# Technical Design 30 Seconds

Minka Firth AP2

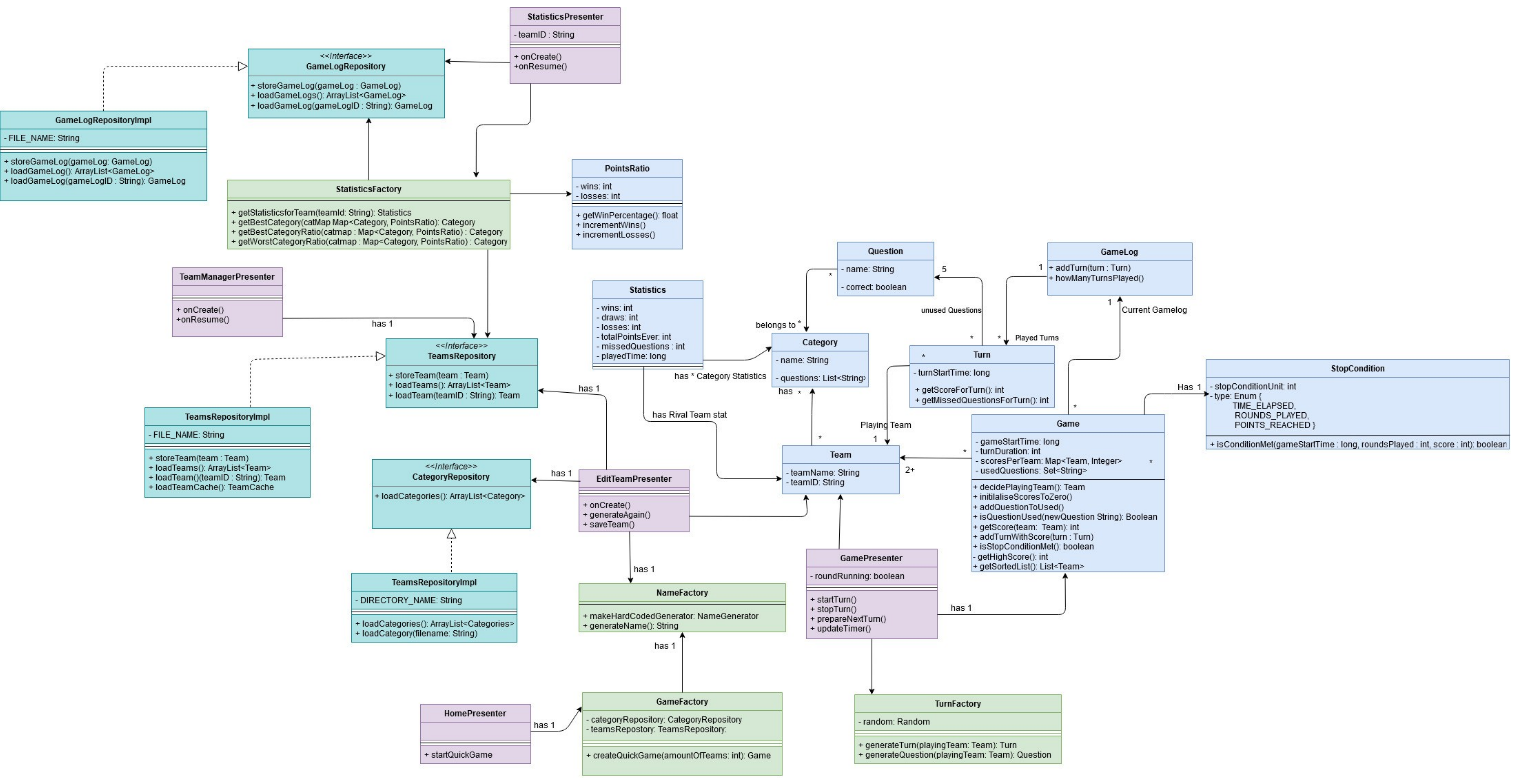
Name: Minka Firth AP2
Date: November 2020
Semester coach: B.S.H.T. Michielsen, M.W.A. Meulenbroeks

## Table of Contents

| Version | Date | Teachers | Author | changes |
|---|---|---|---|---|
| **1.0** | 13-10-2020 | B.S.H.T. Michielsen M.W.A Meulenbroeks | M.Firth | • First version that includes Flow Charts and Architectural Structure Diagram |
| **2.0** | 23-10-2020 | B.S.H.T. Michielsen M.W.A Meulenbroeks | M.Firth | • Added Class Diagram and description • Added Persistance of Data • Added Legend to Flow Chart |
| **2.1** | | B.S.H.T. Michielsen M.W.A Meulenbroeks | M.Firth | • Interfaces definition |
| **3.0** | 07-01-2021 | B.S.H.T. Michielsen M.W.A Meulenbroeks | M.Firth | • Overhaul of class diagrams. • Added chapter for modularity • Added snippets of code for Algorithmic process. |
| **3.1** | 12-01-2021 | B.S.H.T. Michielsen M.W.A Meulenbroeks | M.Firth | • Updated Class Diagrams. • Updated Modularity. |
| **4.0** | 19-01-2021 | B.S.H.T. Michielsen M.W.A Meulenbroeks | M.Firth | • Updated Modularity after adding a fourth module. • Added Chapter for DesignPatters. • Updated first two Class Diagrams. |

# Class Diagram

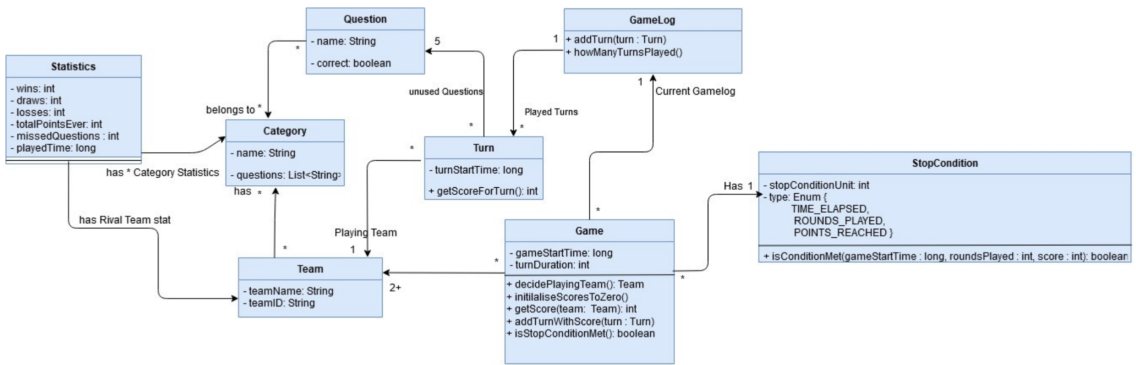
Below is a Diagram of the application. While making this, it quickly became clear that it would become a confusing diagram. I have therefore decided to split it into multiple diagrams. Even so, I have decided to include this diagram in the document, to give a visual representation of how the various classes interact with each other. The blue classes are the Data classes and the green classes are Behaviour classes. The purple classes are the Presenters. These presenters all have their own Activity and View interfaces (see other diagrams). The teal coloured are interfaces and their implementations.


