TP2

Semester : Fall 2021

Team Romeo

Chun-An Bau - 2165883

Guilhem Dubois - 1897198

Matyas Jesina - 2165646

Group : 01

LOG8430E - Software Architecture and Advanced Design

Polytechnique Montréal

November 8th 2021

Table of Contents

Measuring the design quality of AmazeFile

Most of the following metrics have been found using the MetricsReloaded plugin on Android Studio. The TCC metric was found using the MetricsTree plugin. The LOD metric had to be calculated from four other metrics that are available through MetricsReloaded: percentage of commented methods, percentage of commented classes, number of methods and number of classes. The descriptions come from the slides shown in class.

Metric Name	Description	Value	Level
LOC	Numbers of return characters	119.95	Class average
		3	Class DataUtilsHolder
		1575	Class MainActivity
		15.53	Method average
		310	Method onBindViewHolder (RecyclerAdapter)
SIZE2	Number of attributes and methods of a class	115.46	Class average
		13	Class DataUtilsHolder
		1023	Class WarnableTextInputLayout
NOM	Number of methods declared in a class	11.34	Class average
		67	Class MainActivity
		37	Class RecyclerAdapter
CC	Number of independent paths	3.23	Method average
		87	Method onBindViewHolder (RecyclerAdapter)
WMC	Sum of the cyclomatic complexities of methods in a class	19.01	Class average
		168	Class RecyclerAdapter
RFC	Number of public methods of a class and all methods accessed directly by these public methods	29.16	Class average
		437	Class MainActivity
DIT	Maximal inheritance length from a derived class to a basic class	2.23	Class average
		12	Class MainActivity
NOC	Number of classes directly	0.29	Class average

	derived from a basic class	15	Class SFtpClientTemplate
CBO	Numbers of classes coupled with another class	10.65	Class average
		104	Class MainActivity
		57	Class HybridFile
Ca	Number of external classes that are coupled with internal classes in relation to a package	168.18	Package average
		987	Package filesystem
		2055	Package utils
LCOM	Degree of cohesiveness of a class according to the Hitz and Montazeri definition	1.72	Class average
		20	Class FileUtils
		7	DecryptService
TCC	Ratio of directly connected methods over all possible connections	0.2996	Class AppsListFragment
		1.0	Class CloudHandler
LOD	Percentage of methods and classes that are not commented	0.2908	Project average
		0.1897	Methods only
		0.7432	Classes only

As we can see with the metrics LOC, SIZE2 and NOM, the project AmazeFile has a pretty big size. Most methods and classes are not overly big, but some of them are, which can make the program harder to understand and to modify.

The metrics CC, WMC and RFC reflect on the complexity of AmazeFile. On average, the methods are a bit too complex, but it's mainly the classes that are. This could be caused by the size as explained above, where the methods and classes might be too big. The complexity could be reduced by extracting the methods and classes into multiple smaller components.

The inheritance seems well designed in this application, which helps the testability among other qualities. The maximum length from a derived class to a basic class is 12, which isn't bad considering it's for the MainActivity, for which it makes sense to have multiple levels of inheritance. A lot of classes are derived from a base class, which helps to respect the Open-Closed Principle (OCP) since polymorphism can be used.

The CBO and Ca metrics reflect on the coupling between classes, while LCOM and TCC are about the cohesion in a class. There is a high coupling from external classes using some

from the utils package, which could be desirable in this case as they are tools that are reused. However, there is also a high coupling between quite a few of other classes, which might be less desirable since making a change in one class might require considering that change in other classes. Those instances of high coupling go against the modifiability of the application. In general, the high cohesion and low coupling GRASP principles seem to be respected, meaning the responsibilities are often well separated.

Finally, the classes in the project are well documented, but a lot of methods are not. The long term maintainability would benefit from having more comments on methods since that would make the methods easier to understand, and to learn for new developers.

The design properties are presented, and then are used to calculate the quality attributes. The metrics have been calculated with the same tools as described in the first section and with the software Understand.

Design Property	Design Metric	Description	Value	Level
Design Size	DSC	Total number of classes in the design	422	Project
Hierarchies	NOH	Number of class hierarchies in the design	50	Project
Abstraction	ANA	Average number of classes from which a class inherits information	1.52	Project
Encapsulation	DAM	Ratio of the number of private attributes over the total number of attributes	65.44%	Project
Coupling	DCC	Number of classes that a class is related to	10.65	Class average
Cohesion	CAM	Cohesion among methods of class	0.914	Project
Composition	MOA	Number of data declaration whose types are user defined classes	6.40	Class average

Inheritance	MFA	Ratio of methods inherited by a class over all methods accessible by member methods of the class	13.79%	Project
Polymorphism	NOP	Number of methods that can exhibit polymorphic behavior	14.90	Class average
Messaging	CIS	Number of public methods	8.18	Class average
Complexity	NOM	Number of methods	11.34	Class average

Quality criteria name	Value
Functionality	109.03
Reusability	212.66
Understandability	-151.42
Flexibility	8.15
Efficiency	4.72
Extensibility	2.95

The values obtained in the section before are not specific values but rather to give a general idea of the presence of different qualities in the program. We can see an emphasis on functionality and reusability, meaning the code is mostly clean and well written. However, there's not a lot of documentation which reflects the low understandability value. Flexibility, efficiency and extensibility are all positive but not by much. They could probably be improved although the values don't seem to indicate they're a big problem in the application, especially compared to understandability.

