NOVA SCHOOL OF SCIENCE & TECHNOLOGY

THE

S TEAM

Students:

Francisco Parrinha	- 58360
Bernardo Atalaia	- 59962
Martin Magdalinchev	- 58172
Carlos Soares	- 57006
Pedro Inácio	- 59184

Index

Code Smells

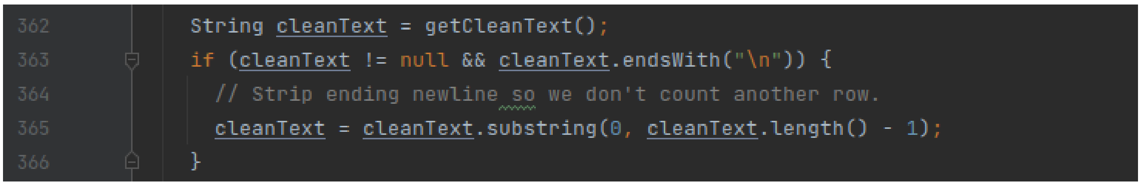
Francisco Parrinha – 58360

1. **Long Method:**

- Method *computePrefHeight*” in: *ganttproject/src/main/java/biz.gantproject/lib/fx/treetable/LabeledSkinBase.java*

The method starts at line number 350 and ends at number 397, containing 47 lines. There are several refactoring possibilities. The code within lines number 362 and 366 serves as an auxiliary operation to the method. This operation is to strip strings.

The following block of code could be removed from this function to a new *getStrippedString* method, and later be called on the original *computePrefHeight* function. The code to be refactored:

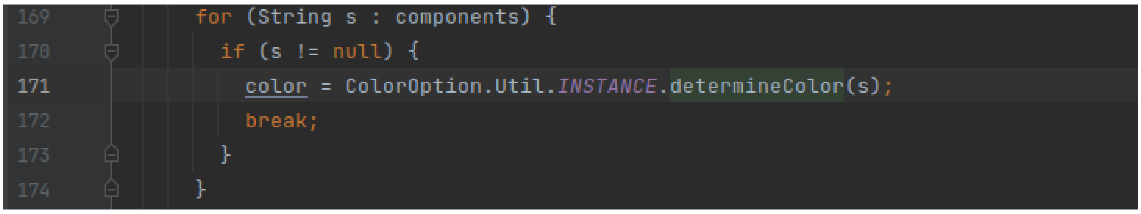


```
362    String cleanText = getCleanText();
363    if (cleanText != null && cleanText.endsWith("\n")) {
364      // Strip ending newline so we don't count another row.
365      cleanText = cleanText.substring(0, cleanText.length() - 1);
366    }
```

- Método *parse* em: *biz.ganttproject.core/src/main/java/biz/ganttproject/core/chart/render/RectangleRenderer.java*

Although the parse function is not too long, it contains an auxiliary operation and could be refactored.

The block of code shown below (linhas 169 - 174) could be moved to a new method, *gerColor*, for example. This would improve the project 's modularity. The new method would then be called on the original function.
The code to be refactored:



```
169    for (String s : components) {
170      if (s != null) {
171        color = ColorOption.Util.INSTANCE.determineColor(s);
172        break;
173      }
174    }
```

2. **Comments:**

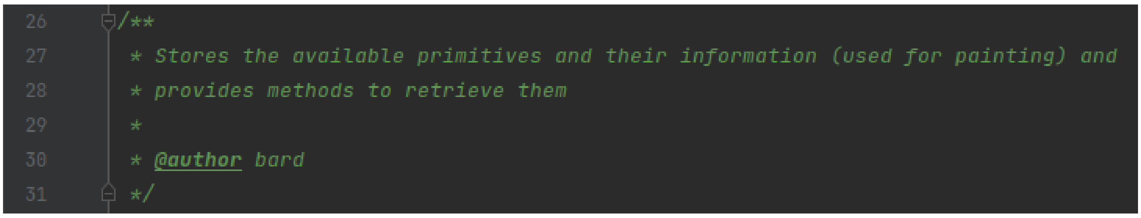
- Classe *Canvas* em:
  biz.ganttproject.core/src/main/java/biz/ganttproject/core/chart/canvas/

This class contains several nested classes The entire *Canvas* class only contains one comment (line 26 -31) and it does not have any *Javadoc* written excluding this one.

All methods and nested classes are to be commented out, making the code understanding much harder.

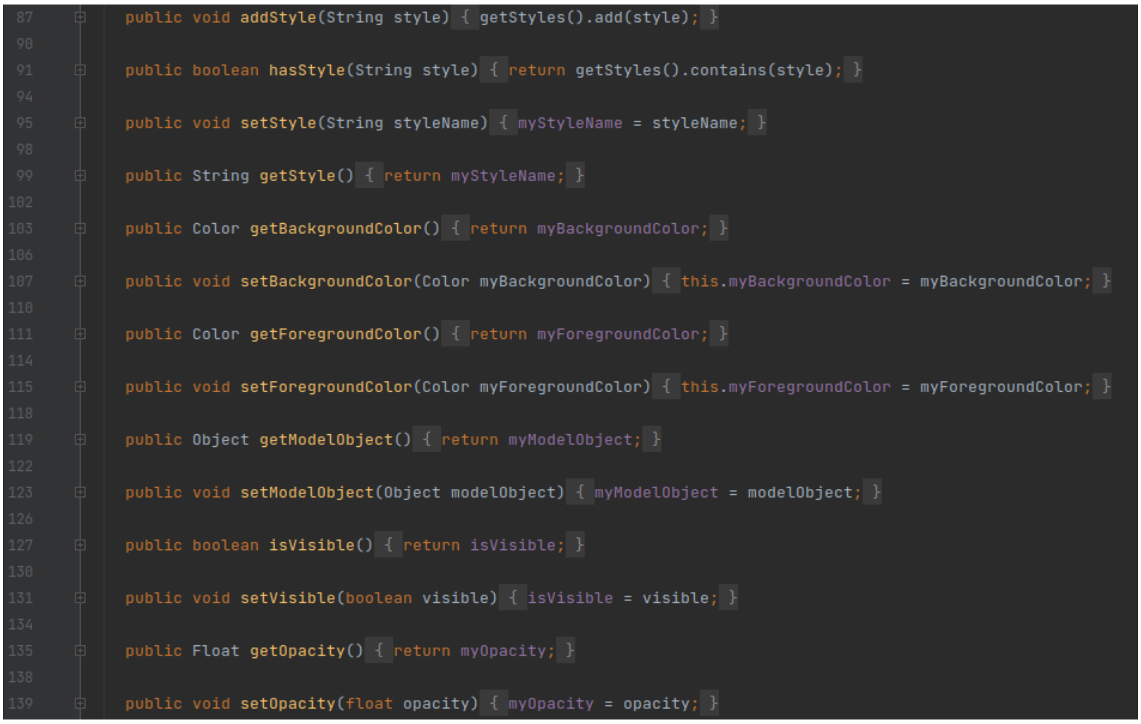
Creating the Javadoc is enough to refactor this code-smell. Here are some examples of this anti-pattern

1. The only comment:

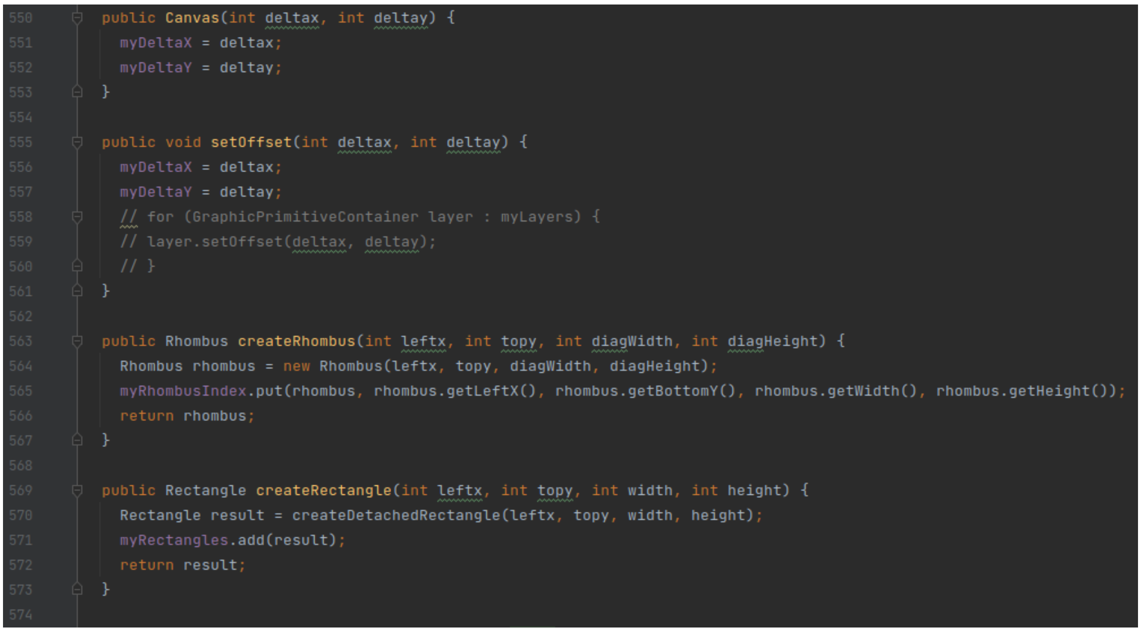


```
26  /**
27   * Stores the available primitives and their information (used for painting) and
28   * provides methods to retrieve them
29   *
30   * @author bard
31   */
```

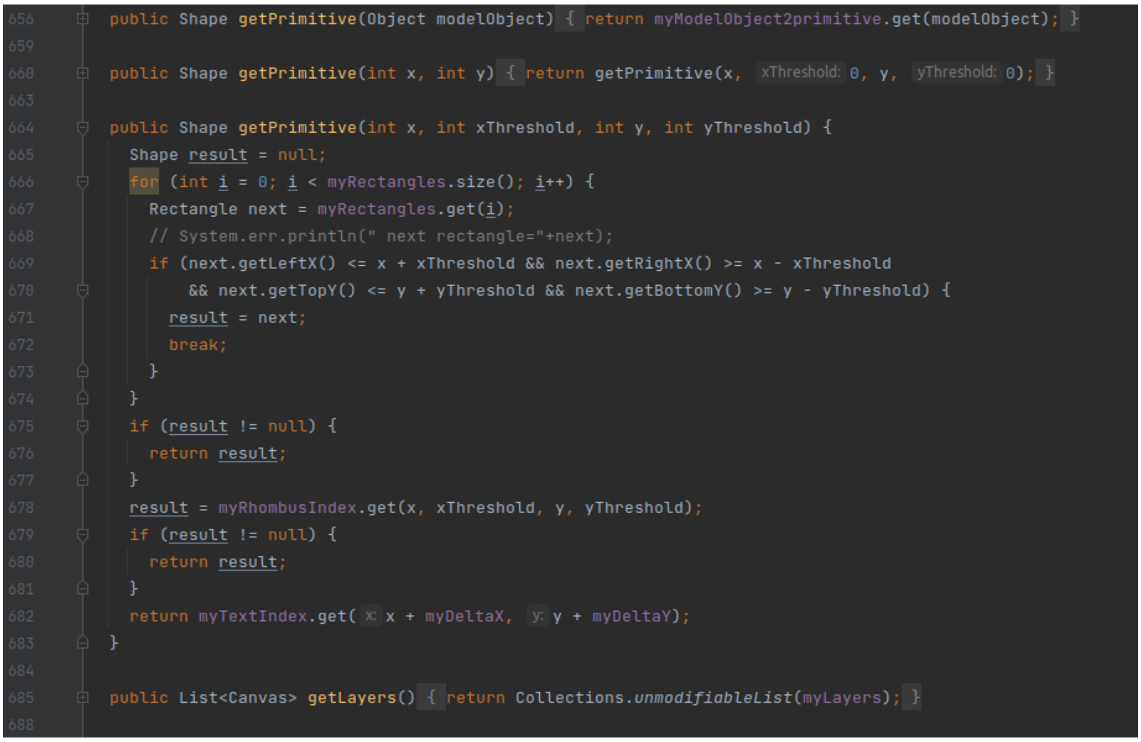
2. Some examples of the lack of code:



```
public void addStyle(String style) { getStyles().add(style); }

public boolean hasStyle(String style) { return getStyles().contains(style); }

public void setStyle(String styleName) { myStyleName = styleName; }

public String getStyle() { return myStyleName; }

public Color getBackgroundColor() { return myBackgroundColor; }

public void setBackgroundColor(Color myBackgroundColor) { this.myBackgroundColor = myBackgroundColor; }

public Color getForegroundColor() { return myForegroundColor; }

public void setForegroundColor(Color myForegroundColor) { this.myForegroundColor = myForegroundColor; }

public Object getModelObject() { return myModelObject; }

public void setModelObject(Object modelObject) { myModelObject = modelObject; }

public boolean isVisible() { return isVisible; }

public void setVisible(boolean visible) { isVisible = visible; }

public Float getOpacity() { return myOpacity; }

public void setOpacity(float opacity) { myOpacity = opacity; }
```



```java
public Canvas(int deltax, int deltay) {
  myDeltaX = deltax;
  myDeltaY = deltay;
}

public void setOffset(int deltax, int deltay) {
  myDeltaX = deltax;
  myDeltaY = deltay;
  // for (GraphicPrimitiveContainer layer : myLayers) {
  // layer.setOffset(deltax, deltay);
  // }
}

public Rhombus createRhombus(int leftx, int topy, int diagWidth, int diagHeight) {
  Rhombus rhombus = new Rhombus(leftx, topy, diagWidth, diagHeight);
  myRhombusIndex.put(rhombus, rhombus.getLeftX(), rhombus.getBottomY(), rhombus.getWidth(), rhombus.getHeight());
  return rhombus;
}

public Rectangle createRectangle(int leftx, int topy, int width, int height) {
  Rectangle result = createDetachedRectangle(leftx, topy, width, height);
  myRectangles.add(result);
  return result;
}
```



```java
public Shape getPrimitive(Object modelObject) { return myModelObject2primitive.get(modelObject); }

public Shape getPrimitive(int x, int y) { return getPrimitive(x, xThreshold: 0, y, yThreshold: 0); }

public Shape getPrimitive(int x, int xThreshold, int y, int yThreshold) {
  Shape result = null;
  for (int i = 0; i < myRectangles.size(); i++) {
    Rectangle next = myRectangles.get(i);
    // System.err.println(" next rectangle="+next);
    if (next.getLeftX() <= x + xThreshold && next.getRightX() >= x - xThreshold
        && next.getTopY() <= y + yThreshold && next.getBottomY() >= y - yThreshold) {
      result = next;
      break;
    }
  }
  if (result != null) {
    return result;
  }
  result = myRhombusIndex.get(x, xThreshold, y, yThreshold);
  if (result != null) {
    return result;
  }
  return myTextIndex.get( x: x + myDeltaX, y: y + myDeltaY);
}

public List<Canvas> getLayers() { return Collections.unmodifiableList(myLayers); }
```

