CSE307: Software Engineering

Code Smell Detection using SonarLint and CheckStyle

Submitted By:

Sub-Section: A2

Group Members:

1. Rasman Mubtasim Swargo (ID: 1705051)
2. Apurba Saha (ID: 1705056)

Department: CSE

Level 3 Term 1

Date of submission: 28 July, 2021

Overview

Code Smells are issues with the style of the code that makes code harder to read, understand, and maintain. They are super common. Code smells are usually not bugs. They are not technically incorrect and do not prevent the program from functioning. Instead, they indicate weaknesses in design that may slow down development or increase the risk of bugs or failures in the future.

In this assignment, we are checking some of our past codes for bad code smell. For C++ codes, we are using SonarLint and for Java codes, we are using CheckStyle and built-in Intellij tool.

All the codes can be found in the following github repository
https://github.com/diponsaha007/Code-Smell

Note: As all the codes lack comments, function and class documentation, consistent naming, we will not mention these everytime.

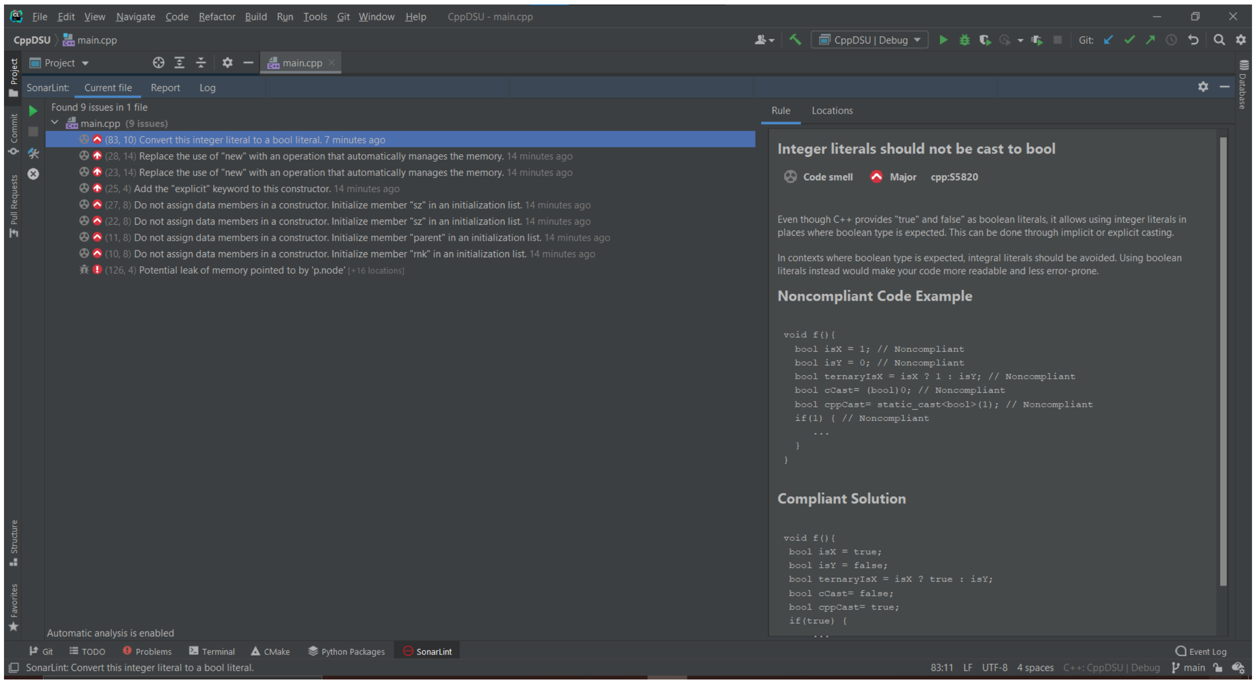
<u>CPP Codes</u>

Code 1:

https://github.com/diponsaha007/Code-Smell/blob/main/CppDSU/main.cpp

This code implements a simple Disjoint Set Union (DSU) class in C++. When we analyzed the code with SonarLint the following issues shown in the image were raised.

Link to image for better quality:
https://github.com/diponsaha007/Code-Smell/blob/main/CppDSU/CppDSU.png



```
Found 9 issues in 1 file
main.cpp (9 issues)
(83, 10) Convert this integer literal to a bool literal. 7 minutes ago
(28, 14) Replace the use of "new" with an operation that automatically manages the memory. 14 minutes ago
(23, 14) Replace the use of "new" with an operation that automatically manages the memory. 14 minutes ago
(25, 4) Add the "explicit" keyword to this constructor. 14 minutes ago
(27, 8) Do not assign data members in a constructor. Initialize member "sz" in an initialization list. 14 minutes ago
(22, 8) Do not assign data members in a constructor. Initialize member "sz" in an initialization list. 14 minutes ago
(11, 8) Do not assign data members in a constructor. Initialize member "parent" in an initialization list. 14 minutes ago
(10, 8) Do not assign data members in a constructor. Initialize member "rnk" in an initialization list. 14 minutes ago
(126, 4) Potential leak of memory pointed to by 'p.node' [+16 locations]
```

Integer literals should not be cast to bool

Code smell   Major   cpp:S5820

Even though C++ provides "true" and false" as boolean literals, it allows using integer literals in places where boolean type is expected. This can be done through implicit or explicit casting.

In contexts where boolean type is expected, integral literals should be avoided. Using boolean literals instead would make your code more readable and less error-prone.

Noncompliant Code Example

```
void f(){
  bool isX = 1; // Noncompliant
  bool isY = 0; // Noncompliant
  bool ternaryIsX = isX ? 1 : isY; // Noncompliant
  bool cCast= (bool)0; // Noncompliant
  bool cppCast= static_cast<bool>(1); // Noncompliant
  if(1) { // Noncompliant
     ...
  }
}
```

Compliant Solution

```
void f(){
 bool isX = true;
 bool isY = false;
 bool ternaryIsX = isX ? true : isY;
 bool cCast= false;
 bool cppCast= true;
 if(true) {
     ...
```

Automatic analysis is enabled

SonarLint: Convert this integer literal to a bool literal.

83:11 LF UTF-8 4 spaces C++: CppDSU | Debug main

Overview of the bad smells detected:

- **Dynamically allocated memory is not released:** At line 23 and 28, we allocate memory with new operator for node* but there is no deallocation of memory creating potential memory leak. Dynamically allocated memory should be released.

```
node= new Node[sz];
```

- **Memory is managed manually:** Data structures such as vector or list can be used for automatic memory management.