StatisticsPresenter
- teamID : String
+ onCreate()
+onResume()
<<Interface>>
GameLogRepository
+ storeGameLog(gameLog : GameLog)
+ loadGameLogs(): ArrayList<GameLog>
+ loadGameLog(gameLogID : String): GameLog
GameLogRepositoryImpl
- FILE_NAME: String
+ storeGameLog(gameLog: GameLog)
+ loadGameLog(): ArrayList<GameLog>
+ loadGameLog(gameLogID : String): GameLog
StatisticsFactory
+ getStatisticsforTeam(teamId: String): Statistics
+ getBestCategory(catMap Map<Category, PointsRatio): Category
+ getBestCategoryRatio(catmap : Map<Category, PointsRatio) : Category
+ getWorstCategoryRatio(catmap : Map<Category, PointsRatio) : Category
PointsRatio
- wins: int
- losses: int
+ getWinPercentage(): float
+ incrementWins()
+ incrementLosses()
TeamManagerPresenter
+ onCreate()
+onResume()
has 1
<<Interface>>
TeamsRepository
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam(teamID : String): Team
TeamsRepositoryImpl
- FILE_NAME: String
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam()(teamID : String): Team
+ loadTeamCache(): TeamCache
<<Interface>>
CategoryRepository
+ loadCategories(): ArrayList<Category>
TeamsRepositoryImpl
- DIRECTORY_NAME: String
+ loadCategories(): ArrayList<Categories>
+ loadCategory(filename: String)
has 1
has 1
EditTeamPresenter
+ onCreate()
+ generateAgain()
+ saveTeam()
has 1
NameFactory
+ makeHardCodedGenerator: NameGenerator
+ generateName(): String
has 1
HomePresenter
+ startQuickGame
has 1
GameFactory
- categoryRepository: CategoryRepository
- teamsRepostory: TeamsRepository
+ createQuickGame(amountOfTeams: int): Game
Statistics
- wins: int
- draws: int
- losses: int
- totalPointsEver: int
- missedQuestions : int
- playedTime: long
has * Category Statistics
has Rival Team stat
Question
- name: String
- correct: boolean
*
5
belongs to *
Category
- name: String
- questions: List<String>
has *
unused Questions
GameLog
+ addTurn(turn : Turn)
+ howManyTurnsPlayed()
1
1
Current Gamelog
*
*
Played Turns
Turn
- turnStartTime: long
+ getScoreForTurn(): int
+ getMissedQuestionsForTurn(): int
StopCondition
- stopConditionUnit: int
- type: Enum {
TIME_ELAPSED,
ROUNDS_PLAYED,
POINTS_REACHED }
+ isConditionMet(gameStartTime : long, roundsPlayed : int, score : int): boolean
Has 1
Playing Team
1
*
Team
- teamName: String
- teamID: String
2+
*
Game
- gameStartTime: long
- turnDuration: int
- scoresPerTeam: Map<Team, Integer>
- usedQuestions: Set<String>
*
+ decidePlayingTeam(): Team
+ initilaliseScoresToZero()
+ addQuestionToUsed()
+ isQuestionUsed(newQuestion String): Boolean
+ getScore(team: Team): int
+ addTurnWithScore(turn : Turn)
+ isStopConditionMet(): boolean
- getHighScore(): int
+ getSortedList(): List<Team>
GamePresenter
- roundRunning: boolean
+ startTurn()
+ stopTurn()
+ prepareNextTurn()
+ updateTimer()
has 1
TurnFactory
- random: Random
+ generateTurn(playingTeam: Team): Turn
+ generateQuestion(playingTeam: Team): Question

## Data

These Object classes are mostly connected with regular association arrows. This is because most of these relations are just a visualisation of a field. Instead of saying that object *Game* has a List of Played Rounds, I have drawn an association arrow (as in accordance with Bazzz). Naturally, all of these fields within the object classes are private.  I will go into detail for a few of these classes where I feel clarification is needed.

The most important one of these is **Game**. It has a List of Playing Teams and a **GameLog**, called *currentGameLog*. The long, *startTime,* is timestamp, this is to keep track of when each game started. It also has an integer, called *roundDuration*.  This int is given by the user when starting a normal game. It decides how long each round should last.



So far, **Game** only has one method: *decidePlayingTeam().* Teams take turns, which means that each **Round** the application has to decide what Team is playing. In **GameLog** is a method called, *howManyRoundsPlayed().* This method returns an integer that represents how many rounds have been played.  *DecidePlayingTeam()* then takes this int and the size of the List of teams and then uses the remainder operation to figure out which team should be playing. (eg. if there has been 4 rounds, en 3 teams, the remainder would be 1. This means the first team is playing.)

