BORG CALENDAR M4

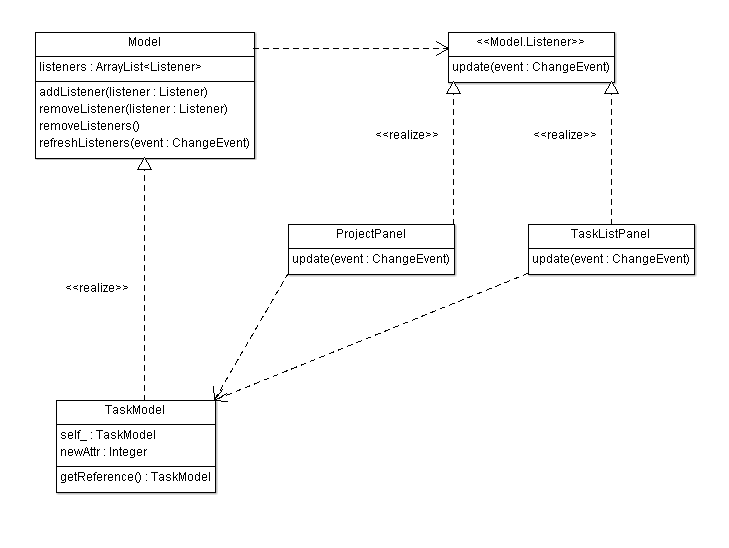
Student Name and Number

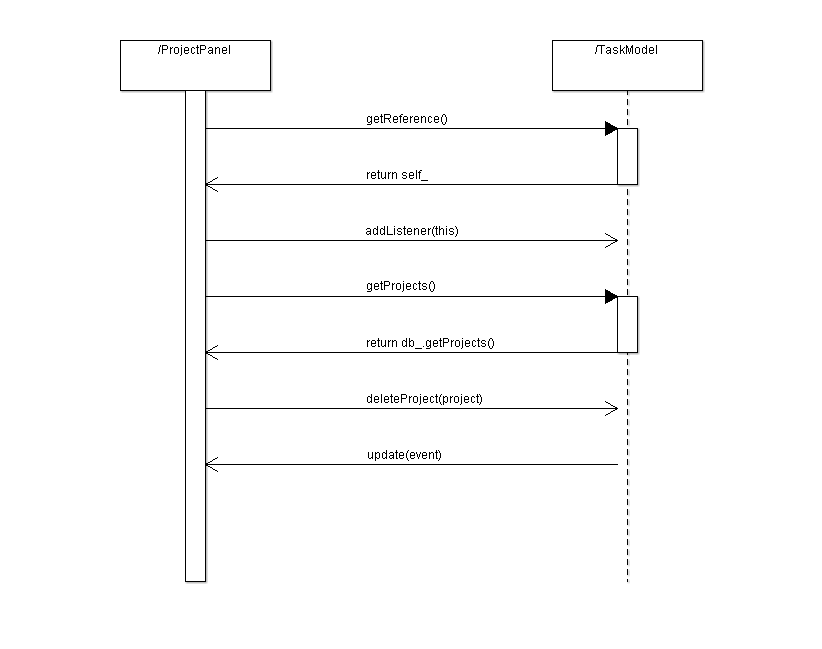
Hamid Shahrestani (9729747)  
Anjaneyulu Bodepudi (5973775)   
Manouchehr Azizi (5232287)   
Viet Hung Nguyen (9816240)

# Identify Substantial Patterns

**MVC Pattern (Viet Hung Nguyen)**

MVC design pattern [3] is implemented in BORG to separate between models and UI. However it combines view and control into a UI class. A UI class get data from a model and display data to users. From user interaction, UI class manipulates its model. Then the model will then notify all listening UI about its changes.



Observer pattern is used to allow views getting notified by model changes [1].

Singleton pattern is used to allow UI classes have access to models [2].

The way of applying MVC design pattern in BORG with combined view and control is similar to the Document View pattern in MFC framework from Microsoft [4].

Using MVC in BORG allow separating between models and UI classes so that we can easily write unit tests for model classes and protect models from UI changes.

In general, the separation between view and control allows having different views, for example supporting different environment such as desktop, web browsers, mobile browsers etc., with the same control logic and allows testing controller’s logic independent with views. In BORG, view and controller are combined. So testing control logic would not be easy for BORG.

*Tools for reverse engineering:*

* AgroUML: a tool to draw class diagrams
* Intellij Idea: an IDE with strong support for navigating around the code.

*Code snippets:*

public abstract class Model {

private ArrayList<Listener> listeners;

public void addListener(Listener listener) {…}

protected void refreshListeners(ChangeEvent event) {…}

public void removeListener(Listener listener) {…}

protected void removeListeners() {…}

protected void refreshListeners(ChangeEvent event) {…}

}

public interface Listener {

public abstract void update(ChangeEvent event);

}

public class ProjectPanel … implements Model.Listener {

public void update(ChangeEvent event) {

refresh();

}

}

public class TaskModel extends Model … {

static private TaskModel self\_ = new TaskModel();

private TaskModel() {…}

static public TaskModel getReference() {

return (self\_);

}

}

# Implement a refactoring (7 marks)

Think small! But a rename is not going to be sufficient. You will hand in a patchset. Each patch is a diff of the system showing how the system has changed. The change log will describe why the change was made and its impact on the system. The patches in the patchset should follow a logical order. Each patch must be small, independent, and complete. Err on the side of making **many** small, independent complete changes. Ideally, you would also provide a test to show that the behaviour has not changed. Each patch should be in an individual file with the name Patchset#-#.patch (ASCII text only). Group them in a zip file. Use git diff to make the patch files.

Patchset 0/2: Refactor class VideoAudio into two classes

1-3 Paragraphs describing how each of the changes (patches in the set) tie together (no diff)

Patchset 1/2: Create new class Audio

1-3 Paragraphs describing why

Diff showing lines changed

Patchset 2/2: Move method audioRecord into Audio class

1-3 Paragraphs describing why

Diff showing lines changed

# References

[1] Observer design pattern. <http://sourcemaking.com/design_patterns/observer>

[2] Singleton design pattern. <http://sourcemaking.com/design_patterns/singleton>

[3] Model View Control design patter. <http://en.wikipedia.org/wiki/Model-view-controller>

[4] MFC framework. <http://msdn.microsoft.com/en-ca/library/k9kb0kba.aspx>