Reviewer: Bernardo Atalaia 59962

1: Good example with a very good explanation, very clear and direct.

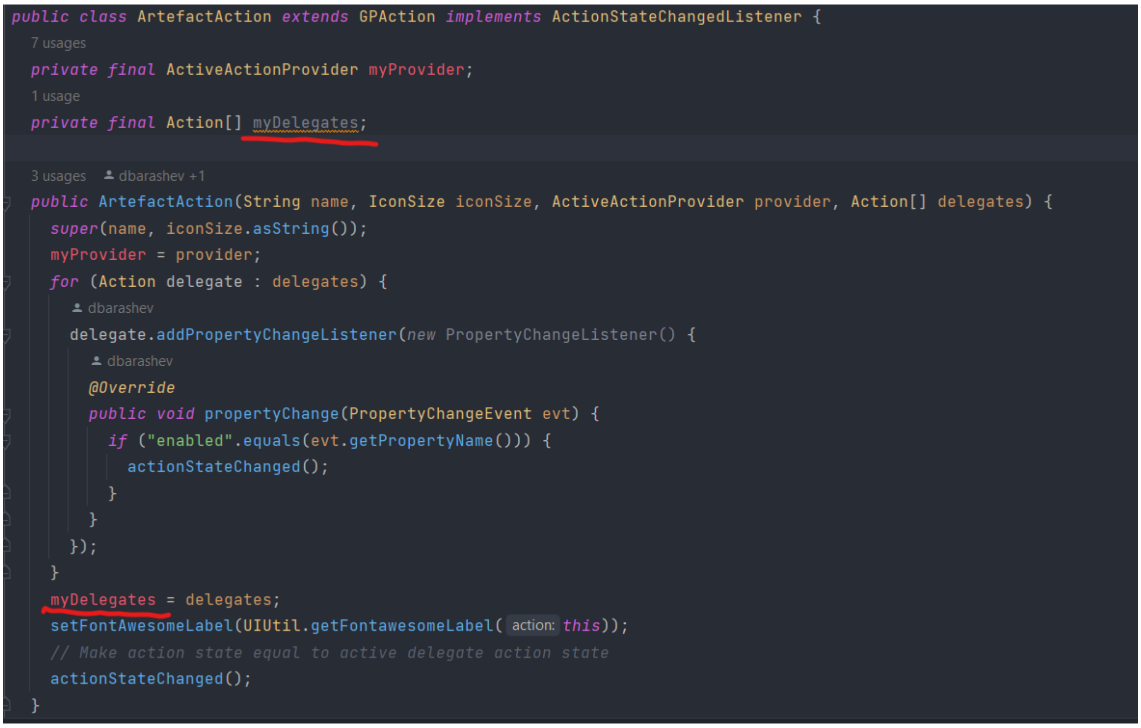
2: It's a good code smell and well explained but i think there are more suitable classes for that, in this case there is only 4 parameters, the thing is that they have big names so it seems like more. Overall good explanation aswell but i think there are better examples.

3: I completly agree with this code smell, also well explained.

Bernardo Atalaia – 59962

1- Dead Code

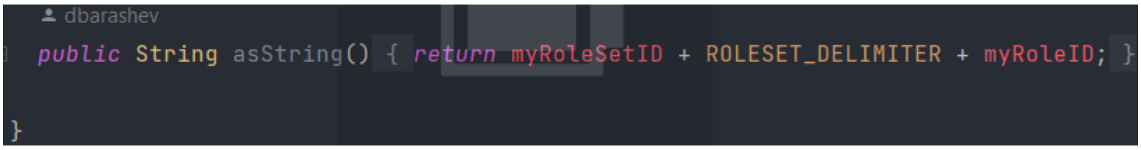
In the ArtefactAction Class shown below, there's a variable that receives a value to be saved but it's not used and cant be accessed by other classes, so there is no point in saving this information as it has no purpose in this class.
This is easily fixed by just deleting this variable.
Path: main\java\net\sourceforge\ganttproject\roles\RolePersistentID.java



```
public class ArtefactAction extends GPAction implements ActionStateChangedListener {
  7 usages
  private final ActiveActionProvider myProvider;
  1 usage
  private final Action[] myDelegates;

  3 usages  dbarashev +1
  public ArtefactAction(String name, IconSize iconSize, ActiveActionProvider provider, Action[] delegates) {
    super(name, iconSize.asString());
    myProvider = provider;
    for (Action delegate : delegates) {
       dbarashev
      delegate.addPropertyChangeListener(new PropertyChangeListener() {
         dbarashev
        @Override
        public void propertyChange(PropertyChangeEvent evt) {
          if ("enabled".equals(evt.getPropertyName())) {
            actionStateChanged();
          }
        }
      });
    }
    myDelegates = delegates;
    setFontAwesomeLabel(UIUtil.getFontawesomeLabel( action: this));
    // Make action state equal to active delegate action state
    actionStateChanged();
  }
```

2- Speculative Generality

The RolePresidentID Class has the method shown below that is never used, probably the author though that it would be needed in the future, but it never did.
Again, the solution is easy, as the deletion of this method has no impact on the App overall.
Path: main\java\net\sourceforge\ganttproject\roles\RolePersistentID.java

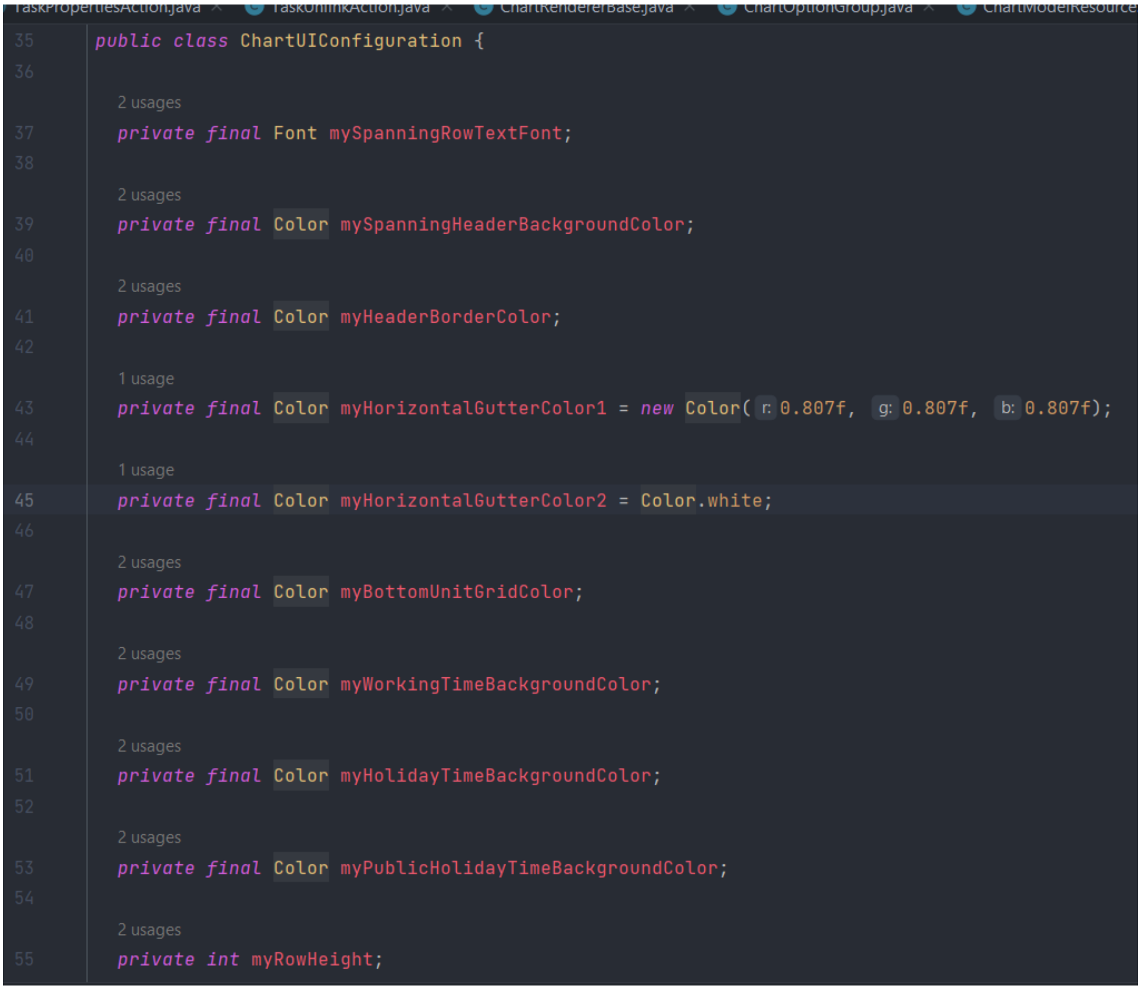


```
   dbarashev
  public String asString() { return myRoleSetID + ROLESET_DELIMITER + myRoleID; }

}
```

3- Primitive Obsession

The ChartUIConfiguration class can be seen as a long class (another code smell) and in this particular case it's mainly because of the extensive number of variables that this class has, most of them even being useless at some point due to lack of usage. This code smell, in this case, can be solved in a couple of ways, as by deleting all the useless primitives, as grouping primitives that are make sense to be used together and don't have a big meaning by their own in a new class englobing all of them so this class only has the new class as it's primitive.

