BORG CALENDAR – M3

Student Name and Number

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

# Description of the software architecture

To create the Actual System Model we used a Reverse Engineering tool namely “Object Aid UML Diagram” which is an Eclipse Plugin. The source code contains 18 packages, but for simplicity we only examine the core entity classes contained within “net.sf.borg.model” and a few classes within “net.sf.borg.model.entity” package. The main correspondence of our modeling is with the classes contained in these packages. The other 16 packages are mostly taking care of UI tasks (10 packages), tools (3 packages), database tools (2 packages), and one controller class (pure fabrication) which is responsible to starting up the model and spawning various threads, including the main UI thread and various timer threads. It also handles shutdown. In total there are 148 classes of which 85 classes are UI classes.

Examining our use cases closely, there are several main entities in the calendar that can be added, removed, edited, etc. These entities are: Project, Task, Subtask, memo and ToDo.

Within our entities package we have data classes that hold data without any functionalities. For example the following classes are included in this package: Project, Task, Subtask, and Memo. We also included these classes in our “Conceptual Class Diagram”. These classes along with classes contained in Model package, are in close relationship that provide core functionalities of the software.

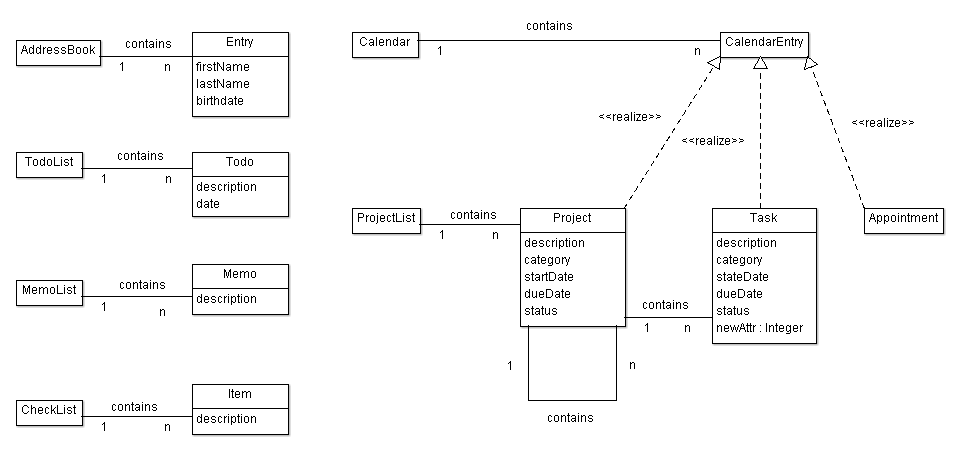
Model classes all extend a model interface. Each model class is a singleton class which provides functionalities to access/manipulate entities for a feature. Their primary function is to do CRUD methods. For example: TaskModel provides functionalities to access/manipulate entities for project management feature; includes: Porject, Task, SubTask.

The following classes are included in the actual class diagram:

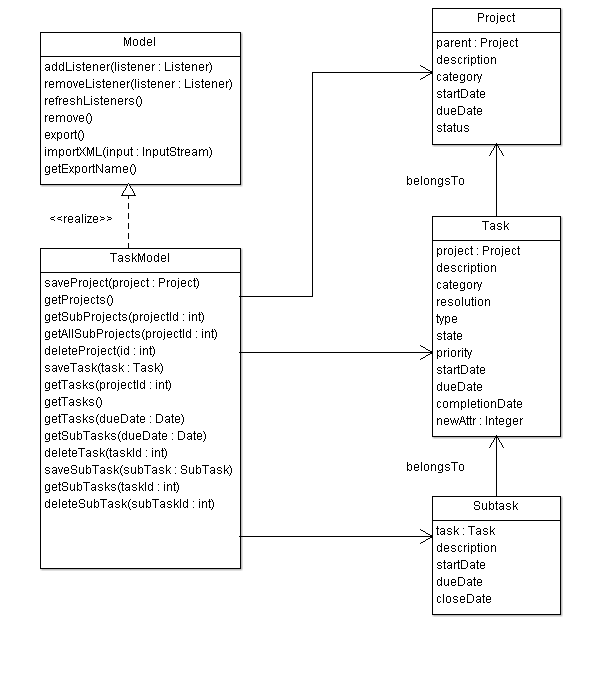
* Model: Abstract class that all the other model classes implement it. It has basic operations like addListener(), removeListener(), import(), export().
* TaskModel: It is a realization class of Model and is used to access/manipulate projects, tasks and subtasks.
* MemoModel: It is a realization of Model with CRUD methods to manipulate Memo objects
* AppointmentModel: It is a realization of Model with CRUD methods to manipulate Appointment objects
* AddressModel: It is a realization of Model with CRUD methods to manipulate Address objects
* CheckListModel: It is a realization of Model with CRUD methods to manipulate CheckList objects
* Other classes: In total there are 18 classes. The other classes included in this package perform utility tasks like searching, import, export, them provider and etc.