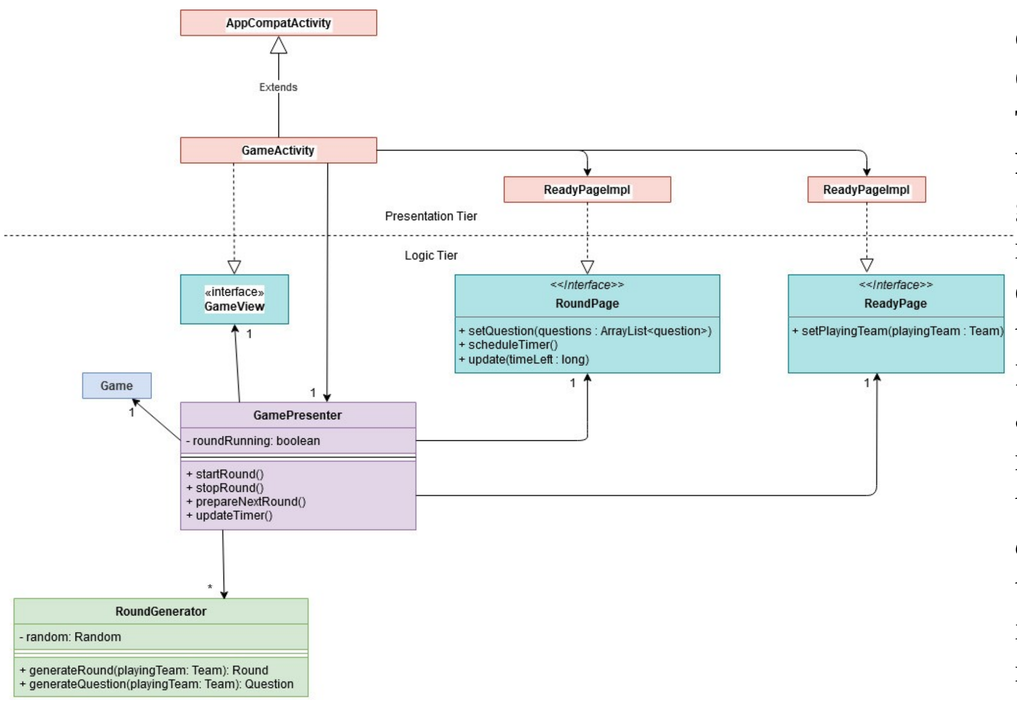
**GameLog** consists of a List of Played Rounds and a List of used Questions. The List of questions is necessary to avoid duplicate questions in the Game. The list of played rounds is important to keep track of what Rounds have been played (with what team, what questions were guessed correctly and by what team.) **Gamelog** has two more methods. The first one, *addRound()* adds a new Round to the List of played Rounds. The last method, *getMostRecentRound(),* is important to figure out the Timer for each Round.

In a previous version, I had an Enum called StopConditionType and each **Game** had a stopConditionUnit (int). These were to figure out how long the game would last. (eg. a game would end when either an X amount of rounds are played, X amount of time has passed or, X amount of points have been reached). This is why it's important that each Game has a timestamp. Because these two are always used together and don't mean much by themselves, I have decided to turn them into one class. The StopCondition (both unit and type) are specified by the user (except for a quickGame, in which case the game assumes you want to play for 30 points).

While one could argue that it is more important that a Category is a list of Questions. It is also important to keep in mind that each Question has (at least) one Category. This is important, because I want the game to be able to keep "Statistics" of previous games. Aside from having a Category, each Question has a string(its name) and a boolean. This boolean, "correct" will be set to false at the beginning at each round. When a player has guessed the question correctly, it will be set to true. This way, when **GameLog** keeps a list of used Questions, it will automatically know if they were guessed correctly.

## Presenter Diagrams

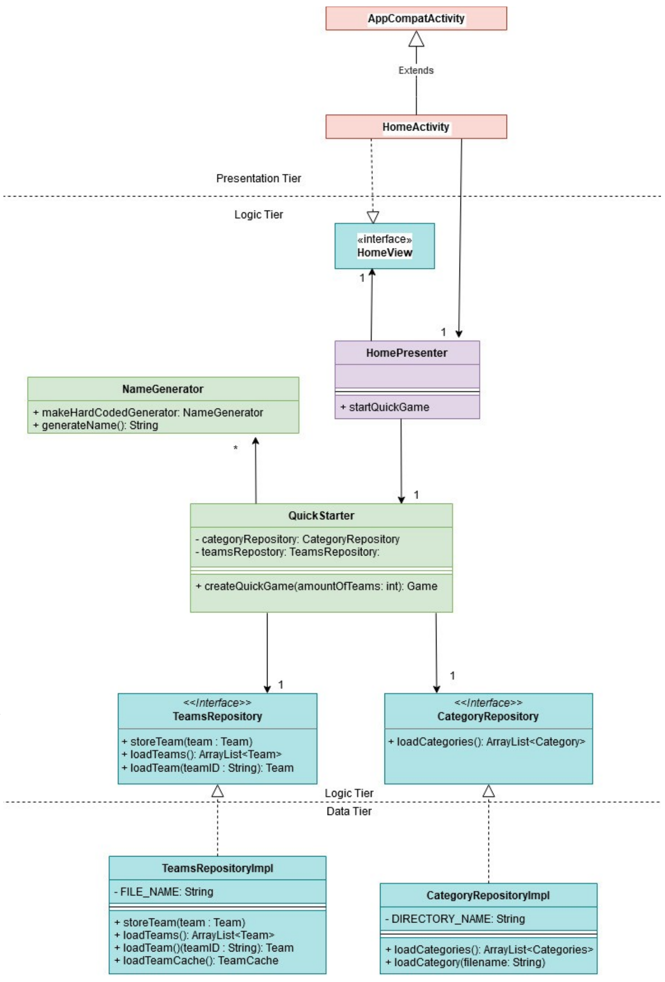
Because I am making an android application, I wanted to make sure I take care of the dependency inversion (see architectural design for more information). For this reason I made separate diagrams for the Presenters. For example, the diagram below is the diagram for the **GamePresenter.** This presenter is the "middle-man" between the data and logic, and the GUI. There are two "pages" connected to the Activity, as these similar to activities (in the way that they display data to the user), I've made interfaces to connect them to the Presenter. This way the presenter isn't actually communicating with the UI tier.