- **"explicit" is not used on single parameter constructors and conversion operators:** At line 25, there is no explicit keyword before single parameter constructor. When the expected parameter is a class with a single-argument constructor, the compiler will implicitly pass the method argument to that constructor to implicitly create an object of the correct type for the method invocation. Alternatively, if the wrong type has a conversion operator to the correct type, the operator will be called to create an object of the needed type. But using implicit conversions makes the execution flow difficult to understand. Readers may not notice that a conversion occurs, and if they do notice, it will raise a lot of questions: Is the source type able to convert to the destination type? Is the destination type able to construct an instance from the source? Is it both? And if so, which method is called by the compiler? Moreover, implicit promotions can lead to unexpected behavior, so they should be prevented by using the explicit keyword on single-argument constructors and conversion operators. Doing so will prevent the compiler from performing implicit conversions.

```
Disjoint_Set(int n)
{
   sz=n;
   node= new Node[sz];
}
```

- **Class members are not initialized in an initialization list:** Member data should be initialized in-class or in a constructor initialization list. At line 10,11,22 and 27 data members are assigned in a constructor. Member data should be initialized in-class or in a constructor initialization list.

```
Node()
{
   rnk=-1;
   parent=-1;
}
```

- **Integer literal is used instead of Bool:** At line 83, integer literal is used inside 'while' condition which should be bool. Even though C++ provides "true" and "false" as boolean literals, it allows using integer literals in places where boolean type is expected. Where boolean type is expected, integral literals should be avoided. Using boolean literals instead would make the code more readable and less error-prone

```
while(1){..}
```

- **Magic Number:** Magic numbers are numeric representations that should be replaced with a symbolic representation. At line 22, we have

```
sz = 100;
```

  Here 100 is the default size when no size is passed in the constructor. However, using numerical constants makes it harder for the reader to understand what the constant signifies. So we could do,

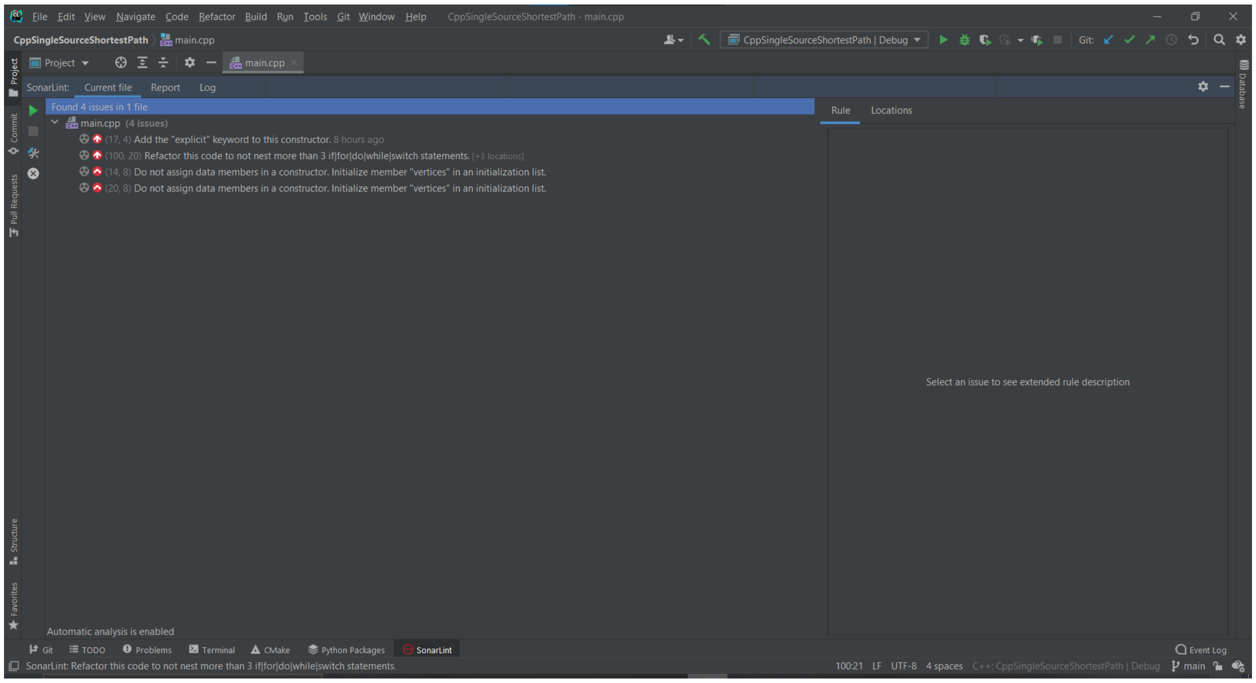
```
#define DEFAULT_SIZE 100
sz =  DEFAULT_SIZE;
```

  This makes the code more readable.

Code 2:
https://github.com/diponsaha007/Code-Smell/blob/main/CppSingleSourceShortestPath/main.cpp

This code implements a graph class which contains two single source shortest path algorithms (Dijkstra and Bellman-Ford) in C++. When we analyzed the code with SonarLint the following issues shown in the image were raised.

Link to image for better quality:
https://github.com/diponsaha007/Code-Smell/blob/main/CppSingleSourceShortestPath/CppSSSP.png


File Edit View Navigate Code Refactor Build Run Tools Git Window Help
CppSingleSourceShortestPath - main.cpp
SonarLint: Current file Report Log
Rule Locations

```
Found 4 issues in 1 file
main.cpp (4 issues)
    (17, 4) Add the "explicit" keyword to this constructor. 8 hours ago
    (100, 20) Refactor this code to not nest more than 3 if|for|do|while|switch statements. [+3 locations]
    (14, 8) Do not assign data members in a constructor. Initialize member "vertices" in an initialization list.
    (20, 8) Do not assign data members in a constructor. Initialize member "vertices" in an initialization list.
Select an issue to see extended rule description
Automatic analysis is enabled
SonarLint: Refactor this code to not nest more than 3 if|for|do|while|switch statements.
100:21 LF UTF-8 4 spaces C++: CppSingleSourceShortestPath | Debug main
```

Overview of the bad smells detected:

- **"explicit" is not used on single parameter constructors and conversion operators:** At line 17, there is no explicit keyword before single parameter constructor.

```
Graph(int n)
{
   adj.resize(n);
   vertices = n;
}
```

- **Class members are not initialized in an initialization list:** At line 14 and 20 data members are assigned in a constructor. Member data should be initialized in-class or in a constructor initialization list.

```
Graph()
{
   vertices = 0;
}
```

- **More than 3 nested if | for | while statements:** At line 94 - 107, we have 4 nested control flow statements. Nested if, for, do, while, switch and try statements is a key ingredient for making what’s known as "Spaghetti code".

```
for (int i = 1; i <= vertices - 1; i++)
{
   for (int a = 0; a < vertices; a++)
   {
       for (pair<int, int> b : adj[a])
       {
           if (dist[b.first] > dist[a] + b.second)
           {
               dist[b.first] = dist[a] + b.second;
               parent[b.first] = a;
           }
       }
   }
}
```

Such code is hard to read, refactor and therefore maintain. Though this might be an overkill here as the algorithm requires 4 nested control flow statements, however we can make it 3. The 2 nested loops at line 96 and 98 basically traverse each edge of the graph. So we can use a define for that like below:

```
#define FOR_EACH_EDGE(a,b,vertices,adj) for (int a = 0; a < vertices; a++) for
(pair<int, int> b : adj[a])
```

