SENG 401 Final Report

AHMED, S, AMER
Undergraduate Student, University of Calgary Software and Electrical Engineering
ahmedsay01@gmail.com

VAIBHAV, D, KAPOOR
Undergraduate student, University of Calgary Software and Electrical Engineering department, vaibhav.kapoor@ucalgary.ca

NICHOLAS, H, LEE
Undergraduate student, University of Calgary Software and Electrical Engineering department, nicholas.lee1@ucalgary.ca

BENJAMIN, J, NIELSEN
Undergraduate student, University of Calgary Software and Electrical Engineering department, benjamin.nielsen@ucalgary.ca

THOMAS (HONG), PAN
Undergraduate Student, University of Calgary Software and Electrical Engineering
hong.pan@ucalgary.ca

MATTHEW, S, WELLS
Undergraduate student, University of Calgary Software and Electrical Engineering department, matthew.wells@ucalgary.ca

The architecture of a software system is always evolving based on changes that can make the software system more efficient and reliable. Analyzing the architecture of a software system from time to time, can lead to discovering better implementations that allow for a more stable and efficient software system. The task at hand involved the analysis of two similar software systems with the difference of one utilizing machine learning. The architecture of each software system was then analyzed to find areas where the architecture can be improved to decrease coupling, reduce redundancy, and increase expandability. To improve the architecture of each software system, software design patterns were utilized to refactor certain code. This report provides information about the software systems chosen, how they were analyzed and what design patterns were implemented to improve the architecture of the system overall.

CCS CONCEPTS • Applied Computing---Computers in other domains---Personal computers and PC applications---Computer games; 500 • Software and its engineering---Software organization and properties---Software systems structures---Software architectures---Publish-subscribe/event-based architectures; 300 • Software and its engineering---Software creation and management---Software post-development issues---Software evolution; 100

1. Introduction

The objective of this project is to learn about how software architecture and design patterns are implemented in real life software. Another objective was to demonstrate how software development tools such as BOA are used.

The first objective was met through analyzing two chosen related software systems (one with ML and one without) in order to create C&C, Uses, Decomposition and Layered Diagrams to visualize the software's architecture. Afterwards, the implementation of the software was analyzed and two design patterns that would improve the implementation were found for each software. The second objective was met through the attempted use of the BOA, and ARCADE to supplement the process of analyzing the two software systems, along with manual analysis methods.

Some of the findings were the fact that the software tools were not found to be significantly helpful when analyzing larger software systems with files of different extensions. Also, many professionally developed software systems (such as the OpenCV library) were written with the proper use of design patterns, so new ones were difficult to find. While the two systems had a similar goal, it was discovered through this project that their implementation and architecture could not be more different.

2. System Overview

2.1. Introduction to Systems

The two systems that were selected for this project were the "Sudoku 16x16 Explainer" and "Snap Solve Sudoku". Both systems contain code that is intended to solve sudoku puzzles, with the "Sudoku 16x16 Explainer" focusing on using known sudoku techniques to solve and demonstrate step by step solutions while the other system "SnapSolve" focusing on using machine learning to recognize puzzles through images and instantly generating solutions. The Sudoku 16x16 Explainer is a Java based application designed to be run as an executable on supporting desktop operating systems whereas the SnapSolve is an android application designed using Java and Kotlin. In this document "Sudoku 16x16 Explainer" will be referred as 'system 1' and the ML system "Snap Solve Sudoku" will be referred as 'system 2'.

2.2. Summary Statistics

Repository 1: Sudoku 16x16 Explainer
This project is able to solve a provided sixteen by sixteen sudoku puzzle, and then give a step by step walkthrough of the techniques that the code used to fill each square.

- Link: https://github.com/1to9only/Sudoku16x16Explainer
- Number of Contributors: 1
- Number of Classes: 101
- Number of Interfaces: 7
- Number of Enums: 2

Repository 2: Snap Solve Sudoku (ML)
This project uses machine learning to recognize an image of a sudoku puzzle, and then solve all available solutions that exist for said puzzle.

- Link: https://github.com/Beebeeoii/snap-solve-sudoku
- Number of Contributors: 1
- Number of Classes: 139
- Number of Interfaces: 3
- Number of Enums: 0

3. First Software System

3.1. Architecture

The decomposition view of Sudoku 16x16 Explainer shows the organization of system code as modules and submodules in a ‘part-of’ hierarchical relationship. From the diagram represented by Figure 3.1, the Sudoku16x16Explainer module can be seen containing the overall system code with all the Java modules. The applet module implements threading and code responsible to run the application on windows, the generator module is responsible for generating sudoku models while the gui module creates all the graphical components accessible to the user such as the board itself, dialogue messages, menu screen, etc. The IO handles the user interaction as the Solver generates solutions and tips based on the user input on the puzzles. Lastly, the tools module contains permutation generators and linked sets used to generate the solutions and map out the puzzle.

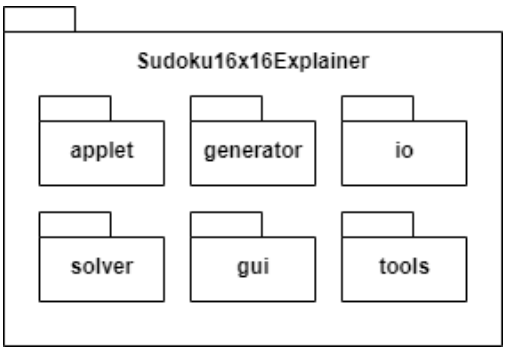

Figure 3.1: Decomposition view shows the major components/modules of Sudoku16x16Explainer

The C&C view of the Sudoku 16x16 Explainer system in the data-flow style shows the computational interaction between the components in the system. As a GUI based app, a lot of data begins and ends at the GUI component which directly communicates back and forth with the IO and Tools components. Saved sudoku boards are read into the software through the IO component (gui passes it the grid to load it into) while the HTML elements specified by the GUI components are loaded in using classes in the Tools component. The Applet module communicates directly with the gui to enable basic applet support to run the application. GUI also communicates with Solver both directly and indirectly though Sudoku and Generator and when checking for hints, it communicates directly with the Solver.

