Task 01 for 6CS002 - Advanced Software Engineering Topics

Name: R.M.D.S.Siriwardhena

Student ID: 2412928

13th November 2023

1) _Q.java

```
1    package base;
2    public class _Q {
3      static transient String[] stuff = {
4        "Progress comes from the intelligent use of experience.","Elbert Hubbard","",
5        "No amount of experimentation can ever prove me right; a single experiment can prove me wrong.","Albert
Einstein","",
6        "To be trusted is a greater compliment than to be loved.","George MacDonald","",
7        "Everything has beauty, but not everyone sees it.","Confucius","",
8        "Pretty much all the honest truth-telling there is in the world is done by children.","Oliver Wendell Holmes","",
9        "A lie cannot live.","Martin Luther King Jr.","",
10        "There are a lot of lies going around...and half of them are true.","Winston Churchill","",
11        "Everyone is a moon and has a dark side which he never shows to anybody.","Mark Twain","",
12        "Lies are usually caused by an undue fear of men.","Nachman of Bratslav","",
13        "If we knew what it was we were doing, it would not be called research, would it?","Albert Einstein",""};
14    }
```

Bad Smells:

Code Lines 2 - 14 :

- The class possesses data but lacks functionality, resembling a "lazy" class that essentially serves as a mere container for data. Given that the _Q class solely maintains a static array of strings without performing any substantial actions, it appears to lack value or usefulness. This observation aligns with the notion of a "Lazy Class" as per the bad smell in software design.
- Apart from setting up the data, the _Q class exclusively carries out the essential function of retaining the array of strings. This aligns with the concept of the "Data Class" bad smell, where the class predominantly houses data without any associated behavior. Essentially, it can be viewed as a data-centric class.
- There are no new remarks for this class. This could be categorized as bad smell since it doesn't improve the documentation of the code. (Although given that this is a simple class this might not be an issue)

2) ConnectionGenius.java

```
1    package base;
2    import java.net.InetAddress;
3    /**
4     * @author Kevan Buckley, maintained by __student
5     * @version 2.0, 2014
6     */
7
8    public class ConnectionGenius {
9
10     InetAddress ipa;
11
12     public ConnectionGenius(InetAddress ipa) {
13       this.ipa = ipa;
14     }
15
16     public void fireUpGame() {
17       downloadWebVersion();
18       connectToWebService();
19       awayWeGo();
20     }
21
22     public void downloadWebVersion(){
23       System.out.println("Getting specialised web version.");
24       System.out.println("Wait a couple of moments");
25     }
26
27     public void connectToWebService() {
28       System.out.println("Connecting");
29     }
30
31     public void awayWeGo(){
32       System.out.println("Ready to play");
33     }
34
35   }
```

Bad Smells:

Code Lines 3 - 5 :

- There aren't many remarks in class. Although the course and its procedures are simple, a comment section at the outset could offer more details regarding the class's goals, the importance of the fireUpGame method, and any design choices.

Code Line 10 :

- The ipa field is public, which might lead to "inappropriate intimacy". Consider making it private and providing a getter method if external classes need access.

Code Lines 12 - 14 :

- The class is more focused on storing data than on providing meaningful behavior, aligning with the "Data Class" bad smell.

