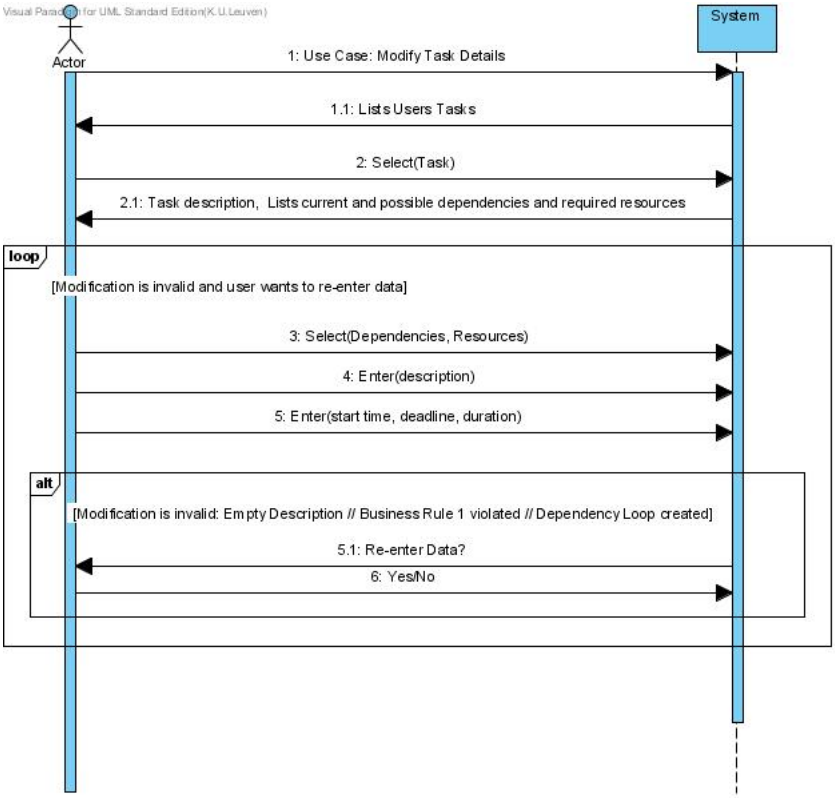

Figure 3: System Sequence Diagram describing the editing of a task

2.1.4 Updating a Task status

Updating a status of a task is integrated in modifying a task, but requires special attention as many different rules apply when adjusting the status of a Task.

Updating the status can directly reflect the status of dependent tasks. When a task was marked Successful, but is reverted to Failed or Unfinished the user will be asked to update the status of all dependent tasks or leave everything including this tasks status unchanged. This is because may have to revert back to Failed or Unfinished because they depended on the successful completion of this task.

2.2 Getting a task overview

When asking for a overview of tasks, it is sometimes easy to sort and/or filter tasks in a different manner. That way you can get a better overview. We provided a handy interface for this in the way of a loop. The user can choose how he wants his tasks sorted: by deadline of by duration. In case the user selects to sort the tasks by deadline, he is asked how many tasks are to be shown. Is duration selected, a minimum and maximum duration is asked. It's made particulary easy to alter and/or add sorting and filtering methods.

