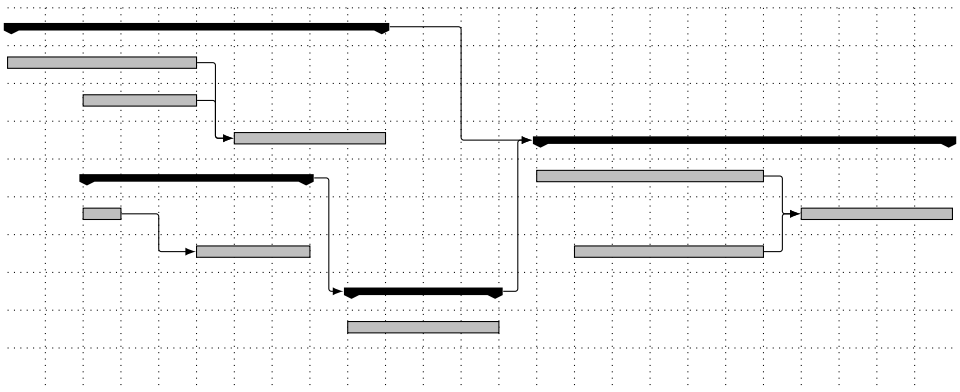

TaskMan: A Project-Oriented Task Manager

Software-ontwerp:
project 2014-2015

Iteration 2

Contents

1 Introduction

For the course *Software-ontwerp: project*, you will design and develop *TaskMan*, a project-oriented task manager.

In Section 2, we explain how the project is organized, discuss the quality requirements for the software you will design and develop, and describe how we evaluate the solutions. In Section 3, we show a diagram of the domain model and explain the problem domain of the application. The use cases are described in detail in Section 4.

2 General Information

In this section, we explain how the project is organized, what is expected of the software you will develop and the deliverables you will hand in.

2.1 Team Work

For this project, you will work in groups of four. Each group is assigned an advisor from the educational staff. If you have any questions regarding the project, you can contact your advisor and schedule a meeting. When you come to the meeting, you are expected to prepare specific questions and have sufficient design documentation available. *If the design documentation is not of sufficient quality, the corresponding question will not be answered.* It is your own responsibility to organize meetings with your advisor and we advise to do this regularly. Experience from previous years shows that groups that regularly meet with their advisors produce a higher quality design.

If there are problems within the group, you should immediately notify your advisor. Do not wait until right before the deadline or the exam!

To ensure that every team member practices all topics of the course, a number of roles are assigned by the team itself to the different members at the start of each iteration (or shortly thereafter in case of the first iteration). A team member that is assigned a certain role will give the presentation or demo corresponding to that role at the end of the iteration. That team member is *not supposed to do all of the work concerning his task*! He must, however, take a coordinating role in that activity (dividing the work, sending reminders about tasks to be done, make sure everything comes together, etc.), and be able to answer most questions on that topic during the evaluation.

The following roles will be assigned round-robin:

Design Coordinator The design coordinator has an active role in making the design of your software. In addition to knowing the design itself, he knows *why* these design decisions were taken, and what the alternatives are.

Testing Coordinator The testing coordinator has an active role in planning, designing, and writing the tests for the software. He knows which kinds of tests are needed (scenario, unit) for testing various parts of the software, which concrete tests are chosen (success cases, failure cases, corner cases, ...), and which techniques are used (mocking, stubbing, ...).

Domain Coordinator The domain coordinator has an active role in interpreting and extending the domain model. He knows which parts of the domain are relevant for the application.

The goal of these roles is to make every team member participate in all aspects of the development of your system. **Every team member must be able to explain the used domain model, the design of the system, and the functioning of your test suite.**

2.2 Iterations

The project is divided into 3 iterations. In the first iteration, you will implement the base functionality of the software. In subsequent iterations, new functionality will be added and/or existing functionality will be changed.

2.3 The Software

The focus of this course is on the *quality* (maintainability, extensibility, stability, readability,...) of the software you write. We expect you to use the development process and the techniques that are taught in this course. One of the most important concepts are the General Responsibility Assignment Software Principles (GRASP). These allow you to talk and reason about an object oriented design. You should be able to explain all your design decisions in terms of GRASP.