3) Domino.java

```
 1  package base;
 2  /**
 3   * @author Kevan Buckley, maintained by __student
 4   * @version 2.0, 2014
 5   */
 6
 7  public class Domino implements Comparable<Domino> {
 8    public int high;
 9    public int low;
10    public int hx;
11    public int hy;
12    public int lx;
13    public int ly;
14    public boolean placed = false;
15
16    public Domino(int high, int low) {
17      super();
18      this.high = high;
19      this.low = low;
20    }
21
22    public void place(int hx, int hy, int lx, int ly) {
23      this.hx = hx;
24      this.hy = hy;
25      this.lx = lx;
26      this.ly = ly;
27      placed = true;
28    }
29
30    public String toString() {
31      StringBuffer result = new StringBuffer();
32      result.append("[");
33      result.append(Integer.toString(high));
34      result.append(Integer.toString(low));
35      result.append("]");
36      if(!placed){
37        result.append("unplaced");
38      } else {
39        result.append("(");
40        result.append(Integer.toString(hx+1));
41        result.append(",");
42        result.append(Integer.toString(hy+1));
43        result.append(")");
44        result.append("(");
45        result.append(Integer.toString(lx+1));
46        result.append(",");
47        result.append(Integer.toString(ly+1));
48        result.append(")");
49      }
50      return result.toString();
51    }
52
53    /** turn the domino around 180 degrees clockwise*/
54
55    public void invert() {
56      int tx = hx;
57      hx = lx;
58      lx = tx;
59
60        int ty = hy;
61        hy = ly;
62        ly = ty;
63      }
64
65      public boolean ishl() {
66        return hy==ly;
67      }
68
69
70      public int compareTo(Domino arg0) {
71        if(this.high < arg0.high){
72          return 1;
73        }
74        return this.low - arg0.low;
75      }
76
77
78
79    }
```

Bad Smells:

Code Lines 1 - 79 :

- The Domino class has numerous fields and methods, possibly exceeding the necessary size for its responsibilities and resulting in a "Large Class" bad smell. Additionally, the class features a relatively high number of fields (such as high, low, hx, hy, lx, ly, placed), which may complicate management and understanding.

Code Lines 2 - 5 :

- Though there are some comments in the code, adding more descriptive comments could improve readability and understanding of the code.

Code Lines 8 - 14 :

- The class uses public fields (high, low, hx, hy, lx, ly, placed), violating encapsulation and contributing to the "Long Parameter List" bad smell.

Code Lines 22 :

- The code utilizes primitive data types (ints) to represent dominoes instead of creating a custom class or enum for better abstraction. This approach can be deemed a "Primitive Obsession" bad smell.

Code Lines 30 - 49 :

- The presence of conditional logic for formatting output within the toString method raises concerns about the code's design. While some level of conditional logic is occasionally necessary, an excessive amount can be indicative of poor design, potentially resulting in maintenance challenges. This situation might be considered an "Inefficient Algorithms" bad smell.

Code Lines 30 - 67 :

- The toString method is lengthy with conditional formatting, possibly causing a "Long Method" code smell. Similarly, the toString and invert methods, while task-specific, could benefit from extracting parts into smaller methods for improved readability.

Code Lines 70 - 75 :

- The compareTo method might have a logical error in its implementation. The comparison should be based on both high and low values. The current implementation only considers high, potentially leading to incorrect sorting. (Inefficient Algorithms)

4) IOLibrary.java

```
1    package base;
2    import java.io.*;
3    import java.net.*;
4    /**
```

```
5     * @author Kevan Buckley, maintained by __student
6     * @version 2.0, 2014
7     */
8
9    public final class IOLibrary {
10     public static String getString() {
11       BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
12       do {
13         try {
14           return r.readLine();
15         } catch (Exception e) {
16         }
17       } while (true);
18     }
19
20     public static InetAddress getIPAddress() {
21       BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
22       do {
23         try {
24           String[] chunks = r.readLine().split("\\.");
25           byte[] data = {
Byte.parseByte(chunks[0]),Byte.parseByte(chunks[1]),Byte.parseByte(chunks[2]),Byte.parseByte(chunks[3])};
26           return Inet4Address.getByAddress(data);
27         } catch (Exception e) {
28         }
29       } while (true);
30     }
31
32   }
```

Bad Smells:

Code Lines 1 - 32 :

- The class exhibits a "Large Class" issue as it violates the Single Responsibility Principle by containing multiple methods handling disparate concerns, including console input and InetAddress.
- The absence of comments throughout the entire class diminishes code readability and makes it challenging for developers to understand the intended functionality of methods and classes.