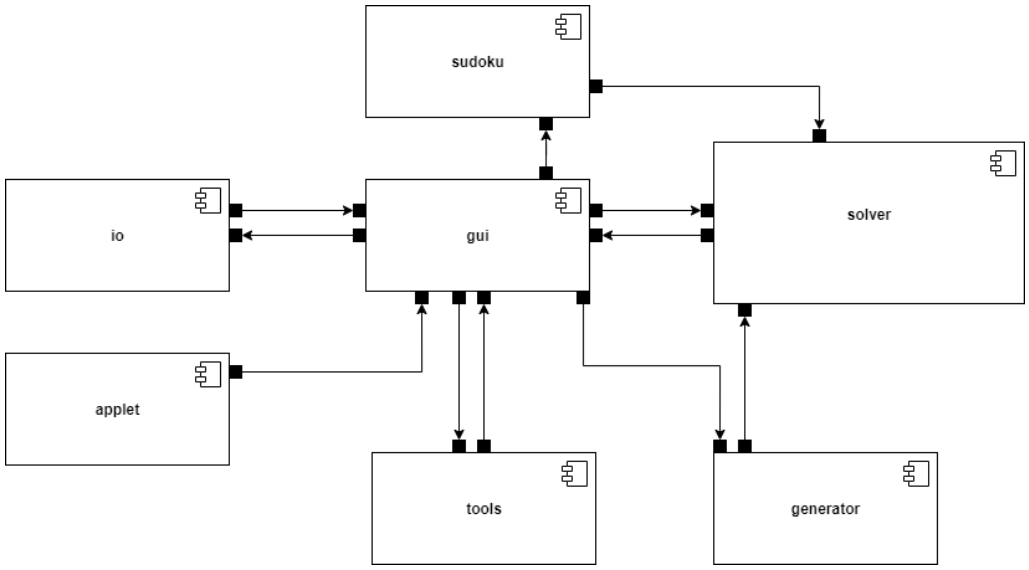

Figure 3.2: High-Level C&C of Sudoku16x16Explainer

3.2. Design Pattern 1: Facade Pattern

The Facade pattern was applied in the tools package in the Sudoku16x16Explainer repository. The class CommonTuples provides functionality that could be hidden behind a unified interface which would make the subsystem easier to use. As a result we decided to implement the facade pattern.

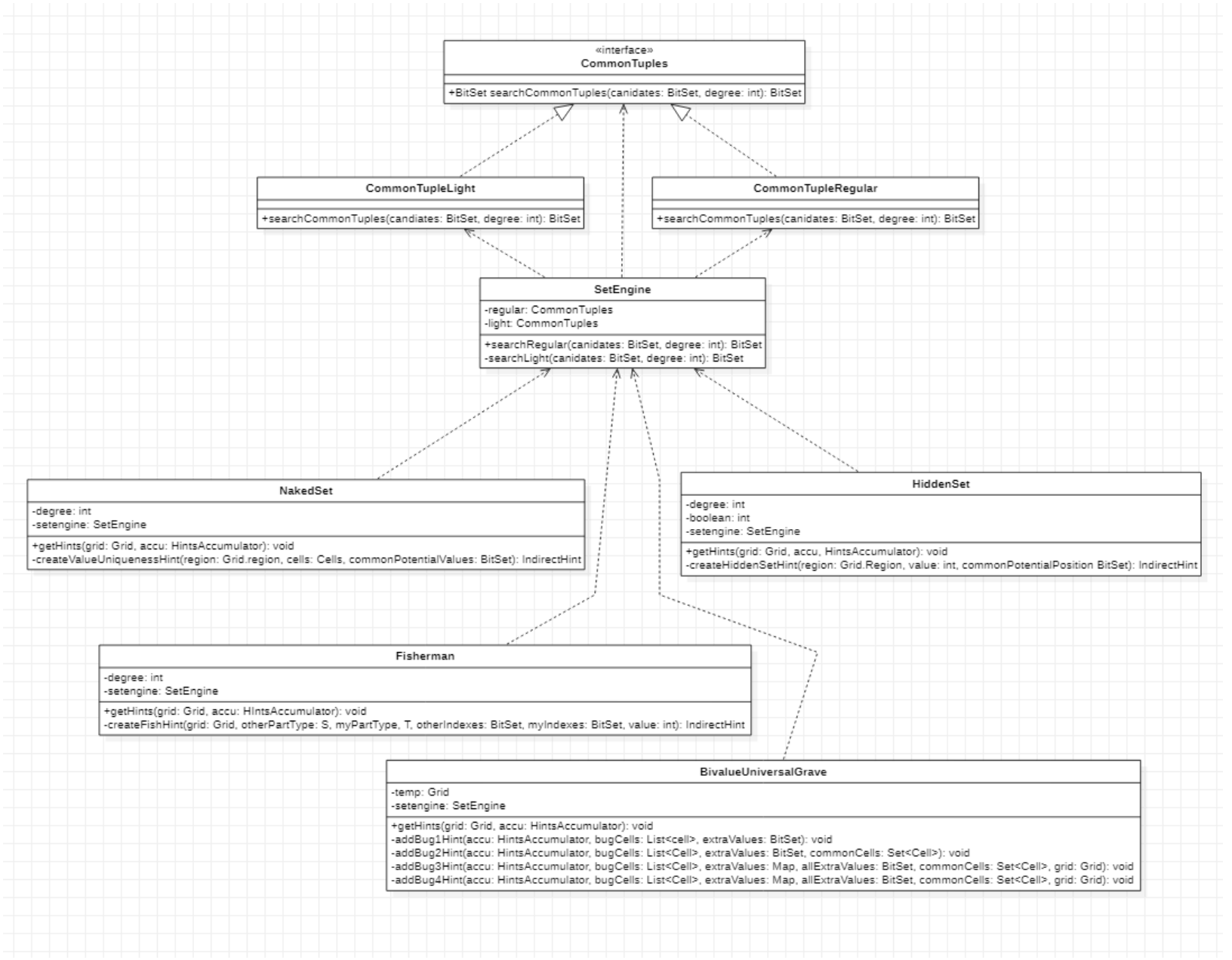

Figure 3.3: Facade Pattern UML diagram

Facade Pattern Implementation Code:

```
public interface CommonTuples{
     BitSet searchCommonTuple(BitSet[] candidates, int degree);
}

public class CommonTuplesLight implements CommonTuples {
    public BitSet searchCommonTuple(BitSet[] candidates, int
degree) {
        BitSet result = new BitSet(16);
        for (BitSet candidate : candidates) {
            result.or(candidate);
            if (candidate.cardinality() == 0)
                return null;
        }
        if (result.cardinality() == degree)
            return result;
        return null;
    }
}

public class CommonTuplesRegular implements CommonTuples {
    public BitSet searchCommonTuple(BitSet[] candidates, int
degree) {
        BitSet result = new BitSet(16);
        for (BitSet candidate : candidates) {
            if (candidate.cardinality() <= 1)
                return null;
            result.or(candidate);
        }
        if (result.cardinality() == degree)
            return result;
        return null;
    }
}

public class SetEngine {
```

```
	private CommonTuples regular, light;

	public SetEngine(){
		this.regular = new CommonTuplesRegular();
		this.light = new CommonTuplesLight();
	}

	public BitSet searchRegular(BitSet[] candidates, int
degree){
		return this.regular.searchCommonTuple(BitSet[]
candidates, int degree);
	}

	public BitSet searchLight(BitSet[] candidates, int degree){
		return this.light.searchCommonTuple(BitSet[]
candidates, int degree);
	}
}
```

3.3. Change in System Architecture due to Design Pattern 1

To implement the facade pattern the code first had to be refactored to allow for a more efficient implementation. Namely the CommonTuples.java class was turned into an interface implemented by two new classes, CommonTuplesLight and CommonTupleRegular. Using the CommonTuples.java interface a new class, SetEngine.java, was added to facilitate the use of the facade pattern. Using this SetEngine class we are able to hide the original complexity of the code and create a more efficient, simple design. Below we can see the modules and components that we affected by the addition of the facade pattern.

