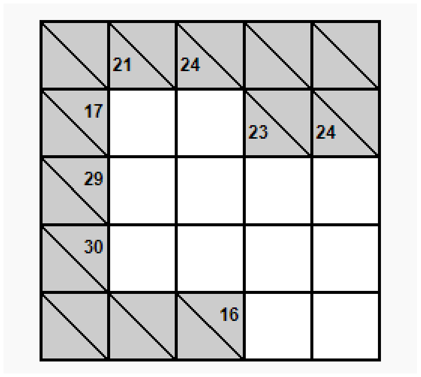
Design Document

Team PI-B

15 March 2020



Kakuro

Team members

Name	ID Number
Sajib Ahmed	40044867
Yaroslav Bilodid	40068605
Jesse Desmarais	40035761
Antoine Farley	40100554
Marc Hegedus	26242219
Katerina Tambakis	27010486

Contents

1 Introduction

The primary goal of this project is to develop the Kakuro game. This game is based on the original Kakuro game with no modification in the logic of the game. The purpose of the design document is to provide all details of the Architectural Design (AD), Detailed Design and Dynamic Design Scenarios.

1.1 Purpose

The purpose of the document is to present the design of the Kakuro game, which is in partial fulfillment of the requirements of building the application. It will provide details on the Architectural Design, Detailed Design and Dynamic Design Scenarios. The Architectural Design will focus on the built of the application with architectural diagrams and Subsystem Interface Specifications. The Detailed Design focuses on the system design describing the Application Core and Puzzle along with UML diagrams and short textual descriptions. Finally, the Dynamic Design Scenarios will list the use cases using UML sequence Diagrams and show various subsystems and units are interacting to achieve a system-level service.

1.2 Scope

The document is intended to provide detailed design specifications of the Kakuro Game that will be used as a basis for the implementation phase. The Architectural Design explains the application in terms of Architectural Diagram and Subsystem Interface Specifications in great detail in order for the implementation team to actually create a game based on the Detailed Design.

