BORG CALENDAR – DESIGN PATTERNS

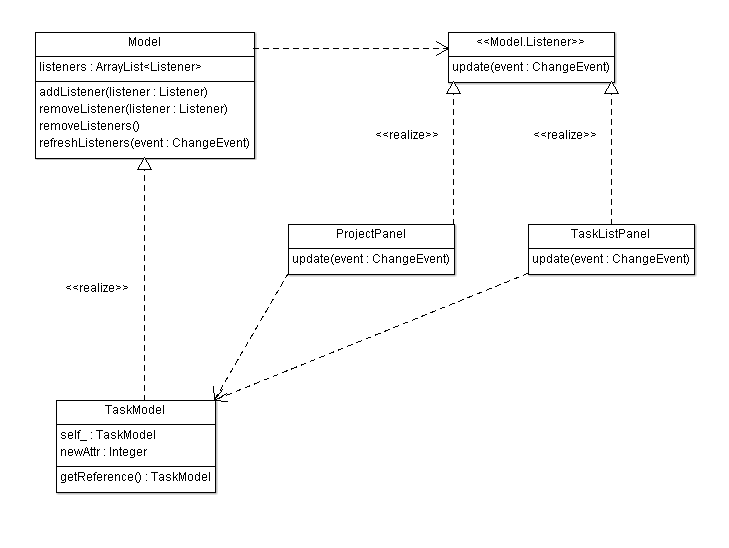
Student Name and Number

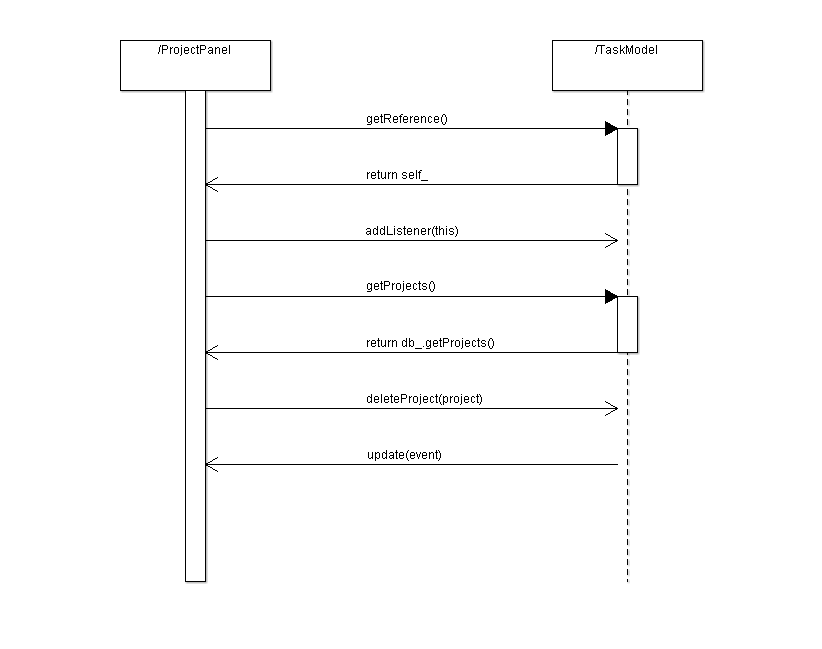
Hamid Shahrestani (9729747)  
Anjaneyulu Bodepudi (5973775)   
Manouchehr Azizi (5232287)   
Viet Hung Nguyen (9816240)

# 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 models. The updated models will then notify all listening UIs about their 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 class TaskModel extends Model … {

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

private TaskModel() {…}

static public TaskModel getReference() {

return (self\_);

}

}

public interface Listener {

public abstract void update(ChangeEvent event);

}

public class ProjectPanel … implements Model.Listener {

public ProjectPanel() {

…

TaskModel.getReference().addListener(this);

…

}

public void update(ChangeEvent event) {

refresh();

}

private void deleteActionPerformed() {

…

TaskModel.getReference().deleteProject(projectId.intValue());

…

}

}

In the code snippets above, we can see that:

* The abstract Model class provides methods for registering/unregistering listeners.
* The concrete TaskModel is a singleton with getReference() method to provide access to its instance.
* The UI class ProjectPanel implement Listener interface and call its refresh() method when being notified by model.