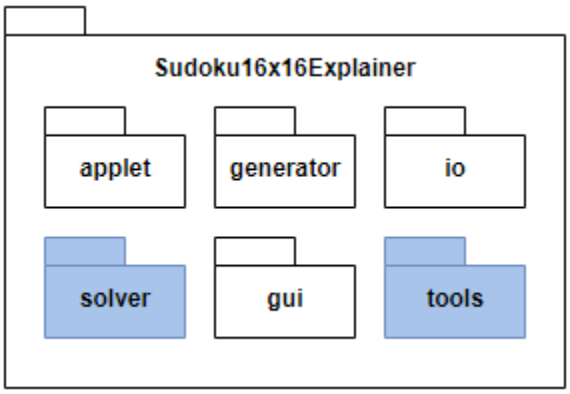

Figure 3.4: Facade Pattern Documentation: Affected Modules

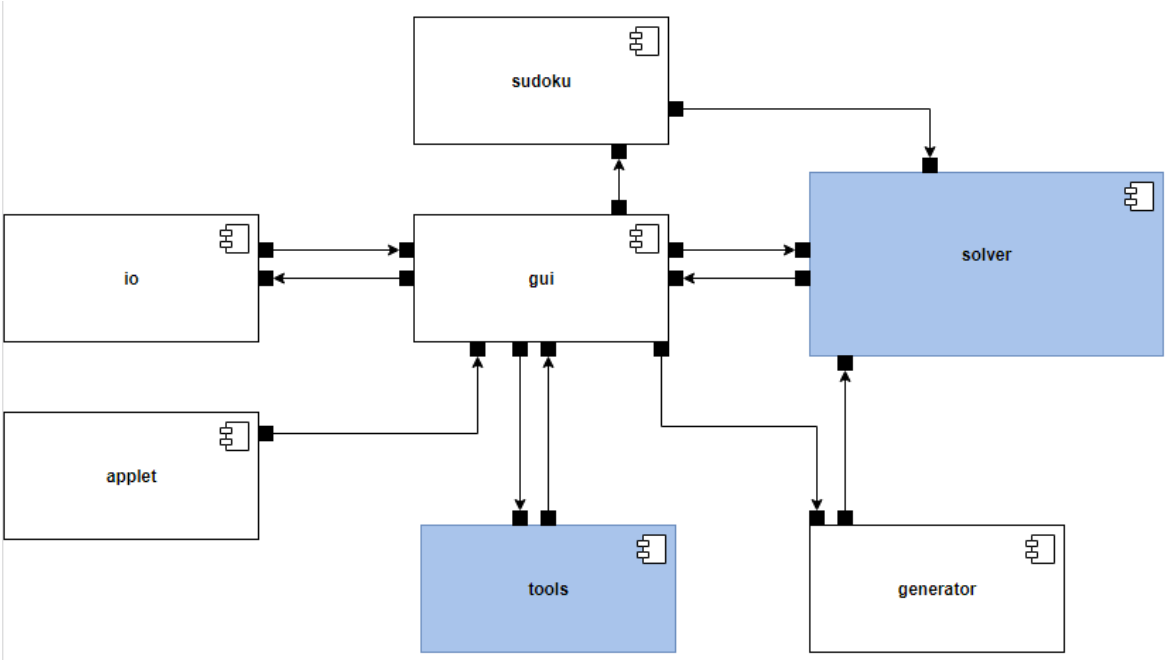

Figure 3.5: Facade Pattern Documentation: Affected Components

3.4. Design Pattern 2: Abstract Factory Pattern

An abstract factory pattern was applied within the solver package of the Sudoku16X16Explainer. Within the solver package there originally were four BugHint classes that were used by the solver in one of its sudoku solving methods. These classes were all child classes of an abstract BugHint class. It was found that the system needed to use one object within this family of objects at any given time, and also that the general solver system did not need to see the actual concrete implementation of the classes in order to operate. It was determined only four classes would be needed to be added for this design pattern: AbstractBugHintFactory, ComplexBugHintFactory, SimpleBugHintFactory, and BugHintProducer.