Then the nested control flow statements can be nested as,

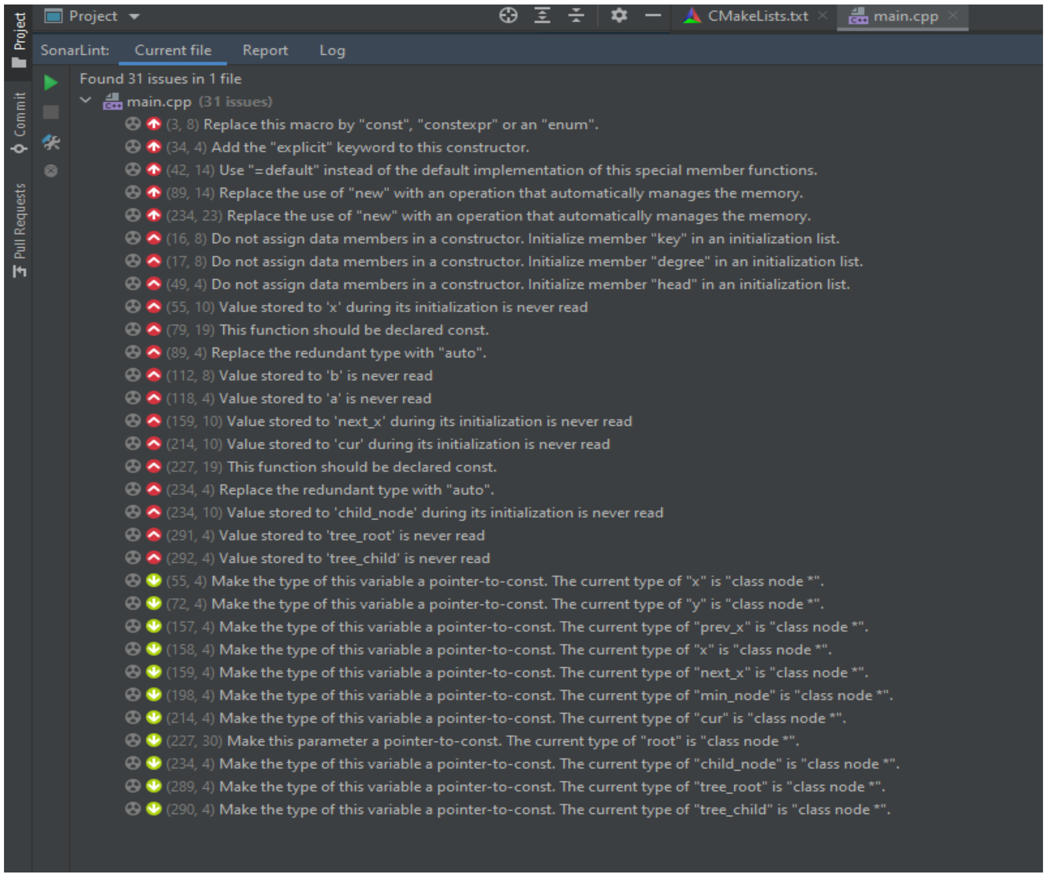
```
for (int i = 1; i <= vertices - 1; i++) //level 1
             FOR_EACH_EDGE(a,b,vertices,adj) //level 2
                    if (dist[b.first] > dist[a] + b.second) //level 3
                    {
                        dist[b.first] = dist[a] + b.second;
                      parent[b.first] = a;
                    }
```

Again, this might be an overkill from the context of the bellman ford algorithm.

Code 3:
https://github.com/diponsaha007/Code-Smell/blob/main/CPPBinomialHeap/main.cpp

This code implements a Binomial Heap in C++. When we analyzed the code with SonarLint the following issues shown in the image were raised.

Link to image for better quality:
https://github.com/diponsaha007/Code-Smell/blob/main/CPPBinomialHeap/CPPBinHeap.png



```
Found 31 issues in 1 file
main.cpp (31 issues)
(3, 8) Replace this macro by "const", "constexpr" or an "enum".
(34, 4) Add the "explicit" keyword to this constructor.
(42, 14) Use "=default" instead of the default implementation of this special member functions.
(89, 14) Replace the use of "new" with an operation that automatically manages the memory.
(234, 23) Replace the use of "new" with an operation that automatically manages the memory.
(16, 8) Do not assign data members in a constructor. Initialize member "key" in an initialization list.
(17, 8) Do not assign data members in a constructor. Initialize member "degree" in an initialization list.
(49, 4) Do not assign data members in a constructor. Initialize member "head" in an initialization list.
(55, 10) Value stored to 'x' during its initialization is never read
(79, 19) This function should be declared const.
(89, 4) Replace the redundant type with "auto".
(112, 8) Value stored to 'b' is never read
(118, 4) Value stored to 'a' is never read
(159, 10) Value stored to 'next_x' during its initialization is never read
(214, 10) Value stored to 'cur' during its initialization is never read
(227, 19) This function should be declared const.
(234, 4) Replace the redundant type with "auto".
(234, 10) Value stored to 'child_node' during its initialization is never read
(291, 4) Value stored to 'tree_root' is never read
(292, 4) Value stored to 'tree_child' is never read
(55, 4) Make the type of this variable a pointer-to-const. The current type of "x" is "class node *".
(72, 4) Make the type of this variable a pointer-to-const. The current type of "y" is "class node *".
(157, 4) Make the type of this variable a pointer-to-const. The current type of "prev_x" is "class node *".
(158, 4) Make the type of this variable a pointer-to-const. The current type of "x" is "class node *".
(159, 4) Make the type of this variable a pointer-to-const. The current type of "next_x" is "class node *".
(198, 4) Make the type of this variable a pointer-to-const. The current type of "min_node" is "class node *".
(214, 4) Make the type of this variable a pointer-to-const. The current type of "cur" is "class node *".
(227, 30) Make this parameter a pointer-to-const. The current type of "root" is "class node *".
(234, 4) Make the type of this variable a pointer-to-const. The current type of "child_node" is "class node *".
(289, 4) Make the type of this variable a pointer-to-const. The current type of "tree_root" is "class node *".
(290, 4) Make the type of this variable a pointer-to-const. The current type of "tree_child" is "class node *".
```

Overview of the bad smells detected:

- **Macros should not be used to define constants:**
  At line 3 we used macro to define "inf". A macro is a textual replacement, which means that it's not respecting the type system, it's not respecting scoping rules… There is no reason not to use a constant instead.

```
#define inf 99999
```

- **"explicit" is not used on single parameter constructors and conversion operators:**
  At line 34, there is no explicit keyword before single parameter constructor.

```
BinomialHeap(node* x);
```

- **Special member function should not be defined unless a non standard behavior is required:**

  At line 42, the default constructor was defined. But all special member functions (default constructor, copy and move constructors, copy and move assignment operators, destructor) can be automatically generated by the compiler if we don't prevent it (for many classes, it is good practice to organize your code so that you can use these default versions.

```
BinomialHeap::BinomialHeap()
{
   head = NULL;
}
```

- **Memory should not be managed manually**

  At line 89, 234.

  If we manage memory manually, it's our responsibility to delete all memory created with new, and to make sure it's deleted once and only once. Ensuring this is done is error-prone, especially when your function can have early exit points.

```
node* x = new node(k);
```

```
node* child_node = new node();
```

- **Member data should be initialized in-class or in a constructor initialization list**

At lines 16, 17, 49.