Code Lines 11 - 21 :

- The initialization of the BufferedReader is repeated in both the getString and getIPAddress methods. This redundancy in code can lead to maintenance challenges and increases the risk of inconsistencies.(Duplicate Code)

Code Lines 12 - 17 & 22 - 29 :

- The identified problem involves an "Infinite Loop - Long Method" issue within the getString and getIPAddress methods. These methods lack explicit exit conditions for their loops (lines 12-17 and 22-29), posing the risk of potentially long-running processes.

Code Lines 11- 21 & 20 -26 :

- The code presents an issue of inappropriate intimacy, particularly in the getIPAddress method , where there is an intimate understanding of the internal structure of InetAddress involving byte manipulation. Additionally, on lines 11 and 21, the class tightly couples with the System.in input stream, reflecting another instance of inappropriate intimacy.

Code Lines 11 - 21 :

- The presence of message chains, particularly in lines 11 and 21, suggests that the class is mediating communication between the BufferedReader and the client code. This intermediary role can lead to increased complexity and potential maintenance challenges. (Message Chains - Middle Man)

Code Lines 20 -30 :

- The code exhibits signs of speculative generality, particularly in methods like getIPAddress (lines 20-26), which seem to have been designed for potential future use without a clear current need. This situation is indicative of a speculative generality bad smell, as the code aims for reusability but lacks specific functionality and proper handling of edge cases. The broader code (lines 10-18 & 20-30) appears to anticipate future requirements without a present necessity.

5) IOSpecialist.java

```
1    package base;
2
3    /**
4     * @author Kevan Buckley, maintained by __student
5     * @version 2.0, 2014
6     */
7
8    public class IOSpecialist {
9      public IOSpecialist() {
10     }
11     public String getString(){
12       return IOLibrary.getString();
13     }
14   }
```

Bad Smells:

Code Lines 1 - 14 :

- The absence of comments throughout the entire class diminishes code readability and makes it challenging for developers to understand the intended functionality of methods and classes.

Code Lines 8 - 14 :

- The IOSpecialist class functions as an intermediary between the client code and the IOLibrary, merely delegating the getString method without adding any substantial value or functionality of its own. This pattern is a clear instance of the "Middle Man" bad smell, where the class serves as an unnecessary intermediary, introducing an extra layer without contributing meaningful enhancements.

6) Location.java

```
1    package base;
2    import java.awt.Color;
3    import java.awt.Graphics;
4    import java.io.BufferedReader;
5    import java.io.InputStreamReader;
6
7    /**
8     * @author Kevan Buckley, maintained by __student
9     * @version 2.0, 2014
10    */
11
12   public class Location extends SpacePlace {
13     public int c;
14     public int r;
15     public DIRECTION d;
16     public int tmp;
17     public enum DIRECTION {VERTICAL, HORIZONTAL};
18
19     public Location(int r, int c) {
20       this.r = r;
21       this.c = c;
22     }
23
24     public Location(int r, int c, DIRECTION d) {
25       this(r,c);
26       this.d=d;
27     }
28
29     public String toString() {
30       if(d==null){
31         tmp = c + 1;
```

```java
32        return "(" + (tmp) + "," + (r+1) + ")";
33      } else {
34        tmp = c + 1;
35        return "(" + (tmp) + "," + (r+1) + "," + d + ")";
36      }
37    }
38
39    public void drawGridLines(Graphics g) {
40      g.setColor(Color.LIGHT_GRAY);
41      for (tmp = 0; tmp <= 7; tmp++) {
42        g.drawLine(20, 20 + tmp * 20, 180, 20 + tmp * 20);
43      }
44      for (int see = 0; see <= 8; see++) {
45        g.drawLine(20 + see * 20, 20, 20 + see * 20, 160);
46      }
47    }
48
49    public static int getInt() {
50      BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
51      do {
52        try {
53          return Integer.parseInt(r.readLine());
54        } catch (Exception e) {
55        }
56      } while (true);
57    }
58  }
```

Bad Smells:

Code Lines 1 - 58 :

- The lack of meaningful comments throughout the entire class diminishes code readability and makes it challenging for developers to understand the intended functionality of methods and classes.
- The Location class exhibits a "Large Class" bad smell, since it may have too many responsibilities due to its multiple fields and methods.