Path: main\java\net\sourceforge\ganttproject\chart\ChartUIConfiguration.java



```
public class ChartUIConfiguration {

  2 usages
  private final Font mySpanningRowTextFont;

  2 usages
  private final Color mySpanningHeaderBackgroundColor;

  2 usages
  private final Color myHeaderBorderColor;

  1 usage
  private final Color myHorizontalGutterColor1 = new Color( r: 0.807f,  g: 0.807f,  b: 0.807f);

  1 usage
  private final Color myHorizontalGutterColor2 = Color.white;

  2 usages
  private final Color myBottomUnitGridColor;

  2 usages
  private final Color myWorkingTimeBackgroundColor;

  2 usages
  private final Color myHolidayTimeBackgroundColor;

  2 usages
  private final Color myPublicHolidayTimeBackgroundColor;

  2 usages
  private int myRowHeight;
```

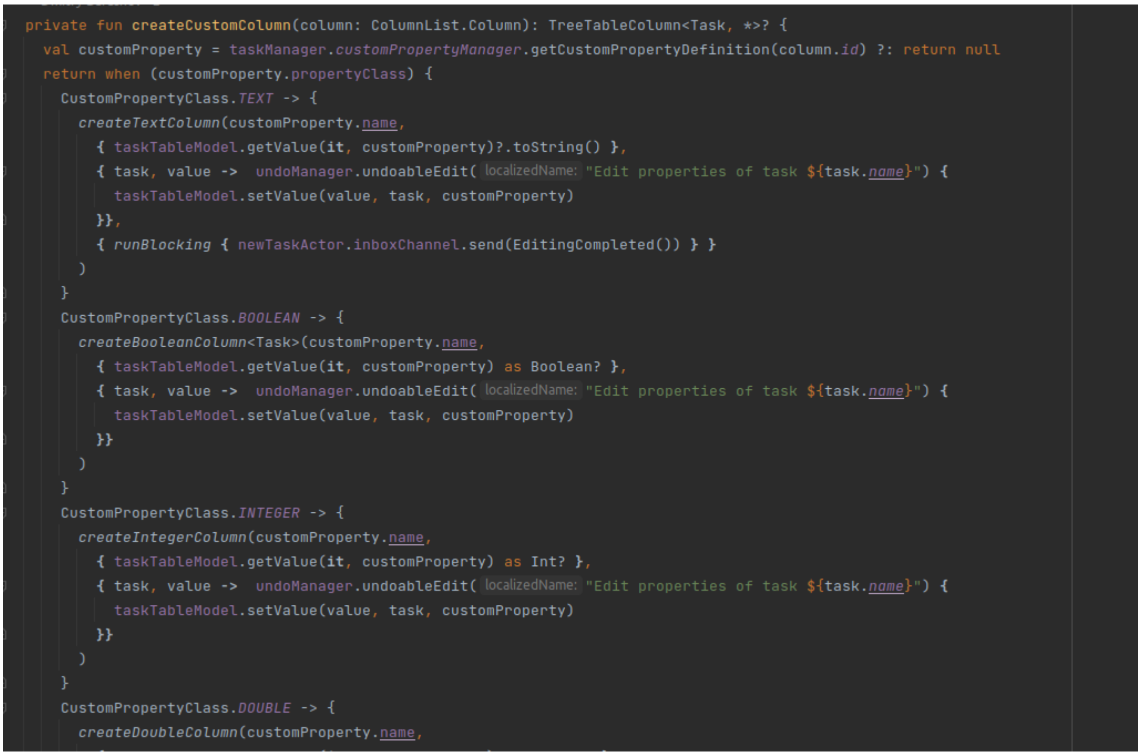
Reviewer: Francisco Parrinha - 58360

Review:
I agree with all the code smells. For each anti-pattern there is a screenshot, an exact location on the codebase and small rationale.
The look of the document is good as well.
Change “Aluno” to “Student” since the doc is written in english.

No Comments

It can be hard for someone else, or even the original developer returning to the code after some time away, to understand what the code is doing or should be doing.

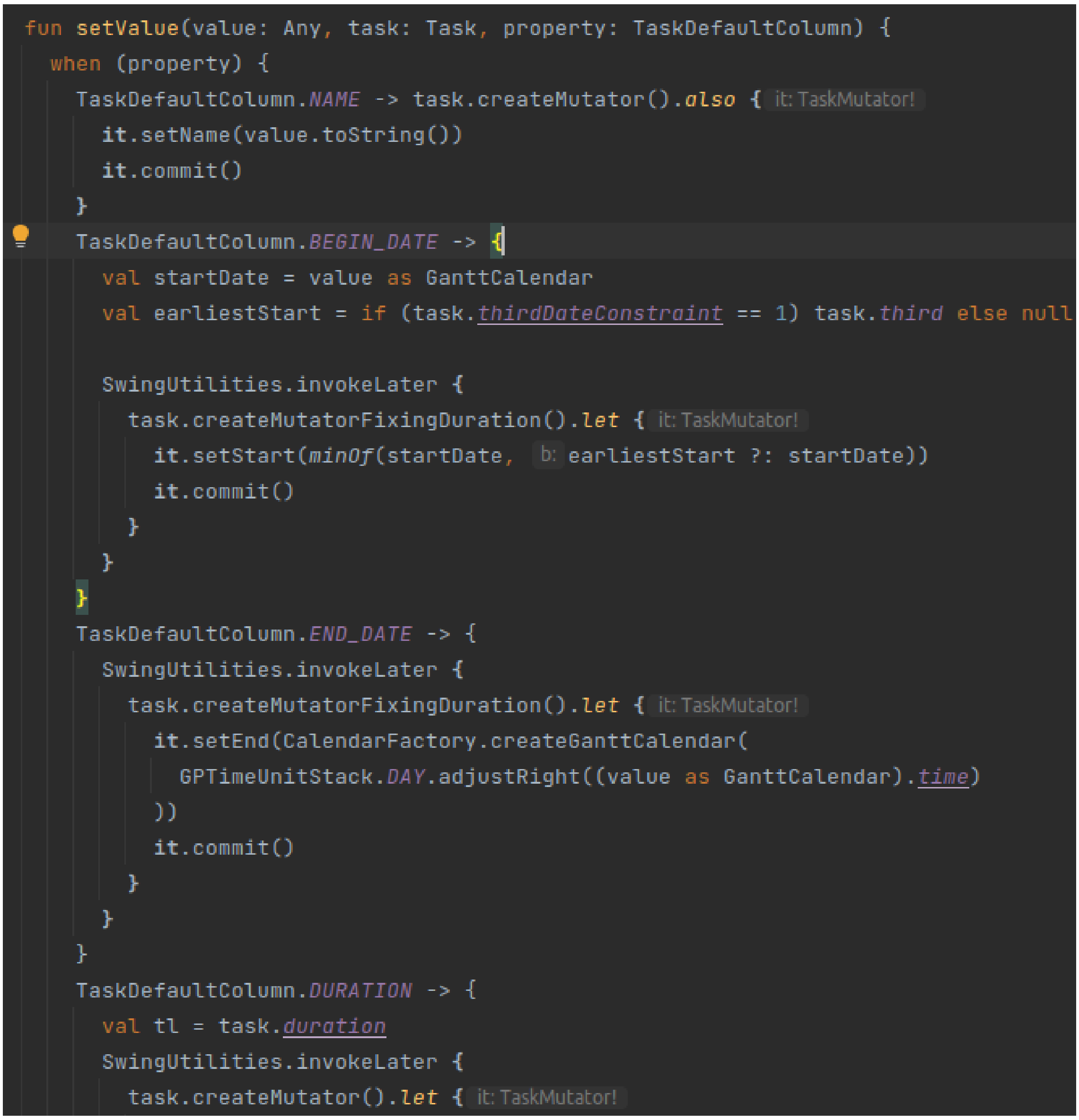
So when you see classes like TaskTable with 1000 lines and almost no comments you know something is going to be hard to understand.



```
private fun createCustomColumn(column: ColumnList.Column): TreeTableColumn<Task, *>? {
  val customProperty = taskManager.customPropertyManager.getCustomPropertyDefinition(column.id) ?: return null
  return when (customProperty.propertyClass) {
    CustomPropertyClass.TEXT -> {
      createTextColumn(customProperty.name,
        { taskTableModel.getValue(it, customProperty)?.toString() },
        { task, value ->  undoManager.undoableEdit( localizedName: "Edit properties of task ${task.name}") {
          taskTableModel.setValue(value, task, customProperty)
        }},
        { runBlocking { newTaskActor.inboxChannel.send(EditingCompleted()) } }
      )
    }
    CustomPropertyClass.BOOLEAN -> {
      createBooleanColumn<Task>(customProperty.name,
        { taskTableModel.getValue(it, customProperty) as Boolean? },
        { task, value ->  undoManager.undoableEdit( localizedName: "Edit properties of task ${task.name}") {
          taskTableModel.setValue(value, task, customProperty)
        }}
      )
    }
    CustomPropertyClass.INTEGER -> {
      createIntegerColumn(customProperty.name,
        { taskTableModel.getValue(it, customProperty) as Int? },
        { task, value ->  undoManager.undoableEdit( localizedName: "Edit properties of task ${task.name}") {
          taskTableModel.setValue(value, task, customProperty)
        }}
      )
    }
    CustomPropertyClass.DOUBLE -> {
      createDoubleColumn(customProperty.name,
```

Long Method

It’s not advisable to create very long methods, its better to just split them into auxiliar functions to allow the code to be easier to read. Not like the method setValue from lines 84 to 167 of TaskTableModel. That could easily be spitted into methods with names referring to functionality.



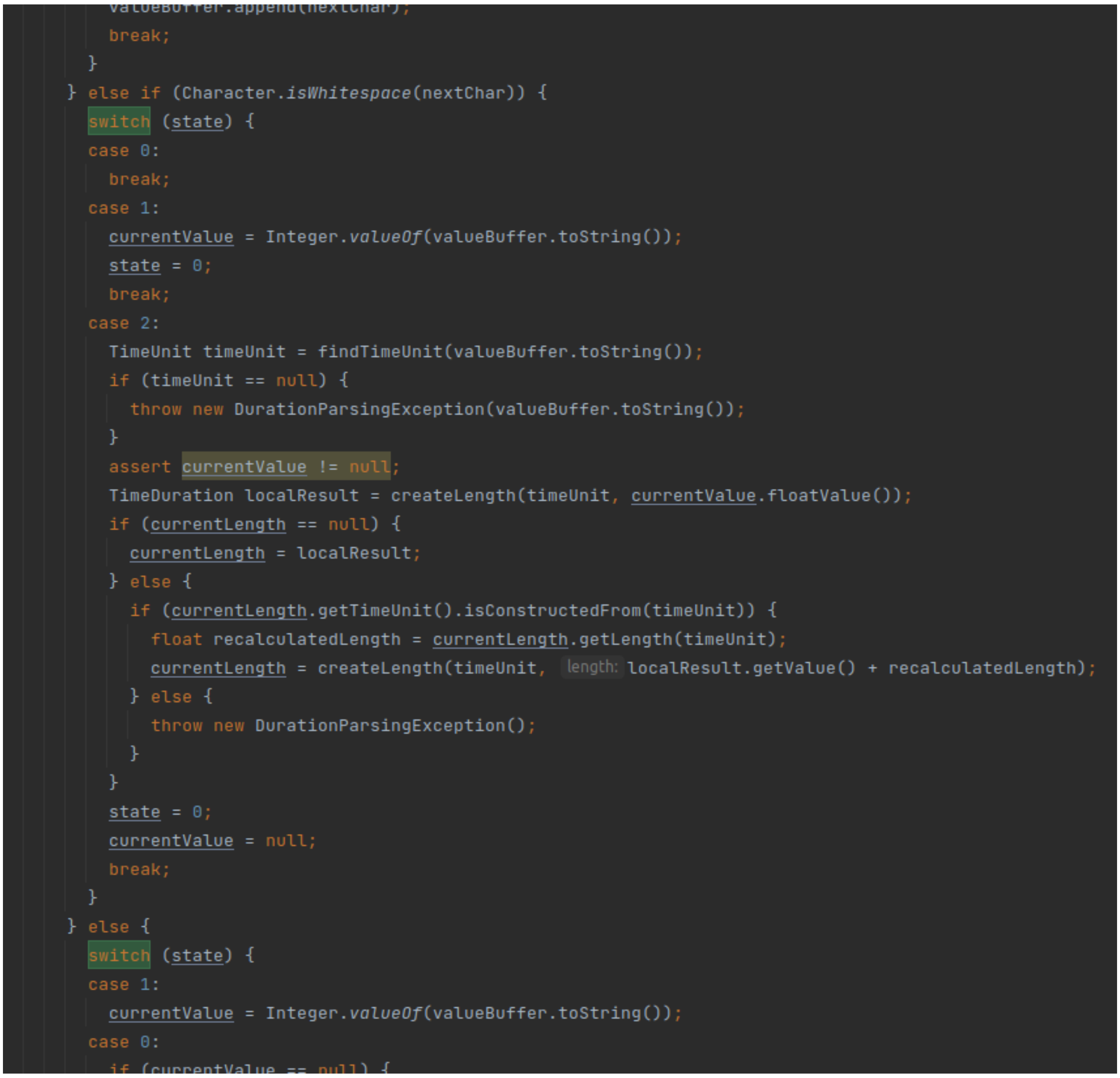
```
fun setValue(value: Any, task: Task, property: TaskDefaultColumn) {
  when (property) {
    TaskDefaultColumn.NAME -> task.createMutator().also { it: TaskMutator!
      it.setName(value.toString())
      it.commit()
    }
    TaskDefaultColumn.BEGIN_DATE -> {
      val startDate = value as GanttCalendar
      val earliestStart = if (task.thirdDateConstraint == 1) task.third else null

      SwingUtilities.invokeLater {
        task.createMutatorFixingDuration().let { it: TaskMutator!
          it.setStart(minOf(startDate,  b: earliestStart ?: startDate))
          it.commit()
        }
      }
    }
    TaskDefaultColumn.END_DATE -> {
      SwingUtilities.invokeLater {
        task.createMutatorFixingDuration().let { it: TaskMutator!
          it.setEnd(CalendarFactory.createGanttCalendar(
            GPTimeUnitStack.DAY.adjustRight((value as GanttCalendar).time)
          ))
          it.commit()
        }
      }
    }
    TaskDefaultColumn.DURATION -> {
      val tl = task.duration
      SwingUtilities.invokeLater {
        task.createMutator().let { it: TaskMutator!
```

Switch statements

Occurs when switch statements are scattered throughout a program. If a switch is changed, then the others must be found and updated as well.