There are three ways to initialize a non-static data member in a class:

- With an in-class initializer
- In the initialization list of a constructor
- In the constructor body

We should use those methods in that order of preference. When applicable, in-class initializers are best, because they apply automatically to all constructors of the class (except for default copy/move constructors and constructors where an explicit initialization for this member is provided). But they can only be used for initialization with constant values.

If our member value depends on a parameter, we can initialize it in the constructor's initialization list. If the initialization is complex, you can define a function to compute the value, and use this function in the initializer list.

```
node(int val = -1, int dgr = 0)
{
   key = val;
   degree = dgr;
   child = NULL;
   sibling = NULL;
   parent = NULL;
}
```

- **Unused assignments should be removed**

  At line 55, 12, 18, 59, 214, 234, 291, 292.

  A dead store happens when a local variable is assigned a value that is not read by any subsequent instruction.

```
node* x = head;
```

- **"auto" should be used to avoid repetition of types**

  At line 89, 234.

When used as a type specifier in a declaration, auto allows the compiler to deduce the type of a variable based on the type of the initialization expression.

```
node* x = new node(k);
```

```
node* child_node = new node();
```

- **Member functions that don't mutate their objects should be declared "const"**

  At line 79, 227.

  No member function can be invoked on a const-qualified object unless the member function is declared "const".

  Qualifying member functions that don’t mutate their object with the "const" qualifier makes your interface easier to understand; you can deduce without diving into implementation if a member function is going to mutate its object

```
void BinomialHeap::binomial_link(node*y, node*z)
{
   y->parent = z;
   y->sibling = z->child;
   z->child = y;
   z->degree = z->degree + 1;
}
```

- **Pointer and reference local variables should be "const" if the corresponding object is not modified**

  At lines 55, 72, 157, 158, 159, 198, 214, 227, 234, 289, 290.

  This rule leads to greater precision in the definition of local variables by making the developer's intention about modifying the variable explicit.

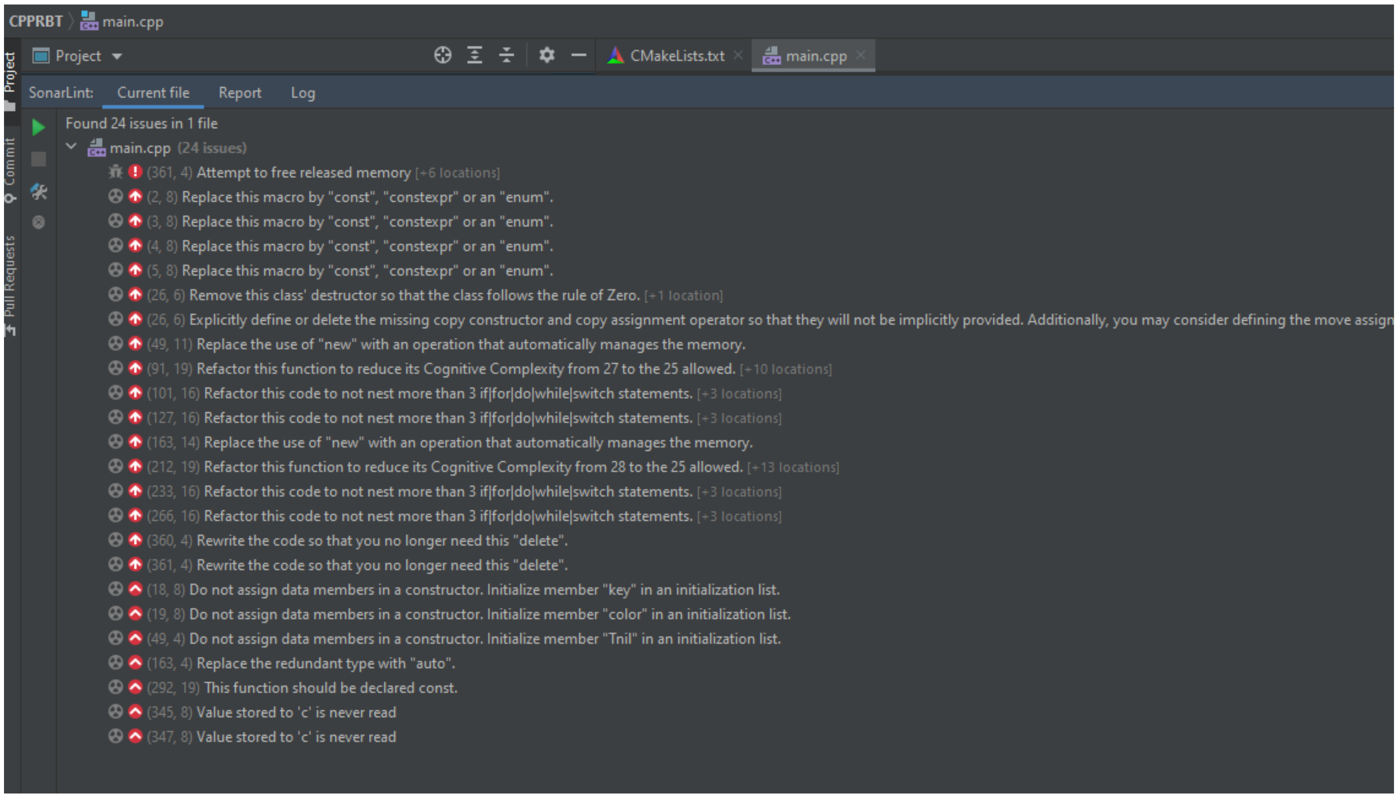
```
node* x = head;
```

Code 4:

https://github.com/diponsaha007/Code-Smell/blob/main/CPPRBT/main.cpp

This code implements a Red Black Tree in C++. When we analyzed the code with SonarLint the following issues shown in the image were raised.

Link to image for better quality:
https://github.com/diponsaha007/Code-Smell/blob/main/CPPRBT/CPPRBT.png



```
Found 24 issues in 1 file
main.cpp (24 issues)
(361, 4) Attempt to free released memory [+6 locations]
(2, 8) Replace this macro by "const", "constexpr" or an "enum".
(3, 8) Replace this macro by "const", "constexpr" or an "enum".
(4, 8) Replace this macro by "const", "constexpr" or an "enum".
(5, 8) Replace this macro by "const", "constexpr" or an "enum".
(26, 6) Remove this class' destructor so that the class follows the rule of Zero. [+1 location]
(26, 6) Explicitly define or delete the missing copy constructor and copy assignment operator so that they will not be implicitly provided. Additionally, you may consider defining the move assign
(49, 11) Replace the use of "new" with an operation that automatically manages the memory.
(91, 19) Refactor this function to reduce its Cognitive Complexity from 27 to the 25 allowed. [+10 locations]
(101, 16) Refactor this code to not nest more than 3 if|for|do|while|switch statements. [+3 locations]
(127, 16) Refactor this code to not nest more than 3 if|for|do|while|switch statements. [+3 locations]
(163, 14) Replace the use of "new" with an operation that automatically manages the memory.
(212, 19) Refactor this function to reduce its Cognitive Complexity from 28 to the 25 allowed. [+13 locations]
(233, 16) Refactor this code to not nest more than 3 if|for|do|while|switch statements. [+3 locations]
(266, 16) Refactor this code to not nest more than 3 if|for|do|while|switch statements. [+3 locations]
(360, 4) Rewrite the code so that you no longer need this "delete".
(361, 4) Rewrite the code so that you no longer need this "delete".
(18, 8) Do not assign data members in a constructor. Initialize member "key" in an initialization list.
(19, 8) Do not assign data members in a constructor. Initialize member "color" in an initialization list.
(49, 4) Do not assign data members in a constructor. Initialize member "Tnil" in an initialization list.
(163, 4) Replace the redundant type with "auto".
(292, 19) This function should be declared const.
(345, 8) Value stored to 'c' is never read
(347, 8) Value stored to 'c' is never read
```

Overview of the bad smells detected:

- **Macros should not be used to define constants**
  At lines 2, 3, 4, 5.

```
#define RED 1
#define BLACK 2
#define DBLACK 3
#define nil -1
```

- **Memory locations should not be released more than once**
  At line 361.