Code Lines 16 - 47 :

- The presence of a temporary field, as indicated by the tmp field in the class, might introduce a "Temporary Field" bad smell. This smell occurs when a field is used only for a brief period and is not part of the class's long-term structure or functionality.

Code Lines 19 - 37 :

- The class has methods that focus on storing data, indicating a potential "Data Class" bad smell.

7) Main.java

`Main.java code file`

Bad Smells:

Code Lines 1 - 887 :

- The lack of meaningful comments throughout the code diminishes its readability, posing challenges for developers in understanding the purpose of methods and classes. (Lines 53, 59, 317, 340, 366, 374, 382, 396, 404, 411, 415): Although commented code can sometimes be useful for future reference, too much commented code might indicate obsolete or unnecessary lines.. Effective code should be comprehensible to all, and the use of comments is recommended but should be applied judiciously and only when necessary.
- The Main class is characterized by a substantial number of fields, methods, and intricate logic, suggesting the potential for a more modular and organized structure. This situation hints at a "Large Class" bad smell, where the class might bear excessive responsibilities due to its multitude of fields and methods.
- Lines with similar patterns or repetitive code blocks.Duplicate code sections may be found across the entire file, notably in similar exception handling blocks and repetitive switch statements.

Code Lines 20 - 24 :

- Repeated use of arrays int[][] could be replaced with a custom class for better encapsulation and readability.
- The representation of player information (playerName) as a simple string might be a candidate for introducing a more structured Player class.

Code Lines 25 - 28 :

- The "Primitive Obsession" smell could be eliminated if primitive types (int, long) were used to represent game state instead of being refactored into more expressive objects or classes.

Code Line 30 :

- The PictureFrame class has an unused field reroll and an unused method reset.
- PictureFrame appears to be intended as a JFrame but is not correctly implemented as one.

Code Lines 32 - 72 :

- The code segments generated by generateDominoes() and generateGuesses() are comparable, generating the "Duplicate code" Bad smell.

Code Lines 181 - 217 :

- Within the methods, there exist closely related parameters, such as x, y, d, and e. This aggregation of related parameters is a potential instance of "Data Clumps." To improve code organization and maintainability, consider encapsulating these parameters into an object that can represent a position and direction, thereby addressing the data clumps issue.
- Methods tend to take multiple parameters contributing to the “long parameter list” bad smell.

Code Lines 219 - 280 :

- There are repeated searches through List<Domino> collections using similar criteria. Contributing to “Data Clumps” bad smell; these could potentially be unified into a single method with more generalized search criteria.
- Several method calls are seen in sequences, which might indicate direct access to internal data of other objects (‘Domino’ class).This leads to “ Message chain” bad smell.If the Domino class is directly accessed by other classes, encapsulating the required functionality within the Domino class itself rather than retrieving its data externally.

Code Lines 283 - 857 :

- The run() method's extensive length, coupled with its execution of multiple tasks, contributes to the "Long Method" bad smell. To address this issue and improve code quality, consider dividing the run() method into smaller, more focused methods. This restructuring is crucial for mitigating the long method smell and, in turn, enhancing both readability and maintainability.
- The presence of divergent change is evident as alterations in both game logic and user interface are managed within the same method. This situation suggests a lack of separation of concerns, potentially leading to difficulties in maintaining and evolving the codebase. Refactoring to isolate and modularize distinct functionalities, enhancing code organization and promoting a more maintainable design.
- The IOSpecialist class acts as a middleman, forwarding calls to IOLibrary.