For example, if conditionals are checking on type codes, or the types of something, then there is a better way of handling the switch statements. It may be possible to reduce conditionals down to a design that uses polymorphism.

So when you go to TaskManagerImpl and see the creadteLength method with 3 switch to the same variable inside of an if else if else, its easy to understand that this code smells.



```
    valueBuffer.append(nextChar);
    break;
  }
} else if (Character.isWhitespace(nextChar)) {
  switch (state) {
  case 0:
    break;
  case 1:
    currentValue = Integer.valueOf(valueBuffer.toString());
    state = 0;
    break;
  case 2:
    TimeUnit timeUnit = findTimeUnit(valueBuffer.toString());
    if (timeUnit == null) {
      throw new DurationParsingException(valueBuffer.toString());
    }
    assert currentValue != null;
    TimeDuration localResult = createLength(timeUnit, currentValue.floatValue());
    if (currentLength == null) {
      currentLength = localResult;
    } else {
      if (currentLength.getTimeUnit().isConstructedFrom(timeUnit)) {
        float recalculatedLength = currentLength.getLength(timeUnit);
        currentLength = createLength(timeUnit, length: localResult.getValue() + recalculatedLength);
      } else {
        throw new DurationParsingException();
      }
    }
    state = 0;
    currentValue = null;
    break;
  }
} else {
  switch (state) {
  case 1:
    currentValue = Integer.valueOf(valueBuffer.toString());
  case 0:
    if (currentValue == null) {
```

Reviewer: Francisco Parrinha 58360

1. It has good code snippets, they are very clear

2. It does not show the exact location of the code smell

3. The explanation of the rationale is very clear

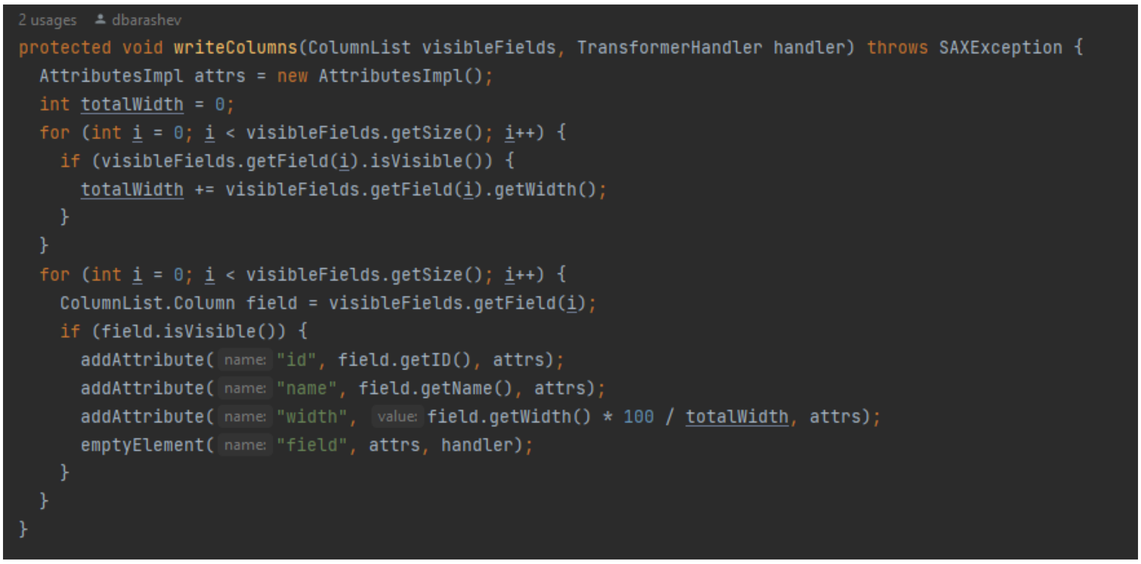
4. It offers refactoring solutions to every code smell

R: I agree with all the code smells in this document

Carlos Soares – 57006

- **Long Method**:

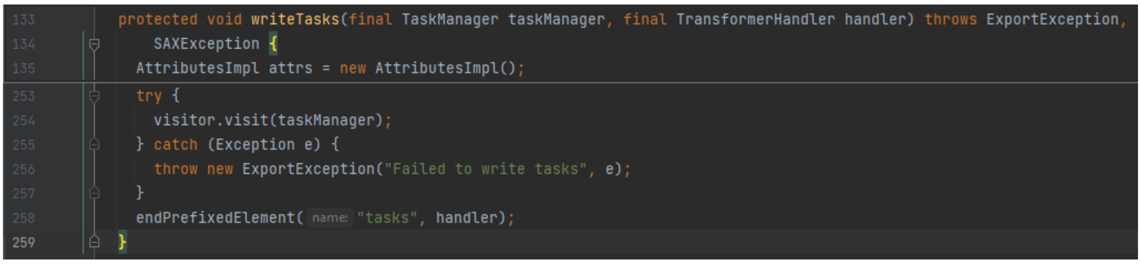
The method presented below, although not extremely long, could clearly be refactored. The first loop to find the value of the variable 'totalWidth' should be written in a separate method.



```
2 usages  dbarashev
protected void writeColumns(ColumnList visibleFields, TransformerHandler handler) throws SAXException {
  AttributesImpl attrs = new AttributesImpl();
  int totalWidth = 0;
  for (int i = 0; i < visibleFields.getSize(); i++) {
    if (visibleFields.getField(i).isVisible()) {
      totalWidth += visibleFields.getField(i).getWidth();
    }
  }
  for (int i = 0; i < visibleFields.getSize(); i++) {
    ColumnList.Column field = visibleFields.getField(i);
    if (field.isVisible()) {
      addAttribute( name: "id", field.getID(), attrs);
      addAttribute( name: "name", field.getName(), attrs);
      addAttribute( name: "width",  value: field.getWidth() * 100 / totalWidth, attrs);
      emptyElement( name: "field", attrs, handler);
    }
  }
}
```

Source: org.ganttproject.impex.htmlpdf/src/main/java/XmlSerializer

The method shown below fits into the same context mentioned above - Long Method - but the reference is here since it is a method that contains 126 lines of code. It could be refactored in many ways.

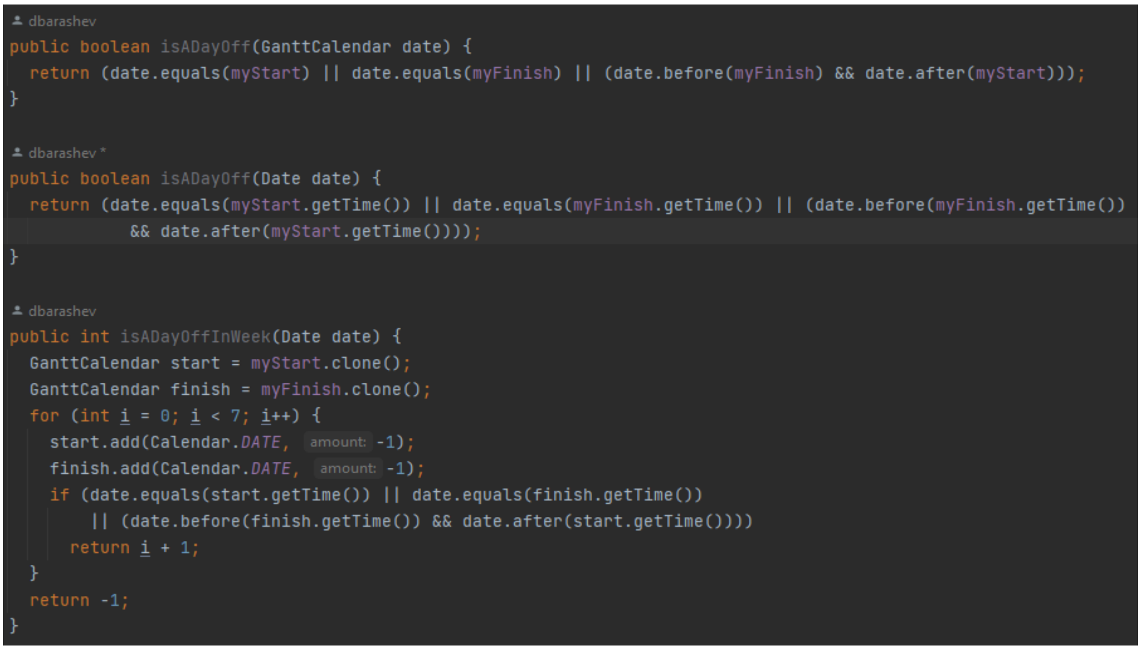


```
  protected void writeTasks(final TaskManager taskManager, final TransformerHandler handler) throws ExportException,
      SAXException {
    AttributesImpl attrs = new AttributesImpl();
    try {
      visitor.visit(taskManager);
    } catch (Exception e) {
      throw new ExportException("Failed to write tasks", e);
    }
    endPrefixedElement( name: "tasks", handler);
  }
```

Source: org.ganttproject.impex.htmlpdf/src/main/java/XmlSerializer

- **Dead Code / Feature Envy / Speculative Generality**:

The piece of code shown below represents several types of code smells. In the GanttDaysOff Class there are 3 methods that have never been used, so it is easy to see Dead Code. It could also be argued that we may be in the presence of a case of Speculative Generality, since these methods could be written to be used later in new functionalities. Analyzing even more deeply the way in which this whole package is built, we could still put these functions in another class where they would make more sense since in this class objects and information from another class are used, namely GanttCalendar.