You are required to provide class and method documentation as taught in previous courses (e.g. the OGP course). When designing and implementing your system, you should use a *defensive programming style*. This means that the *client* of the public interface of a class cannot bring the objects of that class, or objects of connected classes, in an inconsistent state.

Unless explicitly stated in the assignment, you do not have to take into account persistent storage, security, multi-threading, and networking. If you have doubts about other non-functional concerns, please ask your advisor.

2.4 User Interface

You are expected to provide a functional user interface for the specified use cases, with understandable input questions and output statements. This can be as simple as a text-based command-line user interface. Although a user interface is obligatory, its design and code are not considered for grading, i.e. you will not gain any points for making a nicer user interface. You will, however, lose many points if the quality of your software is lacking because you spent too much time on the user interface or if there is too much coupling between the user interface and your domain layer. This coupling should be kept low since "interaction with the user" is just another responsibility and like all responsibilities it should be delegated according to GRASP.

2.5 Testing

All functionality of the software should be tested. **For each use case, there should be a dedicated scenario test class.** For each use case flow, there

should be at least one test method that tests the flow. Make sure you group your test code per step in the use case flow, indicating the step in comments (e.g. `// Step 4b`). Scenario tests should not only cover success scenarios, but also negative scenarios, i.e., whether illegal input is handled defensively and exceptions are thrown as documented. You determine to which extent you use unit testing. The testing coordinator briefly motivates the choice during the evaluation of the iteration.

Tests should have good coverage, i.e. a testing strategy that leaves large portions of a software system untested is of low value. Several tools exist to give a rough estimate of how much code is tested. One such tool is Eclemma[1]. If this tool reports that only 60% of your code is covered by tests, this indicates there may be a serious problem with (the execution of) your testing strategy. However, be careful when drawing conclusions from both reported high coverage and reported low coverage (understand why you should be careful).

The testing coordinator is expected to use a coverage tool and briefly report the results during the evaluation of the iteration.

2.6 UML Tools

There are many tools available to create UML diagrams depicting your design. You are free to use any of these as long as it produces correct UML.

One of these UML tools is Visual Paradigm. Instructions to run Visual Paradigm in the computer labs is described in the following file:
`/localhost/packages/visual_paradigm/README.CS.KULEUVEN.BE`

This file also contains the location of the license key that you can use on your own computer.

2.7 What You Should Hand In

Exactly *one* person of the team hands in an electronic ZIP-archive via Toledo. The archive contains the items below and follows the structure defined below. *Make sure that you use the prescribed directory names!*

- directory `groupXX` (where XX is your group number (e.g. 01, 12, ...))
  - `doc`: a folder containing the Javadoc documentation of your entire system
  - `diagrams`: a folder containing UML diagrams that describe your system (at least one structural overview of your entire design, and sufficient detailed structural and behavioural diagrams to illustrate every use case)
  - `src`: a folder containing your source code
  - `system.jar`: an executable JAR file of your system

When including your source code into the archive, make sure to *not include files from your version control system.* If you use subversion, you can do this with the the `svn export` command, which removes unnecessary repository folders from the source tree.

[1] http://www.eclemma.org

Make sure you choose relevant file names for your analysis and design diagrams (e.g. `SSDsomeOperation.png`). You do **not** have to include the project file of your UML tool, only the exported diagrams.

We should be able to start your system by executing the JAR file with the following command: `java -jar system.jar`.

2.7.1 Late Submission Policy

If the zip file is submitted N minutes late, with $0 \leq N \leq 240$, the score for all team members is scaled by $(240 - N)/240$ for that iteration. For example, if your solution is submitted 30 minutes late, the score is scaled by 87.5%. So the maximum score for an iteration for which you can earn 4 points is reduced to 3.5. If the zip file is submitted more than 4 hours late, the score for all team members is 0 for that iteration.

2.7.2 When Toledo Fails

If the Toledo website is down – and *only* if Toledo is down – at the time of the deadline, submit your solution by e-mailing the ZIP-archive to your advisor. The timestamp of the departmental e-mail server counts as your submission time.