Here are the formulas that define the quality attribute value shown above:

Quality Attribute	Index Computation Equation
Reusability	-0.25 * Coupling + 0.25 * Cohesion + 0.5 * Messaging + 0.5 * Design Size
Flexibility	0.25 * Encapsulation - 0.25 * Coupling + 0.5 * Composition + 0.5 * Polymorphism
Understandability	-0.33 * Abstraction + 0.33 * Encapsulation - 0.33 * Coupling + 0.33 * Cohesion - 0.33 * Polymorphism - 0.33 * Complexity - 0.33 * Design Size
Functionality	0.12 * Cohesion + 0.22 * Polymorphism + 0.22 Messaging + 0.22 * Design Size + 0.22 * Hierarchies
Extendibility	0.5 * Abstraction - 0.5 * Coupling + 0.5 * Inheritance + 0.5 * Polymorphism
Effectiveness	0.2 * Abstraction + 0.2 * Encapsulation + 0.2 * Composition + 0.2 * Inheritance + 0.2 * Polymorphism

We can see that reusability is mostly dependent on messaging and design size. The metrics for those are CIS (number of public methods) and DSC (number of classes). The more the code is divided into separate classes and methods, the easier it is to reuse them. It's also affected by the metrics DCC and CAM which represent the coupling between classes and cohesion inside a class, since more cohesion and less coupling means the methods and classes are better isolated and therefore easier to reuse. Out of the metrics mentioned in question 1, the metrics CBO (numbers of classes coupled with another class), Ca (number of external classes that are coupled with internal classes in relation to a package), LCOM (degree of cohesiveness of a class) and TCC (ratio of directly connected methods over all possible connections) also affect the reusability since the first two are related to coupling and the last two, to cohesion. For example, the class HybridFile has 57 classes coupled with it (CBO) and its package, filesystem, has a high value of 987 external classes that are using its internal classes (Ca). That means there's a lot of coupling related to that class and package, making it harder to reuse its components. Reducing coupling would reduce the CBO and Ca metrics, and at the same time increase reusability.

For the flexibility attribute, encapsulation, composition and polymorphism are positive attributes while coupling reduces flexibility. Therefore the metrics DAM, MOA, NOP and DCC affect the flexibility. However, DIT (Maximal inheritance length from a derived class to a basic class) and NOC (Number of classes directly derived from a basic class) also affect the flexibility since they're related to polymorphism. For example, the SFtpClientTemplate class has a high NOC value with 15 derived uses, overwriting the *execute()* command. The action is then executed through SshClientUtils, which means that the template can be reused for different actions while letting the SshClientUtils class do repetitive work such as connecting

to a client URL, etc. Therefore, the high NOC value helps flexibility by allowing the same template to have different implementations.

For understandability, a lot of factors come into play, encapsulation and cohesion being the important ones. In this case, the RFC metric (number of public methods in a class and methods that are called by those methods) reflects on how good the encapsulation is. A high RFC value means that a lot of methods are public and call other methods, making the class harder to debug. For example, the MainActivity class has a huge value of 437, and is also a really hard class to understand, with almost 2000 lines and around 50 public methods. Some of those methods are also really complex like getStorageDirectories(), which has 40 other methods called in it. Making only the necessary methods public and keeping them simple would reduce the value of RFC and increase the understandability.

Anomalies in AmazeFile

Anomaly 1: Long Method

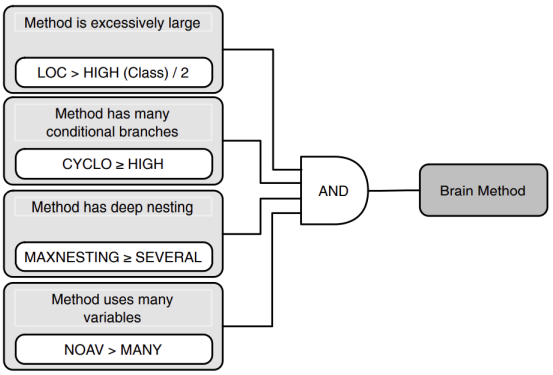
In the class Operations.java, there's a method called mkdir which creates a folder asynchronously based on which platform it is (DropBox, OneDrive, etc.). The values of the thresholds are from the book "Object-Oriented Metrics in Practice", where MAXNESTING and NOAV were given, while LOC and CC were calculated from the formula given in the book.