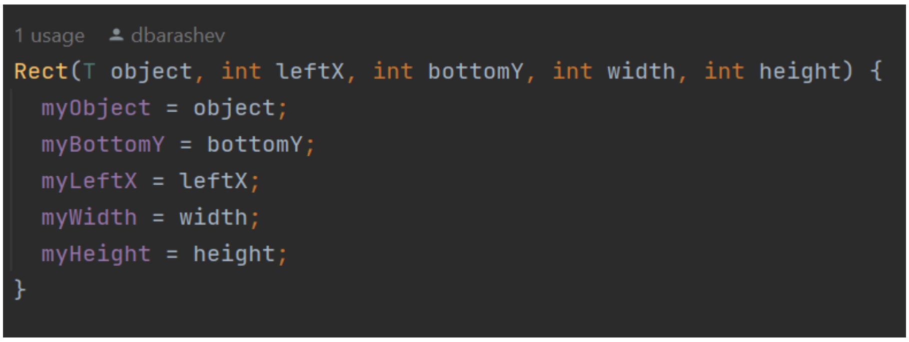


```
dbarashev
public boolean isADayOff(GanttCalendar date) {
  return (date.equals(myStart) || date.equals(myFinish) || (date.before(myFinish) && date.after(myStart)));
}

dbarashev *
public boolean isADayOff(Date date) {
  return (date.equals(myStart.getTime()) || date.equals(myFinish.getTime()) || (date.before(myFinish.getTime())
          && date.after(myStart.getTime())));
}

dbarashev
public int isADayOffInWeek(Date date) {
  GanttCalendar start = myStart.clone();
  GanttCalendar finish = myFinish.clone();
  for (int i = 0; i < 7; i++) {
    start.add(Calendar.DATE, amount: -1);
    finish.add(Calendar.DATE, amount: -1);
    if (date.equals(start.getTime()) || date.equals(finish.getTime())
        || (date.before(finish.getTime()) && date.after(start.getTime())))
      return i + 1;
  }
  return -1;
}
```

Source: biz.ganttproject.core.calendar.GanttDaysOff

- **Data Clamps**:

The next piece of code typically represents an example of Data Clamps. An object should be used here to represent the points in space, thus containing the encapsulated coordinates, not having to pass so many arguments to a function.



```
1 usage  dbarashev
Rect(T object, int leftX, int bottomY, int width, int height) {
  myObject = object;
  myBottomY = bottomY;
  myLeftX = leftX;
  myWidth = width;
  myHeight = height;
}
```

Source: biz.ganttproject.core.char.canvas.DummySpacialIndex

Reviewer: Francisco Parrinha 58360

Points:

1. It shows a good code snippet
2. It shows the exact location of code
3. It shows a good rationale for the chosen code smell
4. It shows good refactoring solutions

R: Overall, I agree with the decisions taken

Martin Magdalinchev – 58172

1. Speculative Generality

There is a function in “TimeUnitGraph” class that is never used. Maybe it was created with some purpose, but that purpose is no longer needed.

```
TimeUnit createTimeUnit(String name, TimeUnit atomUnit, int count) {
  TimeUnit result = new TimeUnitImpl(name, this, atomUnit);
  registerTimeUnit(result, count);
  return result;
}
```

Solution: As it is never used, we can remove this method and the code will continue to make sense.

Path: java/biz/ganttproject/core/time/TimeUnitGraph.java

2. Lack of comments

There can be found another smell code in this same class, which is the lack of comments. The only Javadoc comment in this class is the following:

```
/**
 * Created by IntelliJ IDEA.
 *
 * @author bard Date: 01.02.2004
 */
```

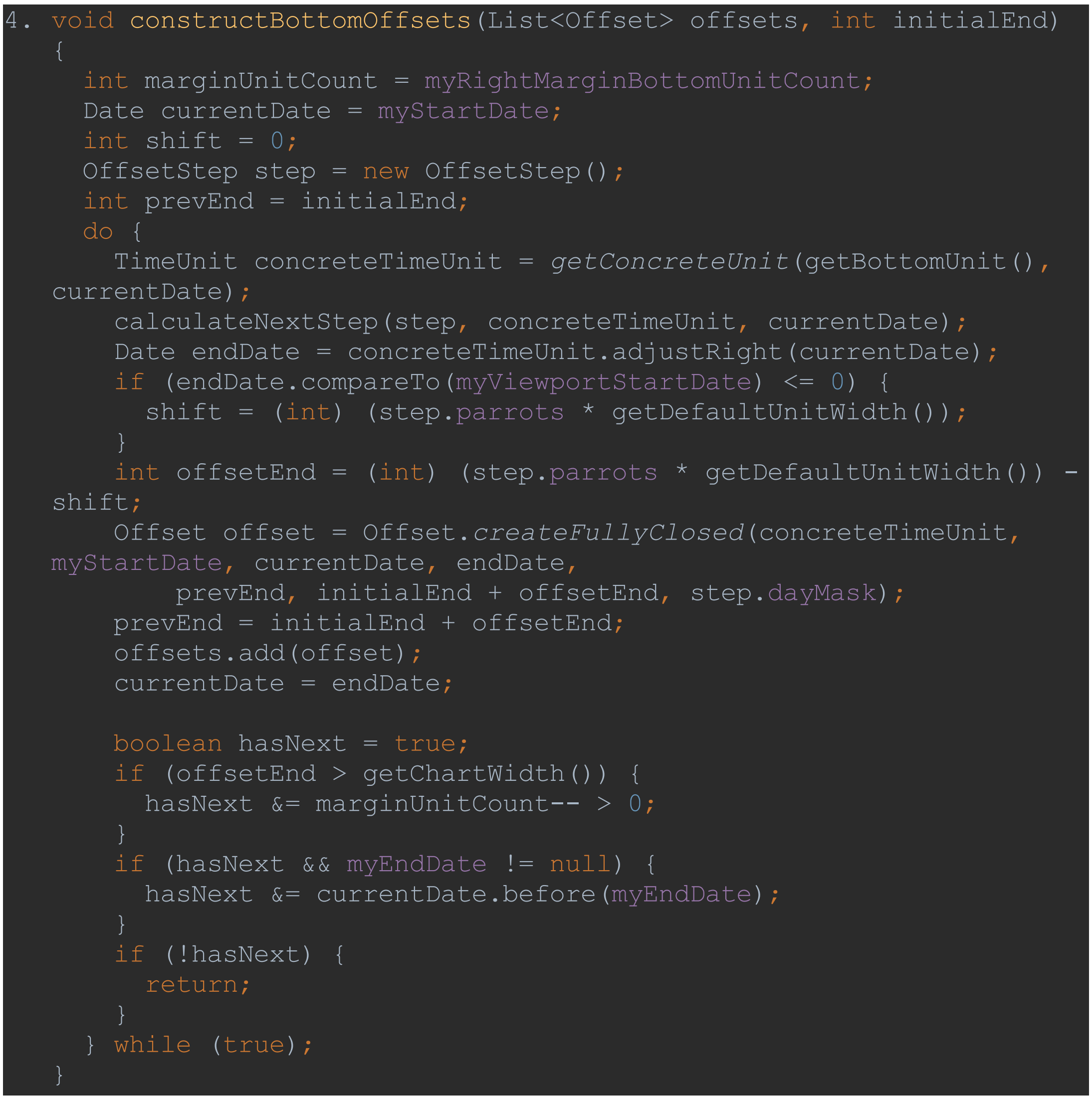
Due to the lack of comments, the code becomes much more difficult and confusing to understand. There is a simple solution: comments all the functions so other programmers can easily understand the existing code and improve it.

This are some functions to illustrate the lack of comments:

```
public TimeUnit createDateFrameableTimeUnit(String name, TimeUnit
atomUnit, int atomCount, DateFrameable framer) {
  TimeUnit result = new TimeUnitDateFrameableImpl(name, this, atomUnit,
framer);
  registerTimeUnit(result, atomCount);
  return result;
}

public TimeUnitFunctionOfDate createTimeUnitFunctionOfDate(String name,
TimeUnit atomUnit, DateFrameable framer) {
  TimeUnitFunctionOfDate result;
  result = new TimeUnitFunctionOfDateImpl(name, this, atomUnit, framer);
  registerTimeUnit(result, -1);
  return result;
}

private void registerTimeUnit(TimeUnit unit, int atomCount) {
  TimeUnit atomUnit = unit.getDirectAtomUnit();
  List<Composition> transitiveCompositions =
myUnit2compositions.get(atomUnit);
  if (transitiveCompositions == null) {
    throw new RuntimeException("Atom unit=" + atomUnit + " is unknown");
  }
```

3. Long method

The following smell code can be found in “OffsetBuilderImpl” class and is identified as long method because of the size of the function. Methods with such a big size can be very confusing and difficult to understand. They can be easily corrected by creating some auxiliar functions and separate the big “problem” in some smaller ones. (The lack of comments of this long method helps even more for the difficult interpretation)



```
4. void constructBottomOffsets(List<Offset> offsets, int initialEnd)
   {
     int marginUnitCount = myRightMarginBottomUnitCount;
     Date currentDate = myStartDate;
     int shift = 0;
     OffsetStep step = new OffsetStep();
     int prevEnd = initialEnd;
     do {
       TimeUnit concreteTimeUnit = getConcreteUnit(getBottomUnit(),
   currentDate);
       calculateNextStep(step, concreteTimeUnit, currentDate);
       Date endDate = concreteTimeUnit.adjustRight(currentDate);
       if (endDate.compareTo(myViewportStartDate) <= 0) {
         shift = (int) (step.parrots * getDefaultUnitWidth());
       }
       int offsetEnd = (int) (step.parrots * getDefaultUnitWidth()) -
   shift;
       Offset offset = Offset.createFullyClosed(concreteTimeUnit,
   myStartDate, currentDate, endDate,
           prevEnd, initialEnd + offsetEnd, step.dayMask);
       prevEnd = initialEnd + offsetEnd;
       offsets.add(offset);
       currentDate = endDate;

       boolean hasNext = true;
       if (offsetEnd > getChartWidth()) {
         hasNext &= marginUnitCount-- > 0;
       }
       if (hasNext && myEndDate != null) {
         hasNext &= currentDate.before(myEndDate);
       }
       if (!hasNext) {
         return;
       }
     } while (true);
   }
```

Path: biz/ganttproject/core/chart/grid/OffsetBuilderImpl.java

Reviewer: Francisco Parrinha 58360

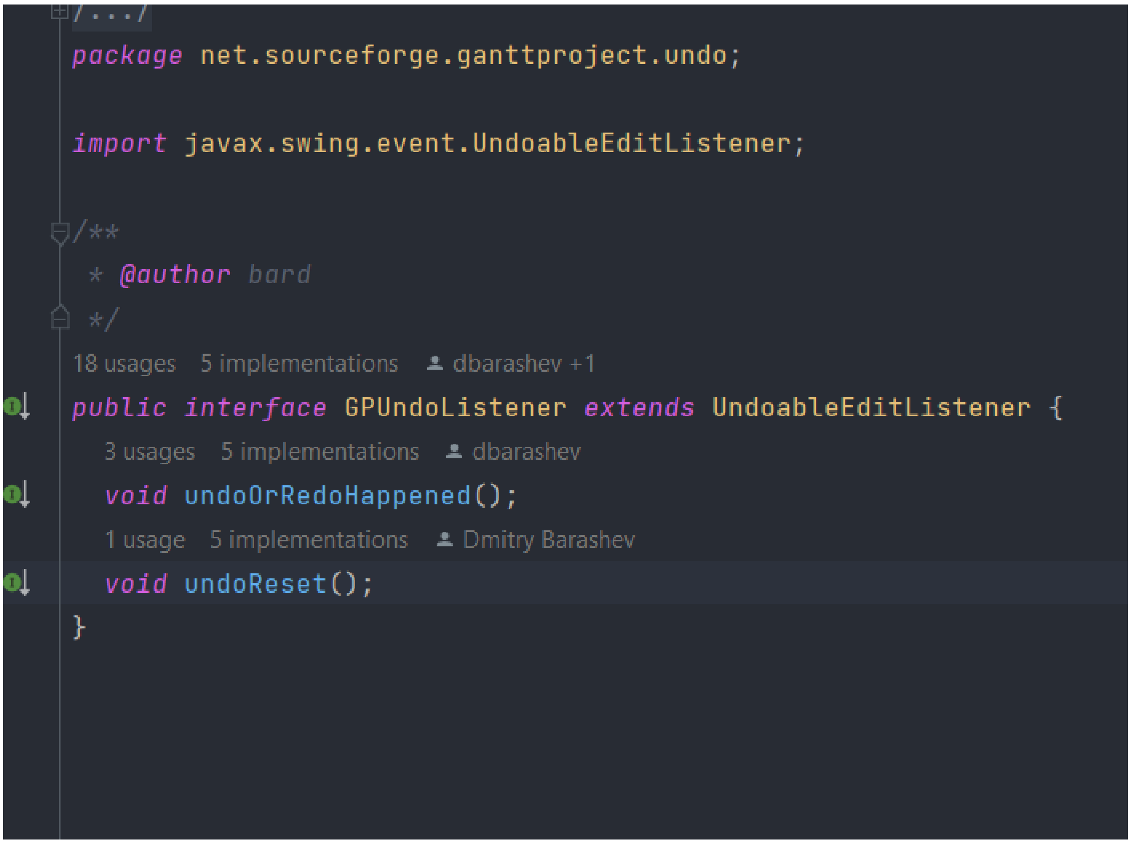
1. Good code snippets. Always displays a screenshot
2. The exact code location is always displayed
3. Good explanation of why it is a code smell
4. Good refactoring solutions

