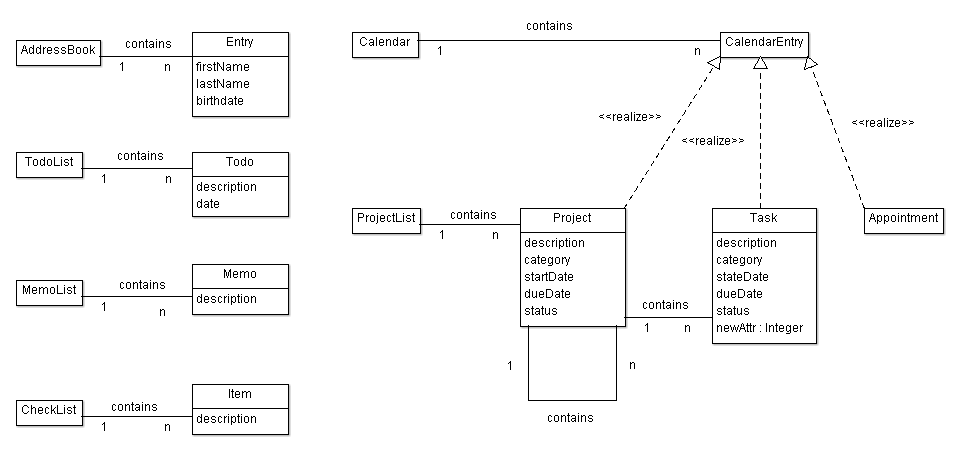
BORG CALENDAR – M3

Student Name and Number

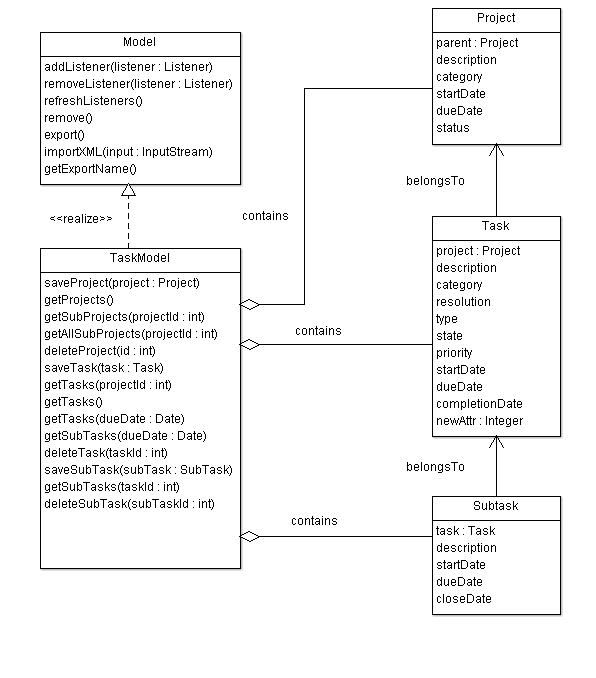
Hamid Shahrestani (9729747)  
Anjaneyulu Bodepudi (5973775)   
Manouchehr Azizi (5232287)   
Viet Hung Nguyen (9816240)

# Class Diagram of Actual System

**Conceptual Diagram**



**Actual Class Diagram**

****

We show here only five classes related to the project management features:

* Model is an abstract class for all model classes. Each model class is a singleton class which provides functionalities to access/manipulate entities for a feature. For example: task model provides functionalities to access/manipulate entities for project management feature; includes: Porject, Task, SubTask.
* TaskModel is a realization class of Model and is used to access/manipulate projects, tasks and subtasks.
* Project is a data class to hold data of a project. It provides no functionality related to a project, for example, get sub projects, get tasks, etc.
* Task is a data class to hold data of a task.
* SubTask is a data class to hold data of sub task.

**Comparing Conceptual Diagram & Actual Class Diagram**

* TaskModel in some degrees acts as ProjectList in conceptual diagram. But they are not totally the same: TaskModel not only contains projects, but also tasks and sub tasks.
* In both conceptual diagram & actual class diagram, a project may contain other projects. But in conceptual diagram, a task cannot contain other tasks. Actual class diagram has sub task class and a task may contain sub tasks.