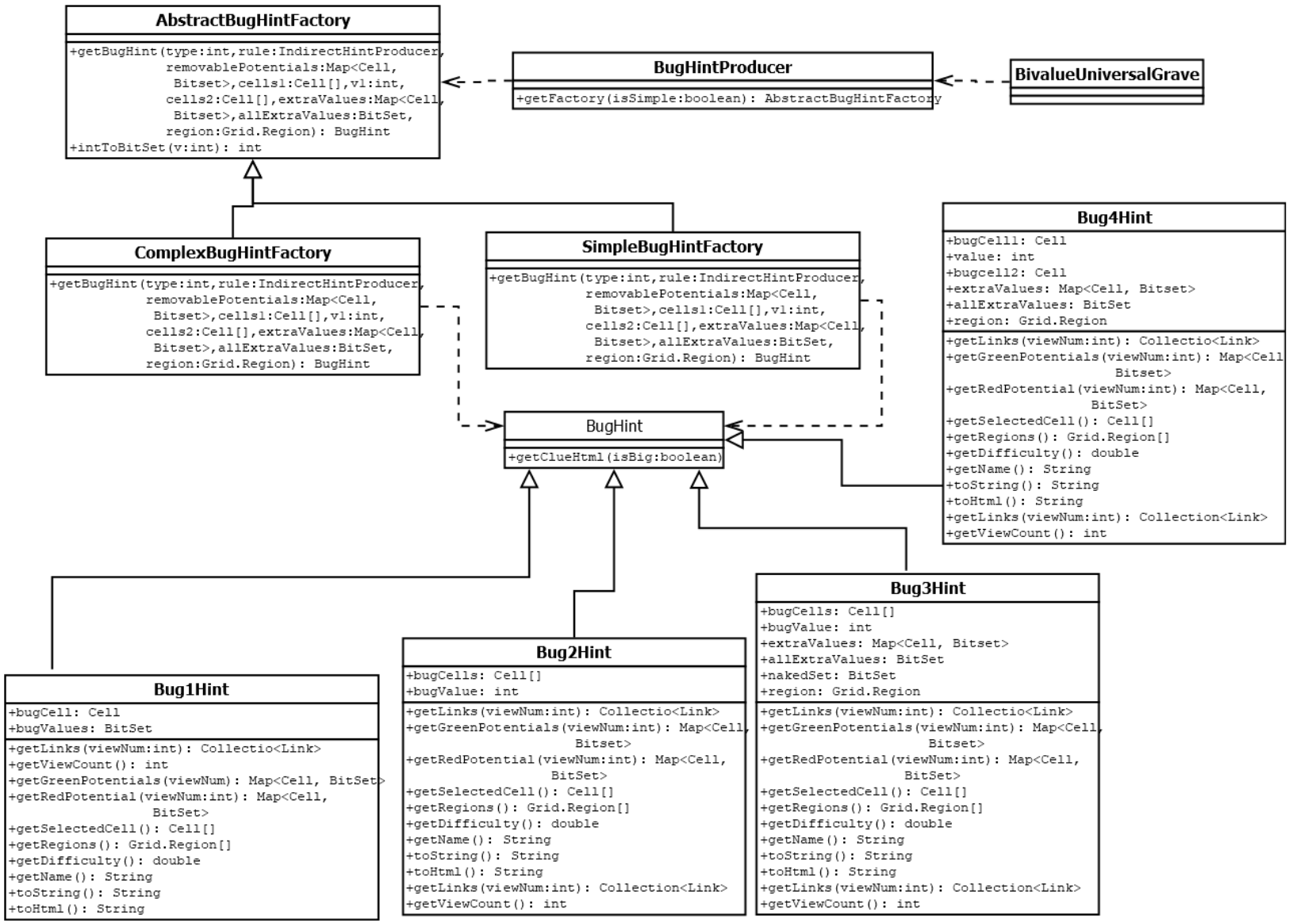

Figure 3.6: Abstract Factory Pattern UML diagram

Abstract Factory Pattern Implementation Code:

```
public abstract class AbstractBugHintFactory{

     public abstract BugHint getBugHing(IndirectHintProducer
rule, Map<Cell, Bitset> removablePotentials,

            Cell[] cells1, int v1, Cell[] cells2, Map<Cell,
BitSet> extraValues,

            BitSet allExtraValues, Grid.Region region);


    public int intToBitSet(int v){

     Bitset b = new BitSet();

     int i = 0;

     while(v != 0){

          if(v %2 != 0)

               b.set(i);

          ++i;

          v = v >>> 1;

     }

     return b;

    }

}


public class ComplexBugHintFactory extends
AbstractBugHintFactory{

     @Override
```

```
     public BugHint getBugHint(int type, IndirectHintProducer
rule, Map<Cell, Bitset> removablePotentials,

            Cell[] cells1, int v1, Cell[] cells2, Map<Cell,
BitSet> extraValues,

            BitSet allExtraValues, Grid.Region region) {

          if (type == 3)

               return new Bug1Hint(rule, removablePotentials,
cells1, cells2, extraValues, allExtraValues, intToBitSet(v1),
region);

          else if (type == 4)

               return new Bug2Hint(rule, removablePotentials,
cells1[0], cells2[0], extraValues, allExtraValues, v1, region);

          else

               return null;

     }

}


public class SimpleBugHintFactory extends
AbstractBugHintFactory{

     @Override

     public BugHint getBugHint(int type, IndirectHintProducer
rule, Map<Cell, Bitset> removablePotentials,

            Cell[] cells1, int v1, Cell[] cells2 = null,
Map<Cell, BitSet> extraValues = null,

            BitSet allExtraValues = null, Grid.Region region =
null) {
```

```
          if (type == 1)
               return new Bug1Hint(rule, removablePotentials,
cells1[0], intToBitSet(v1));
          else if (type == 2)
               return new Bug2Hint(rule, removablePotentials,
cells1, v1);
          else
               return null;
     }
}


public class BugHintProducer{
     public static AbstractBugHintFactory getFactory(boolean
isSimple) {
          if(isSimple)
               return new SimpleBugHintFactory();
          else
               return new ComplexBugHintFactory();
     }
}
```

3.5. Change in System Architecture due to Design Pattern 2

To accomplish the implementation of the abstract factory pattern, new classes had to be created. First an abstract factory class called AbstractBugHintFactory was designed. It contained one method to facilitate the returning of a concrete factory class, and one to help these concrete factories with data conversion necessary to allow the design pattern to work. Then the child classes of ComplexBugHintFactory and SimpleBugHintFactory were created. Bug1Hint and Bug2Hint were classified as simple BugHints - with SimpleBugHintFactory being responsible for their creation - as their operations were significantly less complex than those of Bug3Hint and Bug4Hint. This also meant that Bug3Hint and Bug4Hint were classified as complex, and it would be the responsibility of the ComplexBugHintFactory to ensure their creation. Then class BugHintProducer was created with a single method responsible for either returning a new ComplexBugHintFactory or a new SimpleBugHintFactory. All of this allowed for the creation of BugHint classes to be hidden from class BivalueUniversalGrave, which was the class within the system that made use of them.