Using free(...) or delete releases the reservation on a memory location, making it immediately available for another purpose. So releasing the same memory location twice can lead to corrupting the program's memory.

```
delete Tnil;
```

- **The "Rule-of-Zero" should be followed**
  At line 26.

  Classes that avoid directly handling resources don't need to define any of the special member functions required to properly handle resources: destructor, copy constructor, move constructor, copy-assignment operator, move-assignment operator. That's because the versions of those functions provided by the compiler do the right thing automatically, which is especially useful because writing these functions correctly is typically tricky and error-prone.
  Omitting all of these functions from a class is known as the Rule of Zero because no special function should be defined.

```
class RedBlackTree
{
   node *root;
   node *Tnil;
   void left_rotate(node *x);
   void right_rotate(node *y);
   void insert_fixup(node *z);
   void preorder(node *ptr);
   node* tree_minimum(node *x);
   void rb_transplant(node* u, node* v);
   void delete_fixup(node* x);
   node* tree_search(node*x, int k);
public:
   RedBlackTree();
   void rbinsert(int k);
   void search_value(int k);
   void rbdelete(int k);
   ~RedBlackTree();

};
```

- **When the "Rule-of-Zero" is not applicable, the "Rule-of-Five" should be followed**
  At line 26.

  In C++, we should not directly manipulate resources (a database transaction, a network connection, a mutex lock), but encapsulate them in RAII wrapper classes that will allow us to manipulate them safely. When defining one of those wrapper classes, we cannot rely on the compiler-generated special member functions to manage the class' resources for you. We must define those functions yourself to make sure the class' resources are properly copied, moved, and destroyed.

```
class RedBlackTree
{
   node *root;
   node *Tnil;
   void left_rotate(node *x);
   void right_rotate(node *y);
   void insert_fixup(node *z);
   void preorder(node *ptr);
   node* tree_minimum(node *x);
   void rb_transplant(node* u, node* v);
   void delete_fixup(node* x);
   node* tree_search(node*x, int k);
public:
   RedBlackTree();
   void rbinsert(int k);
   void search_value(int k);
   void rbdelete(int k);
   ~RedBlackTree();

};
```

- **Memory should not be managed manually**
  At lines 49, 163, 360, 361.

  ```
  Tnil = new node(-1,BLACK);
  ```

- **The cognitive Complexity of functions should not be too high**
  At line 91, 212.

Cognitive Complexity is a measure of how hard the control flow of a function is to understand. Functions with high Cognitive Complexity will be difficult to maintain.

```
void RedBlackTree::insert_fixup(node *z)
{
   if(z->parent!=NULL)
   {
       while (z->parent->color == RED)
       {
           if (z->parent == z->parent->parent->left)
           {

               node *y = z->parent->parent->right;
               if (y->color==RED)
               {
                   z->parent->color = BLACK;
                   y->color = BLACK;
                   z->parent->parent->color=RED;
                   z=z->parent->parent;
               }

               else
               {
                   if(z==z->parent->right)
                   {
                       z = z->parent;
                       left_rotate(z);
                   }
                   z->parent->color = BLACK;
                   z->parent->parent->color = RED;
                   right_rotate(z->parent->parent);
               }

           }

           else
           {

               node *y = z->parent->parent->left;
               if (y->color==RED)
               {
                   z->parent->color = BLACK;
                   y->color = BLACK;
                   z->parent->parent->color=RED;
                   z=z->parent->parent;
```

```
                }
                else
                {
                    if(z==z->parent->left)
                    {
                        z = z->parent;
                        right_rotate(z);

                    }
                    z->parent->color = BLACK;

                    z->parent->parent->color = RED;

                    left_rotate(z->parent->parent);

                }
            }
        }

    }
    root->color=BLACK;
}
```

- **Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply**
  At lines 101, 127, 133, 166.

  Nested if, for, do, while, switch and try statements is a key ingredient for making what's known as "Spaghetti code".
  Such code is hard to read, refactor and therefore maintain.

```
void RedBlackTree::insert_fixup(node *z)
{
   if(z->parent!=NULL)
   {
       while (z->parent->color == RED)
       {
           if (z->parent == z->parent->parent->left)
           {
               if (y->color==RED)
               {
               }

               else
               {
                   if(z==z->parent->right)
                   {

                   }
               }

           }

           else
           {
               if (y->color==RED)
               {
               }
               else
               {
                   if(z==z->parent->left)
                   {
                   }

               }
           }
       }

   }
}
```

- **Member data should be initialized in-class or in a constructor initialization list**
  At lines 18, 19, 49.

```
node(int val = -1, int clr = RED)
{
   key = val;
   color = clr;
   left = NULL;
   right = NULL;
   parent = NULL;
}
```

- **"auto" should be used to avoid repetition of types**
  At line 163.

```
node *z = new node(k);
```

- **Member functions that don't mutate their objects should be declared "const"**
  At line 292.

```
void RedBlackTree::search_value(int k)
{
   if(tree_search(root, k)==Tnil) cout<<"False"<<endl;
   else cout<<"True"<<endl;
}
```

- **Unused assignments should be removed**
  At lines 345, 347.