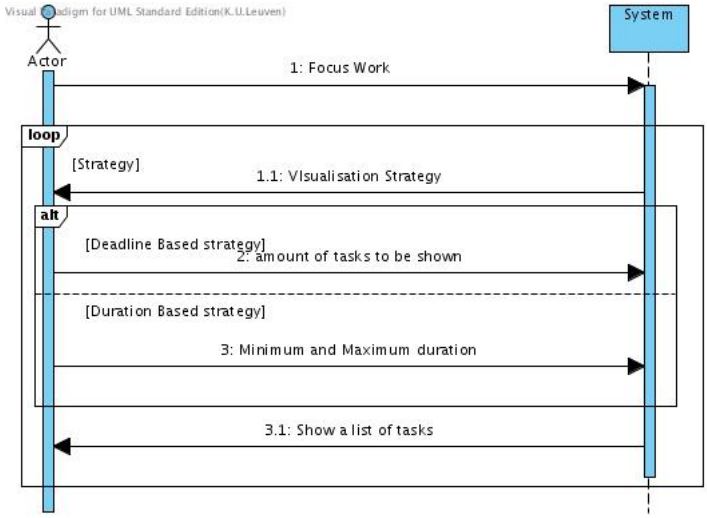

Figure 4: System Sequence Diagram describing the overview of all tasks

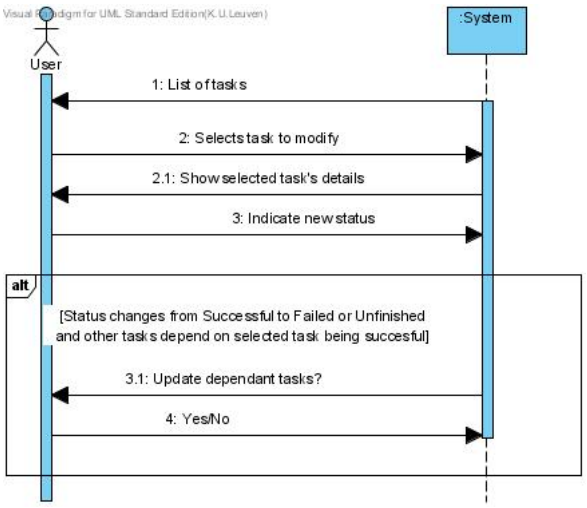

Figure 5: System Sequence Diagram describing the updating of the status of a Task

2.3 Resource Management

2.3.1 Creating Resources

A user can create a resource. When this resource is created, it is added to the system and stored there for later use. The system will only check for a valid description. This means the description can not be empty.

Once created, the resource is available for binding to Tasks, or Reservations.

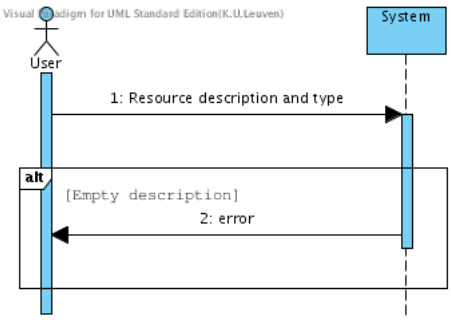

Figure 6: System Sequence Diagram describing the creating of a new resource

2.3.2 Remove Resource

A Resource can be removed. However, the system will first check it's dependency's with Task objects and/or Reservations. If the Resource is required by any of these, the system can not remove the Resource.

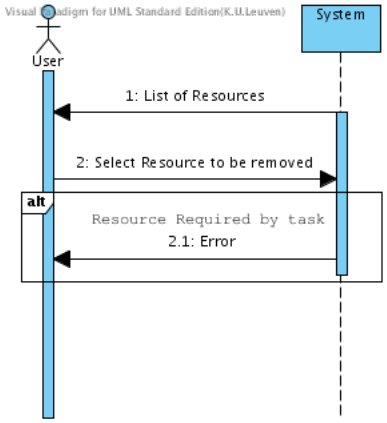

Figure 7: System Sequence Diagram describing the removing of a resource

2.4 Reservations

2.4.1 Create Reservations

Once a Resource is created, the User has the option to make a Reservation for that Resource. When creating a Reservation, the user is asked for the period of time he wants to make the Reservation after being shown a list of current Reservations for the Resource.

After the user enters this data, the system controls the entered data to see if it does not conflict with previous Reservations.

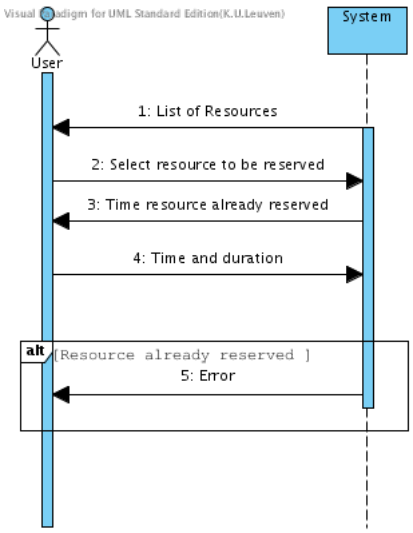

Figure 8: System Sequence Diagram describing the making of a reservation

2.5 Project Management

2.5.1 Create Project

A project can be created. A project can contain many Tasks, but does not have to. Our assumption is that at least one User is allocated to a Project, but several Users can subscribe. The system asks the user for a short description of that Project. If this description was not empty, the system creates the Project.