ProjectPanel user TaskModel.getReference() method to get access to the singleton TaskModel instance.

In ProjectPanel constructor, it calls addListener() to register itself to the model.

When user click delete button, deleteActionPerformed() is called and it invokes the deleteProject() method on the TaskModel instance.

# 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>

# OBSERVER (LISTENER) PATTERN

***Manouchehr Azizi***

**Introduction**

In this document I tried to avoid explaining clear things which is explained in comments of selected code. The observer pattern in this project is used to update multiple Panel view in case of preferences changes. I just briefly explain 3 key points (Publisher, Listener (subscriber), notify) of observer pattern which is implemented in Borg Calendar. After introduction you will see in sequence: publisher, notify, Subscriber, Sequence diagram and Partial class diagram section. For extract class “Aid UML diagram” is used. Following is reference to observer: <http://www.vogella.com/articles/DesignPatternObserver/article.html>, <http://java.dzone.com/articles/design-patterns-uncovered>.

**Publisher**

In “net.sf.borg.common.profs “ class, Listener class is implemented which is responsible to keep list of listeners and Notify listeners of a preferences change (following notifyListeners method).the point is that prefsChanged method is going to be override in all subscriber that we will explain in subscriber section.

**Remark:** prefsChanged method is going to be override to do proper action for each of listener member in case of preference changes.

|  |
| --- |
| /\*\*  \* Interface for classes that want to be notified of preference changes  \*/  **public** **interface** Listener {    /\*\*called when preferences changed.  \*/  **public** **abstract** **void** prefsChanged();  }  /\*\* list of listeners \*/  **static** **private** ArrayList<Listener> *listeners* = **new** ArrayList<Listener>();  /\*\*  \* add a listener  \*  \* **@param** listener the listener  \*/  **static** **public** **void** addListener(Listener listener) {  *listeners*.add(listener);  }  /\*\*  \* Notify listeners of a pref change.  \*/  **static** **public** **void** notifyListeners() {  **for** (**int** i = 0; i < *listeners*.size(); i++) {  Listener v = *listeners*.get(i);  v.prefsChanged();  }  } |

**Notify**

**“Prefs.*notifyListeners*();”** is used in two following class to apply proper action to Observer members:

net.sf.borg.model.Theme

|  |
| --- |
| /\*\*  \* sync with the db. called if the options table is changed by something other than the UI (such as import)  \*/  **public** **static** **void** sync()  {      // notify listeners that Prefs may have changed  **Prefs.*notifyListeners*();**  } |

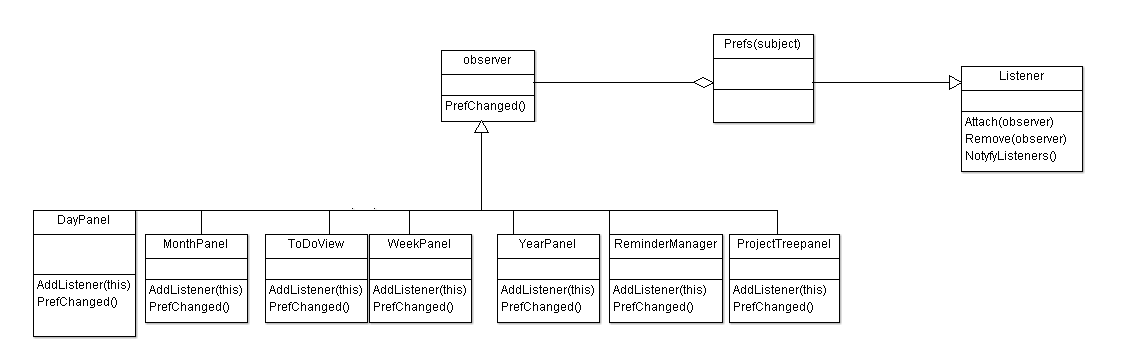
1. net.sf.borg.ui.options.OptionsView

|  |
| --- |
| **private** **void** applyChanges() {  **...**  **Prefs.*notifyListeners*();**    ...  } |