Metric	Value	Threshold
LOC	104	65 (130/2)
CC	26	24.96 (0.24*LOC)
MAXNESTING	6	5
NOAV	12	8

The method has 104 lines of codes, a cyclomatic complexity of 26, a nested depth of 6 and 12 variables accessed. The method clearly does too many things and doesn't seem to be using polymorphism correctly, as there are conditions for each scenario, but they are really similar:

```
} else if (file.isDropBoxFile()) {
   CloudStorage cloudStorageDropbox = dataUtils.getAccount(OpenMode.DROPBOX);
   try {
       cloudStorageDropbox.createFolder(CloudUtil.stripPath(OpenMode.DROPBOX,
file.getPath()));
       errorCallBack.done(file, true);
   } catch (Exception e) {
       e.printStackTrace();
       errorCallBack.done(file, false);
   }
} else if (file.isBoxFile()) {
   CloudStorage cloudStorageBox = dataUtils.getAccount(OpenMode.BOX);
   try {
       cloudStorageBox.createFolder(CloudUtil.stripPath(OpenMode.BOX,
file.getPath()));
       errorCallBack.done(file, true);
   } catch (Exception e) {
       e.printStackTrace();
       errorCallBack.done(file, false);
   }
} else if (file.isOneDriveFile()) {
   ...
} else if ...
```

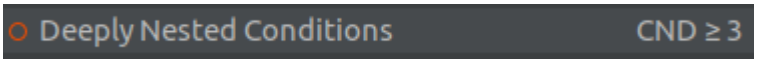
The code smell is Long Method.



The figure above shows the detection strategy for the Long Method code smell. In the case of the mkdir() method, every single metric value is over its threshold. What the class does can't be understood completely from a first glance since there are so many conditional branches and the code is so long. The method could be split into multiple parts to make it more readable.

Anomaly 2: Deeply Nested Conditions

Deeply Nested Conditions, or Dangerously Deep Nesting, is a method level anomaly which means that the nested level of if statements is too high to understand. According to MetricsTree's definition, Condition Nested Depth (CND) of methods should be less than 3.



The value of metrics for the method getFilesList in RootHelper.java are shown below.

Metric	Value	Threshold
CND	3	3

We can see that in the last few lines of the code sketch, the nested level of the if statement is 3, which fits the anomaly Deeply nested conditions.

```
public static ArrayList<HybridFileParcelable> getFilesList(String
path, boolean showHidden, OnFileFound listener) {
    File f = new File(path);
    ArrayList<HybridFileParcelable> files = new ArrayList<>();
    try {
        if (f.exists() && f.isDirectory()) {
            for (File x : f.listFiles()) {
                long size = 0;
                if (!x.isDirectory()) size = x.length();
                HybridFileParcelable baseFile =
                        new HybridFileParcelable(x.getPath(),
                                parseFilePermission(x),
                                x.lastModified(),
                                size, x.isDirectory());
                baseFile.setName(x.getName());
                baseFile.setMode(OpenMode.FILE);
                if (showHidden) {
                    files.add(baseFile);
                    listener.onFileFound(baseFile);
                } else {
                    if (!x.isHidden()) {
                        files.add(baseFile);
                        listener.onFileFound(baseFile);
                    }
                }
            }
        }
    } catch (Exception e) {
    }
    return files;
```

```
}
```

Anomaly 3: Long Parameters List

We detect the anomaly by using the metric profile of MetricsTree. According to the plugin's definition, if the number of parameters is larger or equal to 4, it would be considered a long parameters list. The value of method permissionsToOctalString() is shown in the table below.

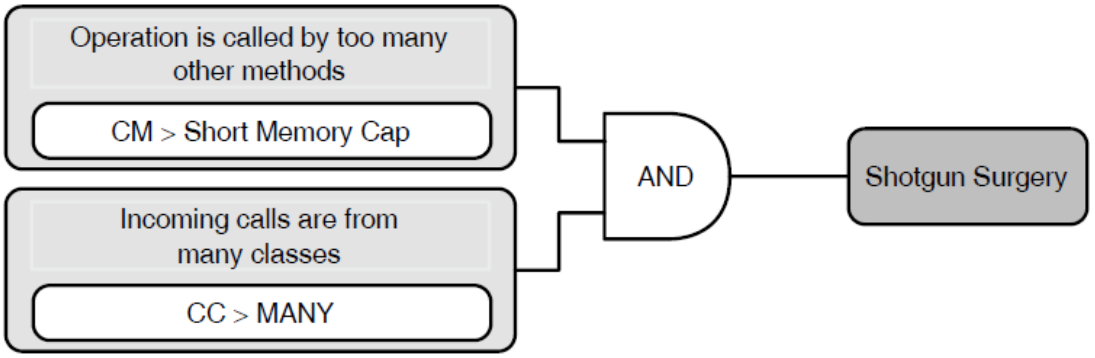
Metric	Value	Threshold
NOPM	9	4

The anomaly is Long Parameters List. The threshold of the anomaly is 4, but the method permissionsToOctalString has 9.

```
public static int permissionsToOctalString(
   boolean ur,
   boolean uw,
   boolean ux,
   boolean gr,
   boolean gw,
   boolean gx,
   boolean or,
   boolean ow,
   boolean ox) {
 int u = getPermissionInOctal(ur, uw, ux) << 6;
 int g = getPermissionInOctal(gr, gw, gx) << 3;
 int o = getPermissionInOctal(or, ow, ox);
 return u | g | o;
}
```