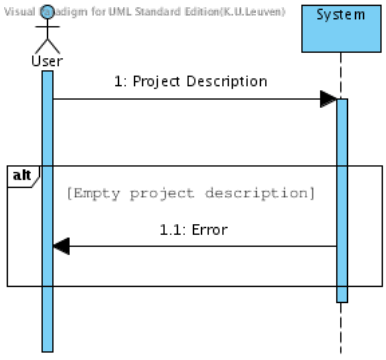

Figure 9: System Sequence Diagram describing the creating of a project

2.5.2 Remove Project

A project can be removed without too many details. If the user wants to remove a project, all of the tasks connected to that project are removed. The user simply selects a Projects and all of the underlying tasks/dependent tasks are removed with it.

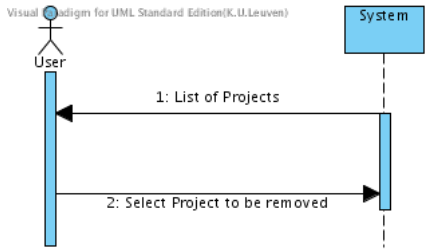

Figure 10: System Sequence Diagram describing the removing of a project

2.6 User Interface

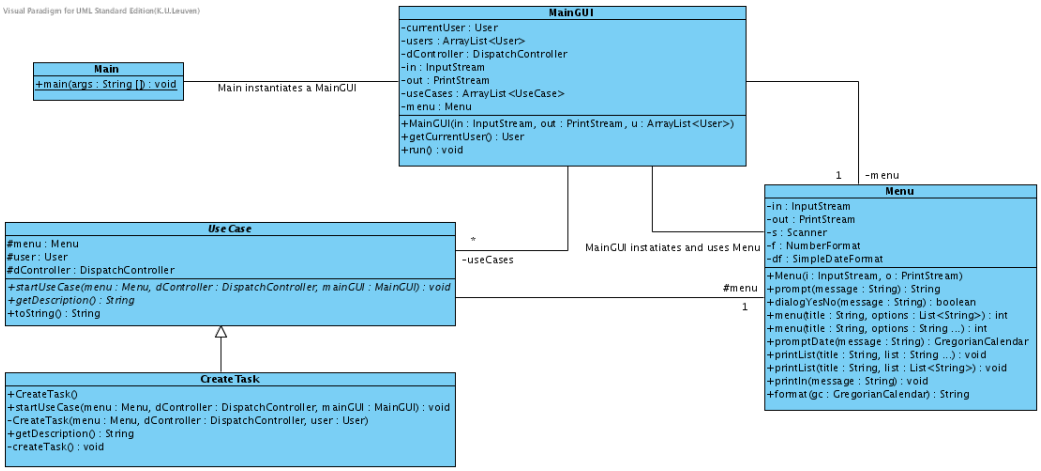

Figure 11: Class diagram of the GUI

The project uses a text based UI. All use case are handled by a subclass of the abstract UseCase class, in the figure the use case for Create Task is given as example. MainGUI keeps a list of instance of these use cases. When the user initiates a use case MainGUI calls the corresponding use case, passing

on the dispatch controller which will be discussed later, the use case then creates a new instance of itself with all field initialized to handle the rest of the use case. The class Menu handles the actual communication with the user through the console, formatting all the in and output.

3 Package Communication

The whole project is divided in four big packages. These packages are 'model', 'controller' and 'gui'. The last package is 'test', which has been separated from the rest of the project for obvious reasons.

We chose to create these packages so everything would be separated from each other. It would be bad practice to let the GUI access the model in a direct way, since this would mean even a small change could result in massive changes throughout the GUI.

This is where the controllers come in. They offer a way of communicating with the model. They basically contain a set of functionalities that are used throughout the program. These will then make the correct calls to the model. This way everything is handled without using direct calls, creating a more persistent system against changes throughout the model.

All these controllers are finally instantiated inside one big container called the dispatch controller. This controller enables the GUI to just instantiate one controller. Its constructor will take care of the rest. The other controllers are stored inside it and can be accessed by calling the correct getters. All the controllers can then easily be passed along the different views using this one object.

4 Class Descriptions

4.1 User

At first, we intended the User class to be responsible for the creation of Projects and Tasks, because a User object contains a list of Tasks and Projects that belong to that User. In the end, we chose not to do so. We felt that that the indirection was not necessary and that it might not benefit the cohesion of the User class.