**Subscriber**

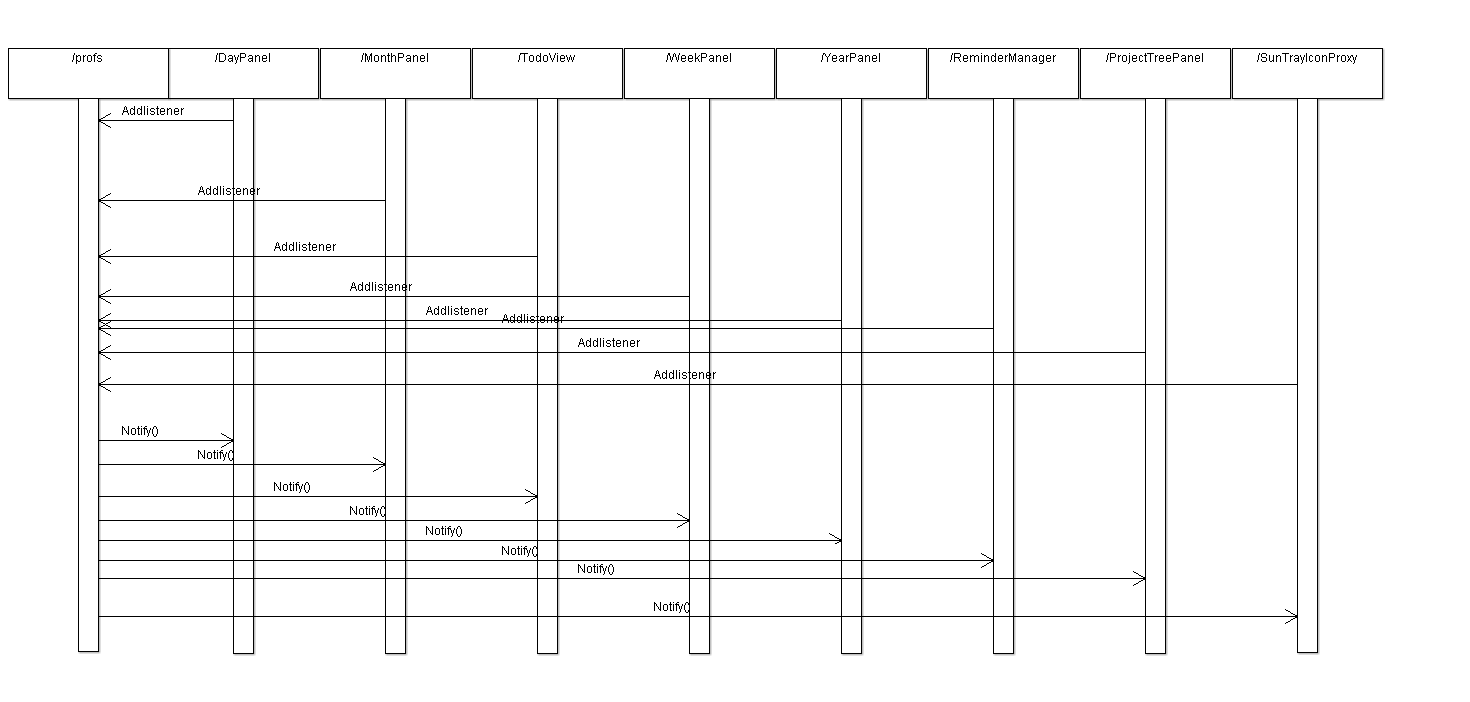
Following class with “Prefs.*addListener*(**this**);”add themselves to Listener list:

* net.sf.borg.ui.calendar.DayPanel
* net.sf.borg.ui.calendar. MonthPanel
* net.sf.borg.ui.calendar. TodoView
* net.sf.borg.ui.calendar. WeekPanel
* net.sf.borg.ui.calendar. YearPanel
* net.sf.borg.ui.popup. ReminderManager
* net.sf.borg.ui.task. ProjectTreePanel
* net.sf.borg.ui. SunTrayIconProxy