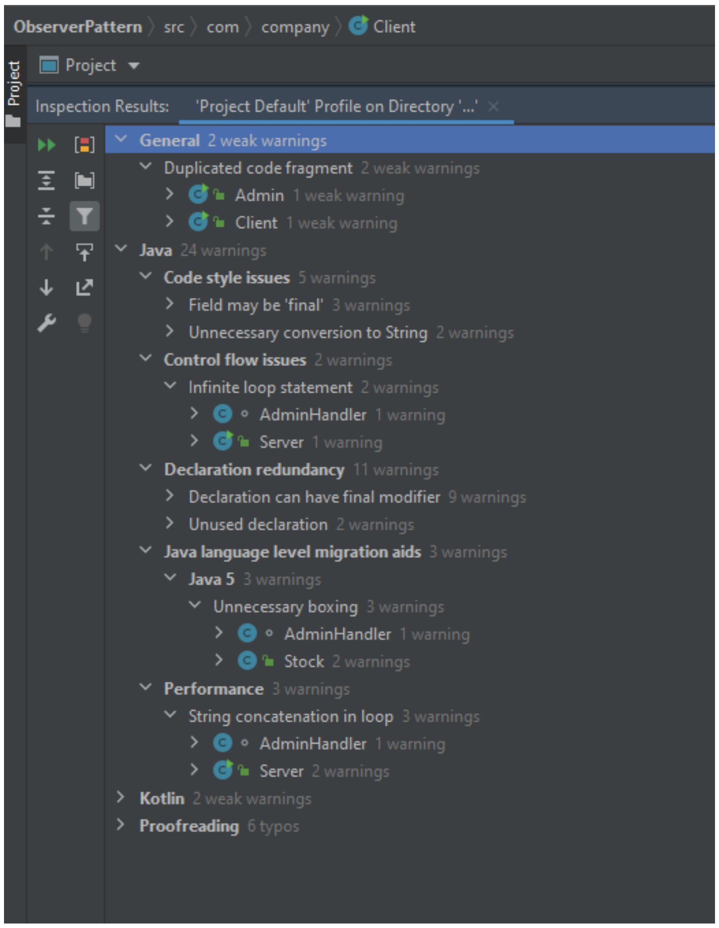
```
if(ptr->color==BLACK)
   c='b';
```

Java Codes:

Code 1:
https://github.com/diponsaha007/Code-Smell/tree/main/JavaObserverPattern/src

This code basically implements a demo Observer Pattern in Java. When we analyzed the code with the default analyzer the following issues shown in the image arose.

Link to image for better quality:
https://github.com/diponsaha007/Code-Smell/blob/main/JavaObserverPattern/JAVAObserverPattern.png

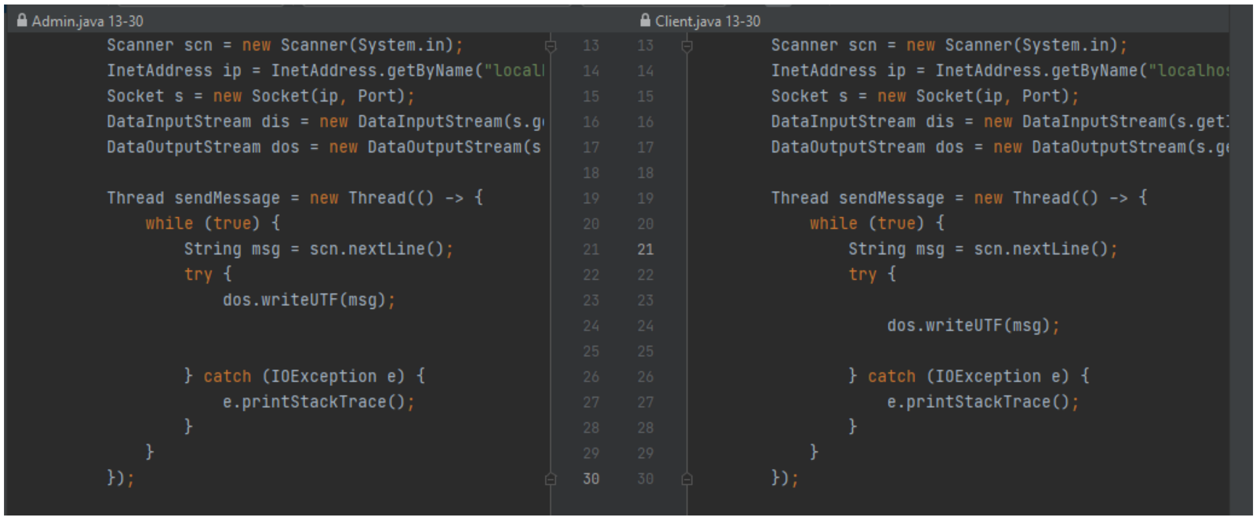

ObserverPattern 〉 src 〉 com 〉 company 〉 Client
Project
Inspection Results:
'Project Default' Profile on Directory '...'

```
General 2 weak warnings
  Duplicated code fragment 2 weak warnings
    Admin 1 weak warning
    Client 1 weak warning
Java 24 warnings
  Code style issues 5 warnings
    Field may be 'final' 3 warnings
    Unnecessary conversion to String 2 warnings
  Control flow issues 2 warnings
    Infinite loop statement 2 warnings
      AdminHandler 1 warning
      Server 1 warning
  Declaration redundancy 11 warnings
    Declaration can have final modifier 9 warnings
    Unused declaration 2 warnings
  Java language level migration aids 3 warnings
    Java 5 3 warnings
      Unnecessary boxing 3 warnings
        AdminHandler 1 warning
        Stock 2 warnings
  Performance 3 warnings
    String concatenation in loop 3 warnings
      AdminHandler 1 warning
      Server 2 warnings
Kotlin 2 weak warnings
Proofreading 6 typos
```

Overview of the bad smells detected:

- **Duplicate code fragment**

  Duplicated blocks of code from the selected scope: the same file, same module, dependent modules, or the entire project.



Admin.java 13-30

```
        Scanner scn = new Scanner(System.in);
        InetAddress ip = InetAddress.getByName("locall
        Socket s = new Socket(ip, Port);
        DataInputStream dis = new DataInputStream(s.g
        DataOutputStream dos = new DataOutputStream(s

        Thread sendMessage = new Thread(() -> {
            while (true) {
                String msg = scn.nextLine();
                try {
                    dos.writeUTF(msg);


                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
```

Client.java 13-30

```
        Scanner scn = new Scanner(System.in);
        InetAddress ip = InetAddress.getByName("localhos
        Socket s = new Socket(ip, Port);
        DataInputStream dis = new DataInputStream(s.getI
        DataOutputStream dos = new DataOutputStream(s.ge

        Thread sendMessage = new Thread(() -> {
            while (true) {
                String msg = scn.nextLine();
                try {

                    dos.writeUTF(msg);

                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        });
```

- **Field may be ‘final’**

  Any fields which may safely be made final. A static field may be final if it is initialized in its declaration or in one static class initializer, but not both. A non-static field may be final if it is initialized in its declaration or in one non-static class initializer or in all constructors.

```
private int id;
```

- **Unnecessary conversion to string**

  Any calls to static methods like String.valueOf() or Integer.toString() used in string concatenations and as arguments to the print() and println() methods of java.io.PrintWriter and java.io.PrintStream, the append() method of java.lang.StringBuilder and java.lang.StringBuffer or the trace(), debug(), info(), warn() and error() methods of org.slf4j.Logger. In these cases the conversion to string will be handled by the underlying library methods and an explicit call to String.valueOf() is not needed.

```
String UpdateMessege = "You subscribed stock "+ StockName +". Its current price: 
"+Float.toString(StockPrice)+", number of available stocks: 
"+Integer.toString(StockCount);
```

- **Infinite loop statement**

  for, while, or do statements which can only exit by throwing an exception. While such statements may be correct, they are often a symptom of coding errors.

```
while (true){...}
```

- **Declaration can have final modifiers**

  Fields, methods or classes, found in the specified inspection scope, that may have a final modifier added to their declarations.

```
Socket s;
```