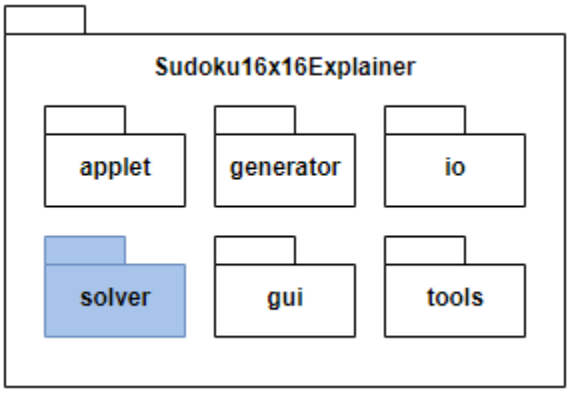

Figure 3.7: Abstract Factory Pattern Documentation: Affected Modules

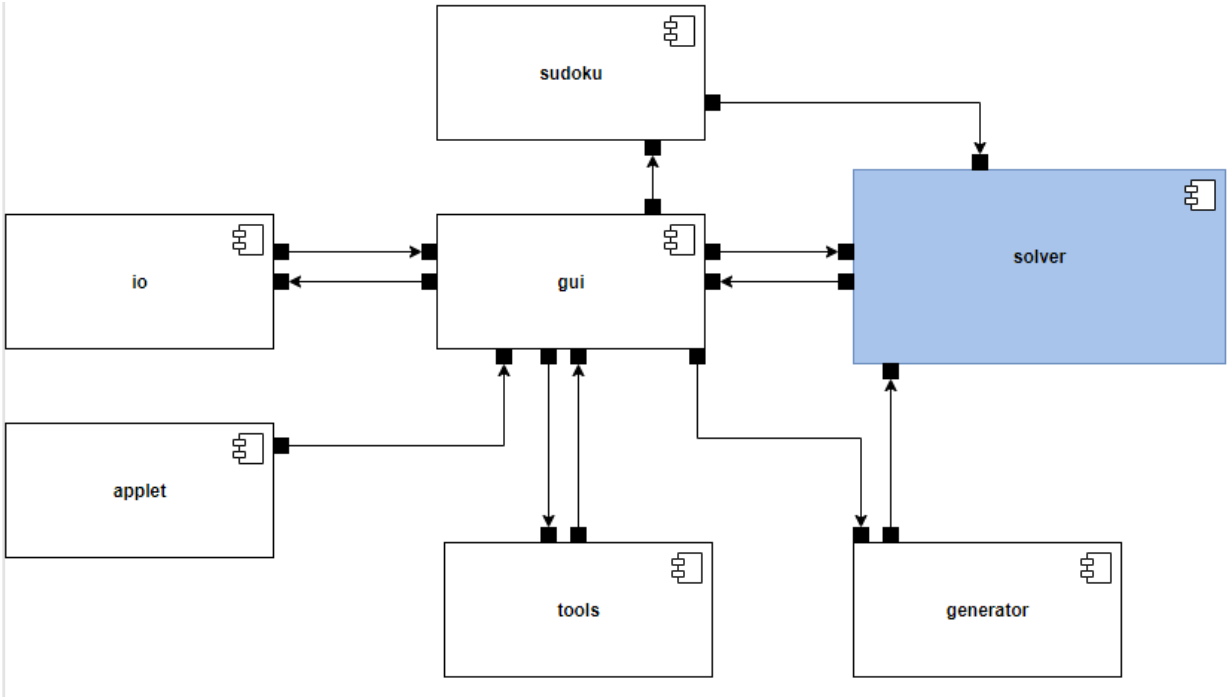

Figure 3.8: Abstract Factory Pattern Documentation: Affected Components

4. Second Software System

4.1. Architecture

The decomposition view of the Snap Solve Sudoku helps display the organization of the system code as modules and submodules in a ‘part-of’ hierarchical relationship. From the diagram represented by figure 4.1, two large modules: SnapSolveSudoku and the openCVLibrary can be seen containing the overall system code. The SnapSolveSudoku module contains code responsible for mapping app functionality and parsing data received from openCV which is a library designed to deal with image processing, video capture and object detection. The SnapSolveSudoku module includes mainly Java code to deal with feature implementation along with some XML and Kotlin code that the system uses to map the android framework of application. The openCV library module contains all Java source code with all modules within the library interacting with core and utils modules for their implementation. Imgcodecs and imgproc modules also interact with the android module which is responsible for the hardware communications such as reading and rendering camera information which the interacting modules process and decode into information meaningful to the program.

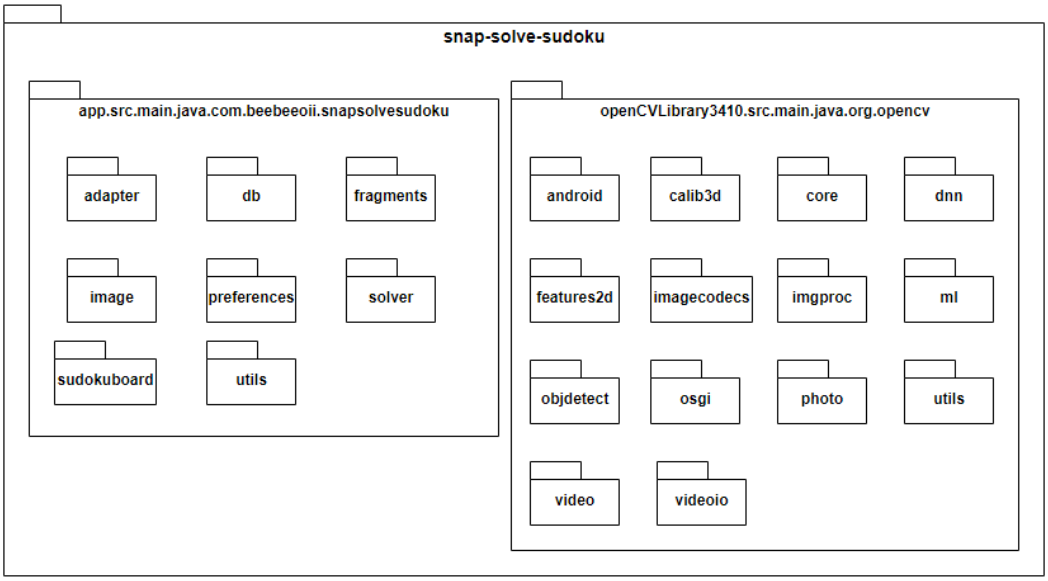

Figure 4.1: Decomposition view shows the major components/modules of SnapSudokuSolver

The C&C view of the Snap Solve Sudoku system in the Data flow style displays the two major components of the system: the openCVLibrary and the SnapSolveSudokuApp. The openCVLibrary component comprises sub components responsible for taking in raw image data and processing them for the SnapSolveSudokuApp through machine learning. The SnapSolveSudokuApp component contains sub components which take the processed data retrieved from the library and give it sudoku context while implementing the features and framework for the user interactable app itself. Most components were found to both send and receive data from the other components that they were attached to and due to the similar nature of the functionality demonstrated by the sub-components, the diagram was simplified from its initial design that showcased all the components to a more clear and less complex diagram. The final C&C diagram as shown by figure 4.2 uses overarching modules to show the overall functionality of the system such that it is easier to interpret at a higher level. The imgproc, calib3d, objdetect, photo, video, videoio, imgcodecs and features2d components in the system were grouped into an encompassing component called image processing and similarly the mll and dnn components were grouped into machine learning due to their similar nature in functionality. In the SnapSolveSudoku component, the SudokuBoard and

solver modules were grouped into one Sudoku App component and the rest under Data Processing.

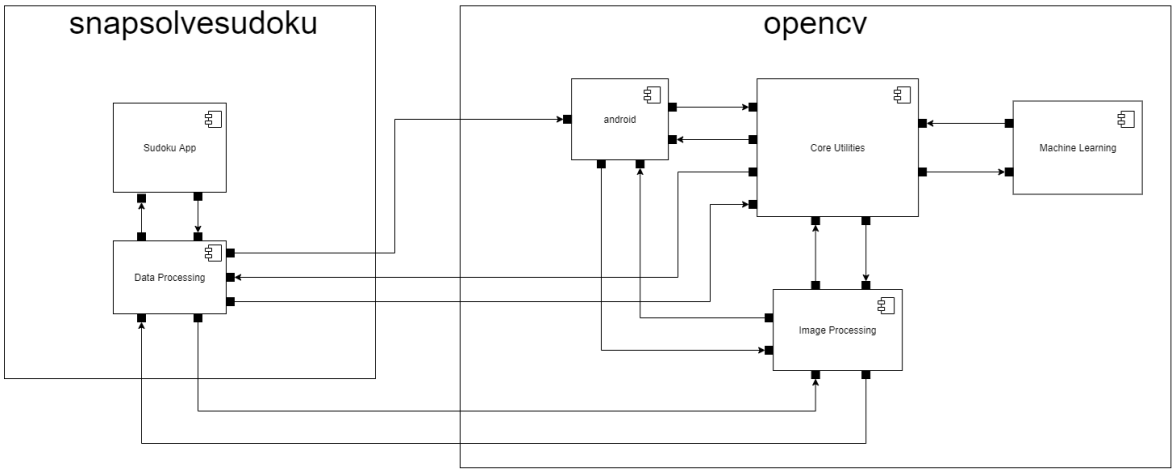

Figure 4.2: High-Level C&C shows the data flow between the SnapSudokuSolver system components

4.2. Design Pattern 1: Factory

The factory design pattern was applied to the Core Package inside the openCVLibrary in the Snap Solve Sudoku (ML Repo) System. The openCV Library has two very similar classes: Rect and Rect2d which only differ by the member variable types used within them. Through the factory pattern, the different Rect classes can be separated from their implementation such that other packages that use the “Rect” family of classes can simply enter their type and values into a factory which then creates the appropriate Rect class for them based on their type of input.

