DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

Dokumentation des Projekts

Wortfinder

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik / Informationstechnik

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Toni Einecker

Abgabedatum 17. Mai 2021

Matrikelnummer	3523850
Kurs	Tinf18B3

Inhaltsverzeichnis

Kapitel 1

Unit Tests

Kapitel 2

Programming Principles

Kapitel 3

Clean Architecture

Zum implementieren der Clean Architecture wurde ein UML Diagramm erstellt (Abb. A.1 im Anhang) um die aktuellen Abhängigkeiten zu sehen, dabei wurden ein paar Abhängigkeitspfeile weggelassen welche von weit außen nach innen gehen und die Lesbarkeit zu sehr verringern.

Für die Struktur der Clean Architektur wurde sich auf 3 Schichten festgelegt, da die Funktionsweise des Programms recht simpel ist. Die innerste Schicht beinhaltet alle Entitäten welche zur strukturierten Datenspeicherung verwendet werden. Die mittlere Schicht umfasst die Anwendungslogik. Die äußere Schicht beinhaltet die Interaktion mit allem Außerhalb, d.h. die Controller zum lesen und schreiben der Daten auf der Festplatte sowie die GUI Elemente mit jeweiligen Controllern.

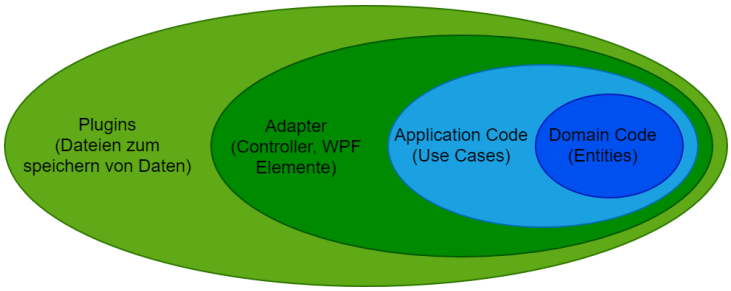

Abbildung 3.1: Aufbau der Schichten der Clean Architecture (link)

Ein großes Problem sind hierbei Abhängigkeiten von Innen nach Außen, wie z.B. die *WordFinder* Klasse (generiert alle möglichen Buchstabenkombinationen in einem Spielfeld), welche direkt auf den *DataController* (nutzt die Wortliste auf der Festplatte) zu greift. Die Abhängigkeit wird durch Dependency Inversion, d.h. die Erstellung eines Interfaces welches die äußere Klasse implementiert und die innere nutzt, umgedreht (link zum Commit):

Wenn innere Klassen äußere erzeugen müssen, wie es beim *WordFinder* der Fall ist, wird das Erzeugungsmuster der abstrakten Fabrik verwendet. Dem *WordFinder* wird dann im Konstruktor eine Referenz auf die Fabrik gegeben. Über die Funktion "GetDataController()"gibt die Fabrik dann eine neue Instanz des *DataController* zurück, wobei der Rückgabetyp im Interface der Fabrik als *IDataController* Deklariert ist (link zum Commit):

Außerdem problematisch ist der grundsätzliche Aufbau der Programms sowie speziell der Aufbau des Hauptfensters. Dieses bestand aus einem Grid welches durch den *FieldGenerator* mit Teilfenstern (*LetterBox*, zeigen die Buchstaben an) gefüllt wurde. Jeder *LetterBox* wurde

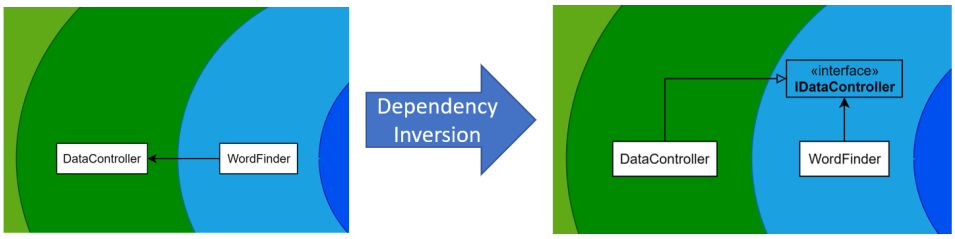

Abbildung 3.2: Beispiel der Dependency Inversion (link)