- **Unused declaration**

  Classes, methods or fields in the specified inspection scope that are not used or not reachable from entry points. It also reports parameters that are not used by their methods and all method implementations/overriders and local variables that are declared but not used. Some unused members might not be reported during in-editor highlighting. Due to performance reasons, a non-private member is checked only when its name rarely occurs in the project.

```
Socket s;
```

- **Unnecessary boxing**

  Explicit boxing, i.e. wrapping of primitive values in objects. Explicit manual boxing is unnecessary under Java 5 and newer, and can be safely removed.

```
StockCount = Integer.valueOf(st.nextToken());
StockPrice = Float.valueOf(st.nextToken());
```

- **String concatenation in loop**

  String concatenation in loops. As every String concatenation copies the whole String, usually it is preferable to replace it with explicit calls to StringBuilder.append() or StringBuffer.append().

```
for(StockInterface si: Server.StockList)
{
   updated+=si.getName()+"\t\t\t\t"+ si.getCount() +"\t\t\t\t"+ si.getPrice()
+"\n";
}
```

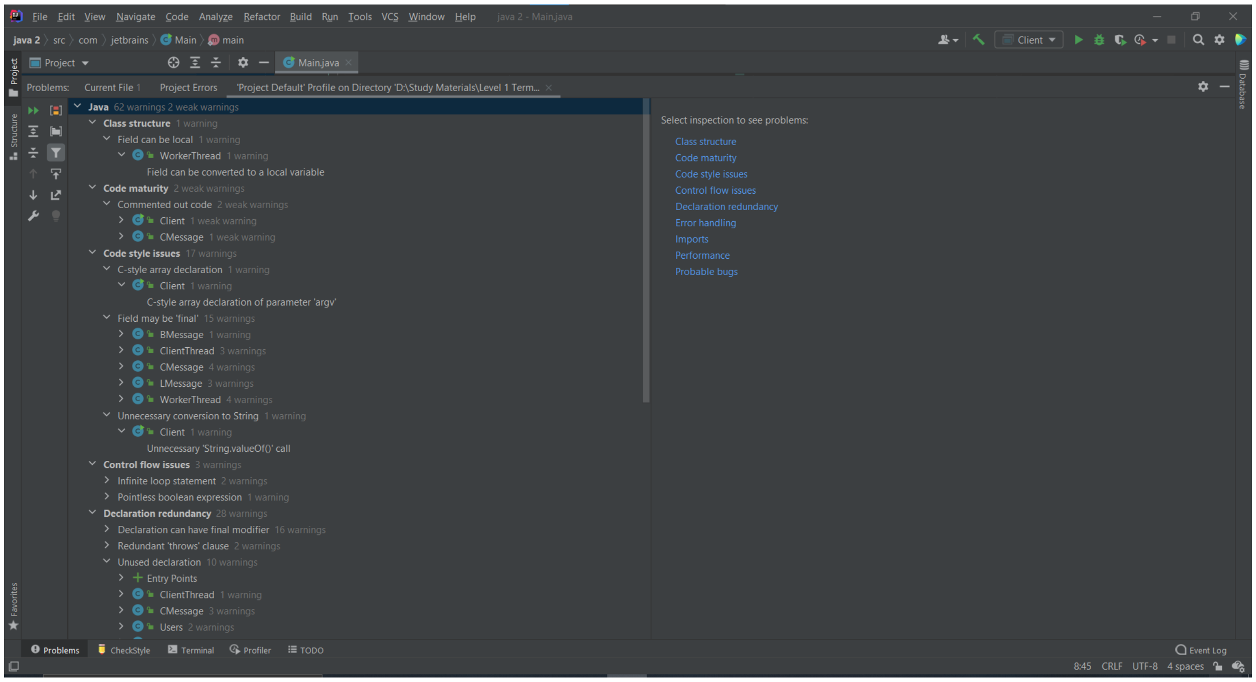
Code 2:
https://github.com/diponsaha007/Code-Smell/tree/main/JavaMessenger/src/com/jetbrains

This code implements a simple messenger using socket programming in Java where users can send messages among themselves. When we analyzed the code with CheckStyle and the default analyzer the following issues shown in the image arose.

Link to image for better quality:

Default analyzer:
https://github.com/diponsaha007/Code-Smell/blob/main/JavaMessenger/Smells%20with%20intellij.png

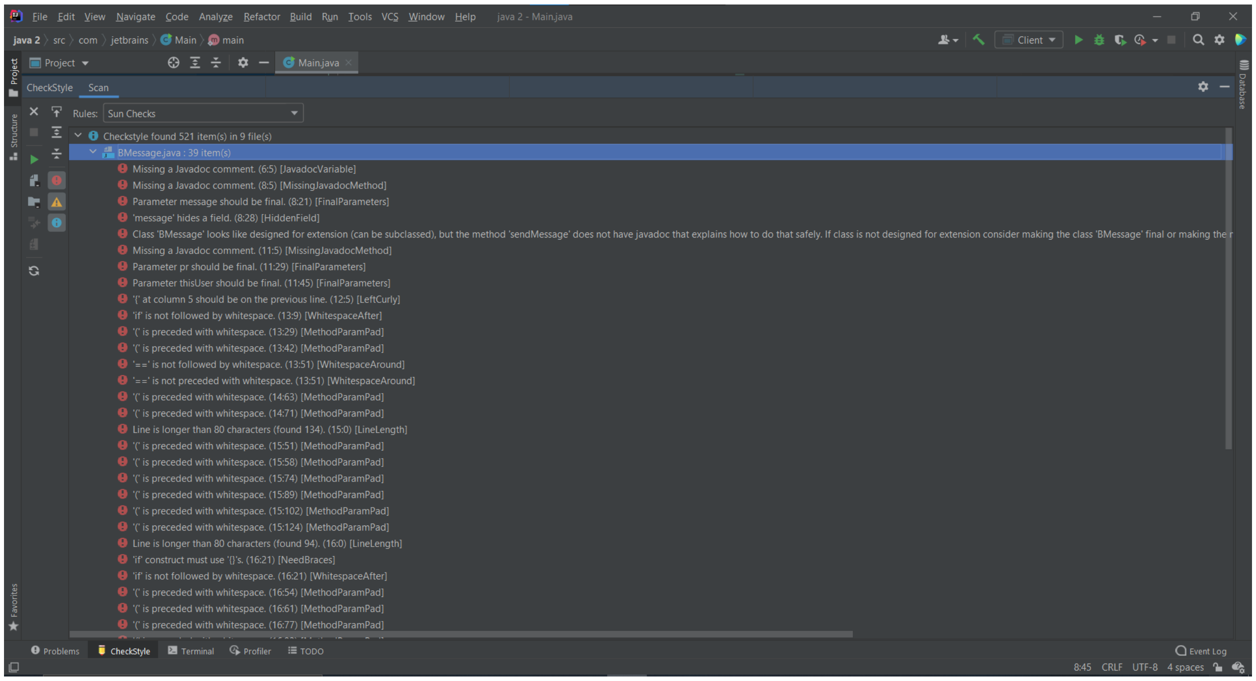
CheckStyle:
https://github.com/diponsaha007/Code-Smell/blob/main/JavaMessenger/Smells%20with%20checkStyle.png


