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 vs. our Conceptual Class Diagram

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.entity” package. The main correspondence of our modeling is with the classes contained in this package. The other 17 packages are mostly taking care of UI tasks (10 packages), tools (3 packages), database tools (2 packages), model(1 package) 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.

There’s a close correspondence between the actual system classes and those of our conceptual model. Of 15 classes we captured in our model, 8 of them exist in the actual system model. These classes are:

|  |  |  |  |
| --- | --- | --- | --- |
| 1. CalendarEntity | 1. Address | 1. Project | 1. Task |
| 1. Memo | 1. CheckList | 1. Item | 1. Todo |

There are 7 classes in the actual system model that are not included in our conceptual model and there are 5 classes in our conceptual model that are not part of the actual system model.

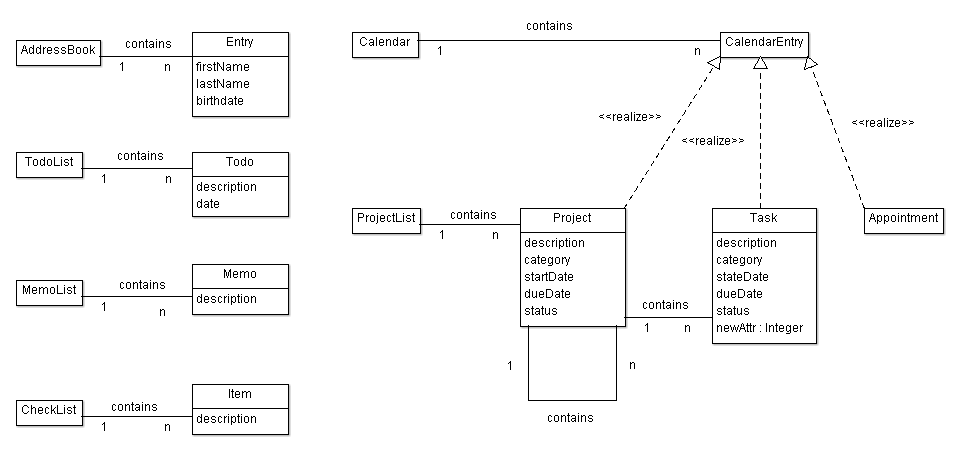
The following are the classes not included in our conceptual model:

* Option: Each BorgOption instance holds a single row from the options table in the database. This is an entity we overlooked.
* Subtask: It represent a task within another task. We figured in our design that we can model this entity with “Task” class. The use of a separate Subtask class makes the design more cohesive but instead there will be more coupling.
* Tasklog: we didn’t really take into account any logging feature but rather captured the main core functions of the software.
* Link: Any association between an entity and another entity will be represented with a Link object. This will increase coupling but might be more cohesive.
* LabelEntity: According to the documentation this class is a non-persisted entity that is used when the model needs to package a transient, calculated entity for the UI, such as a calculated holiday or birthday based on the address book. We didn’t really predicted the design that far.
* KeyedEntity: Abstract base class for all Entities that are keyed by a simple integer key. Again we considered all entities equal without defining any key for them.
* EncryptableEntity: Abstract base class for entities that can have encrypted fields. It is up to the entity specific Decrypt and Encrypt methods to determine which fields are to be encrypted. We overlooked the encryption feature.