Code Lines 323 - 841 :

- The run() method incorporates multiple switch case statements, notably in the c2 switch statement. The extensive use of switch statements, as observed in the code, indicates complex branching logic, contributing to the "Switch Statements" bad smell. Refactoring the code to eliminate or reduce reliance on switch statements, promoting a cleaner and more maintainable code structure.

Code Lines 283 - 842 :

- The run() method's considerable length, coupled with its performance of multiple tasks, constitutes a "Long Method" bad smell. To address this issue and enhance code quality, it is recommended to break down the run() method into smaller, more focused methods. This restructuring not only mitigates the long method smell but also contributes to improved readability and maintainability.
- Some variable names are cryptic or uninformative, such as _$_, c2, cf, pf, etc. Using more descriptive names can significantly improve code readability and comprehension.

- There's a mix of user interface (UI) and game logic present in the run() method, making it challenging to separate the two. Preferably its a good practice to keep the UI and game logic as separate as possible.
- The IOLibrary class has only static methods, indicating a lack of instance-level functionality.Consider converting it to a proper utility class or introducing instance methods.
- The IOSpecialist class is a potential candidate for laziness.If it only delegates calls without adding significant behavior, consider removing it.

Code Lines 769- 853:

- There is a repeated pattern of printing values and flushing the PrintWriter.Consider extracting this pattern into a method to avoid Code duplication.

Code Lines 859 - 861 :

- The main() method instantiates the Main class and starts the game by invoking the run() method which contributes to "Message Chains" bad smell.

Code Lines 863 - 866 :

- drawDominoes and drawGuesses appear to have potentially high parameter lists.
- The method drawDominoes() in Location class has feature envy for '_d' list.
- The Main class has direct access to the private lists of Domino objects (_d and _g) adding "Inappropriate Intimacy" bad smell.
- The chaining of method calls like pf.dp.drawDomino(g, d) might indicate a message chain. Consider encapsulating such behavior or introducing intermediary objects to reduce coupling.

Code Lines 869 - 879 :

- The gecko method seems to involve unnecessary recursion and bit manipulation, which might make it harder to understand and maintain.

8) MultiLingualStringTable.java

```
1    package base;
2   /**
3    * @author Kevan Buckley, maintained by __student
4    * @version 2.0, 2014
5    */
6
7   public class MultiLingualStringTable {
8     private enum LanguageSetting {English, Klingon}
9     private static LanguageSetting cl = LanguageSetting.English;
10    private static String [] em = {"Enter your name:", "Welcome", "Have a good time playing Abominodo"};
11    private static String [] km = {"'el lIj pong:", "nuqneH", "QaQ poH Abominodo"};
12
13    public static String getMessage(int index){
14      if(cl == LanguageSetting.English ){
15        return em[index];
16      } else {
17        return km[index];
18      }
19
20    }
21  }
```

Bad Smells:

Code Lines 2 - 5 :

- While there are comments indicating the author and version, there is no inline documentation for the methods. Consider adding comments to explain the purpose and usage of the class and methods.

Code Lines 10 - 11 :

- The `em` and `km` arrays could be considered data clumps. Consider encapsulating them into a separate class or a dedicated structure if they grow in complexity.

Code Lines 13 - 18 :

- The `getMessage` method accesses the `cl` field, indicating a close relationship with the `LanguageSetting` enum.Consider encapsulating this logic within the `LanguageSetting` enum or a dedicated language-related class.
- The direct access to the `cl` field from outside the class might indicate inappropriate intimacy. Consider providing controlled access to this field, possibly through accessor methods.