Design Patterns

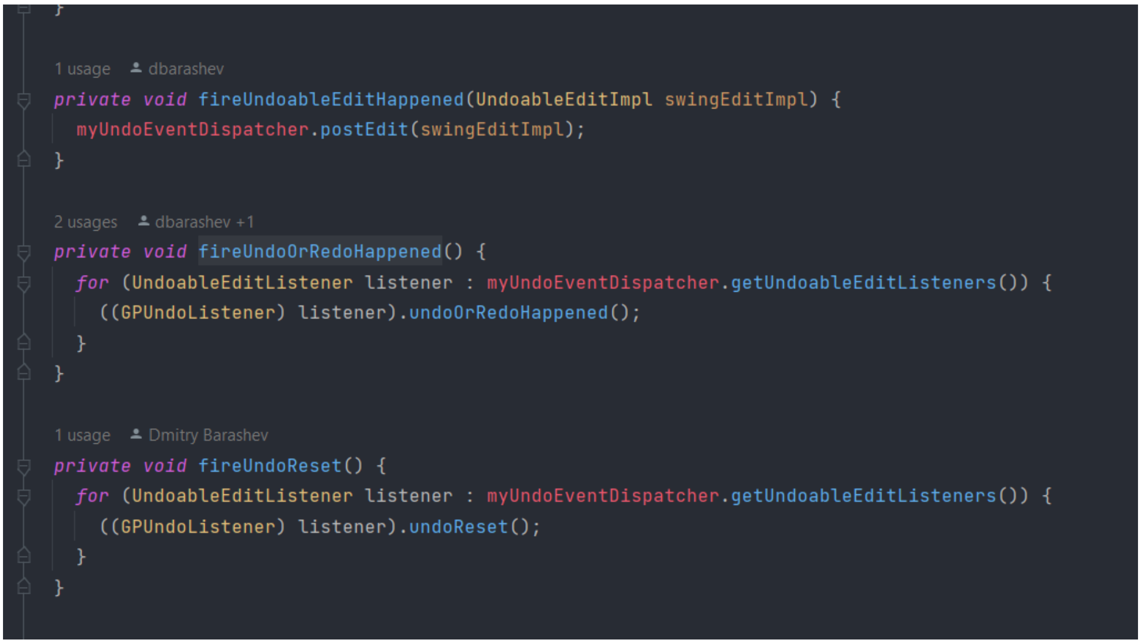
Bernardo Atalaia – 59962

1- Observer

This Design pattern is usually used to observe a specific class, that is notified once a modification has happened in this class. Other classes that are waiting for a specific state or event, can use their observer to observe it and notify once an important action is made.
In the example below, GPUndoListener is an Interface of an observer, any observer class that implements this is going to observe any actions related to “undo”.
In the second example, there is a class (“UndoManagerImpl”) notifying it’s observers (“listeners”) of an action that happened.
Path: main\java\net\sourceforge\ganttproject\undo\GPUndoListener.java
main\java\net\sourceforge\ganttproject\undo\UndoManagerImpl.java



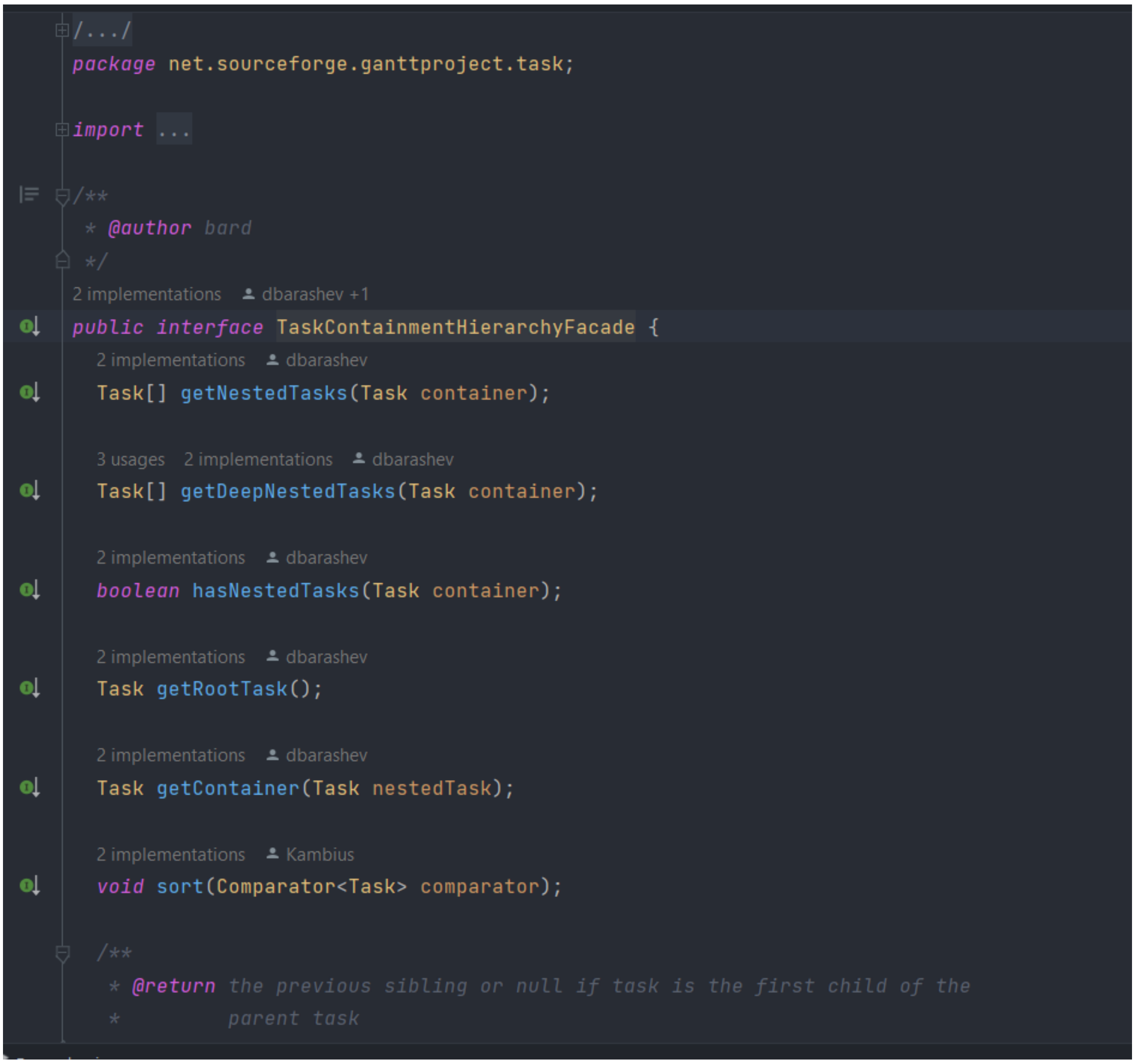
```
/.../
package net.sourceforge.ganttproject.undo;

import javax.swing.event.UndoableEditListener;

/**
 * @author bard
 */
18 usages   5 implementations   dbarashev +1
public interface GPUndoListener extends UndoableEditListener {
  3 usages   5 implementations   dbarashev
  void undoOrRedoHappened();
  1 usage   5 implementations   Dmitry Barashev
  void undoReset();
}
```



```
}

1 usage   dbarashev
private void fireUndoableEditHappened(UndoableEditImpl swingEditImpl) {
  myUndoEventDispatcher.postEdit(swingEditImpl);
}

2 usages   dbarashev +1
private void fireUndoOrRedoHappened() {
  for (UndoableEditListener listener : myUndoEventDispatcher.getUndoableEditListeners()) {
    ((GPUndoListener) listener).undoOrRedoHappened();
  }
}

1 usage   Dmitry Barashev
private void fireUndoReset() {
  for (UndoableEditListener listener : myUndoEventDispatcher.getUndoableEditListeners()) {
    ((GPUndoListener) listener).undoReset();
  }
}
```

2- Facade

This Pattern is really useful to simplify code and it’s understanding, aswell as giving a simplified interface to a complex system. In the example below, the interface TaskContainmentHierarchyFacade does just that.
Path:
main\java\net\sourceforge\ganttproject\task\TaskContainmentHierarchyFacade.java

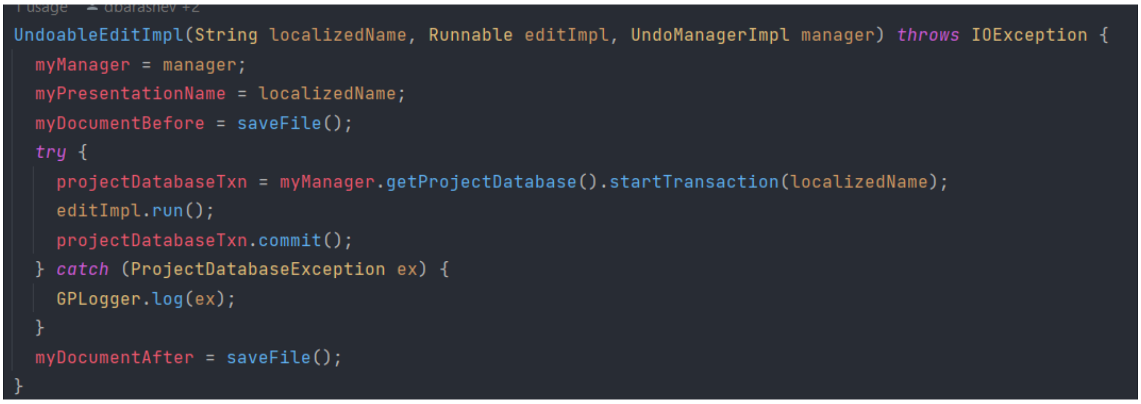


```
/.../
package net.sourceforge.ganttproject.task;

import ...

/**
 * @author bard
 */
2 implementations  dbarashev +1
public interface TaskContainmentHierarchyFacade {
  2 implementations  dbarashev
  Task[] getNestedTasks(Task container);

  3 usages  2 implementations  dbarashev
  Task[] getDeepNestedTasks(Task container);

  2 implementations  dbarashev
  boolean hasNestedTasks(Task container);

  2 implementations  dbarashev
  Task getRootTask();

  2 implementations  dbarashev
  Task getContainer(Task nestedTask);

  2 implementations  Kambius
  void sort(Comparator<Task> comparator);

  /**
   * @return the previous sibling or null if task is the first child of the
   *         parent task
```

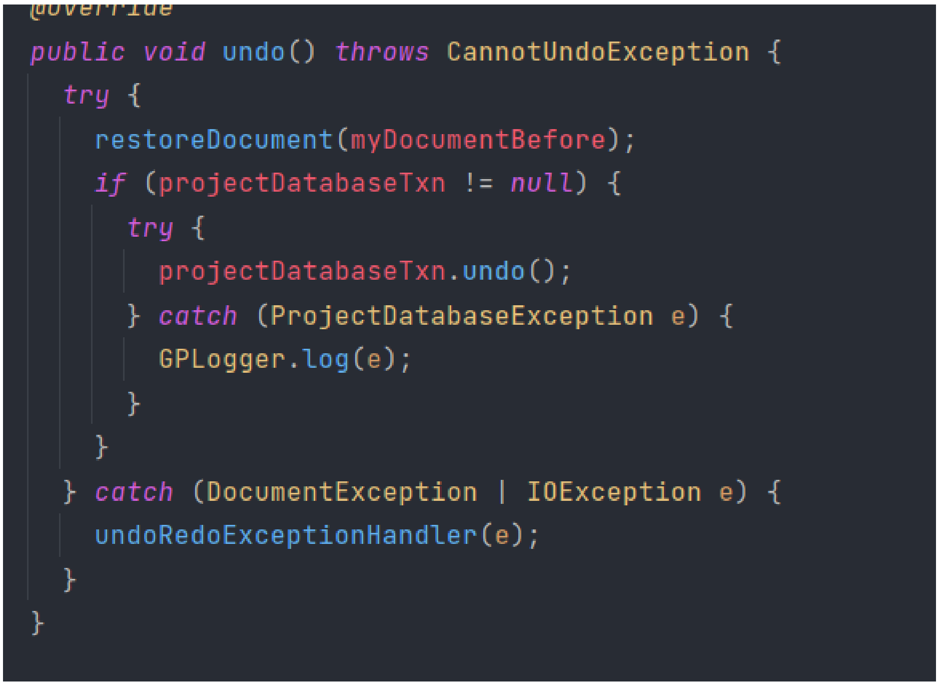
3- Memento Pattern

This pattern is basically a backup of a class, or something stored by the class itself of a previous state/version that can be brought back if for some reason an undo is needed.