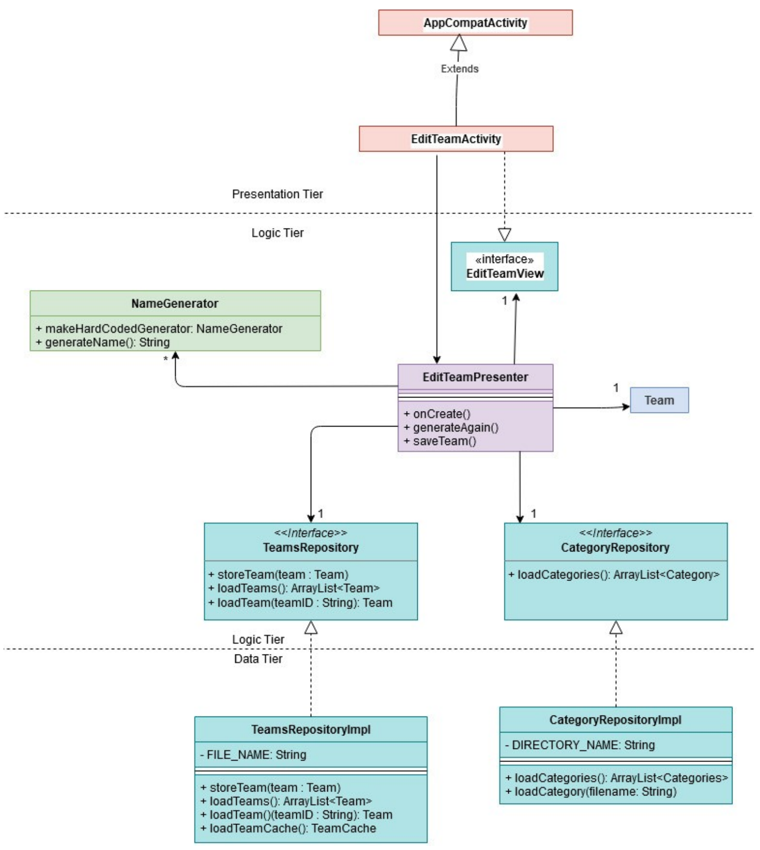
The **GamePresenter** also has one **Game** (this is the same Game from the data diagram). This **Presenter** has a **RoundGenerator.** This class generates the questions for each round. Then with those questions, a playing team and a timestamp, it will generate a Round. The **Presenter** is then able to start, stop and prepare rounds. *PrepareRounds()* uses the method, *decidePlayingTeam()*. The last thing the **GamePresenter** does is *updateTimer()*. This Method makes the actual 30 seconds timer.

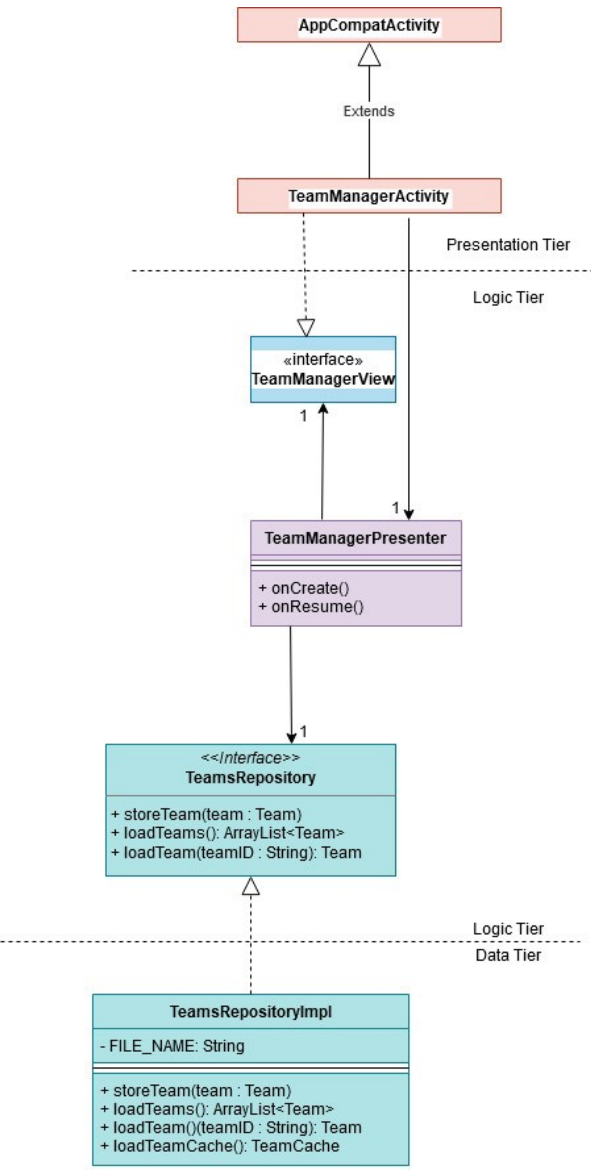
The **HomePresenter** is the only presenter with the logic class, **QuickStarter**. As I've mentioned, a QuickGame is like a thirty seconds game, but instead of users selecting teams with categories, the user just gives the application the *AmountOfTeams* it will be playing with. The **QuickStarter** will use this integer to make X amount of teams, using the **NameGenerator**. The other settings are all determined by the application (round timer, stopcondition etc.) This QuickStarter will save the created teams to the TeamsRepository. It will also load all the categories from the CategoryRepository (see Interfaces chapter for more information).

**NameGenerator** is a very simple logic class with three lists.  The **NameGenerator** randomly chooses a random String from all three lists to generate a TeamName.


AppCompatActivity
Extends
HomeActivity
Presentation Tier
Logic Tier
«interface» HomeView
HomePresenter
+ startQuickGame
NameGenerator
+ makeHardCodedGenerator: NameGenerator
+ generateName(): String
QuickStarter
- categoryRepository: CategoryRepository
- teamsRepostory: TeamsRepository:
+ createQuickGame(amountOfTeams: int): Game
<<Interface>> TeamsRepository
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam(teamID : String): Team
<<Interface>> CategoryRepository
+ loadCategories(): ArrayList<Category>
Logic Tier
Data Tier
TeamsRepositoryImpl
- FILE_NAME: String
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam()(teamID : String): Team
+ loadTeamCache(): TeamCache
CategoryRepositoryImpl
- DIRECTORY_NAME: String
+ loadCategories(): ArrayList<Categories>
+ loadCategory(filename: String)
1
1
1
*
1
1

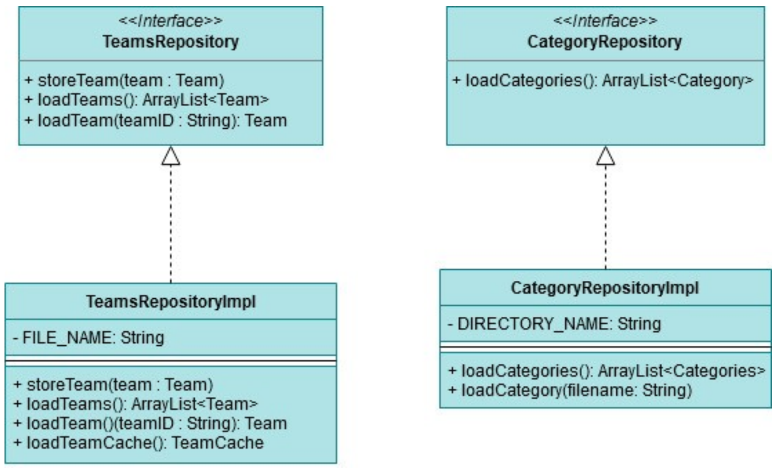
The next Presenter is the **EditTeamPresenter**. This presenter makes it possible to save new Teams and edit already existing Teams. If the Presenter has been given a TeamID, it knows what to Team to edit. If no TeamID was given, it will simply make a new Team. The moment it makes a Team, it will load all the categories from the **CategoryRepository**, and the user will have the opportunity to select the categories they will play with. It stores the new/edited Team to the **TeamsRepository**.