While not described in the project assignment, we assumed the system could become a multi-user system in the next iteration(s). We therefore based our design on this.

Every User object is responsible for keeping track of the Projects and Tasks that belong to him. The system can ask the User object to return a list of these Projects or Tasks.

4.2 Task manipulation

Tasks are collected in the User. They contain a list of Resources the User might require to execute the Task. A Task has attributes which define it's own description, a Start and End time as well as a Duration. Also, a Task has a list of Tasks on which the Task may depend. This means the status of the Task at hand is dependent on the Status of it's dependent Tasks.
Keeping Low Coupling and High Cohesion in mind, Task has the following responsibilities:

- Keeping track of its name, start date, due date, duration and status
- Keeping track of the resources required to execute the task
- Keeping track of its dependencies, as well as the tasks that depend on this task
- Checking for the business rule 1 and preventing the construction of loops in the dependency scheme
- Updating the status of dependent tasks when necessary

A Task is not an Information Expert or Creator of Resources. As described in the next chapter, a Resource has its own management. We do feel it is necessary for the Task to know about its Resources.
However, a Task is Information Expert about Task itself. Therefore, we felt that the Task class should be responsible for enforcing business rule 1, preventing the construction of dependency loops, and updating the status of dependent tasks when necessary (as described in the use case 'update task status').

While designing, it was suggested that perhaps the Task class has two distinct responsibilities this way: one concerning its own details, and one concerning the way it interacts with other Tasks in the dependency graph. In the end,

we chose to stick to one single Task class, as we felt that the cohesion of this class was sufficiently high.

4.3 Resource manipulation

Resources can be accessed via Tasks. This is because a Task can define which resources are required for that Task. However when a Resource is just created, or when a Task to which a Resource was allocated gets removed, the Resource will not be referenced to by any Task. The object would not exists. *A resource is therefore stored in a RepositoryManager. This is explained in the section about 'Collecting Data'.*

A Resource itself is an Information Expert as well as a Creator for Reservations. We decided to put the responsibility of creating and storing Reservations in Resource because a Resource needs this information to check it's own availability, as displayed in the diagram 'Create Reservation'.