Anomaly 4: Shotgun surgery

Shotgun surgery is represented by too many incoming dependencies. According to the Object-Oriented Metrics in Practice book, the detection strategy according to thresholds is as follows:

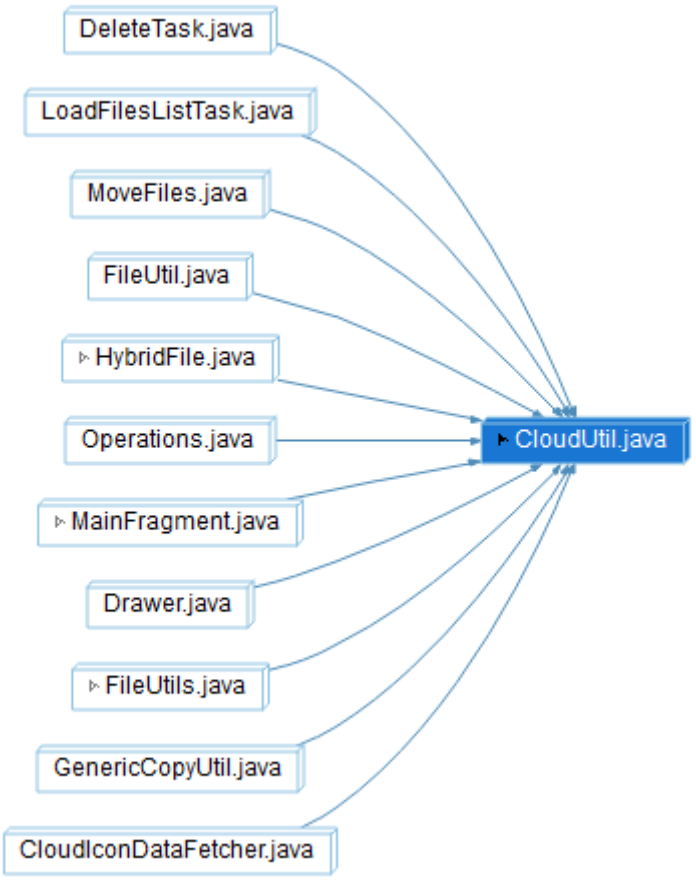


Detection strategy for Shotgun surgery

Numeric Value	Semantic Label
0	NONE
1	ONE/SHALLOW
2 – 5	TWO, THREE/FEW/ SEVERAL
7 – 8	Short Memory Capacity

Numeric values for thresholds

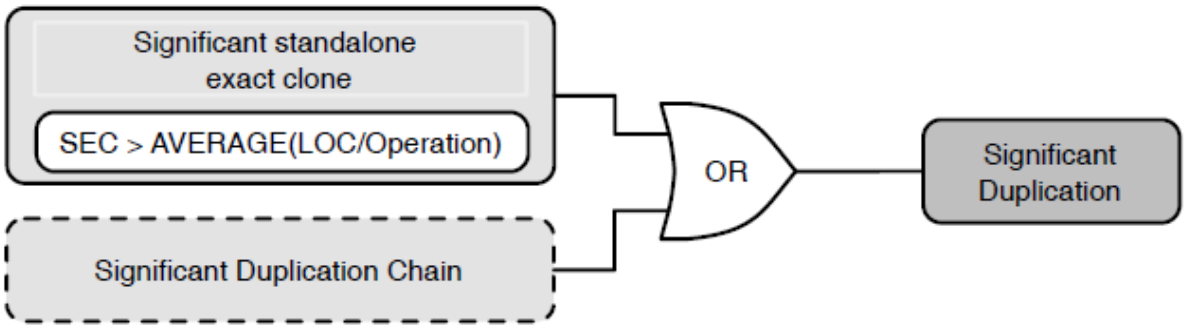
Since many classes depend on these methods, even a simple minor change could break many parts of the application. This situation is clearly visible in a call graph of `CloudUtil.java`. As we can see on the following picture, methods from CloudUtil are called from 11 different classes, often from multiple methods. This fits into the detection strategy shown above.



Incoming dependencies for CloudUtil.java

Anomaly 5: Significant duplication

Code duplication reduces maintainability by introducing unnecessary overhead for developers while changing functionality. It also increases the codebase size without introducing new features. Since only one class should offer a particular functionality, we need to eliminate duplicated code. We're going to focus on one of the possible strategies of finding a significant duplication:



A significant exact clone (SEC) is a group of consecutive duplicate lines of code. To be considered a significant duplication, this clone must be longer than the average of Lines of code divided by operation length.

If we look into the code of `StreamServer.java` and `CloudStreamServer.java`, we can see that both files contain private class `HttpSession`. This class is approximately 460 lines long and exactly the same in both files, except for a single method's argument. According to the detection method pictured above, since SEC = LOC in this case, it qualifies as a significant duplication.