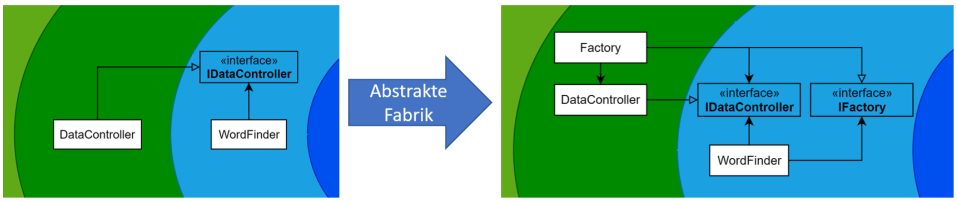

Abbildung 3.3: Beispiel der Dependency Inversion (link)

die Instanz des *WordBuilder* gegeben, welchen sie aufrufen wenn sie angeklickt werden.

Zur Verbesserung der Trennung zwischen dem Application Code und der GUI, wird das Grid direkt im Code Behind des *MainWindow* durch kaskadieren von speziellen WPF Elementen aufgebaut. Die Ansteuerung des *MainWindow* erfolgt dann über einen neuen *MainWindowController* dem im Aufruf nur die Größe des Spielfelds und die Buchstaben übergeben werden. Somit erfolgt die Erstellung der GUI separiert vom restlichen Anwendungscode.

Vor der Implementierung wurde zuerst das erstellte UML Diagramm abgeändert. Zuerst wurde dabei die Stuktur geändert, dann alle Abhängigkeitspfeile welche nach außen zeigen mittels Dependency Inversion umgedreht und als letztes an notwendigen Stellen eine Abstrakte Fabrik hinzugefügt. Das daraus entstandene Diagramm ist im Anhang als Abbildung **??**.

Mit diesen Vorgaben wurde das Diagramm umstrukturiert, sodass alle Klassen an ihrem sinnvollsten Ort sind. Da das Programm vorher ständig mit Funktionen ergänzt wurde bei denen lediglich Wert auf Funktionalität gelegt wurde, sind im Nachhinein betrachtet unnötig komplizierte Strukturen entstanden. Diese werden daher bei der Umstrukturierung des Diagramms zur Clean Architecture auch gleich geändert. Ein Beispiel hierfür ist die *FindableWords* Klasse, welche alle im aktuelle laufenden Spiel findbaren Wörter beinhaltet. Diese wurde erst später hinzugefügt, da davor direkt jedes Wort in einer Wortliste mit gültigen Wörtern überprüft wurde. Sowie die *GameGrid* Klasse, welche (unter anderem) die Größe und Buchstaben des Spielfelds beinhaltet. Die findbaren Wörter sowie die Buchstaben im aktuellen Spiel wurden in eine neue Klasse *Game* verschoben welche alle Daten über Spiel enthält. Somit können Spiele auch im Vorhinein generiert und dann geladen werden.

Kapitel 4

Code Smell Refactoring

4.0.1 Long Method

Die Klasse *WordGenerator* ist dafür zuständig alle möglichen Wörter in einem Raster von Buchstaben zu finden. Dafür hat sie zwei Funktionen welche jeweils ca. 50 Zeilen lang sind. Beide Funktionen sind durch viele Schleifen sehr unübersichtlich und fallen unter die Long Method Code Smells. Zum Refactoren des Code Smells wird die Funktion zuerst mittels Extract Method in kleinere Funktionen unterteilt. Dabei wurde direkt noch die If-Abfrage zur besseren Lesbarkeit umgedreht. Die extrahierten Methoden sind in Abb.4.1 in Grün und Orange markiert.(Link zum Commit)

Abbildung 4.1: Anwendung des Extract Method Refactoring (link)