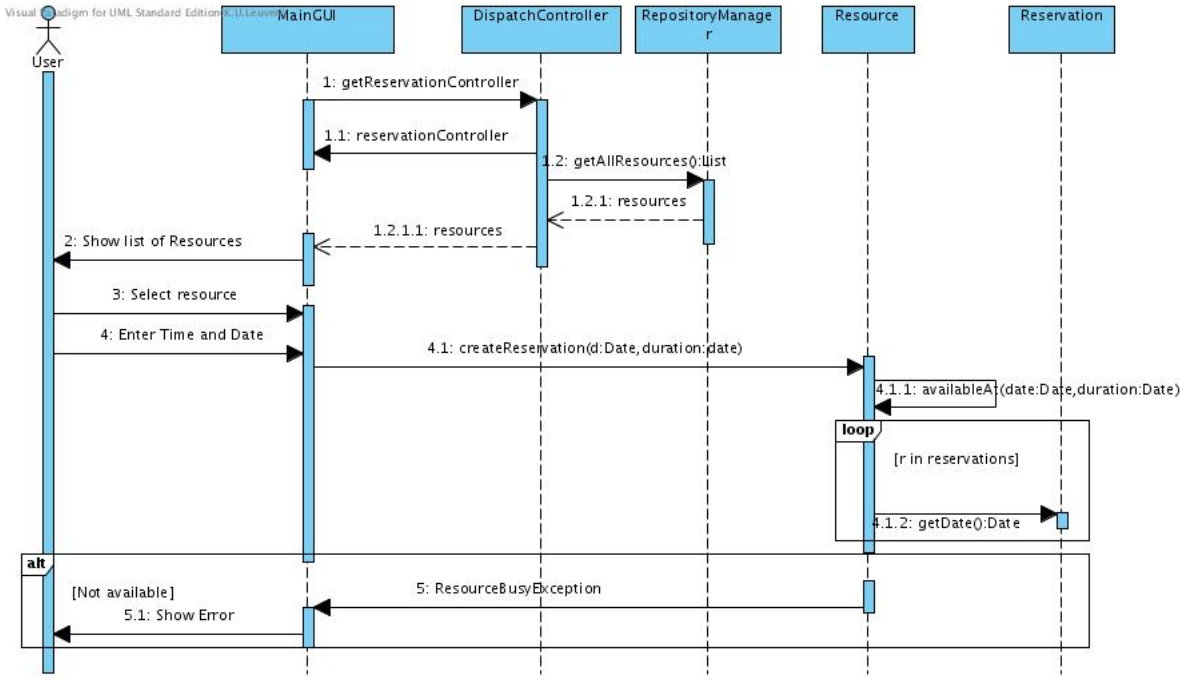

Figure 12: Create Reservation Sequence Diagram

4.4 Project manipulation

A project is an instance that could contain Tasks. It does not have any binding to other objects and can not be stored in an object. Therefore, a Project is stored the same way a Resource is. By using repositories. See the subsection 'Collecting Data'.

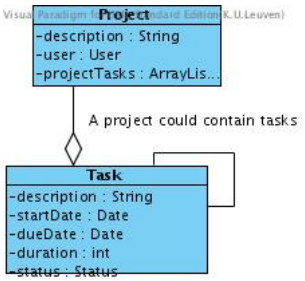

Figure 13: Project Class Diagram

The responsibilities of the Project class are as follows:

- Keeping track of its own details (description)
- Keeping track of the Tasks that are in the project
- Binding or removing Tasks from or to the project

When the application user creates a Project, he becomes a owner of that Project, this is how the User acts as a container of the Project Object. Tasks can be added to the Project however the User wishes to do this. However, it's not the User object that calls the Project constructor. It is our opinion that it is not the User's responsibility to create this object, rather then it's the Project's responsibility to aggregate the User.

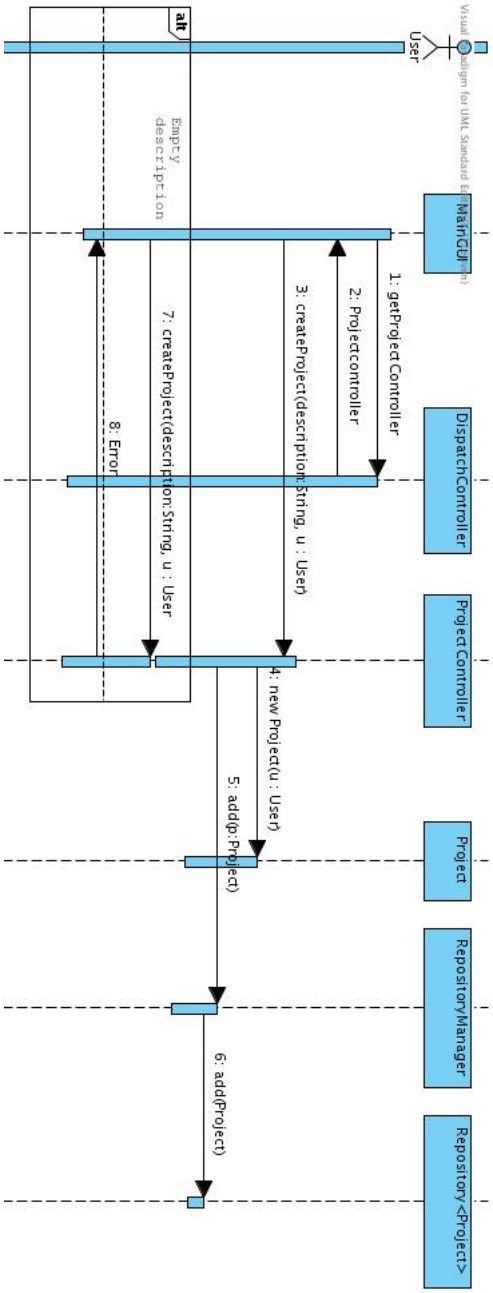

Figure 14: Create Project Sequence Diagram

Therefore, it's the Project's constructor that gets an argument User. This User object is then asked to add Project to it's list of Projects.