As the class UndoableEditImpl has in the example below, the old document saved in case an undo is needed along side with the method undo that brings the old document back.



```
1 usage  dbarashev +2
UndoableEditImpl(String localizedName, Runnable editImpl, UndoManagerImpl manager) throws IOException {
  myManager = manager;
  myPresentationName = localizedName;
  myDocumentBefore = saveFile();
  try {
    projectDatabaseTxn = myManager.getProjectDatabase().startTransaction(localizedName);
    editImpl.run();
    projectDatabaseTxn.commit();
  } catch (ProjectDatabaseException ex) {
    GPLogger.log(ex);
  }
  myDocumentAfter = saveFile();
}
```



```
@Override
public void undo() throws CannotUndoException {
  try {
    restoreDocument(myDocumentBefore);
    if (projectDatabaseTxn != null) {
      try {
        projectDatabaseTxn.undo();
      } catch (ProjectDatabaseException e) {
        GPLogger.log(e);
      }
    }
  } catch (DocumentException | IOException e) {
    undoRedoExceptionHandler(e);
  }
}
```

Reviewer: Francisco Parrinha 58360

Points:

1. It illustrates the code snippet
2. The location is very clear
3. The rationale is very well explained. It also explains what the design pattern is

R: Overall, I agree with the chosen design patterns

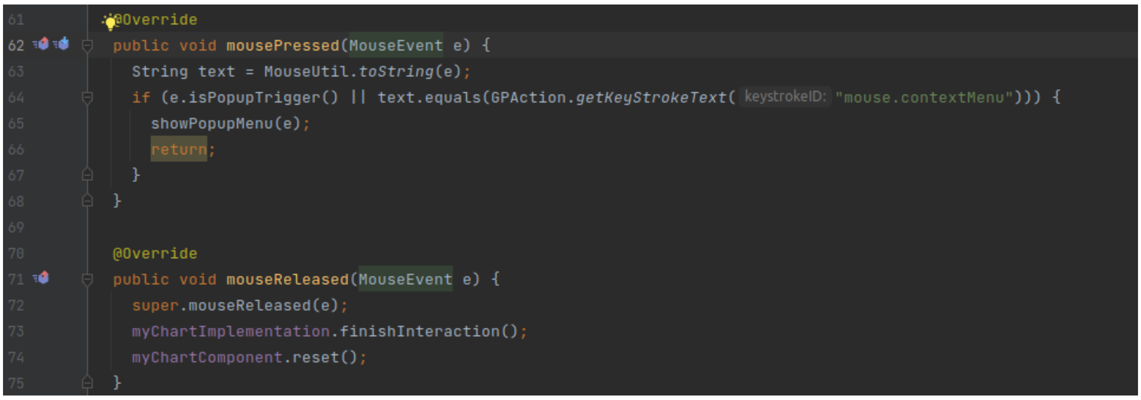
Francisco Parrinha – 58360

1. Observer:

- Class *MouseListenerBase* in: *ganttproject/src/main/java/net/sourceforge/ganttproject/chart/mouse/MouseListenerBase.java*

An observer pattern is a design pattern where an object, named *subject*, maintains a list of its dependencies. also known as *observers*.

The following class is an observer. The following code snippet shows the update function used to inform the subject:



```
@Override
public void mousePressed(MouseEvent e) {
  String text = MouseUtil.toString(e);
  if (e.isPopupTrigger() || text.equals(GPAction.getKeyStrokeText( keystrokeID: "mouse.contextMenu"))) {
    showPopupMenu(e);
    return;
  }
}

@Override
public void mouseReleased(MouseEvent e) {
  super.mouseReleased(e);
  myChartImplementation.finishInteraction();
  myChartComponent.reset();
}
```

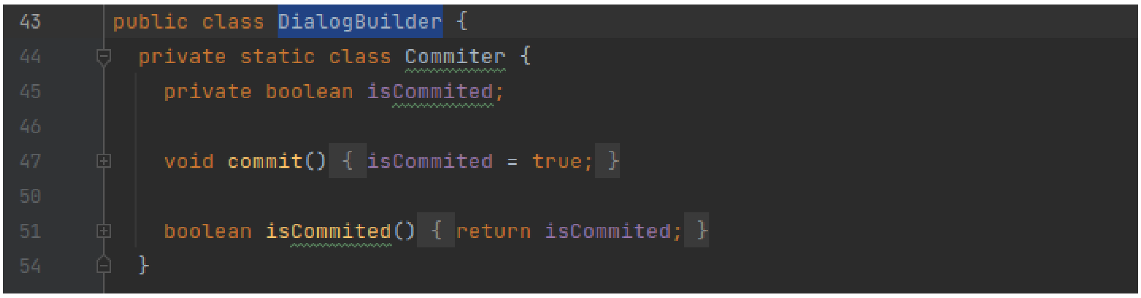
2. Builder:

- Class *DialogBuilder* in: *ganttproject/src/main/java/net/sourceforge/ganttproject/DialogBuilder.java*

A builder pattern is a design pattern that offers a flexible solution to various object-oriented programming. Its intent is to separate the construction of an object from its realization.

The following class is a builder class. It simplifies the construction of dialog objects.

The design pattern:



```
public class DialogBuilder {
  private static class Commiter {
    private boolean isCommited;

    void commit() { isCommited = true; }

    boolean isCommited() { return isCommited; }
  }
```

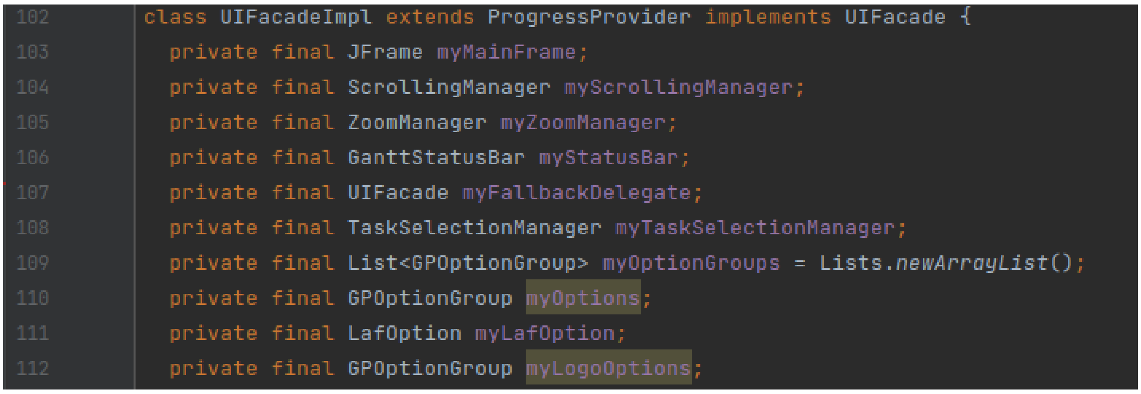
2. Facade:

- Class *UIFacadeImpl* in: *ganttproject/src/main/java/net/sourceforge/ganttproject/UIFacadeImpl.java*

A facade pattern is a design pattern commonly used in object-oriented programming. A facade is an object that serves as a front-facing interface, making complex code more accessible. It can improve its readability and its usability.

Objects instantiated from this class are facades. This class serves the purpose explained above by adding several minimal methods that have larger and more complex implementations in other classes.

A screenshot of the class:



```
class UIFacadeImpl extends ProgressProvider implements UIFacade {
  private final JFrame myMainFrame;
  private final ScrollingManager myScrollingManager;
  private final ZoomManager myZoomManager;
  private final GanttStatusBar myStatusBar;
  private final UIFacade myFallbackDelegate;
  private final TaskSelectionManager myTaskSelectionManager;
  private final List<GPOptionGroup> myOptionGroups = Lists.newArrayList();
  private final GPOptionGroup myOptions;
  private final LafOption myLafOption;
  private final GPOptionGroup myLogoOptions;
```

Reviewer: Pedro Inácio 59184

1, 2 and 3: Every single designed pattern is well identified, with reasonable references and very briefely reviews the pattern in the beggining wich is always a plus.

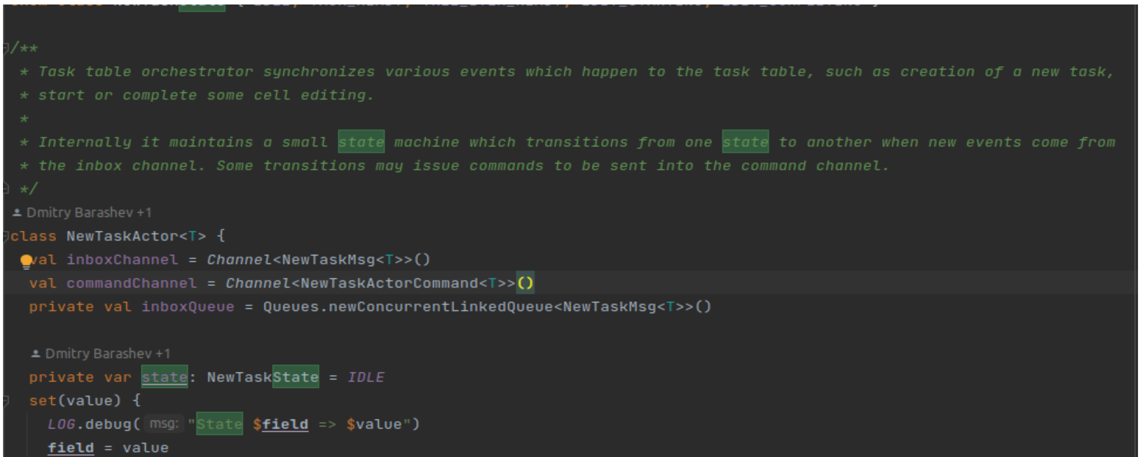
Pedro Inácio 59184

State pattern

A state pattern is present when the class depends on a series of states to decide how to act facing a situation.

This design pattern is used in package ganttview, with the goal of maintaining “context” in the actor of its current state. In fact this pattern here is so clear that the comment on top of the class NewTaskActor says it all:

```
/**
 * Task table orchestrator synchronizes various events which happen to the task table, such as creation of a new task,
 * start or complete some cell editing.
 *
 * Internally it maintains a small state machine which transitions from one state to another when new events come from
 * the inbox channel. Some transitions may issue commands to be sent into the command channel.
 */
```

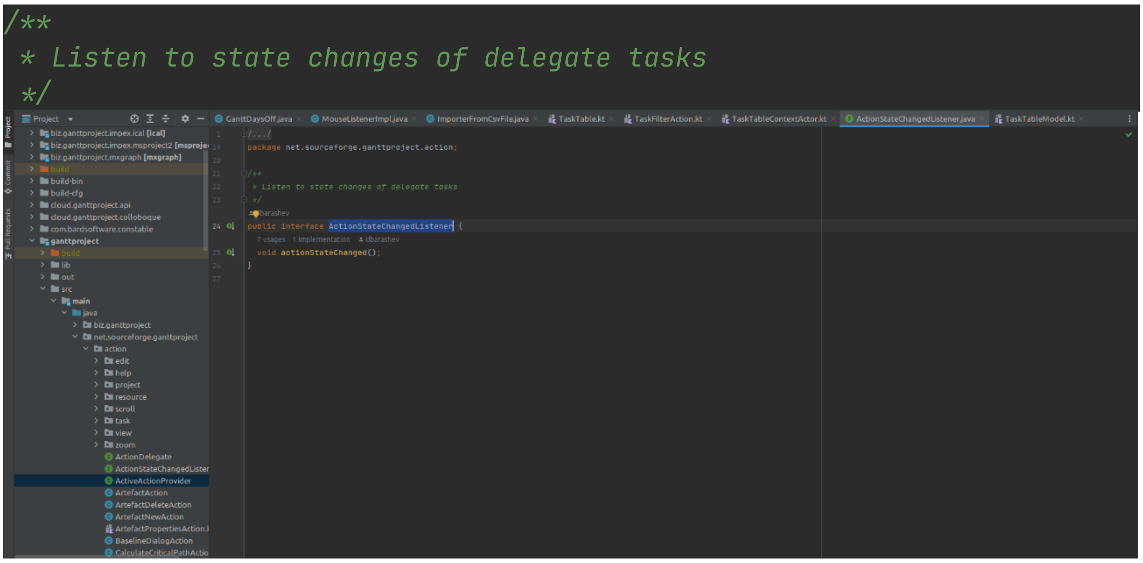


```
/**
 * Task table orchestrator synchronizes various events which happen to the task table, such as creation of a new task,
 * start or complete some cell editing.
 *
 * Internally it maintains a small state machine which transitions from one state to another when new events come from
 * the inbox channel. Some transitions may issue commands to be sent into the command channel.
 */
Dmitry Barashev +1
class NewTaskActor<T> {
  val inboxChannel = Channel<NewTaskMsg<T>>()
  val commandChannel = Channel<NewTaskActorCommand<T>>()
  private val inboxQueue = Queues.newConcurrentLinkedQueue<NewTaskMsg<T>>()

  Dmitry Barashev +1
  private var state: NewTaskState = IDLE
  set(value) {
    LOG.debug( msg: "State $field => $value")
    field = value
```

Observer pattern

The Observer design pattern is a pattern where a subject keeps a list of observers and notifies them of any state changes, usually by calling one of their methods. The observers rely on the subject to inform them of changes in the subject's state.