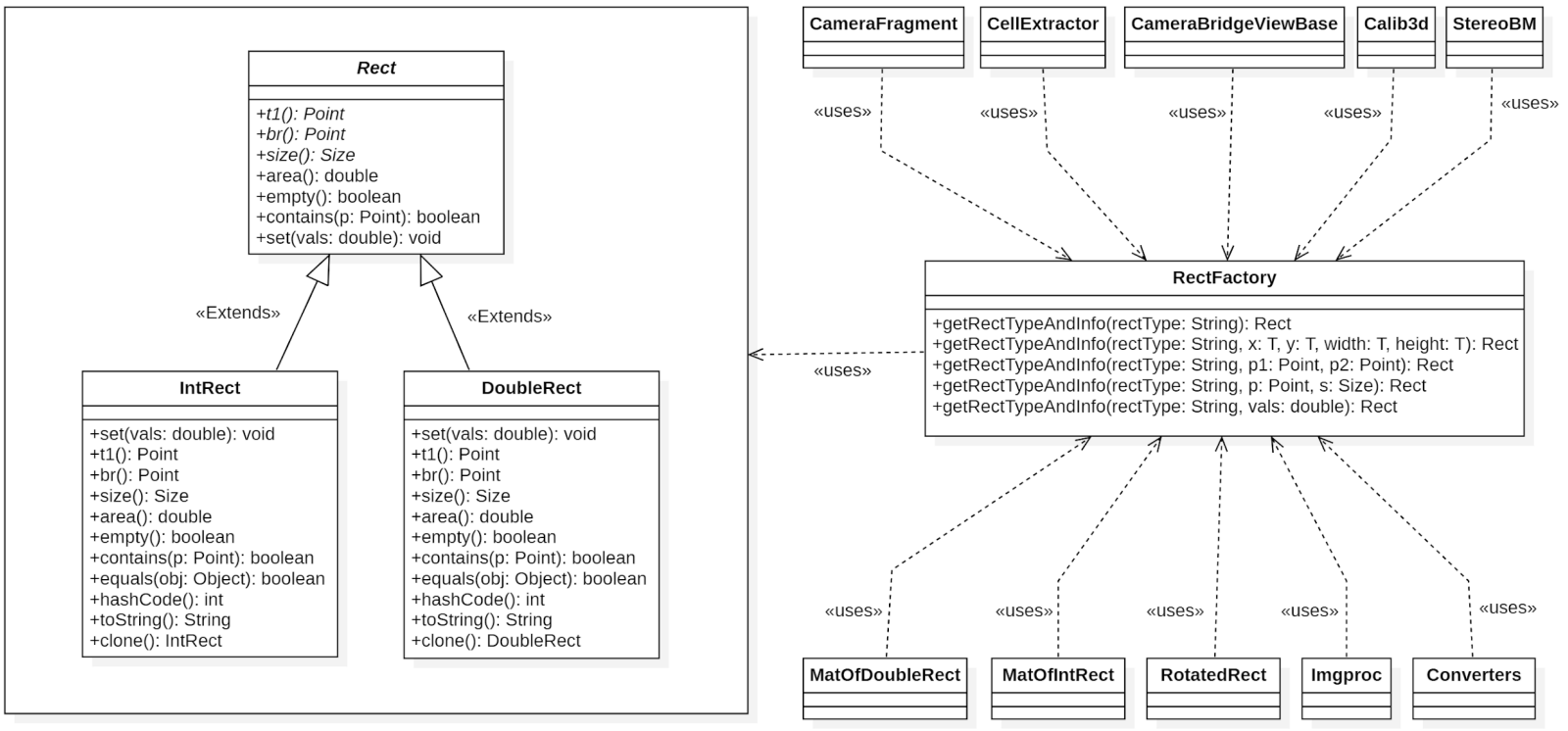

Figure 4.3: Factory Pattern UML

Factory Pattern Implementation Code:

```
//complete implementation of abstract Rect class
package org.opencv.core;

//javadoc:Rect_
public abstract class Rect {

     public int x, y, width, height;

    public abstract void set(double[] vals);

    public abstract Point tl();

    public abstract Point br();

    public abstract Size size();

    public abstract double area();

    public abstract boolean empty();

    public abstract boolean contains(Point p);

}


//concrete type of Rect with integers (only heading shown)
package org.opencv.core;

public class IntRect extends Rect{


//concrete type of Rect with doubles (only heading shown)
package org.opencv.core;

public class DoubleRect extends Rect{
```

```
    public double x, y, width, height;


//complete implementation of Factory
package org.opencv.core;

public class RectFactory{
     public Rect getRectTypeAndInfo(String rectType){
          if(rectType.equals("int"))
               return new IntRect();
          else if(rectType.equals("double"))
               return new DoubleRect();
          else
               return null;
     }

     public Rect getRectTypeAndInfo(String rectType, double x,
double y, double width, double height){
          if(rectType.equals("int"))
               return new IntRect(x, y, width, height);
          else if(rectType.equals("double"))
               return new DoubleRect(x, y, width, height);
          else
               return null;
     }

     public Rect getRectTypeAndInfo(String rectType, Point p1,
Point p2){
          if(rectType.equals("int"))
               return new IntRect(p1, p2);
          else if(rectType.equals("double"))
               return new DoubleRect(p1, p2);
          else
               return null;
     }
```

```
     public Rect getRectTypeAndInfo(String rectType, Point p,
Size s){
          if(rectType.equals("int"))
               return new IntRect(p, s);
          else if(rectType.equals("double"))
               return new DoubleRect(p, s);
          else
               return null;
     }

     public Rect getRectTypeAndInfo(String rectType, double[]
vals){
          if(rectType.equals("int"))
               return new IntRect(vals);
          else if(rectType.equals("double"))
               return new DoubleRect(vals);
          else
               return null;
     }
}
```

4.3. Change in System Architecture due to Design Pattern 1

Two additional classes were added to act as interfaces to the factory. Previously, external classes would directly call the constructors/members of Rect and Rect2D directly. All external references were altered to calls to the new factory class, and calls to member functions were relegated to the abstract Rect class. This hides the specific implementation of the Rect classes, and further abstracts them from a direct use by the author.