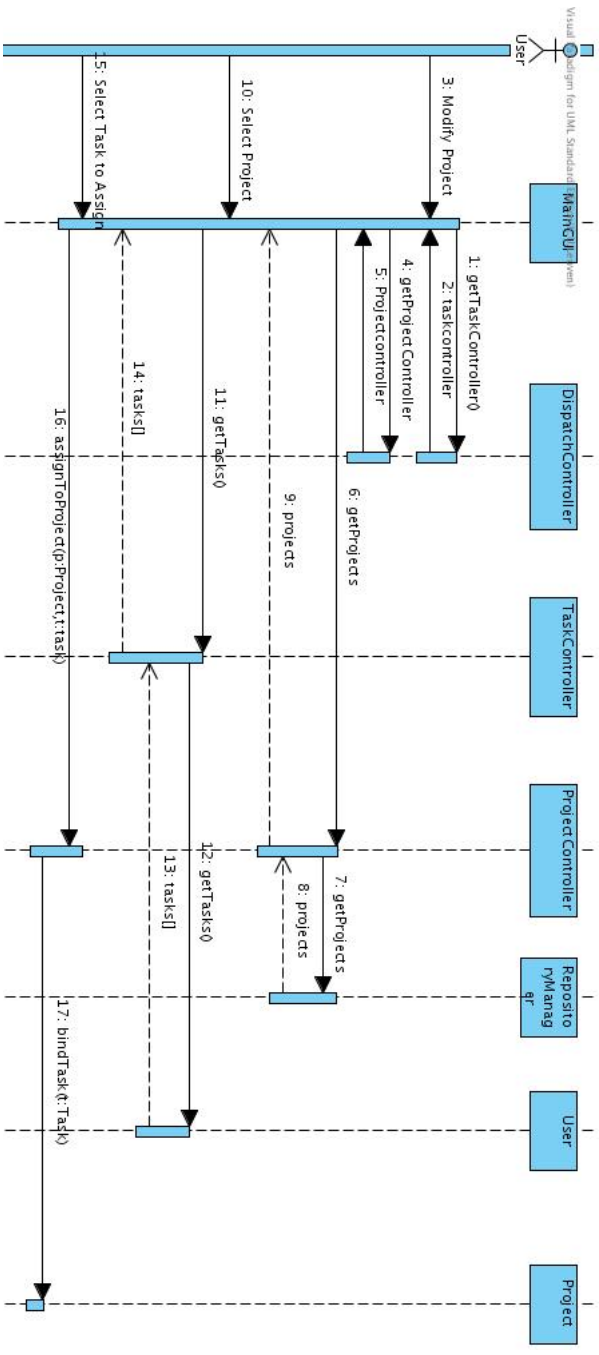

Figure 15: Assign Task to Project Sequence Diagram

It is also the Project's responsibility to bind a Task to the Project. We feel that this direction should be maintained because a Project 'contains' or

'aggregates' a Task. This binding is done in the Project's constructor.

5 Software Structures

5.1 Task states

Hier komt iets over het gebruik van State pattern. Ook observer zou ik hier zetten

5.2 Task overview

As discussed, we want to show the list of tasks in different ways. We might want them sorted by deadline or by duration. However, there is no reason the functionality should be limited to just these two.
Also, there is no reason to clutter the controller with these algorithms. We tried to find a way to split up the following functionalities:

- *Getting a list of tasks*
- *Sorting that list*
- *Filtering*
- *Returning the list*

We found our solution in the Strategy pattern: We start off by creating an instance of FocusWork. This instance is injected with an implementation of FocusStrategy and the current User. The instance of FocusWork will be responsible for getting the list of tasks. Since the sorting and filtering can differ from strategy to strategy, they are handled by the injected instance of FocusStrategy. That way, neither controller nor FocusWork (context) are responsible, or even aware, of how these algorithms work.
There was however the issue of creating an instance of FocusWork. We found it bad design to let both GUI nor TaskController call the constructor of an implementation of FocusStrategy. This would mean higher coupling between those classes. It would also mean a larger footprint when adding new FocusStrategies, more classes had to be adjusted. We therefore opted for using the Factory Method pattern to create instances of FocusWork with an injected Strategy.

The class FocusFactory takes care of Strategy constructors and injecting. The GUI now simply calls a method in this class, a pre-fabricated FocusWork is returned.
All this combined makes sure that when we create a new implementation of FocusStrategy, the footprint is returned to a minimum: We have to adjust the GUI to make sure it asks the right questions, and the FocusFactory is adjusted so right constructor is called in the right way.