9) PictureFrame.java

```java
package base;
import java.awt.*;

import javax.swing.*;
/**
 * @author Kevan Buckley, maintained by __student
 * @version 2.0, 2014
 */

public class PictureFrame {
  public int[] reroll = null;
  public Main master = null;

  class DominoPanel extends JPanel {
    private static final long serialVersionUID = 4190229282411119364L;

    public void drawGrid(Graphics g) {
      for (int are = 0; are < 7; are++) {
        for (int see = 0; see < 8; see++) {
          drawDigitGivenCentre(g, 30 + see * 20, 30 + are * 20, 20,
              master.grid[are][see]);
        }
      }
    }



    public void drawHeadings(Graphics g) {
      for (int are = 0; are < 7; are++) {
        fillDigitGivenCentre(g, 10, 30 + are * 20, 20, are+1);
      }

      for (int see = 0; see < 8; see++) {
        fillDigitGivenCentre(g, 30 + see * 20, 10, 20, see+1);
      }
    }

    public void drawDomino(Graphics g, Domino d) {
      if (d.placed) {
        int y = Math.min(d.ly, d.hy);
        int x = Math.min(d.lx, d.hx);
        int w = Math.abs(d.lx - d.hx) + 1;
        int h = Math.abs(d.ly - d.hy) + 1;
        g.setColor(Color.WHITE);
        g.fillRect(20 + x * 20, 20 + y * 20, w * 20, h * 20);
        g.setColor(Color.RED);
        g.drawRect(20 + x * 20, 20 + y * 20, w * 20, h * 20);
        drawDigitGivenCentre(g, 30 + d.hx * 20, 30 + d.hy * 20, 20, d.high,
            Color.BLUE);
        drawDigitGivenCentre(g, 30 + d.lx * 20, 30 + d.ly * 20, 20, d.low,
            Color.BLUE);
      }
    }

    void drawDigitGivenCentre(Graphics g, int x, int y, int diameter, int n) {
      int radius = diameter / 2;
      g.setColor(Color.BLACK);
      // g.drawOval(x - radius, y - radius, diameter, diameter);
      FontMetrics fm = g.getFontMetrics();
      // convert the string to an integer
      String txt = Integer.toString(n);
      g.drawString(txt, x - fm.stringWidth(txt) / 2, y + fm.getMaxAscent() / 2);
    }

    void drawDigitGivenCentre(Graphics g, int x, int y, int diameter, int n,
        Color c) {
      int radius = diameter / 2;
      g.setColor(c);
      // g.drawOval(x - radius, y - radius, diameter, diameter);
      FontMetrics fm = g.getFontMetrics();
      String txt = Integer.toString(n);
      g.drawString(txt, x - fm.stringWidth(txt) / 2, y + fm.getMaxAscent() / 2);
    }

    void fillDigitGivenCentre(Graphics g, int x, int y, int diameter, int n) {
      int radius = diameter / 2;
      g.setColor(Color.GREEN);
      g.fillOval(x - radius, y - radius, diameter, diameter);
      g.setColor(Color.BLACK);
      g.drawOval(x - radius, y - radius, diameter, diameter);
      FontMetrics fm = g.getFontMetrics();
      String txt = Integer.toString(n);
      g.drawString(txt, x - fm.stringWidth(txt) / 2, y + fm.getMaxAscent() / 2);
    }

    protected void paintComponent(Graphics g) {
      g.setColor(Color.YELLOW);
      g.fillRect(0, 0, getWidth(), getHeight());

      // numbaz(g);
      //
      // if (master!=null && master.orig != null) {
      // drawRoll(g, master.orig);
      // }
      // if (reroll != null) {
```

```
96        // drawReroll(g, reroll);
97        // }
98        //
99        // drawGrid(g);
100       Location l = new Location(1,2);
101
102       if (master.mode == 1) {
103         l.drawGridLines(g);
104         drawHeadings(g);
105         drawGrid(g);
106         master.drawGuesses(g);
107       }
108       if (master.mode == 0) {
109         l.drawGridLines(g);
110         drawHeadings(g);
111         drawGrid(g);
112         master.drawDominoes(g);
113       }
114     }
115
116     public Dimension getPreferredSize() {
117       // the application window always prefers to be 202x182
118       return new Dimension(202, 182);
119     }
120   }
121
122   public DominoPanel dp;
123
124   public void PictureFrame(Main sf) {
125     master = sf;
126     if (dp == null) {
127       JFrame f = new JFrame("Abominodo");
128       dp = new DominoPanel();
129       f.setContentPane(dp);
130       f.pack();
131       f.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
132       f.setVisible(true);
133     }
134   }
135
136   public void reset() {
137     // TODO Auto-generated method stub
138
139   }
140
141 }
```