AppCompatActivity
Extends
EditTeamActivity
Presentation Tier
Logic Tier
«interface» EditTeamView
1
NameGenerator
+ makeHardCodedGenerator: NameGenerator
+ generateName(): String
*
EditTeamPresenter
+ onCreate()
+ generateAgain()
+ saveTeam()
1
Team
1
1
<<Interface>> TeamsRepository
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam(teamID : String): Team
<<Interface>> CategoryRepository
+ loadCategories(): ArrayList<Category>
Logic Tier
Data Tier
TeamsRepositoryImpl
- FILE_NAME: String
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam()(teamID : String): Team
+ loadTeamCache(): TeamCache
CategoryRepositoryImpl
- DIRECTORY_NAME: String
+ loadCategories(): ArrayList<Categories>
+ loadCategory(filename: String)

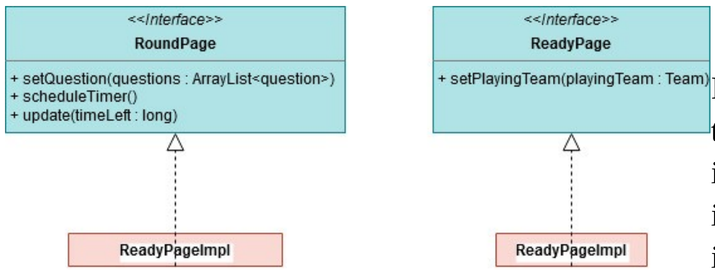


The last of these Presenters is the **TeamManagerPresenter** (below). This is probably the simplest of the presenters, because its only function is to load the teams from the **TeamsRepository.** In the activity it will show all the Teams and these are selectable for the user to edit them in the **EditTeamPresenter.**

## Interfaces Definition


<<Interface>>
TeamsRepository
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam(teamID : String): Team
<<Interface>>
CategoryRepository
+ loadCategories(): ArrayList<Category>
TeamsRepositoryImpl
- FILE_NAME: String
+ storeTeam(team : Team)
+ loadTeams(): ArrayList<Team>
+ loadTeam()(teamID : String): Team
+ loadTeamCache(): TeamCache
CategoryRepositoryImpl
- DIRECTORY_NAME: String
+ loadCategories(): ArrayList<Categories>
+ loadCategory(filename: String)

In the previous Chapter I mentioned several interfaces without going into detail. I will be doing that here. These interfaces are very important, as they are the classes that saves and loads the Teams and Categories from and to Json Files. The reason I've decided to make interfaces out of these classes is to avoid dependencies between different tiers. The implementation of these interfaces are the classes from the Data Tier, while the parent is actually in the logic Tier. The actual interface has these methods that are listed, but the implementation of these interfaces actually make it work.


<<Interface>>
RoundPage
+ setQuestion(questions : ArrayList<question>)
+ scheduleTimer()
+ update(timeLeft : long)
ReadyPageImpl
<<Interface>>
ReadyPage
+ setPlayingTeam(playingTeam : Team)
ReadyPageImpl

The same can be said about the "Page Interfaces". The interfaces themselves are in the logic tier of the application, while the implementation of these interfaces exist only in the presentation tier. As you can see, these interfaces mostly deal with setting the right information into the Textviews on the page. The RoundPage also has important methods dealing with the visual aspects of the 30 seconds timer. Every second the Timer schedules itself to update again in a second. This way every second is actually counting down from 30.

## SOLID

**Single-responsibility principle**

As soon as notice that I might be repeating several functions in a certain combination, I will try to combine these in a separate class. A good example for this is in the

**Open-closed principles**

This seems more like a principle that should be applied to bigger projects with multiple independent modules, so there will be room for expansion without having to modify the source. In my showcase I am using a JSON file to store the settings, which means that I can change these settings without having to recompile the code.

**Liskov substitution Principle**

The Liskov substitution principle has always baffled me a bit. I thought that the whole point of an interface is, is that it can be replaced by its implementing classes. Therefore it seems that each implementing class should have the same functions as the Interface (it might have additional functions, but it should be able to do all the initial functions as well).