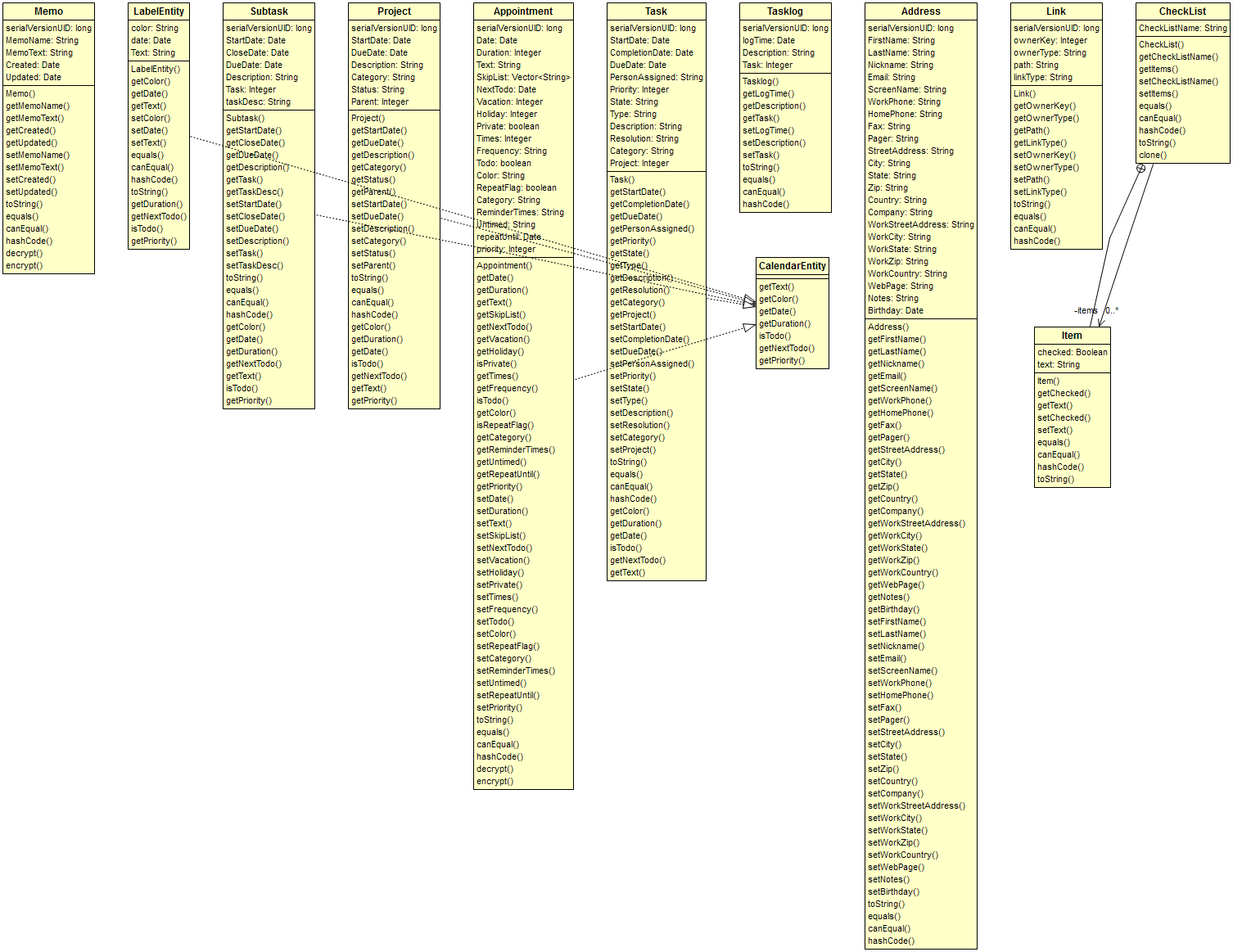
These extra classes are mainly not included in our design because we started from scratch and captured a simple basic set of entities based on Gall’s Law. This is a complex system that evolved for many years and these features are added in later versions.

There are 5 classes in our conceptual model that are not seen in the actual system model. These classes are:

* + AddressBook, ToDoList, ProjectList, MemoList: These classes wrap a collection object that holds a reference to the individual objects. Eg. Address, ToDo, Project and Memo. We intended to present collection objects in their respective entity classes as opposed to using arrays or vectors which is the solution for the actual system design.
  + Calender: This represents the calendar itself that is in association with calendar entity. In the actual system design this class is not present.
  1. Our Conceptual Class Diagram from M2



* 1. Actual System Class Diagram



* 1. Example of source

|  |
| --- |
| **public** **class** Task **extends** KeyedEntity<Task> **implements** CalendarEntity {  **private** String Description;  **private** String Resolution;  **private** String Category;  **private** Integer Project;  @Override  **public** String getColor()  {  **return** "navy";  }  @Override  **public** Integer getDuration()  {  **return** **new** Integer(0);  }  @Override  **public** Date getDate(){ **return** getDueDate(); }  @Override  **public** **boolean** isTodo(){ **return** **true**; }  @Override  **public** Date getNextTodo(){ **return** **null**; }  @Override  **public** String getText(){  // return the text as it should appear on the calendar  String showabb = Prefs.*getPref*(PrefName.*TASK\_SHOW\_ABBREV*);  String abb = "";  **if** (showabb.equals("true"))  abb = "BT" + getKey() + " ";  String de = abb + getDescription();  String tx = de.replace('\n', ' ');  **return** tx;  }  @Override  **protected** Task clone() {  Task dst = **new** Task();  dst.setKey( getKey());  dst.setStartDate( getStartDate() );  dst.setCompletionDate( getCompletionDate() );  dst.setDueDate( getDueDate() );  dst.setPersonAssigned( getPersonAssigned() );  dst.setPriority( getPriority() );  dst.setState( getState() );  dst.setType( getType() );  dst.setDescription( getDescription() );  dst.setResolution( getResolution() );  dst.setCategory( getCategory() );  dst.setProject( getProject() );  **return**(dst);  }  } |

|  |
| --- |
| **public** **interface** CalendarEntity {  **public** String getText();  **public** String getColor();  **public** Date getDate();  **public** Integer getDuration();  **public** **boolean** isTodo();  **public** Date getNextTodo();  **public** Integer getPriority();  } |

# 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.