2.8 Evaluation

After iteration 1, and again after iteration 2, there will be an intermediate evaluation of your solution. An intermediate evaluation lasts 15 minutes and consists of: a presentation about the design and the testing approach, accompanied by a demo of the tests.

The intermediate evaluation of an iteration will cover only the part of the software that was developed during that iteration. Before the final exam, the *entire* project will be evaluated. It is your own responsibility to process the feedback, and discuss the results with your advisor.

The evaluation of an iteration is planned in the week after that iteration. Immediately after the evaluation is done, you mail the PDF file of your presentation to Prof. Bart Jacobs & Tom Holvoet and your advisor.

2.8.1 Presentation Of The Current Iteration

The main part of the presentation should cover the design. The motivation of your design decisions *must* be explained in terms of GRASP principles. Use the appropriate design diagrams to illustrate how the most important parts of your software work. Your presentation should cover the following elements. Note that these are not necessarily all separate sections in the presentation.

1. A discussion of the high level design of the software (use GRASP patterns). Give a rationale for all the important design decisions your team has made.

2. A more detailed discussion of the parts that you think are the most interesting in terms of design (use GRASP patterns). Again we expect a rationale here for the important design decisions.

3. A discussion of the extensibility of the system. Briefly discuss how your system can deal with a number of change scenarios (e.g. extra constraints, additional domain concepts,...).

4. A discussion of the testing approach used in the current iteration.

5. An overview of the project management. Give an approximation of how many hours each team member worked. Use the following categories: group work, individual work, and study (excluding the classes and exercise sessions). In addition, insert a slide that describes the roles of the team members of the current iteration, and the roles for the next iteration. Note that these slides do not have to be presented, but we need the information.

Your presentation should not consist of slides filled with text, but of slides with clear design diagrams and keywords or a few short sentences. The goal of giving a presentation is to communicate a message, not to write a novel. All design diagrams should be *clearly readable* and use the correct UML notation. It is therefore typically a bad idea to create a single class diagram with all information. Instead, you can for example use an overview class diagram with only the most important classes, and use more detailed class diagrams to document specific parts of the system. Similarly, use appropriate interaction diagrams to illustrate the working of the most important (or complex) parts of the system.

2.9 Peer/Self-assessment

In order for you to critically reflect upon the contribution of each team member, you are asked to perform a peer/self-assessment within your team. For each team member (including yourself) and for each of the criteria below, you must give a score on the following scale: *poor/lacking/adequate/good/excellent*. The criteria to be used are:

- Design skills (use of GRASP and DESIGN patterns, ...)
- Coding skills (correctness, defensive programming, documentation,...)
- Testing skills (approach, test suite, coverage, ...)
- Collaboration (teamwork, communication, commitment)

In addition to the scores themselves, we expect you to briefly explain for each of the criteria why you have given these particular scores to each of the team members. The total length of your evaluation should not exceed 1 page.

Please be fair and to the point. Your team members will not have access to your evaluation report. If the reports reveal significant problems, the project advisor may discuss these issues with you and/or your team. Please note that your score for this course will be based on the quality of the work that has been delivered, and not on how you are rated by your other team members.

Submit your peer/self-assessment by e-mail to both Prof. Bart Jacobs & Tom Holvoet and your project advisor, using the following subject:

[SWOP] peer-/self-assessment of group $groupnumber$ by $firstname$ $lastname$

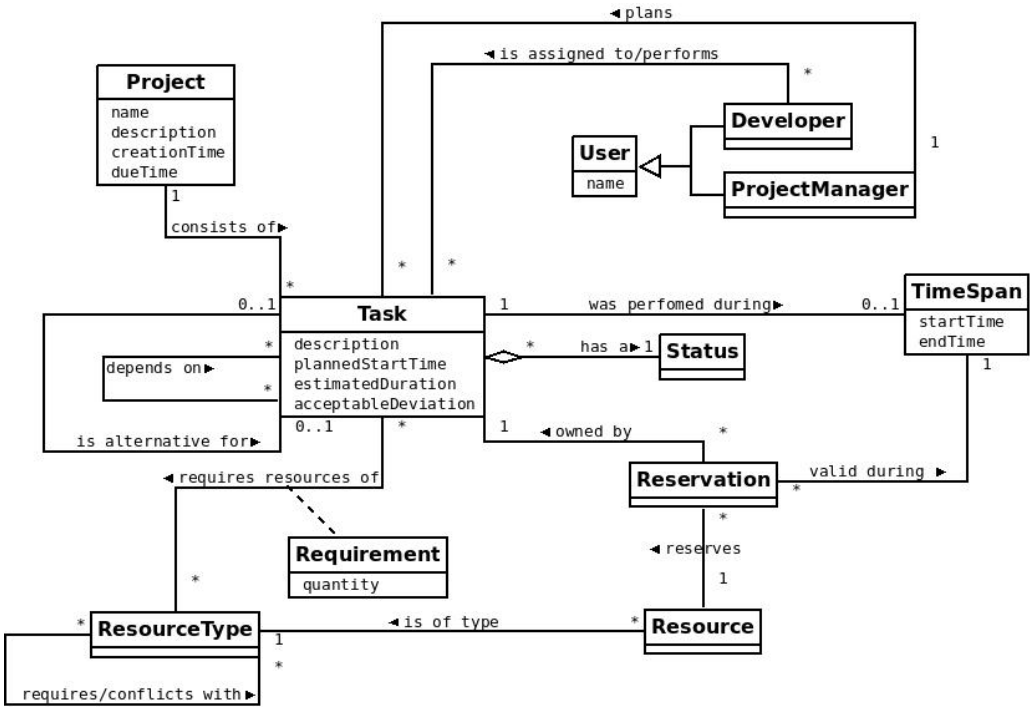

Figure 1: Domain model

2.10 Deadlines

- The deadline for handing in ZIP-archive on Toledo is **24 April, 2015, 6pm**.
- The deadline for submitting your peer/self-assessment is **26 April, 2015, 6pm**, by e-mail to both your project advisor and Prof. Bart Jacobs & Tom Holvoet.

3 Problem Domain Analysis

All extensions to the domain model in this iteration are highlighted in red. In this iteration we introduce resources which have some specific resource type. Tasks require resources to be successfully executed by a team of developers and these resources should be reserved so that they are available when a task is planned. Project managers plan the tasks and make sure that the necessary resources are reserved. A new simulation use case enables a project manager to plan some tasks and revert the planning of the tasks if he fails to come up with a nice schedule.

3.1 Domain Model

Figure 1 shows the domain model of TaskMan.

3.2 Domain Description

The domain model shows the concepts for TaskMan, a project-oriented task manager. This task manager will be used by an independent software developer to

manage his development projects. Below, each of the concepts from the domain model is explained in more detail.

Task: a task is a unit of work that can be performed by a team of developers. A task is assigned to an unfinished project upon creation. Each task has a description, a planned start time, an estimated duration expressed in minutes, and an acceptable deviation, expressed as a percentage (e.g. 10%). The planned start time and the estimated duration together, determine the timespan in which the task is planned to be executed. The estimated duration represents the estimated time it should take to complete the task, and the acceptable deviation the time period within which the task can be finished on time. A task can be finished early (even before the earliest acceptable deviation from the estimated duration), on time (within the acceptable deviation) or with a delay (even later than the latest acceptable deviation). A task can depend on other tasks, meaning that they have to be finished before the execution of the dependent task can start. Naturally, no loops are allowed in the dependency graph. Additionally, when a task fails (see Status), an alternative task can be created to replace the current task. For example, if the task to *deploy apache* is marked as failed, the project manager may create an alternative task *deploy nginx*. This alternative task replaces the failed task with respect to dependency management or determining the project status (ongoing or finished). The time spent on the failed task is however counted for the total execution time of the project.

Project: a project consists of multiple tasks required to complete the project. A project has a name, a description, a creation time, and a due time by which it should be finished. The system should help the project manager to manage his projects, meaning that a project should be marked as either ongoing or finished. A project can only be finished if it has at least one task, and all its tasks are finished (or each failed task has a finished alternative). Additionally, for ongoing projects, the system should indicate whether the project is estimated to finish on time or over time (based on the time spans of the finished tasks, the estimated duration of the unfinished tasks ignoring the acceptable deviation and parallelizing as mush as possible, and using a Monday to Friday working week with 8 hours a day). For finished projects, the system should indicate the total delay that occurred within the tasks of the project, and whether the project finished on time or not (based on the time spans of the tasks within the project).