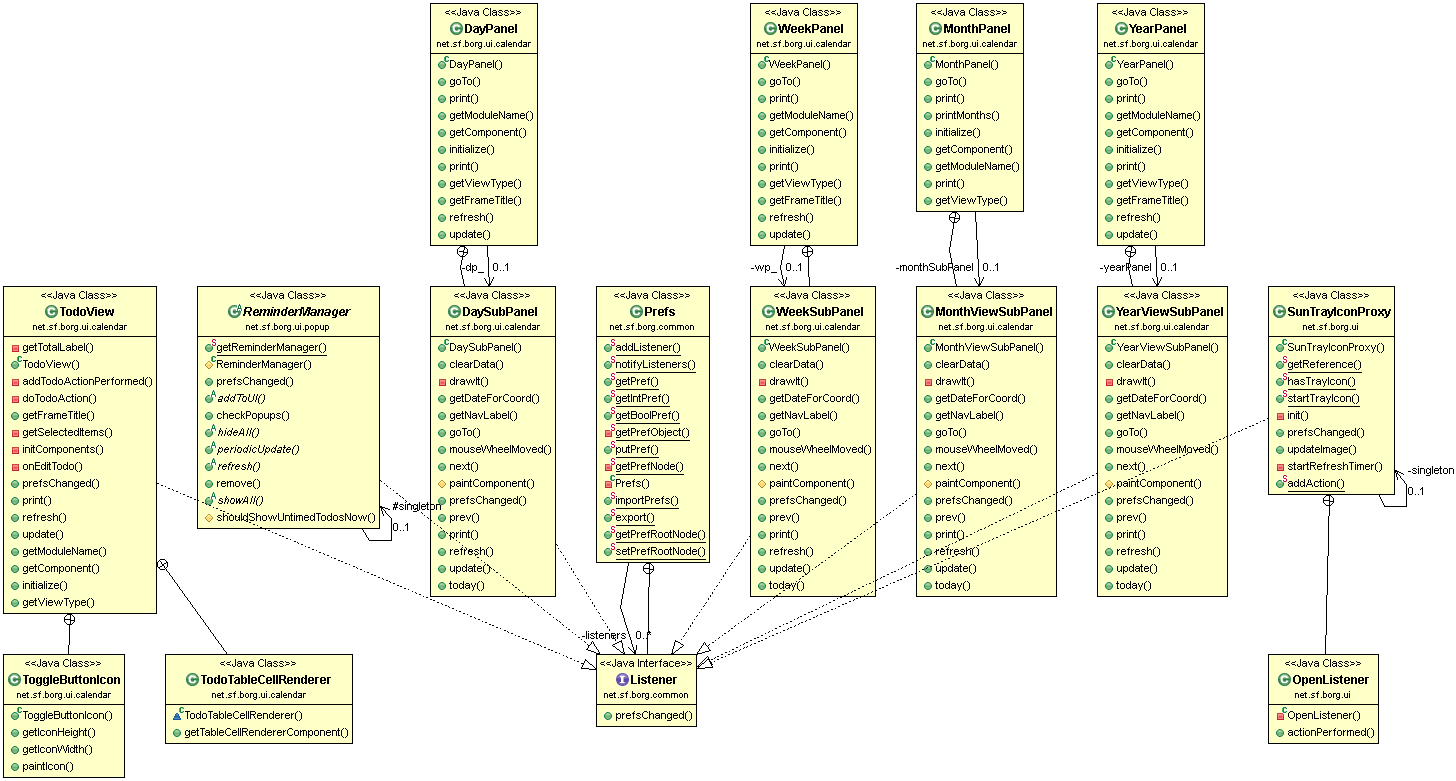
****

**Remark**: in all of upcoming class, “prefsChanged” method is override to do proper action in case of prefrences change.

**Sequence diagram**



**Partial Class diagram**

****

# M4 BORG DESIGN PATTERN

***Hamid Shahrestani Mehr - 9729747***

**Factory Pattern**

**Intro**:

According to Wikipedia “Factory is a creational design pattern to implement the concept of factories and deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The essence of this pattern is to "Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate. The Factory method lets a class defer instantiation to subclasses.”