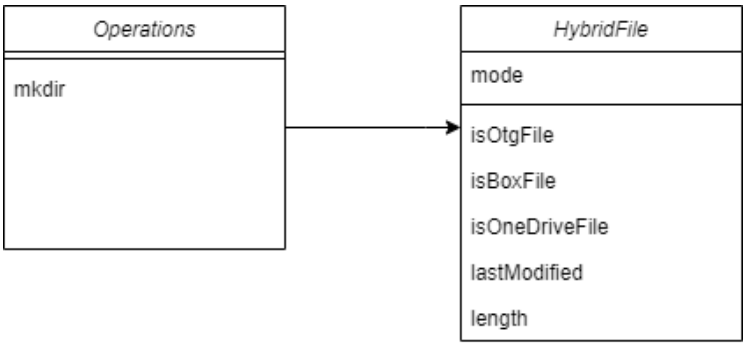
Correcting the Anomalies

Anomaly 1: Long Method

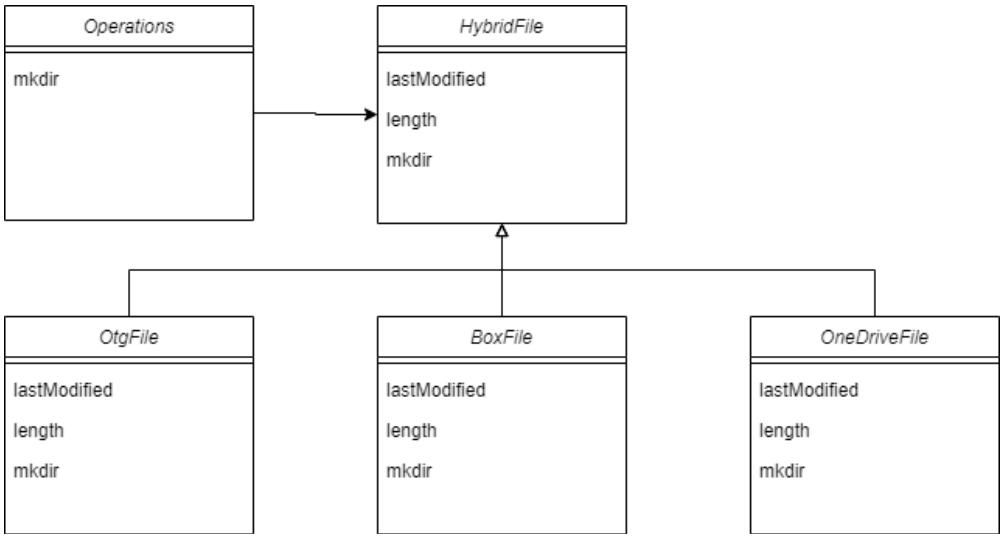
The mkdir method had a lot of conditional statements on the type of folder being created (whether it's in Box, OneDrive, etc.). This made the method long and hard to understand. The problem was not so much that the method is doing too many tasks, but rather that it's doing the simple 'create a directory' task but under each possible condition. Instead of separating the method into multiple methods in the same class, I opted to use polymorphism to have each folder type override a mkdir method. This simplifies the logic of the Operations class mkdir method, while keeping the different mkdir logic for each folder type. It also makes it easier to add a new folder type in future since it would only require overriding the mkdir method, but doesn't affect the one in the Operations class.

Before, the HybridFile class would hold the type of the file in an enum (mode) and have multiple methods to find which type it is. Most of the methods in HybridFile were doing conditional statements on each possible mode to determine how to react, such as in

lastModified and length. The mkdir method also had conditional statements based on the mode, as seen in the question 2b) for that anomaly.



After the refactoring, each file type inherits from HybridFile and overrides the corresponding methods such as lastModified and length.



Now, the mkdir method only needs to call mkdir on the HybridFile, which will be called on the right file type by polymorphism. So instead of having to do:

```
} else if (file.isDropBoxFile()) {
      CloudStorage cloudStorageDropbox =
dataUtils.getAccount(OpenMode.DROPBOX);
      try {
            byte[] tempBytes = new byte[0];
            ByteArrayInputStream byteArrayInputStream = new
ByteArrayInputStream(tempBytes);

cloudStorageDropbox.upload(CloudUtil.stripPath(OpenMode.DROPBOX,
file.getPath()),
                        byteArrayInputStream, 0l, true);
```

```
            errorCallBack.done(file, true);
      } catch (Exception e) {
            e.printStackTrace();
            errorCallBack.done(file, false);
      }
} else if (file.isBoxFile()) {
      CloudStorage cloudStorageBox = dataUtils.getAccount(OpenMode.BOX);
      try {
            byte[] tempBytes = new byte[0];
            ByteArrayInputStream byteArrayInputStream = new
ByteArrayInputStream(tempBytes);
            cloudStorageBox.upload(CloudUtil.stripPath(OpenMode.BOX,
file.getPath()),
                        byteArrayInputStream, 0l, true);
            errorCallBack.done(file, true);
      } catch (Exception e) {
            e.printStackTrace();
            errorCallBack.done(file, false);
      }
} else if (file.isOneDriveFile()) {
```

We can simply do:

```
file.mkdir(context, errorCallBack);
```

With the method simplified, the metric values of the method now become:

Metric	Old Value	New Value	Threshold
LOC	104	20	65 (130/2)
CC	26	3	4.8 (0.24*LOC)
MAXNESTING	6	3	5
NOAV	12	5	8

As we can see, all the metrics values are now in the threshold. The anomaly has therefore been fixed.