TimeSpan: the time span of a (failed or finished) task contains the actual start time and end time. The time spans can be used to calculate the delay of a task, or to assess the estimated time allocated to tasks (to improve future project management). The timestamps are expressed in hours and minutes. Accuracy up to seconds is not required. The time span of different tasks may overlap, except if they depend on each other (if task B depends on task A, then A must have a time span that ends before the time span of B begins).

Status: the status of a task is one of the following: *available*, *unavailable*, *executing*, *finished* or *failed*. A task is available for a specific developer when

it can be executed (i.e. can move to the status executing), meaning that all its dependencies are fulfilled. Concretely, this means that the following conditions must be met:

- The developer for which the task is available is assigned to the task (the use case *Update Task Status* only shows the tasks for the current developer).
- All developers assigned to the task are available at the current system time.
- All resources for the task are available at the current system time.
  - If the current system time falls within a task's planned timespan, all resources need to have the appropriate reservations.
  - If the current system time falls outside the task's planned timespan, a set of required resources needs to be available at the current system time, for at least the estimated duration of the task. For example, if a task is planned for next week, but all resources are available today, it is available for an assigned developer and can be executed today.

Note that during unplanned execution (i.e. current system time falls outside the task's planned timespan), it is possible to use different resources than those that were reserved for the (future) planned timespan, unless the project manager chose specific resource instances for this task (see use case Plan Task). A task is unavailable when the previously mentioned conditions are not met. A task is executing when the developer marks he has started executing the task through the use case *Update Task Status*. If multiple developers are assigned, only one of them has to start the execution to cause change in the status. Other statuses that can be assigned are *finished* and *failed*. A task can only become finished if it was executing.

Resource: executing tasks may require the use of certain resources. For example, to give a demo at a customer's location, the developer may need the *demo kit* and a *car*. Example resources are *cars*, *whiteboards*, *conference rooms*, a *demo kit*, the *distributed testing setup* and the *data center*. All resources belonging to the company are available 24/7. One exception is the *data center*, which is only accessible between 12:00 and 17:00 on working days.

Resource Type: each resource has a specific type, and for each type of resource, numerous instances can exist. For example, the resource type *Car* can consist of numerous cars that are available in the company. Whenever a task is planned, the project manager can choose to assign any resource of a specific type, or can choose a specific instance of this type. When a task is planned, specific resources will be reserved for use during the task's planned timespan (see use case *Plan Task*).
Resources of a specific resource type can conflict with or depend on resources of another type. For example it could be that it is not allowed to reserve 2 conference rooms for the same task or to book a *whiteboard* and a *demo kit* for the same task. A possible example of a dependency of resource types is

the following: when reserving a *conference room* for a specific task, a *demo kit* must also be reserved. These constraints should be flexible and not hard coded as they can be specified in the input file described in subsection 3.4.

Reservation: to ensure that a task can be executed at a given planned timespan, resources can be reserved. A reservation reserves a specific resource for a specific task, for the given timespan on the given day. When a task is marked as *completed* or *failed*, its future reservations are cleared. Note however that reservations that have already been consumed (i.e. when they are partly or completely in the past), should be kept for accounting purposes. When a task is completed in the middle of a reservation, the part that is consumed should be stored, and the remainder (i.e. the part in the future) should be freed. Similarly, when the end time of a reservation has passed, they have to be kept for accounting purposes, but no longer count as a valid resource reservation in the system (i.e. the task associated with the reservation will have no reservations anymore at this point in time.).

User: a user can perform operations within the system, as defined by the use cases.

Developer: a developer is responsible for the day to day tasks within the company, which can include development, deployment, testing, etc. Developers work 8-hours days, from 08:00 to 17:00 with a one hour break between 11:00 and 14:00 (he can choose when he takes his one hour break within this timespan).

Project Manager: a project manager has the capability to assign tasks to concrete developers, and plan tasks within a specific timespan. Essentially, the project manager is responsible for planning and distributing the work that needs to be done. The availability during the working day of a project manager (i.e. his working hours) is not relevant.