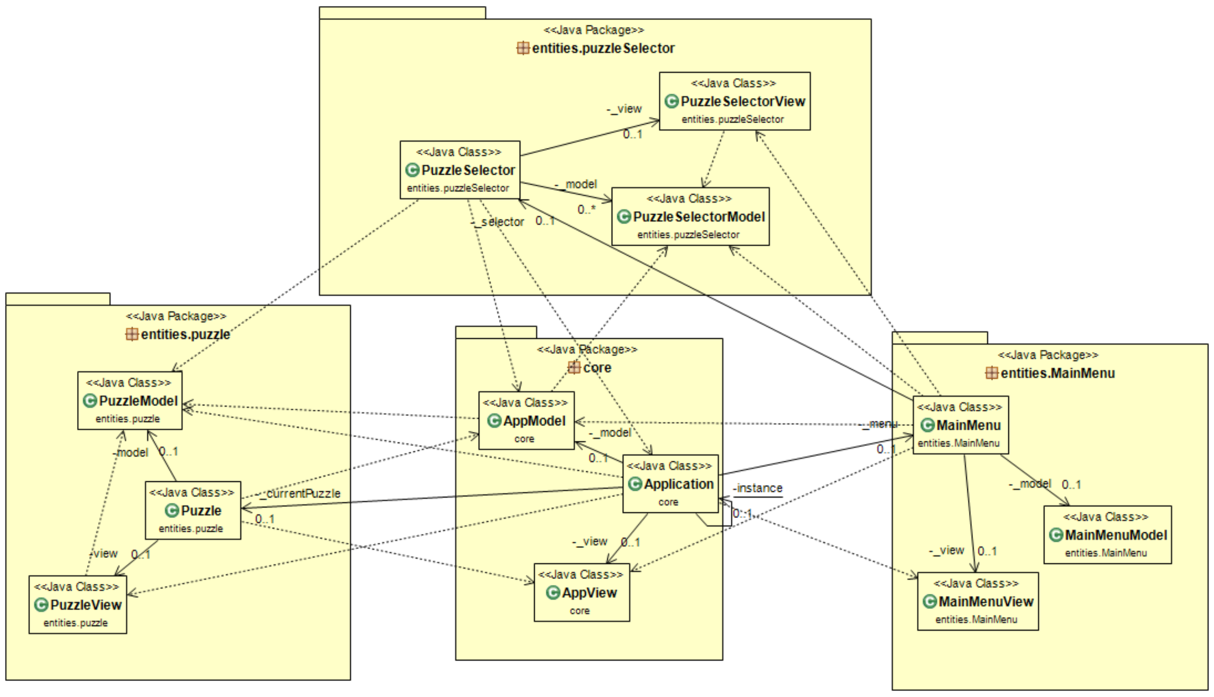
1.3 Context

This document addresses the requirements that will be used as a basis for the implementation phase. A number of figures will be used to describe the main subsystems of the architectural design which are built using MVC design pattern. The Detailed Design describes the system design, describing each subsystem as well as a short textual description for each class. Finally, the Dynamic Design Scenarios will describe each use cases demonstrating how the various subsystems and units are interacting to achieve a system-level service.

2 Architectural Design

Our application is build using 2 main subsystems: AppCore and Puzzle and also has 2 helper subsystems: MainMenu and PuzzleSelector. AppCore is a base of the whole application. MainMenu subsystem is used for interacting with user and proposes all available user cases. PuzzleSelector is used as a prompt to the user to select which puzzle he wants to use. And Puzzle subsystem is build for gameplay, displaying and interacting with the puzzle itself. All the subsystem are built using MVC design pattern so it consists of 3 parts: Controller, Model and View.

2.1 Architectural Diagram



2.2 Subsystem Interface Specifications

Communication between AppCore and Puzzle:

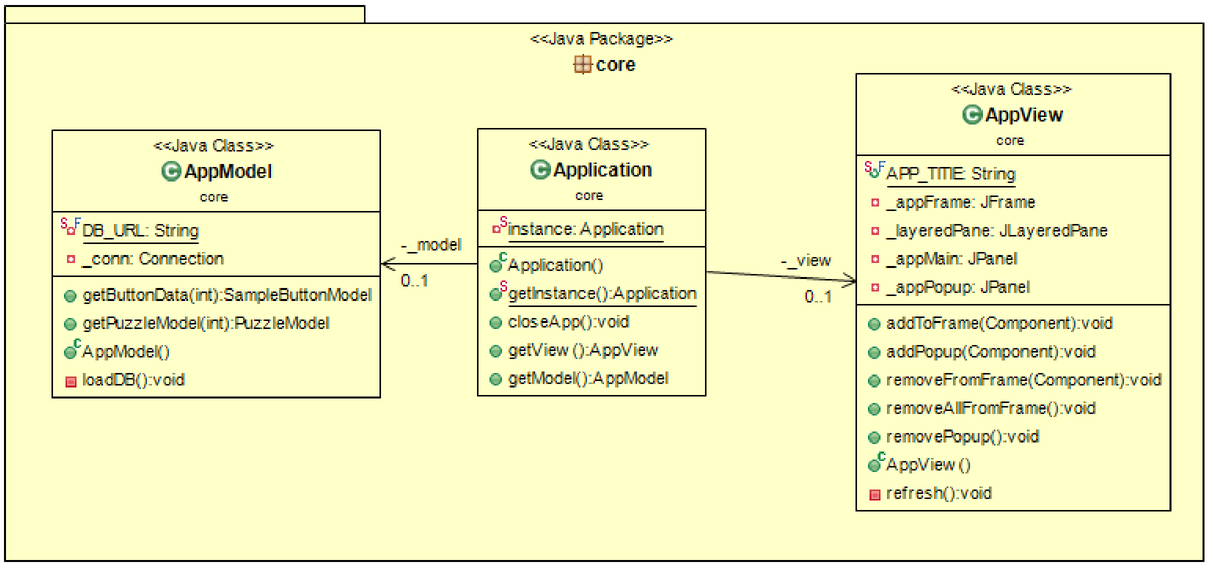
View.addToFrame(view)	This function is being called from inside of Puzzle class to display puzzle to the main frame. It passes puzzleView frame to the AppView to be inserted.
Model.getPuzzleModel(1)	This function is being called from inside of PuzzleModel class to get information about Puzzle selected by user. It passes the id of the puzzle to the AppModel to get information from application Database.
Application.getInstance()	Creates MainMenu instance and it is stored in Application object.
MainMenu()	Creates PuzzleController and assigns call to it to the ButtonListener in the MainMenu.
PuzzleSelector()	Creates Listener for the selector button which calls AppModel to create PuzzleModel and then passes it to Application.
AppSetPuzzle()	Creates a Puzzle instance and puts it to the AppView.