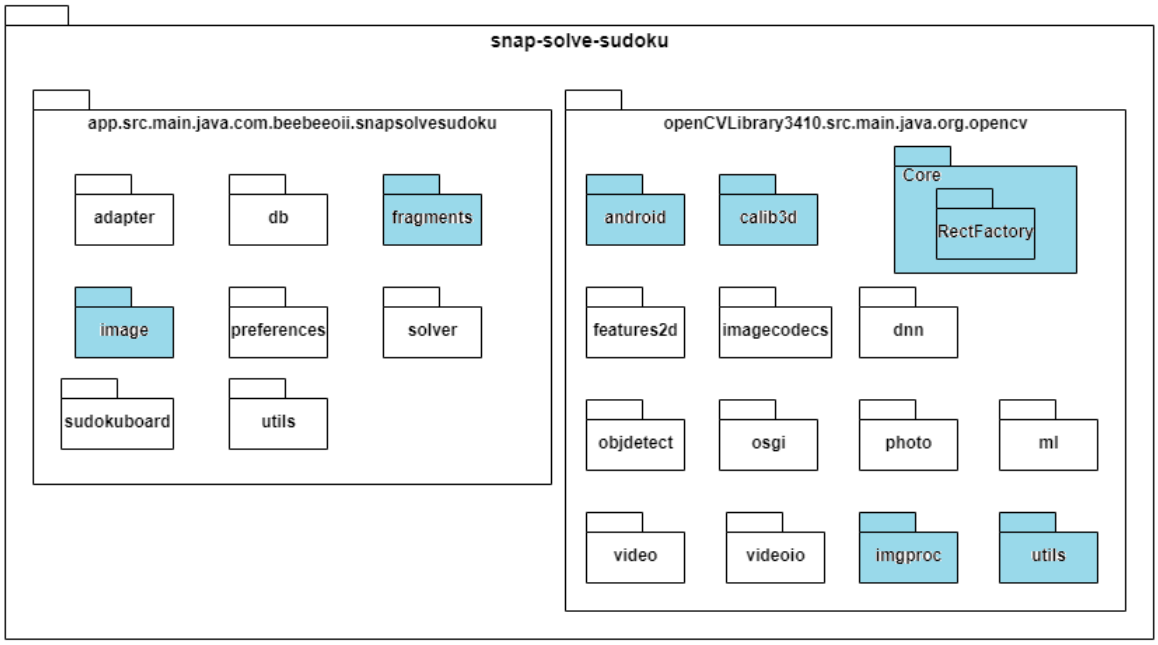

Figure 4.4: Factory Pattern Documentation: Modules Affected

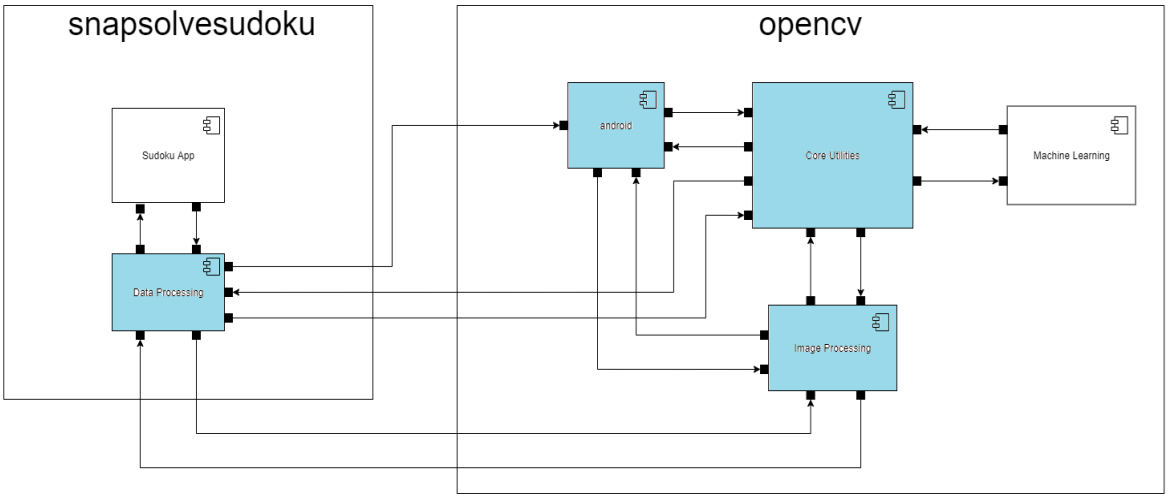

Figure 4.5: Factory Pattern Documentation: Components Affected

4.4. Design Pattern 2: Singleton

A singleton design pattern was implemented in a class called FpsMeter, within the Android package of the openCV library. This class provides a measurement of the speed at which the application is currently running. We deduced that there is no reason to ever have more than a single instance of this class, as the performance it measures is the same for the entire system. Additionally, in the implementation of this class, it is only instantiated to access it's methods, none of which depend on a unique instantiation of the class. By converting FpsMeter to a singleton, it only needs to be instantiated once to give all external references access to it's methods, preventing costly repeated instantiation.

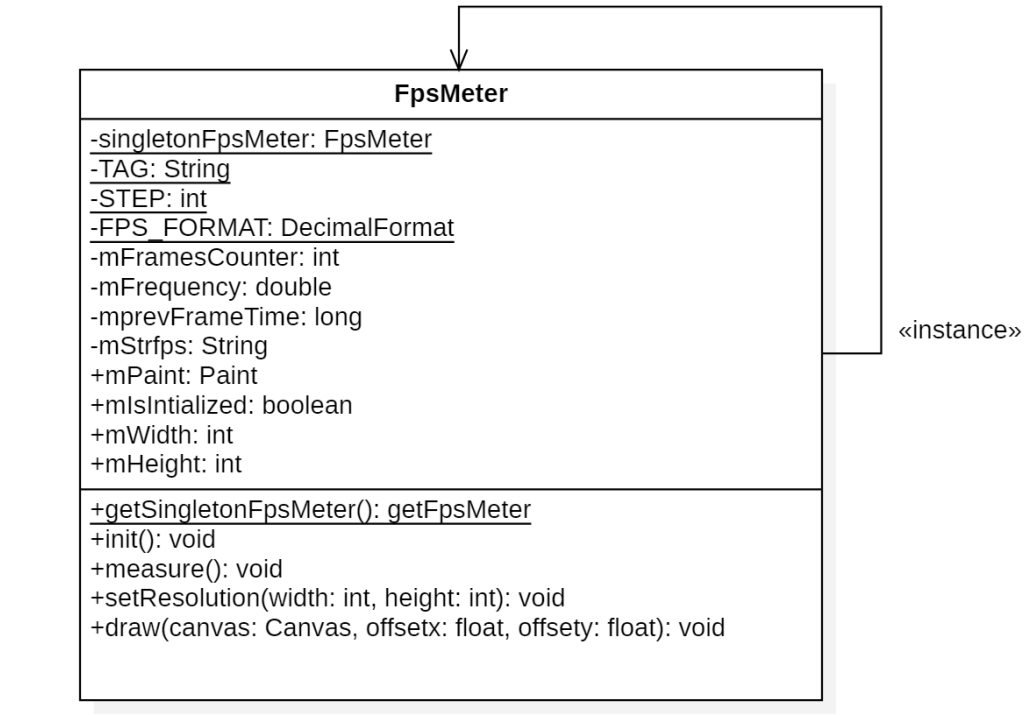

Figure 4.6: Singleton UML

Singleton Pattern Implementation Code:

```
//Showing the changed portions of code implementing the
//singleton pattern

public class FpsMeter {
private static FpsMeter singletonFpsMeter = new FpsMeter();
     public static FpsMeter getSingletonFpsMeter() {
          return singletonFpsMeter;
     }

//This is not the complete implementation of FpsMeter only
//singleton related implementation is shown
}

//Showing the two methods changed to use the singleton pattern.
//within the CameraBridgeViewBase.java
public void enableFpsMeter() {
        if (mFpsMeter == null) {
            mFpsMeter = FpsMeter.getSingletonFpsMeter();
            mFpsMeter.setResolution(mFrameWidth, mFrameHeight);
        }
    }

public void disableFpsMeter() {
     mFpsMeter = null;
}
```

4.5. Change in System Architecture due to Design Pattern 2

While the changes that we made are significant to parts of the system, from an overview, the interaction between high level modules remains the same. Module interaction changed at the level of several classes, but not at a level that is reasonably viewable from a system wide perspective.