Danach fallen die zwei gleichen Schachtelungen der for-Schleifen auf. Dabei wird in *LettersAs2DArray* das Array zu einem 2D Array umstrukturiert, da die Buchstaben in einen Grid angeordnet sind ist so ein Iterieren und suchen von Nachbarn einfacher vorstellbar. Da zwischenzeitlich allerdings die *Coordinate* Klasse eingefügt wurde, welche das Prüfen auf Nachbarschaft übernimmt, lässt sich dieser Code vereinfachen. Das durchlaufen des 2D Arrays (Abb.4.2 Orange markiert) wird somit durch ein einmaliges durchlaufen des Ursprünglichen Arrays ersetzt. Die *LettersAs2DArray* Funktion wird hier nur noch für den Aufruf an die nächste Funktion benötigt und kann nach dessen Refactorings komplett entfernt werden. Außerdem kann die, in Grün markierte, Funktionalität welche überprüft ob ein Wort schon in der List ist ebenfalls durch Extract Method ausgelagert werden. Es ergibt sich somit folgender Code (Link zum Commit):

Abbildung 4.2: Extract Method und Code Vereinfachung (link)

In der zweiten Funktion wird ebenfalls mit Extract Method begonnen (Abb.4.3). Dabei werden zwei verschachtelte Schleifen ausgelagert, in welchen die betroffene Methode Rekursiv wieder aufgerufen wird. Da bei der Rekursion alle Parameter mitgegeben werden müssen ist ein komplettes auslagern unvorteilhaft, da in diese neue Methode dann ebenfalls alle Parameter weitergegeben werden müssten. Daher wird nur der Teil in eine neue Methode verschoben (Orange markiert), welcher alle benachbarten Positionen im Buchstabengitter findet (Link zum Commit):

Danach wird auch hier das zweidimensionale Array durch ein eindimensionales ersetzt. Außerdem lässt sich das innerste foreach durch die Nutzung von der *AddWords* Funktion ersetzen, welche durch Extract Method in der ersten Funktion entstand.

Die Insgesamt Länge des Codes hat sich durch Anwenden der Refactorings nicht verkürzt. Dafür wurde die Lesbarkeit sowie die Wiederverwendbarkeit wesentlich gesteigert. Die fertigen Funktionen sind nachfolgenden Abgebildet (Link zum Commit):

```
public List<Word> CheckRecusive(string word, char[,] letters, List<Coordinate> coordinates, int dic
{
    List<Word> allWords = new List<Word>();
    int currentRow = coordinates[^1].Row;
    int currentColumn = coordinates[^1].Column;
    if (letters[currentRow, currentColumn] != '-')
    {
        int columns = letters.GetLength(0);
        int rows = letters.Length / columns;

        // Add letter to word
        word += letters[currentRow, currentColumn];
        letters[currentRow, currentColumn] = '-';

        // Check if any word begins with this letter string
        int beginningIndex = wordList.FindBeginningLinear(word, dictStartIndex);
        if (beginningIndex >= 0)
        {
            if (wordList.CheckWord(word, beginningIndex))
            {
                allWords.Add(new Word(word, new List<Coordinate>(coordinates)));
            }
            // Add all attached letters and repeat
            for (int i = -1; i <= 1; i++)
            {
                for (int j = -1; j <= 1; j++)
                {
                    int nextRow = currentRow + i;
                    int nextColumn = currentColumn + j;
                    // If cell is in bound
                    if (nextRow < rows && nextRow >= 0 && nextColumn < columns && nextColumn >= 0)
                    {
                        List<Coordinate> newCoordinates = new List<Coordinate>(coordinates)
                        {
                            new Coordinate(nextRow, nextColumn)
                        };
                        foreach (Word foundWord in CheckRecusive(word, (char[,])letters.Clone(), ne
                        {
                            allWords.Add(foundWord);
                        }
                    }
                }
            }
        }
    }
    return allWords;
}
```

```
public List<Word> CheckRecusive(string word, int size, char[,] letters, List<Coordinate> coordinates, int dict
{
    List<Word> allWords = new List<Word>();
    int currentRow = coordinates[^1].Row;
    int currentColumn = coordinates[^1].Column;
    if (letters[currentRow, currentColumn] != '-')
    {
        int columns = letters.GetLength(0);
        int rows = letters.Length / columns;

        // Add letter to word
        word += letters[currentRow, currentColumn];
        letters[currentRow, currentColumn] = '-';

        // Check if any word begins with this letter string
        int beginningIndex = wordList.FindBeginningLinear(word, dictStartIndex);
        if (beginningIndex >= 0)
        {
            if (wordList.CheckWord(word, beginningIndex))
            {
                allWords.Add(new Word(word, new List<Coordinate>(coordinates)));
            }

            List<Coordinate> neightbourCoordinates = GetNeighbourCoordinates(currentRow, currentColumn, size);
            foreach(Coordinate coordinate in neightbourCoordinates)
            {
                List<Coordinate> newCoordinates = new List<Coordinate>(coordinates) { coordinate };
                foreach (Word foundWord in CheckRecusive(word, size, (char[,])letters.Clone(), newCoordinates,
                {
                    allWords.Add(foundWord);
                }
            }
        }
    }
    return allWords;
}
1 reference
private List<Coordinate> GetNeighbourCoordinates(int row, int column, int size)
{
    List<Coordinate> coordinates = new List<Coordinate>();
    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            int nextRow = row + i;
            int nextColumn = column + j;
            if (IsInBound(nextRow, nextColumn, size))
            {
                coordinates.Add(new Coordinate(nextRow, nextColumn));
            }
        }
    }
    return coordinates;
}
1 reference
private bool IsInBound(int row, int column, int size)
{
    return row < size && row >= 0 && column < size && column >= 0;
}
```