The whole refactoring would be huge, since multiple new classes would need to be created, the HybridFile class would need to be refactored completely but also there would need to be a way to create the correct instances of HybridFile depending on the mode, which would

then be used every time a HybridFile is created, for example by using the factory design pattern. In my case, I focused on only the mkdir method, so I added the new mkdir method in HybridFile and overrided it in every new class that inherits it. The quality criterias would be different if the whole refactor was applied.

Design Property	Design Metric	Old Value	New Value	Level
Design Size	DSC	422	429	Project
Hierarchies	NOH	50	50	Project
Abstraction	ANA	1.52	1.66	Project
Encapsulation	DAM	65.44%	65.44%	Project
Coupling	DCC	10.65	10.55	Class average
Cohesion	CAM	0.914	0.915	Project
Composition	MOA	6.40	6.32	Class average
Inheritance	MFA	13.79%	24.70%	Project
Polymorphism	NOP	14.90	9.05	Class average
Messaging	CIS	8.18	8.05	Class average
Complexity	NOM	11.34	11.16	Class average

Quality criteria name	Old Value	New Value
Functionality	109.03	109.25
Reusability	212.66	216.12
Understandability	-151.42	-151.75

Flexibility	8.15	5.21
Efficiency	4.72	3.59
Extensibility	2.95	0.20

The functionality is a bit higher, while the other quality criterias are lower. This is because in the refactoring I did, of only mkdir, HybridFile keeps its mode logic on top of the new polymorphism logic, for the reason explained above of complexity of the refactor. The functionality is therefore higher since there's less coupling and a bigger design size. The others could be higher after the big HybridFile refactor since it would increase the cohesion, inheritance and polymorphism values.

Anomaly 2: Deeply Nested Conditions

Several solutions fix the anomaly, such as combine tests, early return, and inline function calls with boolean short-circuiting. We found that there are at least two solutions that we could apply to the method.

Early return

We can refactor this condition by returning the empty files list immediately while the if statement is false. After that, the code inside the if block doesn't need to be encapsulated.

```
public static ArrayList<HybridFileParcelable> getFilesList(String
path, boolean showHidden, OnFileFound listener) {
   File f = new File(path);
   ArrayList<HybridFileParcelable> files = new ArrayList<>();
   try {
       if (f.exists() && f.isDirectory()) {
           ...
       }
   } catch (Exception e) {
   }
   return files;
}
```

Combine tests

The else block will only handle the condition that the if statement inside the block is true, so we can combine them.

```
...
                if (showHidden) {
                    files.add(baseFile);
                    listener.onFileFound(baseFile);
                } else {
                    if (!x.isHidden()) {
                        files.add(baseFile);
                        listener.onFileFound(baseFile);
                    }
                }
...
```

Before

```
public static ArrayList<HybridFileParcelable> getFilesList(String
path, boolean showHidden, OnFileFound listener) {
    File f = new File(path);
    ArrayList<HybridFileParcelable> files = new ArrayList<>();
    try {
        if (f.exists() && f.isDirectory()) {
            for (File x : f.listFiles()) {
                long size = 0;
                if (!x.isDirectory()) size = x.length();
                HybridFileParcelable baseFile =
                        new HybridFileParcelable(x.getPath(),
                                parseFilePermission(x),
                                x.lastModified(),
                                size, x.isDirectory());
                baseFile.setName(x.getName());
                baseFile.setMode(OpenMode.FILE);
                if (showHidden) {
                    files.add(baseFile);
                    listener.onFileFound(baseFile);
                } else {
                    if (!x.isHidden()) {
                        files.add(baseFile);
                        listener.onFileFound(baseFile);
                    }
                }
            }
        }
    } catch (Exception e) {
    }
    return files;
}
```

After

```
public static ArrayList<HybridFileParcelable> getFilesList(String
path, boolean showHidden, OnFileFound listener) {
    File f = new File(path);
    ArrayList<HybridFileParcelable> files = new ArrayList<>();
    try {
        if (!f.exists() || !f.isDirectory()) return files;
        for (File x : f.listFiles()) {
            long size = 0;
            if (!x.isDirectory()) size = x.length();
            HybridFileParcelable baseFile = new
                        HybridFileParcelable(x.getPath(),
                         parseFilePermission(x),
                        x.lastModified(), size, x.isDirectory());
            baseFile.setName(x.getName());
            baseFile.setMode(OpenMode.FILE);
            if (showHidden) {
                files.add(baseFile);
                listener.onFileFound(baseFile);
            } else if (!x.isHidden()) {
                    files.add(baseFile);
                    listener.onFileFound(baseFile);
            }
        }
    } catch (Exception e) {
    }
    return files;
}
```

With the method simplified, the metric values of the method now become:

Metric	Old Value	New Value	Threshold
CND	3	1	3

After refactoring, the metric number is below the threshold.