In BorgCalendar, GridBagConstraints class is used numerously in many classes. The GridBagConstraints class specifies constraints for components that are laid out using the GridBagLayout class. To facilitate instantiation of objects from this class, a factory class is designed. This class is used 39 times within UI packages to instantiate GridBagConstraints objects. GridBagConstraints object can be created in several ways with different fields populated. Factory method encapsulates the process of creation and saves lots of time.

To capture the following interactions I used Object Aid UML tool.

**Code Snippets**

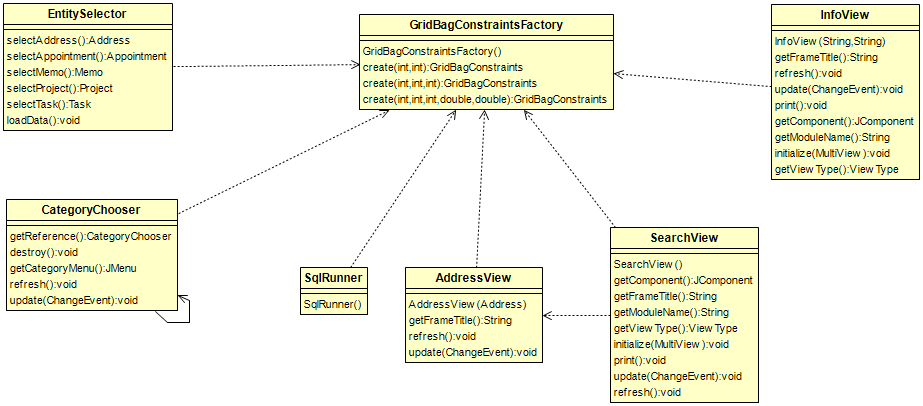
For example in the following code snippet, JPanel has an add method that takes GridBagConstaint object as an argument. To instantiate objects of this type, a factory class is used.

|  |
| --- |
| private JPanel getTopPanel() {  if (topPanel == null) {  topPanel = new JPanel();  topPanel.setLayout(new GridBagLayout());  topPanel.add(buttonPanel, GridBagConstraintsFactory.create(0, 1,  GridBagConstraints.BOTH, 1.0, 0.0));  topPanel.add(getJScrollPane(), GridBagConstraintsFactory.create(0,  0, GridBagConstraints.BOTH, 1.0, 1.0));  }  return topPanel;  } |

|  |
| --- |
| public class GridBagConstraintsFactory {  static private final Insets defaultInsets = new Insets(4, 4, 4, 4);  public static GridBagConstraints create(int x, int y) {  GridBagConstraints gbc = new GridBagConstraints();  gbc.gridx = x;  gbc.gridy = y;  gbc.insets = defaultInsets;  return gbc;  }    public static GridBagConstraints create(int x, int y, int fill) {  GridBagConstraints gbc = create(x, y);  gbc.fill = fill;  return gbc;  }  public static GridBagConstraints create(int x, int y, int fill,  double weightx, double weighty) {  GridBagConstraints gbc = create(x, y, fill);  gbc.weightx = weightx;  gbc.weighty = weighty;  return gbc;  }  } |

**Class Diagram**

For example the picture below displays some of the classes that use this factory class.



**Sources:**

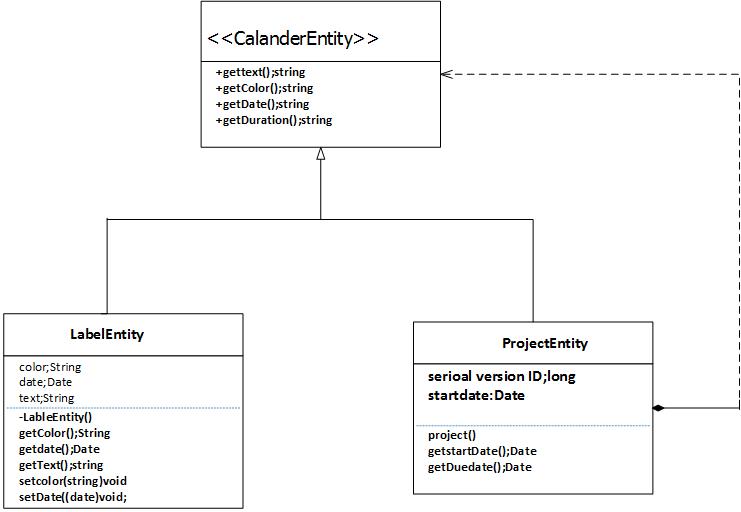
# Design Patterns: Elements of Reusable Object-Oriented Software