Abbildung 4.3: Extract Method (link)

```
public List<Word> GetAllWords(char[] letters, int fieldSize)
{
    if (wordList.Loaded())
    {
        List<Word> allWords = new List<Word>();
        for(int i = 0; i < letters.Length; i++)
        {
            int row = i / fieldSize;
            int column = i % fieldSize;
            List<Coordinate> coordList = new List<Coordinate> { new Coordinate(row, column) };
            List<Word> newWords = FindWordsRecusive("", fieldSize, (char[])letters.Clone(), coordList, 0);
            AddWords(newWords, allWords);
        }
        return allWords;
    }
    return new List<Word>();
}
2 references
private void AddWords(List<Word> newWords, List<Word> allWords)
{
    foreach (Word word in newWords)
    {
        if (WordNotFound(word, allWords))
        {
            allWords.Add(word);
        }
    }
}
1 reference
private bool WordNotFound(Word newWord, List<Word> allWords)
{
    foreach (Word existingWord in allWords)
    {
        if (existingWord.Name.Equals(newWord.Name))
        {
            return false;
        }
    }
    return true;
}
```

```
public List<Word> FindWordsRecusive(string word, int size, char[] letters, List<Coordinate> coordinates, int dictStartIndex)
{
    List<Word> allWords = new List<Word>();
    int position = coordinates[^1].Row * size + coordinates[^1].Column;
    if (letters[position] != '-')
    {
        word += letters[position];
        letters[position] = '-';
        int beginningIndex = wordList.FindBeginningLinear(word, dictStartIndex);
        if (beginningIndex >= 0)
        {
            if (wordList.CheckWord(word, beginningIndex))
            {
                allWords.Add(new Word(word, new List<Coordinate>(coordinates)));
            }
            List<Coordinate> neightbourCoordinates = GetNeighbourCoordinates(position, size);
            foreach(Coordinate coordinate in neightbourCoordinates)
            {
                List<Coordinate> newCoordinates = new List<Coordinate>(coordinates) { coordinate };
                List<Word> words = FindWordsRecusive(word, size, (char[])letters.Clone(), newCoordinates, beginningIndex);
                AddWords(words, allWords);
            }
        }
    }
    return allWords;
}
1 reference
private List<Coordinate> GetNeighbourCoordinates(int position, int size)
{
    List<Coordinate> coordinates = new List<Coordinate>();
    int row = position / size;
    int column = position % size;
    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            int nextRow = row + i;
            int nextColumn = column + j;
            if (IsInBound(nextRow, nextColumn, size))
            {
                coordinates.Add(new Coordinate(nextRow, nextColumn));
            }
        }
    }
    return coordinates;
}
1 reference
private bool IsInBound(int row, int column, int size)
{
    return row < size && row >= 0 && column < size && column >= 0;
}
```