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
java 2 - Main.java
java 2 › src › com › jetbrains › Main › main
Project
Main.java
Client
Problems: Current File 1 Project Errors 'Project Default' Profile on Directory 'D:\Study Materials\Level 1 Term...
Problems CheckStyle Terminal Profiler TODO
Event Log
8:45 CRLF UTF-8 4 spaces

```
Java 62 warnings 2 weak warnings
  Class structure 1 warning
    Field can be local 1 warning
      WorkerThread 1 warning
        Field can be converted to a local variable
  Code maturity 2 weak warnings
    Commented out code 2 weak warnings
      Client 1 weak warning
      CMessage 1 weak warning
  Code style issues 17 warnings
    C-style array declaration 1 warning
      Client 1 warning
        C-style array declaration of parameter 'argv'
    Field may be 'final' 15 warnings
      BMessage 1 warning
      ClientThread 3 warnings
      CMessage 4 warnings
      LMessage 3 warnings
      WorkerThread 4 warnings
    Unnecessary conversion to String 1 warning
      Client 1 warning
        Unnecessary 'String.valueOf()' call
  Control flow issues 3 warnings
    Infinite loop statement 2 warnings
    Pointless boolean expression 1 warning
  Declaration redundancy 28 warnings
    Declaration can have final modifier 16 warnings
    Redundant 'throws' clause 2 warnings
    Unused declaration 10 warnings
      Entry Points
      ClientThread 1 warning
      CMessage 3 warnings
      Users 2 warnings
```

```
Select inspection to see problems:
  Class structure
  Code maturity
  Code style issues
  Control flow issues
  Declaration redundancy
  Error handling
  Imports
  Performance
  Probable bugs
```

Fig (a) : Smells with default intellij analyzer


```
Checkstyle found 521 item(s) in 9 file(s)
  BMessage.java : 39 item(s)
    Missing a Javadoc comment. (6:5) [JavadocVariable]
    Missing a Javadoc comment. (8:5) [MissingJavadocMethod]
    Parameter message should be final. (8:21) [FinalParameters]
    'message' hides a field. (8:28) [HiddenField]
    Class 'BMessage' looks like designed for extension (can be subclassed), but the method 'sendMessage' does not have javadoc that explains how to do that safely. If class is not designed for extension consider making the class 'BMessage' final or making the r
    Missing a Javadoc comment. (11:5) [MissingJavadocMethod]
    Parameter pr should be final. (11:29) [FinalParameters]
    Parameter thisUser should be final. (11:45) [FinalParameters]
    '{' at column 5 should be on the previous line. (12:5) [LeftCurly]
    'if' is not followed by whitespace. (13:9) [WhitespaceAfter]
    '(' is preceded with whitespace. (13:29) [MethodParamPad]
    '(' is preceded with whitespace. (13:42) [MethodParamPad]
    '==' is not followed by whitespace. (13:51) [WhitespaceAround]
    '==' is not preceded with whitespace. (13:51) [WhitespaceAround]
    '(' is preceded with whitespace. (14:63) [MethodParamPad]
    '(' is preceded with whitespace. (14:71) [MethodParamPad]
    Line is longer than 80 characters (found 134). (15:0) [LineLength]
    '(' is preceded with whitespace. (15:51) [MethodParamPad]
    '(' is preceded with whitespace. (15:58) [MethodParamPad]
    '(' is preceded with whitespace. (15:74) [MethodParamPad]
    '(' is preceded with whitespace. (15:89) [MethodParamPad]
    '(' is preceded with whitespace. (15:102) [MethodParamPad]
    '(' is preceded with whitespace. (15:124) [MethodParamPad]
    Line is longer than 80 characters (found 94). (16:0) [LineLength]
    'if' construct must use '{}'s. (16:21) [NeedBraces]
    'if' is not followed by whitespace. (16:21) [WhitespaceAfter]
    '(' is preceded with whitespace. (16:54) [MethodParamPad]
    '(' is preceded with whitespace. (16:61) [MethodParamPad]
    '(' is preceded with whitespace. (16:77) [MethodParamPad]
```

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help java 2 - Main.java
java 2 src com jetbrains Main main
Project
Main.java
CheckStyle
Scan
Rules: Sun Checks
Problems
CheckStyle
Terminal
Profiler
TODO
Event Log
8:45 CRLF UTF-8 4 spaces

Fig (b) : Smells with CheckStyle-IDEA

Overview of the bad smells detected:

- **Field can be local :** The ‘id’ variable in the ‘WorkerThread’ class can be made local in the constructor.

```
public class WorkerThread implements Runnable {
   private int id ;
public WorkerThread(Socket s, int id)throws Exception {
   this.id = id;
}
```

- **C-style array declaration :** Though C-style and Java-style arrays are semantically identical. The “int array[]” syntax was only added to help C programmers get used to java. “int[] array” is much preferable, and less confusing.

```
String argv[]
```

- **Field may be final :** There are many fields in the code that can be made final. A final field cannot have its value changed. A final field must have an initial value assigned to it, and once set, the value cannot be changed again. Making a field final can make the code less error-prone.

```
public class BMessage {
   private String message;
}
```

```
public class ClientThread implements Runnable {
   private PrintWriter outToServer;
   private BufferedReader inFromServer;
   private Socket clientSocket;
}
```

- **Infinite loop statements :** There are infinite loops in “Client” and “Main” class which can’t be completed without throwing an exception.

- **Redundant ‘throws’ clause :** The ‘sendFile’ function inside the ‘CMessage’ class and in the ‘WorkerThread’ constructor the declared exception is never thrown. So redundant ‘throws’ should be removed.

```
public void sendFile(Users thisUser,String string) throws Exception{...}
```

```
public WorkerThread(Socket s, int id)throws Exception{...}
```

- **Unused declaration :** There are some unused declarations in the code that are never used. They should be removed.

```
String string = inFromServer.readLine ();
```

```
private String filename;
```

- **Empty ‘catch’ block :** There are some empty ‘catch’ blocks inside the ‘run’ method of ‘ClientThread’ and ‘WorkerThread’ class. The ‘catch’ parameter should be renamed to ‘ignored’ in case of nothing to be done.

```
catch (Exception e) {}
```

- **Unused imports :** Unused imports should be removed from the code.

```
import java.awt.*;
import java.net.ServerSocket;
import java.nio.file.Files;
```

- **String Concatenation inside loop :** Inside the ‘action’ method of ‘SMessage’ class ‘String s’ is concatenated inside a loop. Variable ‘s’ should be converted into ‘StringBuilder’ as java strings are immutable.

```
s= s+ WorkerThread.getWorkerThreads ().get (i).getThisUser ().getUsername ()+ " ";
```

- **NullPointerException :** Method ‘read’ , ‘write’ and ‘close’ inside ‘ClientThread’ class may produce NullPointerException. So it must be handled with a ‘try catch’ block.

```
while ((count = in.read(bytes)) > 0) {
   out.write(bytes, 0, count);
}

out.close();
```

- **Result of method call ignored :** Result of ‘BufferedInputStream.read()’ is ignored. This is a bad coding style.

```
bis.read (contents, 0, size);
```

- **Long lines :** There are some lines longer than 80 characters. Long lines are hard to follow. So writing long lines should be avoided or these lines should be splitted.