# M4 BORG DESIGN PATTERN

***ANJANEYULU BODEPUDI (5973775)***

COMPOSITE PATTERN

Abstraction: Composite in Java: Model-configurable "Entities" of a Composite

In BORG calendar borg.model.entity package project class and Label Entity classes’ files are implements the << Calendar Entity>> interface. An entity is a lightweight persistence domain object. Typically an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes. The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationship. The below class are composite abstract. Its part whole relationship of the Calendar Entity and project Entity classes.s to the relational data in the underlying data store.

Code snippets

**Package** net.sf.borg.model.entity;

* Interface that needs to be implemented by any entity that can appear on the Calendar.
* This interface would more properly belong in the UI package - but it would take some extra wrapper
* classes that would be a waste

**Public** **interface** Calendar Entity {

**Public** String get Text ();

**Public** String get Color ();

**Public** Date get Date ();

**Public** Integer get Duration();

**public** **boolean** isTodo();

**public** Date getNextTodo();

**public** Integer getPriority();

}

**public** **class** LabelEntity **implements** CalendarEntity {

**private** String color;

**private** Date date;

**private** String Text;

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getDuration()

\*/

@Override

**public** Integer getDuration() {

**return** **new** Integer(0);

}

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getNextTodo()

\*/

@Override

**public** Date getNextTodo() {

**return** **null**;

}

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getTodo()

\*/

@Override

**public** **boolean** isTodo() {

**return** **false**;

}

@Override

**public** Integer getPriority() {

**return** **null**;

}

package net.sf.borg.model.entity;

import java.util.Date;

import javax.xml.bind.annotation.XmlAccessType;

import javax.xml.bind.annotation.XmlAccessorType;

import javax.xml.bind.annotation.XmlRootElement;

import lombok.Data;

import lombok.EqualsAndHashCode;

import net.sf.borg.common.PrefName;

import net.sf.borg.common.Prefs;

/\*\*

\* Project Entity - a project contains tasks and can have child projects

\*/

@XmlRootElement(name="Project")

@XmlAccessorType(XmlAccessType.FIELD)

@Data

@EqualsAndHashCode(callSuper=true)

public class Project extends KeyedEntity<Project> implements CalendarEntity {

private static final long serialVersionUID = -3250115693306817331L;

private Date StartDate;

private Date DueDate;

private String Description;

private String Category;

private String Status;

private Integer Parent;

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.KeyedEntity#clone()

\*/

@Override

protected Project clone() {

Project dst = new Project();

dst.setKey( getKey());

dst.setStartDate( getStartDate() );

dst.setDueDate( getDueDate() );

dst.setDescription( getDescription() );

dst.setCategory( getCategory() );

dst.setStatus( getStatus() );

dst.setParent(getParent());

return(dst);

}

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getColor()

\*/

@Override

public String getColor()

{

// for showing on calendar

// legacy color name - maps to a user-defined color

return "navy";

}

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getDuration()

\*/

@Override

public Integer getDuration()

{

return new Integer(0);

}

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getDate()

\*/

@Override

public Date getDate(){ return getDueDate(); }

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getTodo()

\*/

@Override

public boolean isTodo(){ return true; }

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getNextTodo()

\*/

@Override

public Date getNextTodo(){ return null; }

/\* (non-Javadoc)

\* @see net.sf.borg.model.entity.CalendarEntity#getText()

\*/

@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 = "PR" + getKey() + " ";

String de = abb + getDescription();

String tx = de.replace('\n', ' ');

return tx;

}

@Override

public Integer getPriority() {

return null;

}

}

REFERNCES:

<http://java.uom.gr/~nikos/pattern-detection.html>

<http://sourcemaking.com/design_patterns/composite>

<http://docs.oracle.com/javaee/5/tutorial/doc/bnbqa.html>