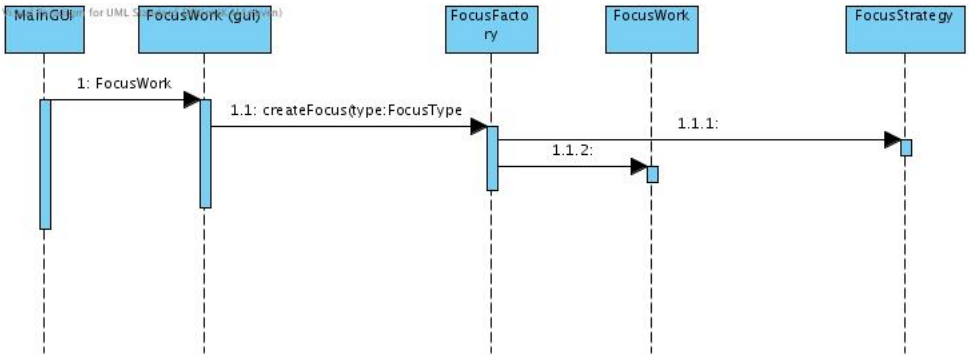

Figure 16: Create a type of FocusWork with an injected Strategy according to a given Type

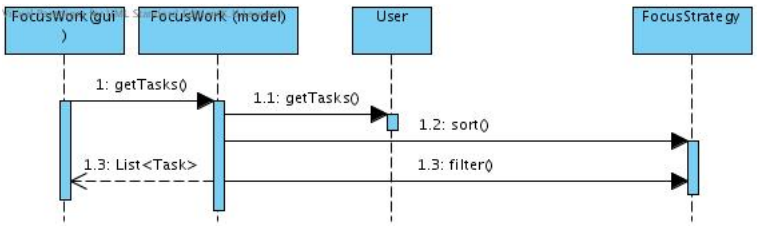

Figure 17: Get a list of tasks, manipulated by a certain Strategy. This strategy is interchangeable

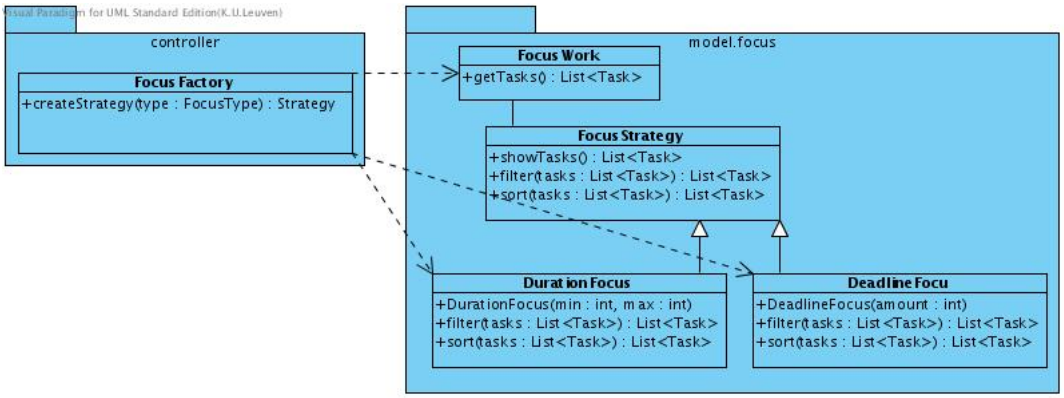

Figure 18: Specialised class diagram explaining the Strategy Pattern

5.3 Linking Tasks by dependency

Uitleggen hoe tasks gelinked worden zonder te veel binding = taskdependencymanager

5.4 Time representation

Clock uitleggen, nut en werking

5.5 Collecting data

Certain types of objects (Resources, Projects) are not required to have a binding with other objects. They can be simply instantiated on their own.
In a domain based model, this was a problem. Objects would be instantiated without a place to put them, they would not be contained. In the first iteration, we solved this by using a Singleton manager for Resource. Projects would be bound to a User.
This seemed bad design, as Singleton was unreliable as an Information Expert. It was too general and would fail in most Unit tests. Storing the Project in the User also seemed bad design, as they originally had no connection to each other.
A solution was found in using Repositories. We created Generic Repositories for Project, Resource and User that were contained in a RepositoryManager. The RepositoryManager would act as an independent Information Expert

and would manage objects in adding and removing them from their respective repositories.
By using overloaded add() and remove() operations, this RepositoryManager stayed easy to use and kept the underlying code abstract. We call this the Facade Pattern.
We use a technique borrowed from the Spring Framework called Dependency Injection to inject a reference to this RepositoryManager in every Controller. This way, they can all access the same information in an Object-Oriented Way.