Bad Smells:

Code Lines 1 - 141 :

- The class has both the `PictureFrame` and `DominoPanel` classes making it a “Large Class” . Separating these into their own files for better organization.
- The class is likely to change for various reasons, such as changes in the drawing logic or changes in the `Domino` class. Separation of concerns to avoid divergent changes.
- The `DominoPanel` class accesses the `master` object frequently leading to “feature envy” bad smell. Consider whether some of these responsibilities could be moved to the `Master` class.

Code Lines 55 - 73 :

- There is a slight code repetition in the `drawDigitGivenCentre` methods, specifically in calculating the text position. This could be refactored for better maintainability.

Code Lines 86 - 114 :

- The `paintComponent` method is relatively long. Consider breaking it down into smaller, more focused methods for better readability and maintainability.
- In lines 90 - 99 there are commented out sections implying obsolete code which can lead to confusion in future maintenance , it is recommended to remove such comments.

Code Lines 122 - 134 :

- The `DominoPanel` class has close interactions with the `master` object, which leads to “inappropriate intimacy”. Consider whether these interactions could be reduced.

10)   SpacePlace.java

```
1   package base;
2   /**
3    * @author Kevan Buckley, maintained by __student
4    * @version 2.0, 2014
5    */
6
7   public class SpacePlace {
8     private int xOrg;
9     private int yOrg;
10    private double theta;
```

```java
11    private double phi;
12
13    public SpacePlace() {
14      xOrg = 0;
15      yOrg = 0;
16    }
17
18    public SpacePlace(double theta, double phi) {
19      super();
20      this.theta = theta;
21      this.phi = phi;
22    }
23
24    public int getxOrg() {
25      return xOrg;
26    }
27
28    public void setxOrg(int xOrg) {
29      this.xOrg = xOrg;
30    }
31
32    public int getyOrg() {
33      return yOrg;
34    }
35
36    public void setyOrg(int yOrg) {
37      this.yOrg = yOrg;
38    }
39
40    public double getTheta() {
41      return theta;
42    }
43
44    public void setTheta(double theta) {
45      this.theta = theta;
46    }
47
48    public double getPhi() {
49      return phi;
50    }
51
52    public void setPhi(double phi) {
53      this.phi = phi;
54    }
55
56  }
```

Bad Smells:

Code Lines 1 - 56 :

- The SpacePlace class might be considered somewhat "lazy" as it mainly holds data and lacks significant behavior
- The SpacePlace class could be considered a data class, as it primarily holds data and lacks significant behavior.
- The class lacks comments, which could make it challenging for others (or even the original developer) to understand the code fully.

Code Lines 7 - 11 :

- The fields xOrg, yOrg, theta, and phi are primitive types. Depending on the context, it might be better to use well-named classes or enums to represent these values and avoid primitive obsession.

Code Lines 8 - 11 :

- The SpacePlace class has a default constructor and another constructor that takes theta and phi. If changes related to spatial coordinates or orientation often require modifications to this class, it might be an indication of divergent change.
- Default constructor might be unnecessary, potentially leading to unused or underused methods (Lazy Class).

Code Lines 18 - 22 :

- The class uses primitive types (int and double) to represent spatial coordinates and orientation. Depending on the application's complexity, it might be more suitable to use a dedicated class or structure.
- In the line 19 parameterized constructor, there's a super() call. Since SpacePlace does not extend another class (explicitly), this call is unnecessary and can be considered as an indication of lazy coding.

Code Lines 24 - 56 :

- The class directly exposes its internal state through getter and setter methods, which might be considered a form of inappropriate intimacy.