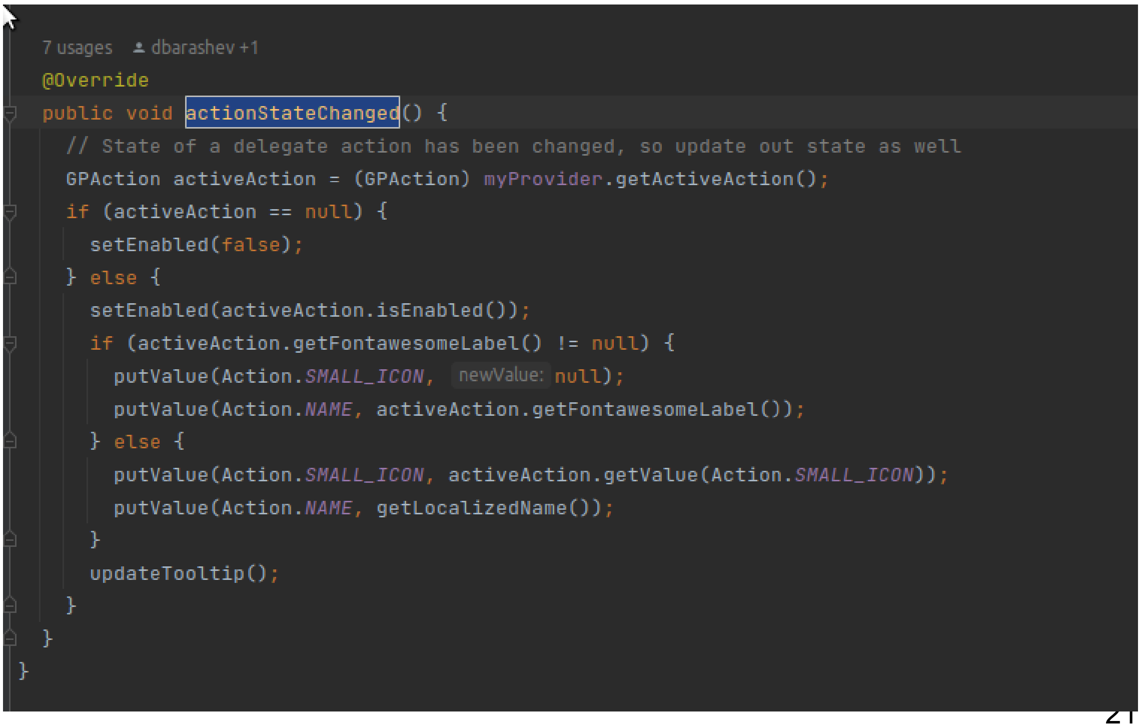
In the interface ActionStateChangedListener `its clear that any class that implements it is an observer for when a state of something changes.`



```
/**
 * Listen to state changes of delegate tasks
 */
```

```
/.../
package net.sourceforge.ganttproject.action;

/**
 * Listen to state changes of delegate tasks
 */
dbarashev
public interface ActionStateChangedListener {
  7 usages   1 implementation   dbarashev
  void actionStateChanged();
}
```

Usage example in ArtefactAction:



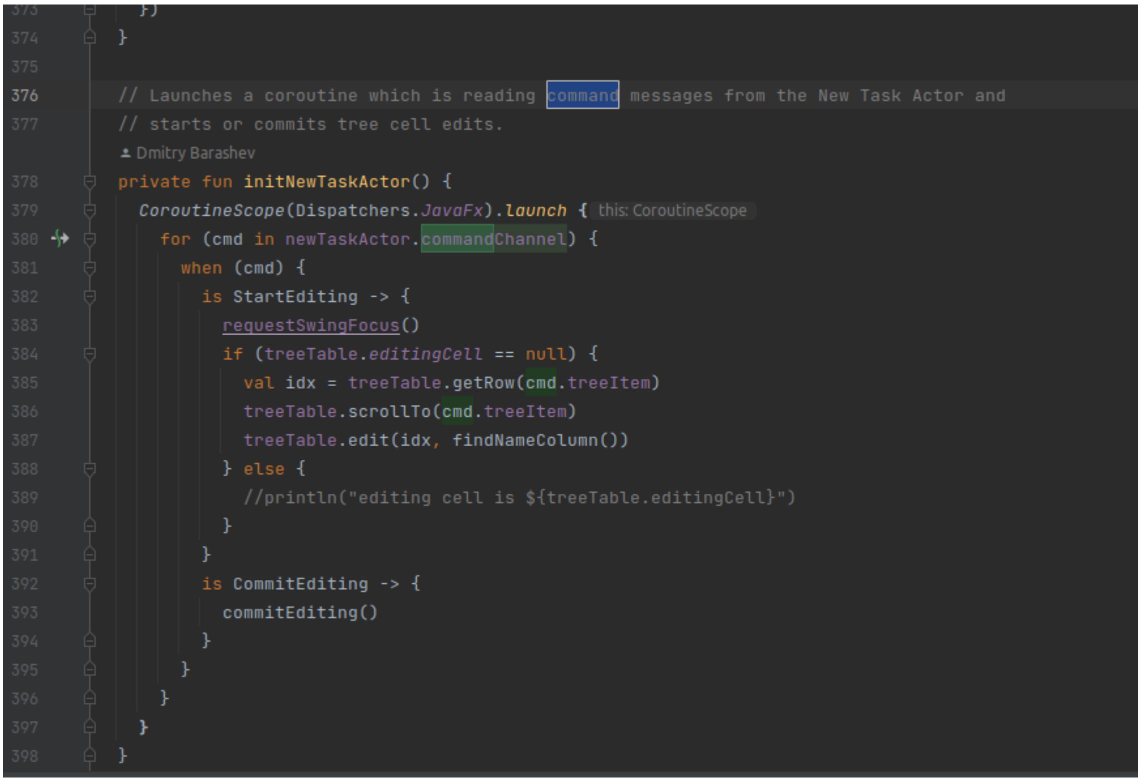
```
  7 usages   dbarashev +1
  @Override
  public void actionStateChanged() {
    // State of a delegate action has been changed, so update out state as well
    GPAction activeAction = (GPAction) myProvider.getActiveAction();
    if (activeAction == null) {
      setEnabled(false);
    } else {
      setEnabled(activeAction.isEnabled());
      if (activeAction.getFontawesomeLabel() != null) {
        putValue(Action.SMALL_ICON,  newValue: null);
        putValue(Action.NAME, activeAction.getFontawesomeLabel());
      } else {
        putValue(Action.SMALL_ICON, activeAction.getValue(Action.SMALL_ICON));
        putValue(Action.NAME, getLocalizedName());
      }
      updateTooltip();
    }
  }
}
```

Command Pattern

The command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time.

In the class TaskTable.kt the method initNewTaskActor checks for external commands to know what to do.

```
// Launches a coroutine which is reading command messages from the New Task Actor and
// starts or commits tree cell edits.
```



```
  })
}

// Launches a coroutine which is reading command messages from the New Task Actor and
// starts or commits tree cell edits.
Dmitry Barashev
private fun initNewTaskActor() {
  CoroutineScope(Dispatchers.JavaFx).launch { this: CoroutineScope
    for (cmd in newTaskActor.commandChannel) {
      when (cmd) {
        is StartEditing -> {
          requestSwingFocus()
          if (treeTable.editingCell == null) {
            val idx = treeTable.getRow(cmd.treeItem)
            treeTable.scrollTo(cmd.treeItem)
            treeTable.edit(idx, findNameColumn())
          } else {
            //println("editing cell is ${treeTable.editingCell}")
          }
        }
        is CommitEditing -> {
          commitEditing()
        }
      }
    }
  }
}
```

Reviewer: Bernardo Atalaia 59962

1: Clear and clean, with a very good example.
2: pretty straight forward information, well explained and a good example too.
3: Good example, pretty good overall aswell.

Martin Magdalinchev – 58172

1. Observer Pattern

The following interface (GPCalendarListener) represents an Observer Pattern because the subject notifies a list of observers about a change that occurred so that the observers can change their status.

```
/**
 * Calendar listeners are notified when calendar is changed, namely,
 * when weekends days change or holidays list change.
 *
 * @author dbarashev (Dmitry Barashev)
 */
public interface GPCalendarListener {
  void onCalendarChange();
}
```

Path: biz/ganttproject/core/calendar/GPCalendarListener.java

2. State Pattern

The class "WorkingUnitCounter" represents a state pattern because allows the object to alter its behaviour based on its state. In this case, the state is represented by the Boolean variable "isMoving":

```
public class WorkingUnitCounter extends ForwardTimeWalker {
  private Date myEndDate;
  private boolean isMoving = true;
  private int myWorkingUnitCounter;
  private int myNonWorkingUnitCounter;

  public WorkingUnitCounter(GPCalendarCalc calendar, TimeUnit
timeUnit) {
    super(calendar, timeUnit);
  }

  @Override
  protected boolean isMoving() {
    return isMoving;
  }

  @Override
  protected void processNonWorkingTime(Date intervalStart, Date
workingIntervalStart) {
    myNonWorkingUnitCounter++;
    isMoving = workingIntervalStart.before(myEndDate);
  }

  @Override
  protected void processWorkingTime(Date intervalStart, Date
nextIntervalStart) {
    myWorkingUnitCounter++;
    isMoving = nextIntervalStart.before(myEndDate);
  }
```

Path: biz/ganttproject/core/calendar/walker/WorkingUnitCounter.java

3. Bridge

In the following two classes can be found a “Bridge” design pattern as both of them implements the same abstract class in a different way:

```
Abstract class: abstract class GPCalendarBase implements GPCalendarCalc
{
```

Path: biz/ganttproject/core/calendar/GPCalendarBase.java

Class 1:

```
public class WeekendCalendarImpl extends GPCalendarBase implements
GPCalendarCalc {
```

```
public List<GPCalendarActivity> getActivities(Date startDate, final
Date endDate) {
  if (getWeekendDaysCount() == 0 && myOneOffEvents.isEmpty() &&
myRecurringEvents.isEmpty()) {
    return myRestlessCalendar.getActivities(startDate, endDate);
  }
  List<GPCalendarActivity> result = new
ArrayList<GPCalendarActivity>();
  Date curDayStart = myFramer.adjustLeft(startDate);
  boolean isWeekendState = (getDayMask(curDayStart) & DayMask.WORKING)
== 0;
  while (curDayStart.before(endDate)) {
    Date changeStateDayStart = doFindClosest(curDayStart, myFramer,
MoveDirection.FORWARD,
        isWeekendState ? DayType.WORKING : DayType.NON_WORKING,
endDate);
    if (changeStateDayStart == null) {
      changeStateDayStart = endDate;
    }
    if (changeStateDayStart.before(endDate) == false) {
      result.add(new CalendarActivityImpl(curDayStart, endDate,
!isWeekendState));
      break;
    }
    result.add(new CalendarActivityImpl(curDayStart,
changeStateDayStart, !isWeekendState));
    curDayStart = changeStateDayStart;
    isWeekendState = !isWeekendState;
  }
  return result;
}
```

Path: biz/ganttproject/core/calendar/WeekendCalendarImpl.java

Class 2:

```
public class AlwaysWorkingTimeCalendarImpl extends GPCalendarBase
implements GPCalendarCalc {
  @Override
  public List<GPCalendarActivity> getActivities(Date startDate, Date
endDate) {
    return Collections.singletonList((GPCalendarActivity) new
CalendarActivityImpl(startDate, endDate, true));
  }
```

Path: biz/ganttproject/core/calendar/AlwaysWorkingTimeCalendarImpl.java

Reviewer: Francisco Parrinha 58360

1. Good code snippet, very clear.
2. Always displays the exact location
3. The rationale is very detailed