Design Property	Design Metric	Old Value	New Value	Level
Design Size	DSC	422	422	Project

Hierarchies	NOH	50	50	Project
Abstraction	ANA	1.52	1.52	Project
Encapsulation	DAM	65.44%	65.91%	Project
Coupling	DCC	10.65	10.7	Class average
Cohesion	CAM	0.914	0.914	Project
Composition	MOA	6.40	6.45	Class average
Inheritance	MFA	13.79%	13.90%	Project
Polymorphism	NOP	14.90	14.96	Class average
Messaging	CIS	8.18	8.18	Class average
Complexity	NOM	11.34	11.34	Class average

Quality criteria name	Old Value	New Value
Functionality	109.03	109.04
Reusability	212.66	212.64
Understandability	-151.42	-151.452
Flexibility	8.15	8.19
Efficiency	4.72	4.745
Extensibility	2.95	2.96

Only reusability and understandability are regressed. However, both of the values are relatively larger than others, making the influence less significant. Our refactoring reduces the level of nested condition blocks, which improves flexibility and efficiency the most.

Anomaly 3: Long Parameters List

According to the course slides, a long parameters list can be refactored by *Introduce parameter object*, which means to merge several parameters with similar purpose or attribute into a single object. The parameters before refactoring stand for read permission for the owner, write permission for the owner, execute permission for the owner, read permission for those in the same group, and so on. In other words, the first 3 parameters record the permission regarding the owner, the next 3 parameters are related to those in the same group, and the last 3 are the permission for other users.

```
public static int permissionsToOctalString(
   boolean ur,
   boolean uw,
   boolean ux,
   boolean gr,
   boolean gw,
   boolean gx,
   boolean or,
   boolean ow,
   boolean ox)
```

Therefore, we build a class to store the permission digits for a specific role.

```
public class Permissions {
   private boolean read;
   private boolean write;
   private boolean execute;

   public Permissions(boolean r, boolean w, boolean e) {
       read = r;
       write = w;
       execute = e;
   }

   public boolean getRead() { return read; }
   public boolean getWrite() { return write; }
   public boolean getExecute() { return execute; }
}
```

Before refactoring, we have to put 9 parameters when we call the method.

```
int per =
   RootHelper.permissionsToOctalString(
       readown.isChecked(),
       writeown.isChecked(),
       exeown.isChecked(),
       readgroup.isChecked(),
       writegroup.isChecked(),
       exegroup.isChecked()),
       readother.isChecked(),
       writeother.isChecked(),
       exeother.isChecked())
   );
```

After refactoring, only 3 parameters are needed, which is better to read and understand the purpose.

```
Permissions owner = new Permissions(readown.isChecked(),
                                    writeown.isChecked(),
                                    exeown.isChecked());
Permissions group = new Permissions(readgroup.isChecked(),
                                    writegroup.isChecked(),
                                    exegroup.isChecked());
Permissions other = new Permissions(readother.isChecked(),
                                    writeother.isChecked(),
                                    exeother.isChecked());

int per = RootHelper.permissionsToOctalString(owner, group, other);
```

With the method simplified, the metric values of the method now become:

Metric	Old Value	New Value	Threshold
NOPM	9	3	4

After refactoring, the metric number is below the threshold.

Design Property	Design Metric	Old value	New value	Level

Design Size	DSC	422	423	Project
Hierarchies	NOH	50	50	Project
Abstraction	ANA	1.52	1.52	Project
Encapsulation	DAM	65.44%	65.5%	Project
Coupling	DCC	10.65	10.61	Class average
Cohesion	CAM	0.914	0.913	Project
Composition	MOA	6.40	6.35	Class average
Inheritance	MFA	13.79%	13.76%	Project
Polymorphism	NOP	14.90	14.91	Class average
Messaging	CIS	8.18	8.19	Class average
Complexity	NOM	11.34	11.35	Class average