3.3 Example Scenario

Below is an example scenario of a project and its tasks that need to be managed with *TaskMan*:

- Current time: 09/02/2015, 08:00, Project Manager (project/task creation)
  - Create project *MobileSteps*: develop mobile app for counting steps using a specialized bracelet (Start 09/02/2015, Due 13/02/2015)
    - Create task 1: design system, 480 minutes, 0% margin, requires *whiteboard*
    - Create task 2: implement system in native code, 128 minutes, 50% margin, depends on task 1
    - Create task 3: test system, 480 minutes, 0% margin, depends on task 2, requires *distributed testing setup*
    - Create task 4: write documentation, 480 minutes, 0% margin, depends on task 2

  - Plan task 1 on 09/02/2015 at 09:00, with developer X and one resource of type *whiteboard*
  - Plan task 2 on 10/02/2015 at 10:00, with developer X and developer Y
  - Plan task 3 on 12/02/2015 at 09:00, with developer X and one resource of type *distributed testing setup*
  - Plan task 4 on 13/02/2015 at 12:00, with developer X

- Current time: 10/02/2015, 08:00 (finished one task, started second task)
  - Project *MobileSteps*: ongoing, on time (Due 13/02/2015)
    * Task 1 finished: design system, 480 minutes, 0% margin, started 09/02/2015 08:00, finished 09/02/2015 16:00
    * Task 2 executing : implement system in native code, 128 minutes, 50% margin, depends on task 1
    * Task 3 unavailable: test system, 480 minutes, 0% margin, depends on task 2
    * Task 4 unavailable: write documentation, 480 minutes, 0% margin, depends on task 2

- Current time: 11/02/2015, 08:00 (failed one task)
  - Project *MobileSteps*: ongoing, on time (Due 13/02/2015)
    * Task 1 finished: design system, 480 minutes, 0% margin, started 09/02/2015 08:00, finished 09/02/2015 16:00
    * Task 2 failed : implement system in native code, 128 minutes, 50% margin, depends on task 1, started 10/02/2015 08:00, finished 10/02/2015 16:00
    * Task 3 unavailable: test system, 480 minutes, 0% margin, depends on task 2
    * Task 4 unavailable: write documentation, 480 minutes, 0% margin, depends on task 2

- Current time: 11/02/2015, 08:00 (created and planned alternative task)
  - Project *MobileSteps*: ongoing, over time (Due 13/02/2015 (8 working hours short))
    * Task 1 finished: design system, 480 minutes, 0% margin, started 09/02/2015 08:00, finished 09/02/2015 16:00
    * Task 2 failed : implement system in native code, 128 minutes, 50% margin, depends on task 1, started 10/02/2015 08:00, finished 10/02/2015 16:00
    * Task 3 unavailable: test system, 480 minutes, 0% margin, depends on task 5
    * Task 4 unavailable: write documentation, 480 minutes, 0% margin, depends on task 5

        * Task 5 available: implement system with phonegap, 480 minutes, 100% margin, depends on task 1, alternative to task 2

- Current time: 13/02/2015, 16:00 (finished project)
  - Project *MobileSteps*: finished, on time (Due 13/02/2015)
    * Task 1 finished: design system, 480 minutes, 0% margin, started 09/02/2015 08:00, finished 09/02/2015 16:00
    * Task 2 failed : implement system in native code, 128 minutes, 50% margin, depends on task 1, started 10/02/2015 08:00, finished 10/02/2015 16:00
    * Task 3 finished: test system, 480 minutes, 0% margin, depends on task 5, started 12/02/2015 08:00, finished 12/02/2015 16:00
    * Task 4 finished: write documentation, 480 minutes, 0% margin, depends on task 5, started 13/02/2015 08:00, finished 13/02/2015 16:00
    * Task 5 finished: implement system with phonegap, 480 minutes, 100% margin, depends on task 1, alternative for task 2, started 11/02/2015 08:00, finished 11/02/2015 16:00

3.4 System Initialisation

It must be possible to initialize your system with data from a file. An example file and example code for parsing the file are on Toledo.

Note that this feature is essential for evaluation purposes, as we will provide you with a data file in the same format to use during the final project demonstration.

4 Use Cases

Figure 2 shows the use case diagram for TaskMan. The following sections describe the various use cases in detail.