**TaskModel class**

|  |
| --- |
| public class TaskModel extends Model implements … {  public void saveProject(Project p) throws Exception {}  public Collection<Project> getProjects() throws Exception {}  public Collection<Project> getSubProjects(int projectid) throws Exception {}  public Collection<Project> getAllSubProjects(int projectid){}  public void deleteProject(int id) throws Exception {}  public void savetask(Task task) throws Exception {}  public Collection<Task> getTasks() throws Exception {}  public Collection<Task> getTasks(int projectid) throws Exception {}  public Collection<Task> get\_tasks(Date d) {}  public void saveSubTask(Subtask s) throws Exception {}  public Collection<Subtask> getSubTasks() throws Exception {}  public void deleteSubTask(int id) throws Exception {}  } |

**Project class**

|  |
| --- |
| public class Project extends KeyedEntity<Project> implements CalendarEntity {  private Date StartDate;  private Date DueDate;  private String Description;  private String Category;  private String Status;  private Integer Parent;    public void setStartDate(Date value) {  StartDate = value;  }  public Date getStartDate() {  return StartDate;  }  // … other getters/setters  } |

# Code Smells and Possible Refactoring

1. Looking at TaskModel class we can see that it’s quite long: there is Large Class code smell here. Refactoring to fix this code smell in TaskModel:

* daysBetween(Date start, Date dd), daysLeft(Date dd) is not the responsibilities of TaskModel and can be moved out of it.
* …

1. In AppointmentModel class , method do\_todo

**public** **void** do\_todo(**int** key, **boolean** del, Date date) **throws** Exception has two different polymorphic methods which are distinguished with “if then else”:

* Delete : delete the todo when all done
* repeatSet: date date of the repeat that is being marked as done. If null, then the next todo is the one. If set, then all todos up to and including the date are marked as done

We can Extract two classes and instead of “if then Else” we implement it with

Polymorphism (strategy):

<<interface>> do\_todo

delete

repeatSet

1. In Day class, there’s an addToDay method that is quite long. We can reduce its size by introducing shorter methods within it. It also lacks enough comments, so by introducing self –explanatory methods, we make the code more comprehensible.

**private** **static** **void** addToDay(Day day, Collection<Integer> l, **int** year,

**int** month, **int** date) **throws** Exception

It can be shortened by introducing at least three short methods:

The top part of the method consists of the code to indicate whether a flag is public or private. We can introduce the method setAccessLevel (or something like that) to refactor this part of the code. This will increase the cohesion.

In the middle of the code there’s a very complicated method to indicate whether the loop should be continued or not. This is a complicated logic, because of the use of Boolean flags. We can simplify it by introducing another method and creating a class that contains the access level flags. Finally the bottom part can be shortened by adding three more methods.

* + - addAppointmentToDay(Appointment apt)
    - setVacation(Appointment apt)
    - setHoliday(Appointment apt)

1. In TaskTypes class, we have a toXml method that is reducing the cohesion of this class.

We move this method to a new class called TaskTypeSerializer and delegate this task to this class. We just need to introduce an instance of this class in our TaskTypes class and call its toXml method.

1. getInfo method in TaskModel class is not making the class cohesive. We move it to another class called TaskModelInformation to make the class more cohesive. We delegate the task to an instance of the TaskModelInformation class that we introduce in TaskModel class.
2. There are some database related methods in TaskModel class that are making this class too big. We need to move all these methods to another class and delegate all the responsibilities of these methods to an instance of the newly created class. We can call it TaskModelDB and move the following methods to that class:
   * + beginTransaction
     + commitTransaction
     + rollbackTransaction
     + addLog
     + saveLog
3. There’s an importXml method in TaskModel class that is not cohesive at all. We want to move it to a TaskModelXmlImporter class and delegate the responsibility of importing xml to this class. Beside lack of cohesive structure, the method is too long and it is using methods that take care of database related tasks. We first need to shorten the method by introducing shorter methods, and delegating the database related tasks to some other objects and classes. We can introduce the following methods:
   1. Unmarshal : it encapsulates the top part of the importXml method.
   2. executeSql: it wraps the following 5 lines that does database related tasks. Next, we move these tasks, to a different class that only does database related tasks to increase cohesion.
   3. handleOldImports: The middle part of the method can be wrapped in this method.
   4. importIntoEmptyDb: the last 40 lines of the method can be wrapped in this method.