**Interface segregation Principles**

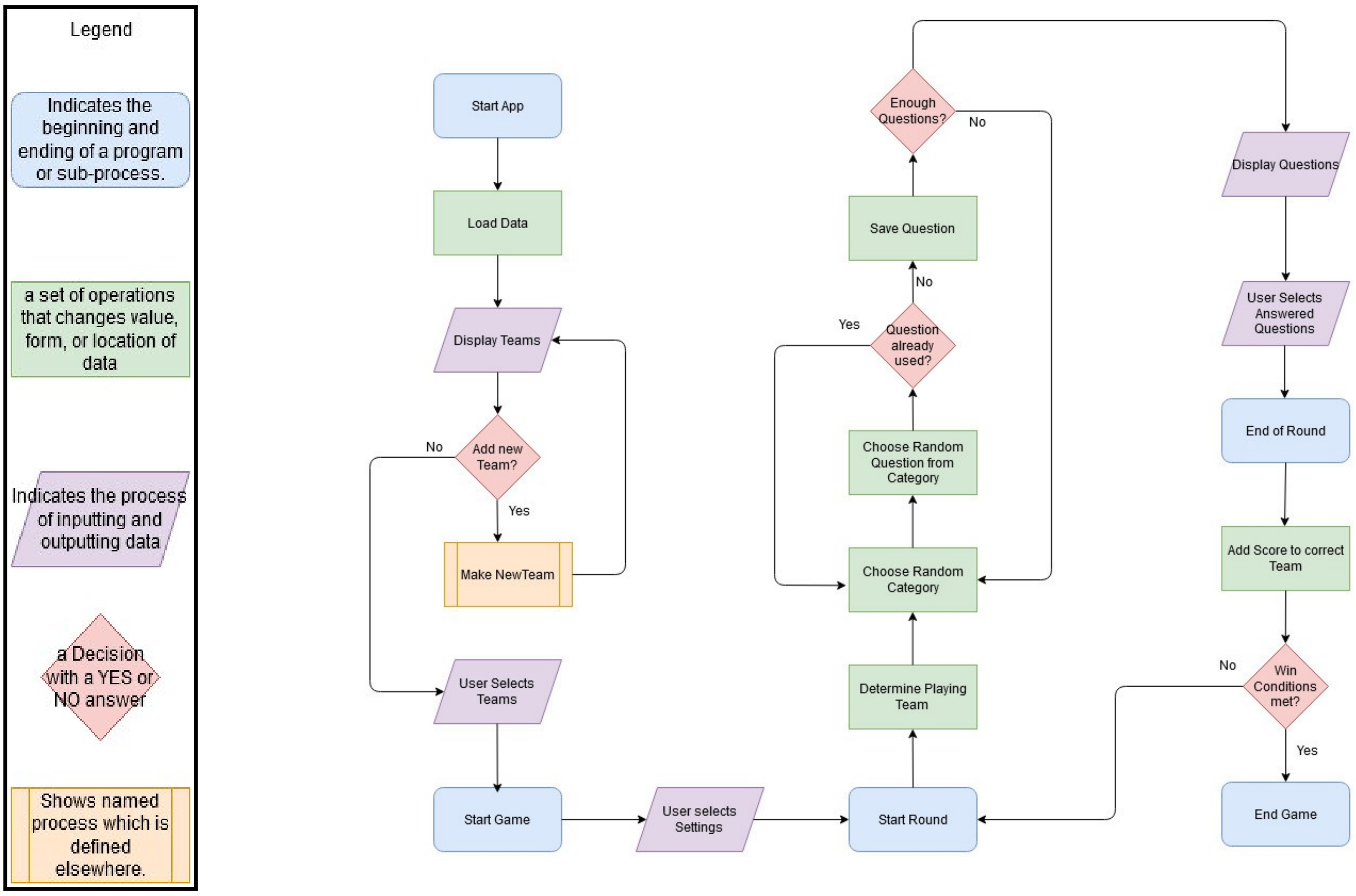
Since my interfaces are designed with the Single-Responsibility principle, this is not going to be a problem.

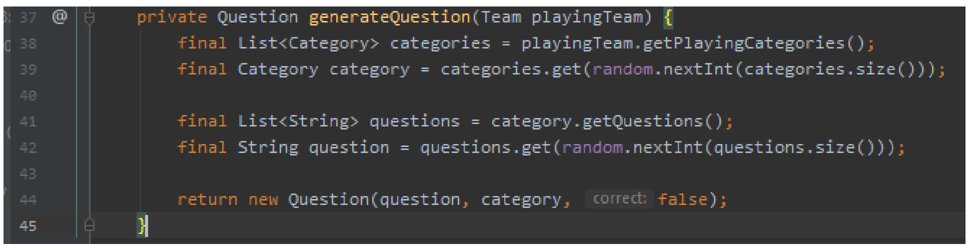
**Dependency Inversion Principle**

In the chapter, Architectural Structure I go into how I am planning on using a presenter that has two interfaces. These interfaces are implemented by classes outside of these layers. In this way I want to avoid dependencies between the different layers in the application. This presenter will use different “logic” classes, where the actual magic happens. In this way I want to avoid dependencies between the different layers and modules in the application.

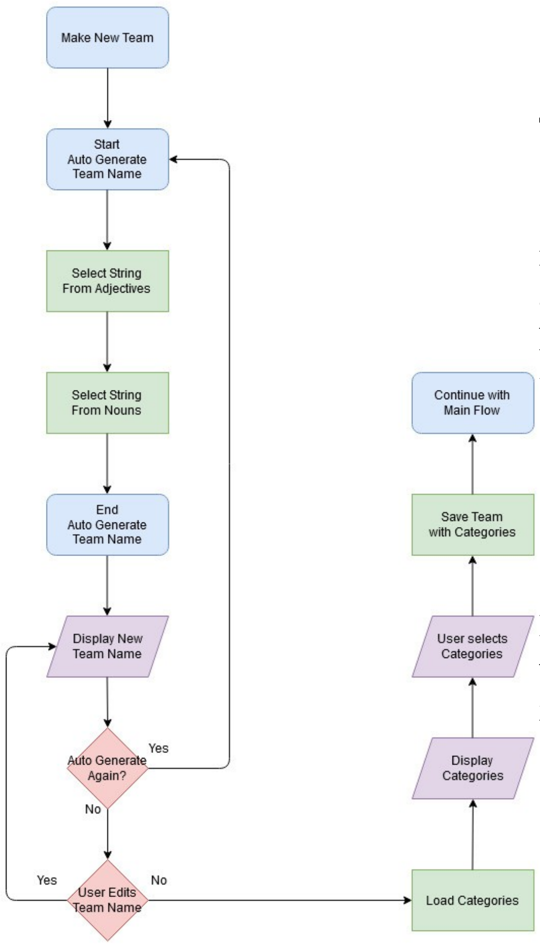
# Algorithmic Processes

In this chapter I will be going more into the algorithmic processes going on in the showcase. I will be doing this while working with a flowchart. This is the first version of the main Flow Chart. “Make New Teams” Has a small flowchart on the next page. I’ve decided to make a smaller flowchart, outside of the main one, to keep it somewhat cleaner. To play a round of 30 Seconds, we must have 5 questions. First the system must determine what Team is playing and then based on the team, it decides what Category to use. From that category, it will pick a random word. To avoid duplicate questions within a game, the system checks if the question has been used already. If it has been used, the system will choose another Category to pick a word from. If the question hasn’t been used yet, it will save this question and then check if it has got enough questions. If there are enough questions, it will continue with the flow. If there aren’t enough questions, it will loop back to “Choose Random Category” to fill in enough questions.