4.1 Use Case: Show Projects

Primary Actor: User.

Main Success Scenario:

1. The user indicates he wants to see an overview of all projects in the system.
2. The system shows a list of projects with their status.
3. The user selects a project to view more details.
4. The system presents an overview of the project details, including a list of tasks with their task details (i.e. status, percentage of overdueness and an indication if the deviation is within or outside accepted deviation) and the estimated end time of a project assuming that unplanned tasks or unfinished tasks with a planned time span before the current system time, end now.

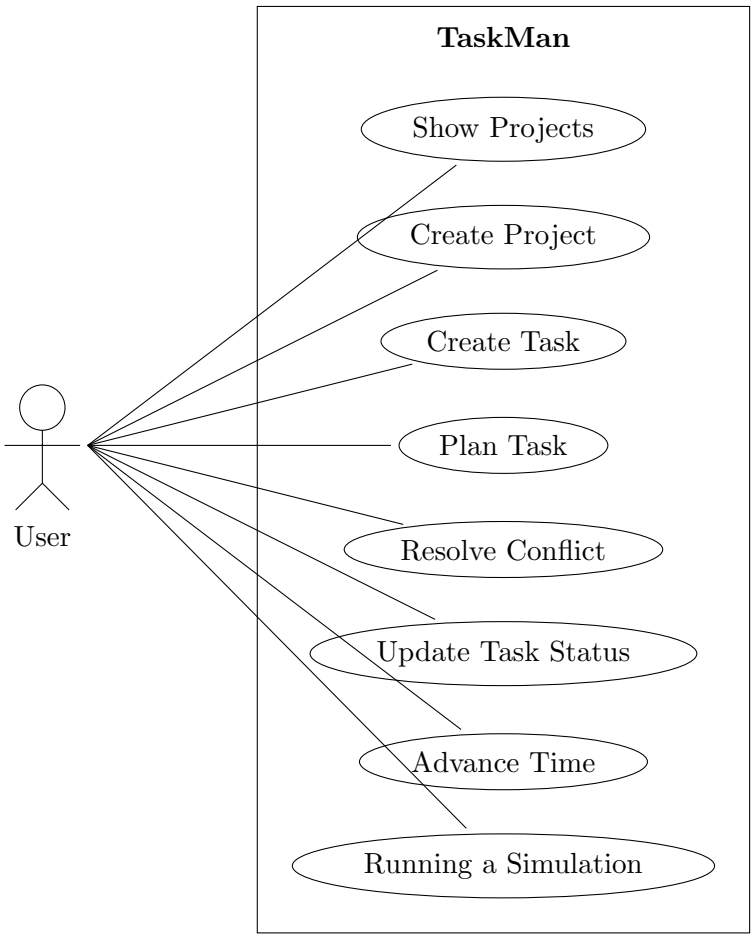

Figure 2: Use case diagram for TaskMan.

5. The user selects a task to view more details.
6. The system presents an overview of the task details.

Extensions:

3-5a. The user indicates he wants to leave the overview:
   1. The use case ends here.

4.2 Use Case: Create Project

Primary Actor: Project Manager.

Success End Condition: A new project has been created.

Main Success Scenario:

1. The user indicates he wants to create a new project.
2. The system shows the project creation form.
3. The user enters all the project details.
4. The system creates the project.

Extensions:

3a. The user indicates he wants to cancel creating the project:
   1. The use case ends here.

4a. The entered data is invalid:
   1. The system indicates the error and the use case returns to step 2.

4b. There are conflicts or missing requirements amongst resource types:
   1. The system describes the conflict or missing requirement and the use case returns to step 2.

4.3 Use Case: Create Task

Primary Actor: Project Manager.

Success End Condition: A new task was created and assigned to a project.

Main Success Scenario:

1. The user indicates he wants to create a new task.
2. The system shows the task creation form, allowing the user to enter all required details, including the required resource types and their necessary quantity (see section 3.2)
3. The user enters all the task details.
4. The system creates the task, assigns it to the selected project and updates required information.

Extensions:

3a. The user indicates he wants to cancel creating the task:

1. The use case ends here.

4a. The entered data is invalid:

1. The system indicates the error and the use case returns to step 2.

4.4 Use Case: Plan Task

Primary Actor: Project Manager.