Abbildung 4.4: Long Method Code Smell nach dem Refactoring (link)

4.0.2 Duplicated Code

Es werden sowohl vorgeladenen Spiele, wie auch Highscores auf der Festplatte lokal gespeichert und beides soll nicht editierbar sein. Daher lädt und speichert die Klasse *ScoreDataController* die Highscores sowie *GameDataController* die vorbereiteten Spiele. Beide beinhalten dabei den gleichen Code für die Ent-/Verschlüsselung. Dieser Duplicated Code wird mittels Extract Method in eine neue Klasse ausgelagert welche die Ent- und Verschlüsselung übernimmt. (Commit 86e635fc6ddbd436ca012c21e3a00b9246248855)

Kapitel 5

Entwurfsmuster

Anhang A

Abbildungen

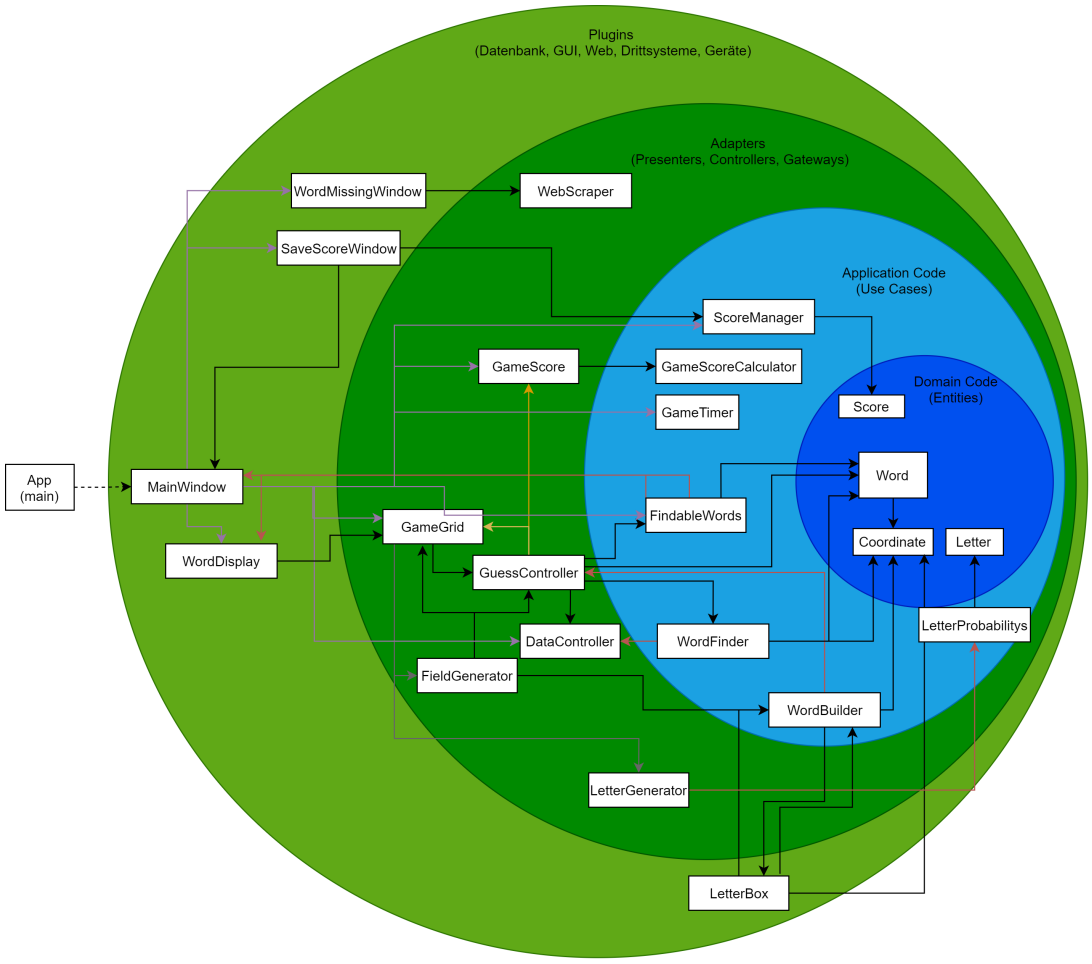

Abbildung A.1: UML Klassendiagramm vor der Clean Architecture (link)