Legend
Indicates the beginning and ending of a program or sub-process.
a set of operations that changes value, form, or location of data
Indicates the process of inputting and outputting data
a Decision with a YES or NO answer
Shows named process which is defined elsewhere.
Start App
Load Data
Display Teams
Add new Team?
No
Yes
Make NewTeam
User Selects Teams
Start Game
User selects Settings
Start Round
Determine Playing Team
Choose Random Category
Choose Random Question from Category
Question already used?
Yes
No
Save Question
Enough Questions?
No
Display Questions
User Selects Answered Questions
End of Round
Add Score to correct Team
Win Conditions met?
No
Yes
End Game

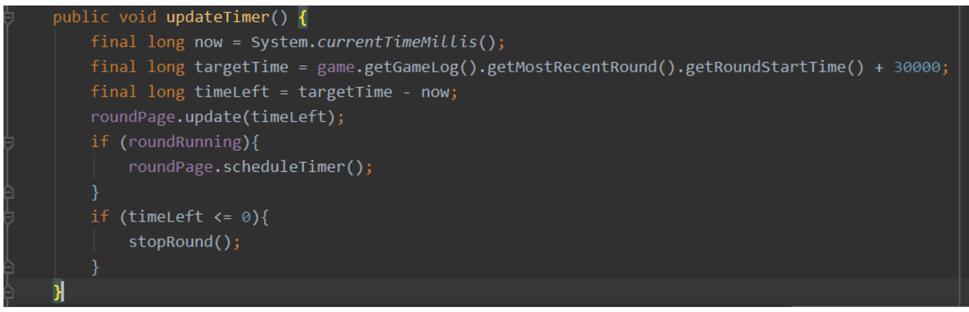


```java
    private Question generateQuestion(Team playingTeam) {
        final List<Category> categories = playingTeam.getPlayingCategories();
        final Category category = categories.get(random.nextInt(categories.size()));

        final List<String> questions = category.getQuestions();
        final String question = questions.get(random.nextInt(questions.size()));

        return new Question(question, category, correct: false);
    }
```

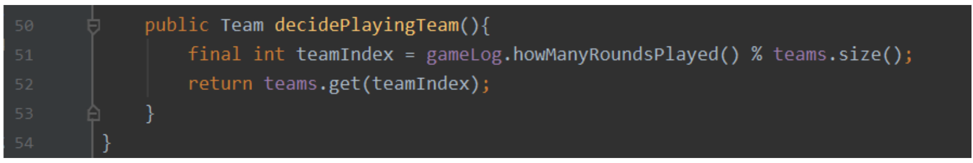


This is the Flow Chart for making new teams. New teams can only be made before a game has begun. The System will auto generate a name, by choosing from lists of adjectives and a nouns. Users then have the option to auto-generate a name again, or edit the name. After deciding on the name, users will have to choose categories for each team. When this process is finished, the Flow will be continued in the Main Flow Chart.

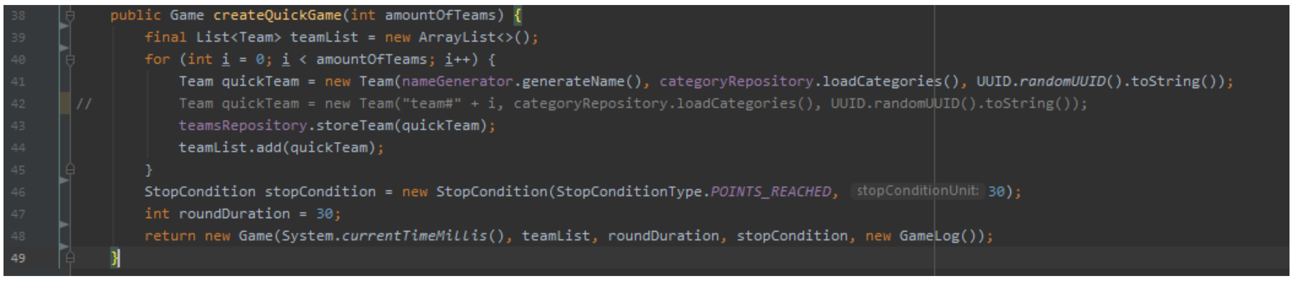
Below I have included some snippets of code to show some of the important Methods from the class diagram already implemented.



```
public void updateTimer() {
    final long now = System.currentTimeMillis();
    final long targetTime = game.getGameLog().getMostRecentRound().getRoundStartTime() + 30000;
    final long timeLeft = targetTime - now;
    roundPage.update(timeLeft);
    if (roundRunning){
        roundPage.scheduleTimer();
    }
    if (timeLeft <= 0){
        stopRound();
    }
}
```

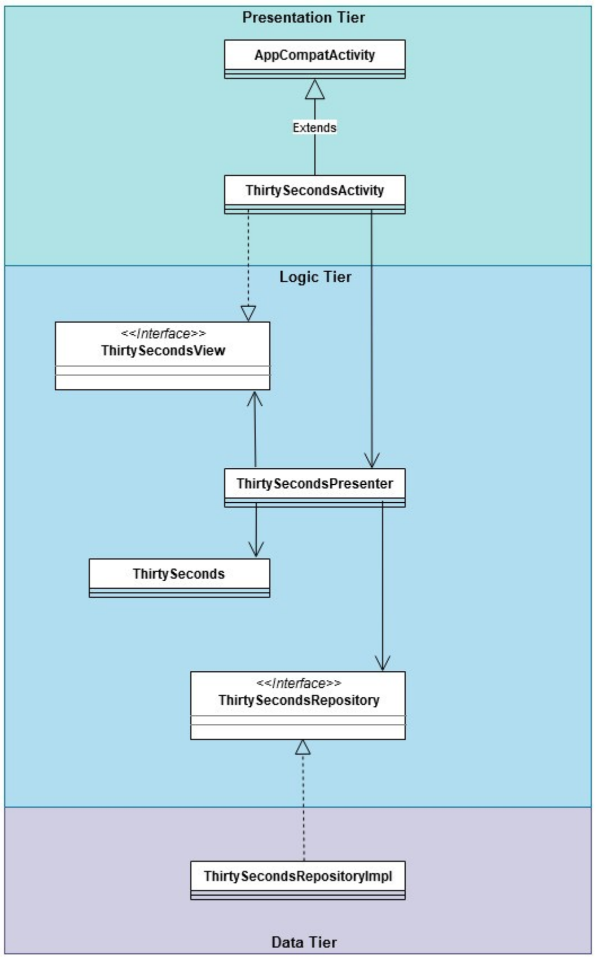


```
    public Team decidePlayingTeam(){
        final int teamIndex = gameLog.howManyRoundsPlayed() % teams.size();
        return teams.get(teamIndex);
    }
}
```



```
    public Game createQuickGame(int amountOfTeams) {
        final List<Team> teamList = new ArrayList<>();
        for (int i = 0; i < amountOfTeams; i++) {
            Team quickTeam = new Team(nameGenerator.generateName(), categoryRepository.loadCategories(), UUID.randomUUID().toString());
//            Team quickTeam = new Team("team#" + i, categoryRepository.loadCategories(), UUID.randomUUID().toString());
            teamsRepository.storeTeam(quickTeam);
            teamList.add(quickTeam);
        }
        StopCondition stopCondition = new StopCondition(StopConditionType.POINTS_REACHED, stopConditionUnit: 30);
        int roundDuration = 30;
        return new Game(System.currentTimeMillis(), teamList, roundDuration, stopCondition, new GameLog());
    }
```