Success End Condition: A task has been planned in a specific timeslot

Main Success Scenario:

1. The user indicates he wants to plan a task.
2. The system shows a list of all currently unplanned tasks and the project they belong to
3. The user selects the task he wants to plan.
4. The system shows the first three possible starting times (only considering exact hours, e.g. 09:00, and counting from the current system time) that a task can be planned (i.e. enough resource instances and developers are available)
5. The user selects a proposed time.
6. The system confirms the selected planned timespan and shows the required resource types and their necessary quantity as assigned by the project manager when creating the task. For each required resource type instance to perform the task, the system proposes a specific resource to make a reservation for.
7. The user allows the system to select the required resources.
8. The system shows a list of developers
9. The user selects the developers to perform the task.
10. The system makes the required reservations and assigns the selected developers.

Extensions:

3-9a. The user indicates he wants to cancel planning a task:

1. The use case ends here.

5a. The user indicates he wants to select another time:

1. The user enters the time and continues with step 6.

6a. The task's reservations conflict with another task:

1. The use case *resolve conflict* is executed.

8a. The user wants to allocate a specific resource instances for this task:

1. The user chooses the resource type he wants to allocate.

2. The system presents the list of available resources of the selected resource type.
3. The user chooses the desired resource.
4. *The use case returns to step 6.*

10a. The task's assigned developers conflict with another task:

1. The use case *resolve conflict* is executed.

4.5 Use Case: Resolve Conflict

Primary Actor: Project Manager.

Precondition: During the planning of a task, a conflict with other tasks has been detected

Success End Condition: A conflict between tasks has been resolved

Main Success Scenario:

1. The system shows the tasks whose reservations conflict with the task currently being planned.
2. The user chooses to move the task currently being scheduled.
3. *This use case ends, and the flow returns to step 4 in "Plan Task".*

Extensions:

2a. The user chooses to move the conflicting tasks:

1. The system executes step 4-7 from the use case *Plan Task* for each of the conflicting tasks. This process can be recursive.

4.6 Use Case: Update Task Status

Primary Actor: Developer.

Success End Condition: The status of a task has been updated.

Main Success Scenario:

1. The user indicates he wants to update the status of a task.
2. The system shows a list of all available tasks and the project they belong to
3. The user selects the task he wants to change.
4. The system shows the update form, allowing the user to select the new status and enter the required status details (see section 3.2).
5. The user selects a status and enters all the required details.
6. The system updates the task status.

Extensions:

3-5a. The user indicates he wants to cancel updating the task's status:

1. The use case ends here.

6a. The entered data is invalid:

1. The system indicates the error and the use case returns to step 4.

4.7 Use Case: Advance Time

Primary Actor: User.

Success End Condition: The internal clock of the system has been updated.

Main Success Scenario:

1. The user indicates he wants to modify the system time.
2. The system allows the user to choose a new timestamp.
3. The user enters a new timestamp.
4. The system updates the internal clock and updates all domain objects that depend on this clock.

Extensions:

3a. The user indicates he wants to cancel advancing the time:

1. The use case ends here.

4a. The entered timestamp is invalid (e.g. timestamp in the past):

1. The system indicates the error and the use case returns to step 2.

4.8 Use Case: Running a Simulation

Primary Actor: Project Manager.

Success End Condition: After a successful simulation, the system is updated according to the simulated use cases. After a canceled simulation, the system is unaltered.

Main Success Scenario:

1. The user indicates he wants to start a simulation.
2. The system show the possible use cases during the simulation (i.e. the currently supported use cases are Show Projects, Create Task, Plan Task)
3. The user selects and performs one of the possible use cases
4. The system asks if the user wants to continue the simulation, to cancel the simulation or to realize the simulation as if the executed use cases where actually performed outside a simulation
5. The user decides to realize the simulation
6. The system updates so that the simulation becomes real.

Extensions:

3-5a. The user indicates he wants to cancel the simulation:

1. The system restores its state from before the simulation.

5a. The user indicates he wants to continue the simulation:

1. The use case return to step 2.

Good luck!
The SWOP Team members