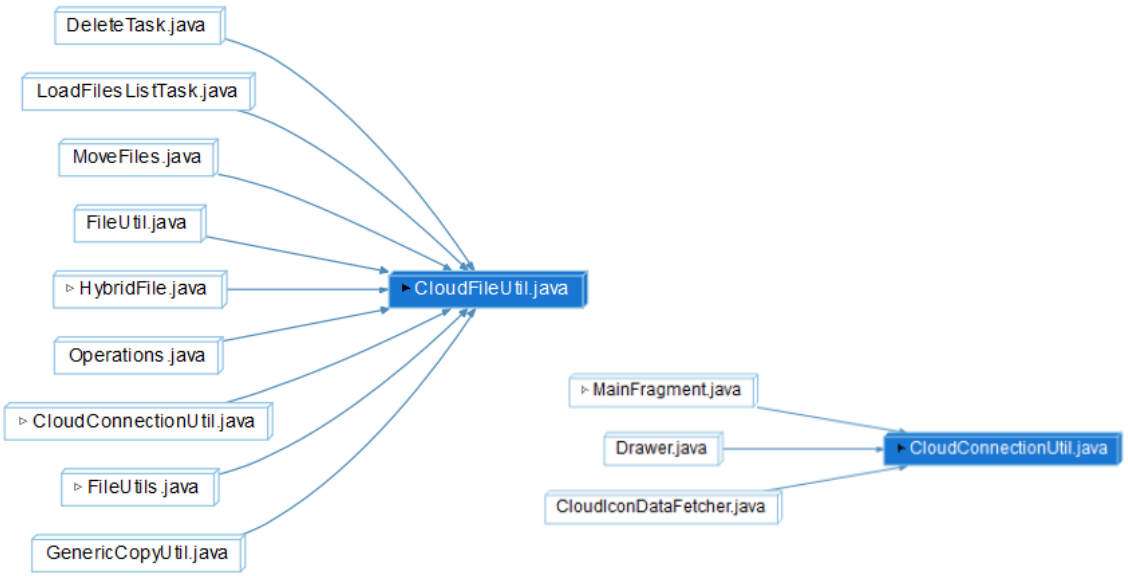
Quality criteria name	Old value	New value
Functionality	109.03	109.28
Reusability	212.66	213.26
Understandability	-151.42	-151.786
Flexibility	8.15	8.14
Efficiency	4.72	4.71
Extensibility	2.95	2.99

According to the quality criteria above, we can see that the refactor, introducing parameter

objects, improves functionality and reusability. However, the efficiency regresses since a new object has to be created while calling the method. Understandability also decreases a little due to a similar reason.

Anomaly 4: Shotgun surgery

Shotgun surgery disharmony can be solved by splitting a class into multiple logical chunks and separating them into individual classes. `CloudUtils` class provides two types of functions - checking files and connecting to cloud systems. We'll split the class into two according to these functions: `CloudFileUtil` and `CloudConnectionUtil`. We can see the resulting call dependencies below:



Callby graphs for CloudFileUtil and CloudConnectionUtil.

We've successfully moved some dependencies and responsibilities into a separate class. All 9 classes on the left side call a single method of `CloudFileUtil`, so this cannot be split further. Even though this change is not very significant, it still improves the overall maintainability and reduces ripple effects during code changes, as we can see from the following metrics table:

Design Property	Design Metric	Old value	New value	Level
Design Size	DSC	422	427	Project
Hierarchies	NOH	50	50	Project
Abstraction	ANA	1.52	1.54	Project
Encapsulation	DAM	65.44%	65.44%	Project
Coupling	DCC	10.65	10.70	Class average
Cohesion	CAM	0.914	0.914	Project
Composition	MOA	6.40	6.42	Class average
Inheritance	MFA	13.79%	13.81%	Project
Polymorphism	NOP	14.90	15.25	Class average
Messaging	CIS	8.18	8.37	Class average
Complexity	NOM	11.34	11.61	Class average

Quality criteria name	Old value	New value
Functionality	109.03	110.25
Reusability	212.66	215.24
Understandability	-151.42	-153.30
Flexibility	8.15	8.32
Efficiency	4.72	4.80

Extensibility	2.95	3.11

Our changes improved most of the quality criteria values. Getting rid of shotgun surgery and splitting classes into separate parts with their own responsibilities improved the extensibility and reusability of the application, other parameters changed only by a small amount.

Anomaly 5: Significant duplication

We can eliminate this disharmony by maintaining only one instance of the HttpSession class. We'll do this by extracting this class into a new file and deleting the duplicated code from both previous files. During this process, we can also find more duplicated code - this has been extracted into new classes: `HTTPResponseCode`, `HTTPSession`, `Response`. Classes with related dependencies had to be modified accordingly. In total, 12 files have been changed, with 679 lines added and 1230 lines deleted. Based solely on these numbers, we might suspect that some duplicated code has been removed. We can see new project statistics below:

Design Property	Design Metric	Old value	New value	Level
Design Size	DSC	422	423	Project
Hierarchies	NOH	50	52	Project
Abstraction	ANA	1.52	1.48	Project
Encapsulation	DAM	65.44%	65.96%	Project
Coupling	DCC	10.65	10.67	Class average
Cohesion	CAM	0.914	0.914	Project
Composition	MOA	6.40	6.40	Class average

Inheritance	MFA	13.79%	14.41%	Project
Polymorphism	NOP	14.90	14.46	Class average
Messaging	CIS	8.18	8.14	Class average
Complexity	NOM	11.34	11.23	Class average

Quality criteria name	Old value	New value
Functionality	109.03	109.58
Reusability	212.66	213.131
Understandability	-151.42	-151.46
Flexibility	8.15	7.77
Efficiency	4.72	4.57
Extensibility	2.95	2.56

By removing significant duplication, we've improved reusability of the code. Other metrics however seem to have changed negatively - by removing methods from classes and reducing the codebase size we've apparently reduced extensibility and flexibility of the code. We believe this is only a mistake in metrics interpretation, as these calculations can't exactly reflect the precise state of the codebase.