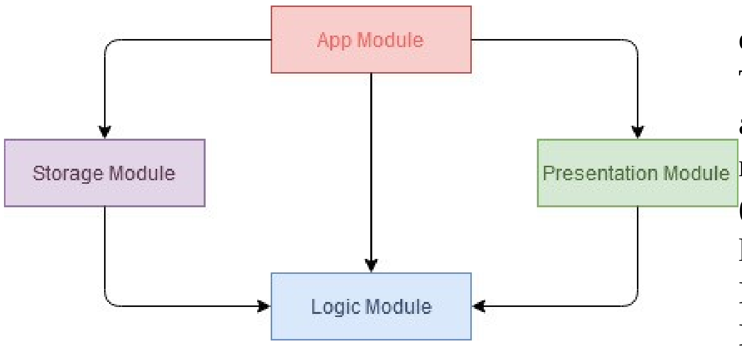
## Architectural Structure



This is my architectural class diagram. Since I am making an android application, I wanted to make sure the Logic Tier doesn't depend on the supporting tiers. I'm planning on avoiding this with a Presenter (this presenter is the same ThirtySecondsPresenter from the class diagram). The Presenter is the only class in the application that interacts with "Thirty Seconds" (this is where the magic happens). The Presenter *has* two interfaces, the ThirtySecondsView, and a ThirtySecondsRepository. ThirtySecondsView is *implemented* by the ThirtySecondsActivity in the Presentation Tier. The ThirtySecondsRepository is *implemented* by the ThirtySecondsRepositoryImpl in the Data Tier. As shown in the diagram, all the arrows from the Data Tier and the Presentation Tier are pointing into the Logic tier. This way, the Data- and Presentation Tier depend on the Logic Tier and not the other way around.

This is just a visual representation of how it would work. In the Class diagrams you can see the actual presenters and their activities.

## Modularity



While I still think it is a bit of an overkill, I have decided to divide my application into four modules. The App module is the "start-up" module of the application. It has dependencies on all the other modules, to start them up. The Presentation Module (the UI) and the Storage Module (the reading and loading of files) both have dependencies on the Logic Modules (this is where the magic happens). The Logic Module doesn't have any dependencies. All the other Modules need an Android library, but the Logic Module only has a java plug-in. This way I make really sure there are no dependencies on the android specific libraries. To start the application I'm using an abstract Factory (see Design Patterns).

## Persistence of Data

For the Thirty Seconds showcase, I will have to load lists of Questions. Since I want to avoid having my code full with these lists, I am planning on using JSON files. The reason I want to store them in a JSON file, is so I can easily add or remove Categories, without having to change the actual code of the program.

I have decided to use JSON files because they seem to be one of the most common data formats. The fact that JSON files are derived from JavaScript will not be a problem, as they're compatible with Java.

I am also planning to storing the (played) GameLogs to another JSON file, so these can later on be read to view the statistics of previously played games. It will keep lists of played rounds, used questions and playing teams, so certain statistics could be calculated.

## Design Patterns

In the previous chapter I keep mentioning Presenters. These are part of the Model-View-Presenter Pattern. While doing some research I came across a lot of discussion whether a Model-View-Presenter is an architectural Design Pattern or not. I decided that at this moment, those discussions weren't really valuable to me, and for the sake of this document, I will simply pretend it is.

*"In MVP, the presenter assumes the functionality of the "middle-man". In MVP, all presentation logic is pushed to the presenter."*

While there are a lot of similarities between an MVP and a Facade class, there are a few fundamental differences. As I understand a facade class would be an interface that connects multiple classes from one subsystem to various client classes. This way it shields both sub systems from each other.

However, an MVP is different because every Activity (like pages) in the Application, needs it's own MVP. Every time I want to display a Team in the Edit Team Activity, I load this Team through the Edit Team Presenter.

Another important difference is that Facade classes don't really have any Logic in them. Presenters however are the classes where the Logic is delivered to the Presentation Layer. As seem in the diagrams, I only have methods that start, stop or make other methods, objects and/or classes.

Before I would simply start the application through the Presentation Module and just make these Presenters whenever I would make the activities. The Presenters are in the Logic Module however. So instead of making them directly in the Activity, I have decided to make use of an Abstract Factory. In the Start-up Module there are two classes, MyApplicationImpl and PresenterFactoryImpl. These are implemented from the MyApplication and PresenterFactory in the Presentation Module. In the Presenter Module an Activity would use an instance of the actual interface and make the associating Presenter.

# Sources

UML distilled by Martin Fowler, 1997

  Abstract Factories

Wiki flowchart

  https://en.wikipedia.org/wiki/Flowchart

wiki klassendiagram

  https://en.wikipedia.org/wiki/Class_diagram

BAZZZ.nl

  https://bazzz.nl/

Canvas

  https://fhict.instructure.com/courses/10297/modules

Wiki SOLID

  https://en.wikipedia.org/wiki/SOLID

Wiki Interfaces

  https://en.wikipedia.org/wiki/Interface_(Java)

Wiki MVP

  https://en.wikipedia.org/wiki/Method_(computer_programming)

Wiki Facade Class Pattern

  https://en.wikipedia.org/wiki/Facade_pattern

Wiki Abstract Factories Patterns

  https://en.wikipedia.org/wiki/Abstract_factory_pattern