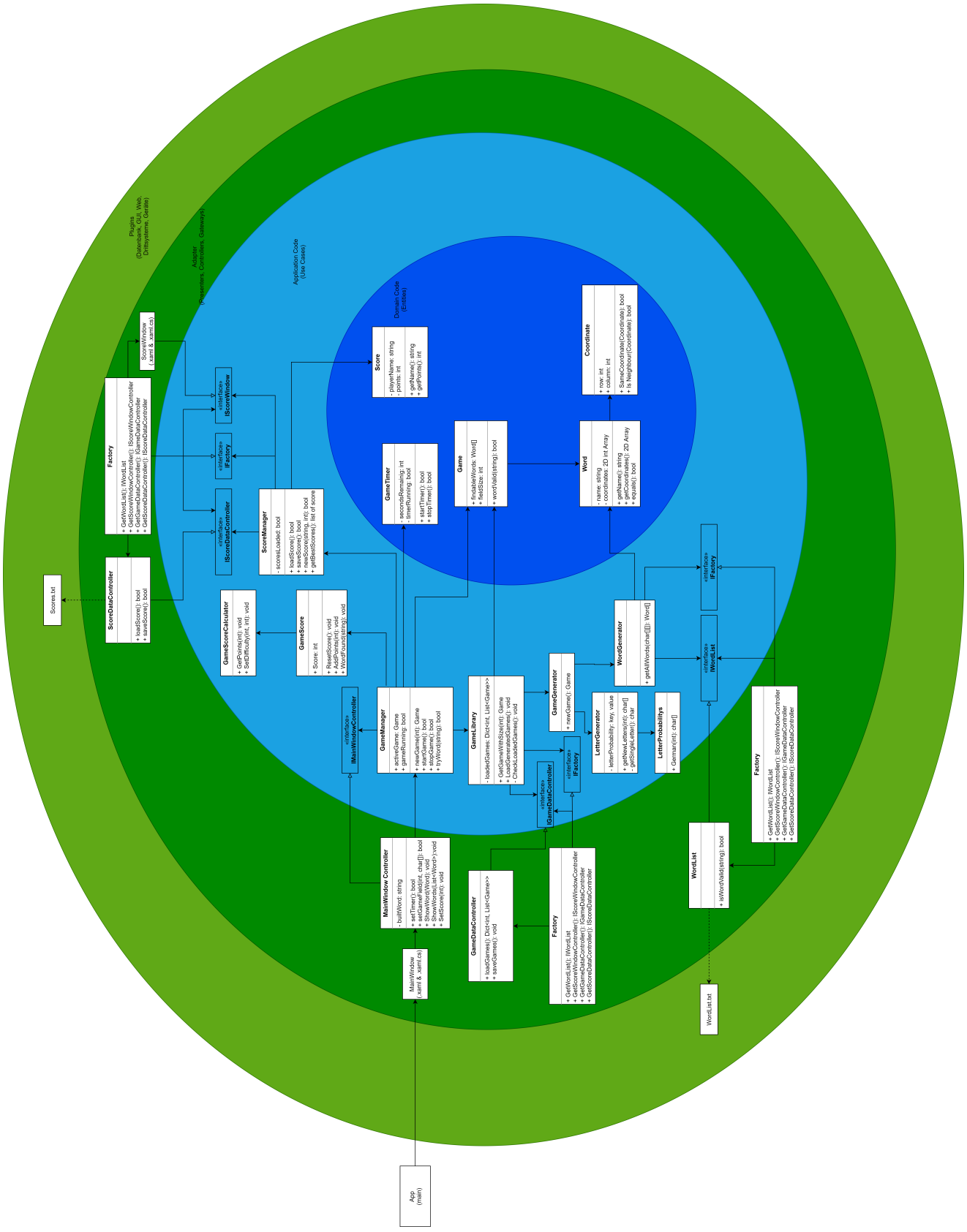

Abbildung A.2: UML Diagramm nach der Clean Architecture (link)