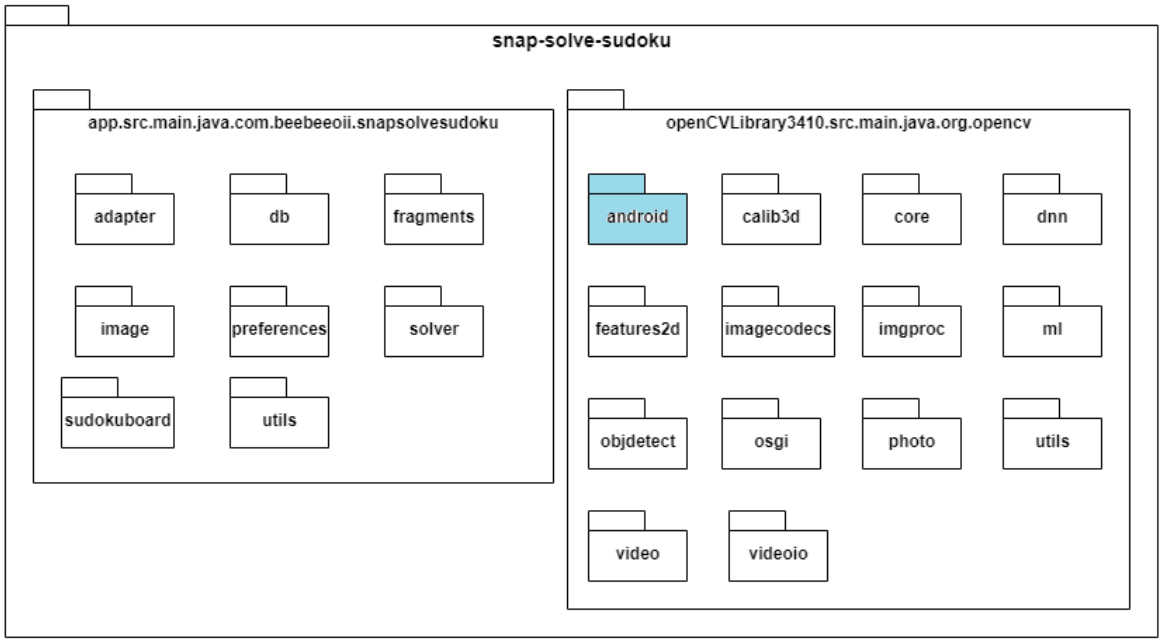

Figure 4.7: Singleton Pattern Documentation: Modules Affected

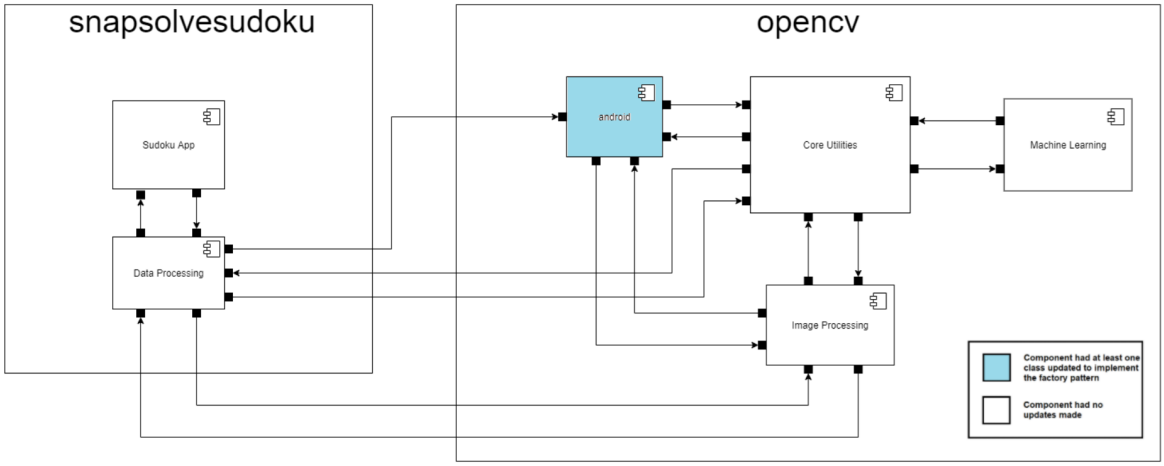

Figure 4.8: Singleton Pattern Documentation: Components Affected

5. Comparison

5.1. Before Refactoring

Both systems provide solutions to sudoku puzzles, but Sudoku16x16Explainer relies on a user to manually input the puzzle while SnapSolveSudoku uses the openCV library to recognize a sudoku board through the camera. This difference constitutes the bulk of the difference between the two systems, they share a similar logical core for solution generation, but SnapSolveSudoku relies on openCV for input recognition. Additionally, Sudoku16x16Explainer provides step by step solutions to puzzles, adding many modules to the solving logic compared to SnapSolveSudoku.

5.2. After Refactor

Our refactorings mainly focused on inefficiency within modules rather than their system wide interaction with each other. Thus, nothing has changed in a comparison between the systems at a system wide level.

6. Conclusion

6.1. Key Findings

As far as we could tell, both of the repositories that we selected were not created by professionals, but by relatively inexperienced amateurs, one of which was in high school. However, even with a seeming lack of experience, when we sought to find spaces to implement new design patterns, we often found they were already present, though often by a different name. It would seem that a number of design patterns arise quite naturally when trying to write quality code, even for someone who calls them by the wrong name.

6.2. Future Work

The modifications made to the openCV library used by snapSolveSudoku could be utilized by other systems that implement it. Additionally, the improvements that we made for both systems would be beneficial for scalability, making it easier for the original author to enhance the scope of their project if they wanted to.

7. Contributions

The sections in this report were authored as follows.

Section 1: Collaboration

Section 2: Collaboration

Section 3 -

- 3.1: Vaibhav Kapoor
- 3.2: Nicholas Lee
- 3.3: Ahmed Amer
- 3.4: Matthew Wells
- 3.5: Matthew Wells

Section 4 -

- 4.1: Vaibhav Kapoor
- 4.2: Benjamin Nielsen
- 4.3: Ahmed Amer
- 4.4: Nicholas Lee
- 4.5: Thomas (Hong) Pan

Section 5 -

- 5.1: Benjamin Nielsen
- 5.2: Thomas (Hong) Pan

Section 6: Collaboration

All Coding was done in peer programming, but only written and committed by one group member per design pattern.