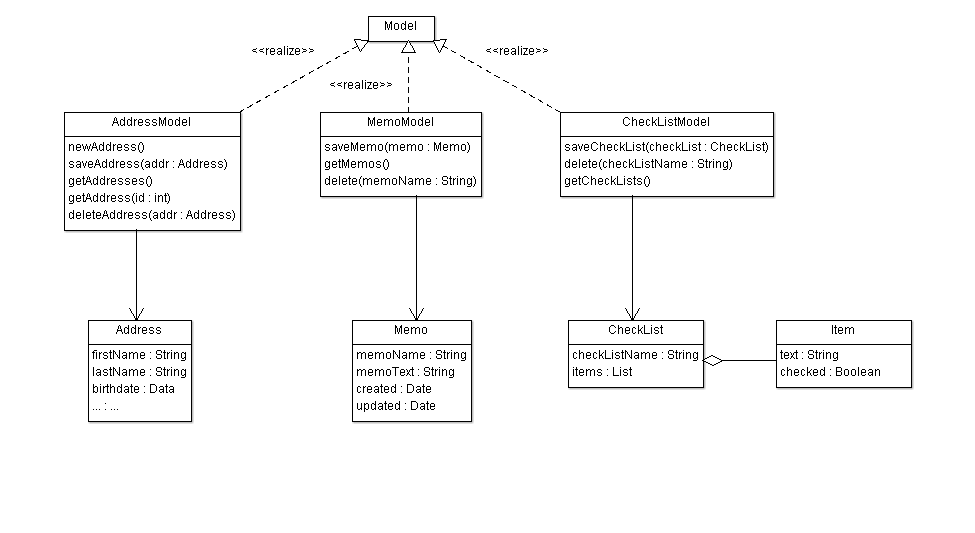
Each of these model classes are in association with their corresponding data class e.g. MemoModel with Memo, TaskModel with Task and etc.

* 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.
* In actual class diagram, we don’t have TodoList and Todo: Borg implementation use appointments and tasks with due dates.
  1. Our Conceptual Class Diagram from M2



* 1. Actual System Class Diagram





* 1. Example of source

We intend to portray the relationship between the following classes: Task, Subtask, and Project in the following code snippet. The comments in the code provide more detailed information.

|  |
| --- |
| **public** **class** Task **extends** KeyedEntity<Task> **implements** CalendarEntity {  **private** String Description;  **private** String Resolution;  **private** String Category;  **private** Integer Project;    **public** **boolean** isTodo(){ **return** **true**; }  @Override  **public** Date getNextTodo(){ **return** **null**; }  @Override  **protected** Task clone() {  Task dst = **new** Task();  dst.setKey( getKey());  dst.setDescription( getDescription() );  dst.setResolution( getResolution() );  dst.setCategory( getCategory() );  dst.setProject( getProject() );  // The above line shows that Task is in relation with Project  **return**(dst);  }  } |

|  |
| --- |
| **public** **class** Subtask **extends** KeyedEntity<Subtask> **implements** CalendarEntity {  **private** **static** **final** **long** *serialVersionUID* = -5794908342032518360L;  **private** Integer Task;  **private** String taskDesc = **null**;  @Override  **protected** Subtask clone() {  Subtask dst = **new** Subtask();  dst.setKey(getKey());  dst.setStartDate(getStartDate());  dst.setCloseDate(getCloseDate());  dst.setDueDate(getDueDate());  dst.setDescription(getDescription());  dst.setTask(getTask());  // The above line shows that Task is in relation with Subtask  **return** (dst);  }    } |

# Code Smells and Possible Refactorings

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.
* We can refactor by creating a utility class like TaskUtil with two static methods: daysBetween() and daysLeft().

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.