6 Software Initialization

When the program is started, the first thing that is taken care of is the instantiation of the dispatch controller.

After that, an XML parser object is created to retrieve all the supplied information. The dispatch controller, together with the location of the XML file, are passed to the constructor of the parser. After this it can finally start parsing.

The first thing it will do is find all the resources in the file. It will then instantiate the ResourceManager singleton and start adding these resources. Once that has been completed, it will start looking for projects. It will create all the found instances. After this it looks for all the tasks and, once again, creates them all. Finally all the tasks, resources and projects are bound according to the specifications in the XML file. It will finally return the user object found in the file.

After this the GUI is instantiated. The user object from before is passed along with its constructor and then starts listening for any input.

The system has now fully started and is ready for use.

7 Testing

7.1 Technology

We decided to use JUnit4 for testing. JUnit4 has a few advantages compared to JUnit3. It uses annotations to define a Test. This gives the developer an

easy solution to testing for Exceptions.

7.2 Testing Approach

We decided to go for a defensive and multi-level testing approach. By multi-level we mean that we test both methods in Model classes (such as User, Task, Resource etc.) as well as testing the Controllers that call these methods (TaskController, ResourceController etc). This way we get a good view on where errors are: model, controller or view.
We also tested most methods for both cases. This means that we test both failure and succession of a method. Testing only for success does not guarantee a correct Exception is thrown, or success in all cases.

8 Project Management

8.1 Planning

The planning of our Team Assignment had the following planning:

Activity	From	To
First meeting & Discussion of our views on the project	9/10/2009	9/10/2009
Creation of a draft class diagram & working out System Sequence Diagrams	9/10/2009	12/10/2009
First meeting with our advisor.	12/10/2009	12/10/2009
Individual rework of Class Diagram	12/10/2009	14/10/2009
Comparing results and creation of definitive Class Diagram	14/10/2009	14/10/2009
Creation of Sequence Diagrams	14/10/2009	16/10/2009
Development of Model classes and Controller classes. Building of GUI structure	16/10/2009	19/10/2009
Code review by team and rewriting certain functionality's	19/10/2009	26/10/2009
Start writing report & finishing UML diagrams	24/10/2009	27/10/2009

8.2 Teamwork

We focused on a close teamwork. We started the project with a team discussion on how everyone saw the project and interpreted the assignment. This gave the team a general perspective and good grasp on how we wanted to implement it.
We have 3 weekly physical meetings, where 1 would be with our team advisor. In these meetings, we discussed what everyone had done in the past days. Whenever somebody was unclear, or the group had doubts about which method or pattern would be the best to use, time was never an issue to come to a general solution that seemed best to everyone.
We also used several team collaboration tools provided by Google such as Google Code and Google Groups. This gave advantages such as a mailing list, subversion with the option to review code at each revision, issue lists and hosted files.
We tried to keep all the information as centralized as possible by using a separate Subversion repository for the Visual Paradigm file. This way every team member always had the most up to date diagrams available.
Development of the project was divided in 4 groups of functionality: Controllers, Models, View and Testing. Each member of the team was assigned one of these tasks. Steven Thuriot took on Controllers and the parsing of XML, Kwinten Missiaen Models, Koen Van den dries the View and Bart Vangeneugden Testing. The report was structured and drafted by Bart Vangeneugden however every team member wrote about the part of development he was responsible for.
We had a total of 18 hours meeting physically for the project. Besides that every group member worked at home. A short estimate follows:

Member	Time(+/-)
Kwinten Missiaen	30
Steven Thuriot	25
Koen Van den dries	23
Bart Vangeneugden	25

9 Self-Evaluation

It is our opinion that we had a great team effort. However, we can improve. In this iteration we were too eager to get a first version of Class- and other diagrams ready. It would have been a better choice to make a good class-per-class analysis before drawing.

To conclude we had no real problems regarding teamwork. Also we learned a lot about project organization.