3 Detailed Design

3.1 Application Core

Application Core is the base structure of our program. It uses MVC design pattern and implements 3 classes: AppModel, Application(Controller) and AppView. The purpose of this package is to build and maintain base layer for the application.

3.1.1 Detailed Design Diagram



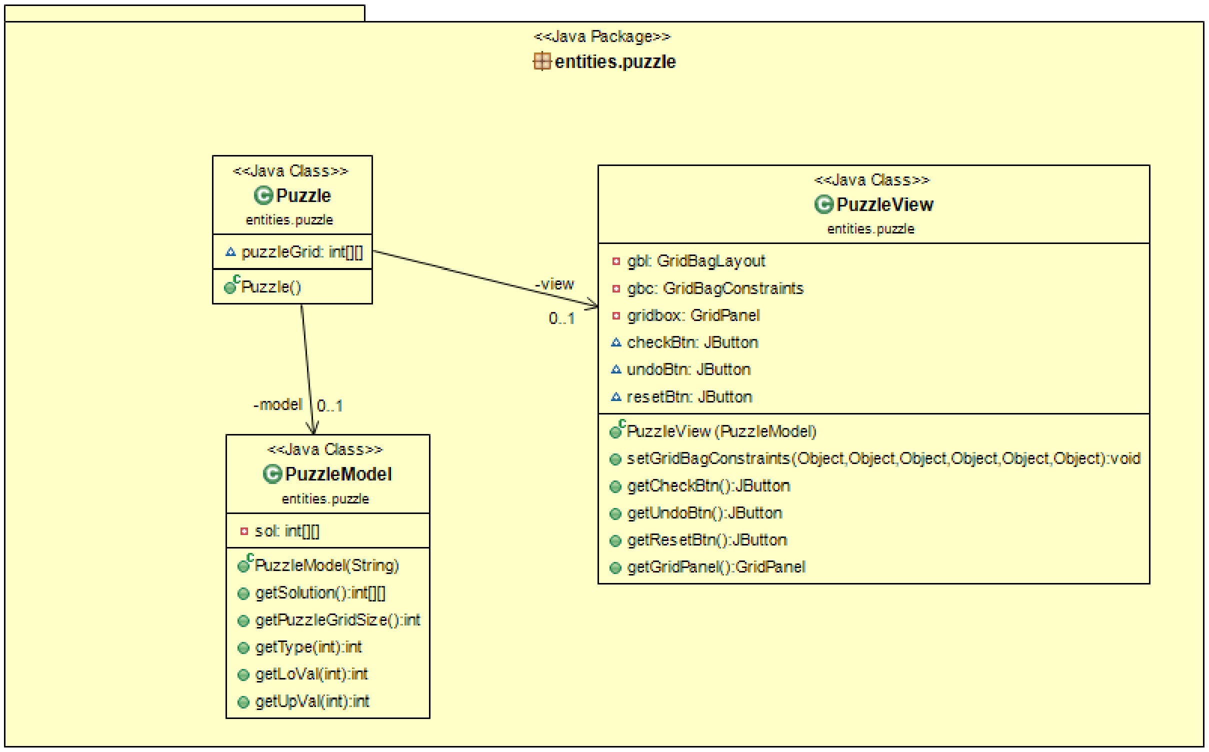
3.1.2 Unit Description

Application	Singleton core class. Gives access to CoreView and Core-Model.
AppView	Used for displaying main frame as a base for menu or puzzle frames.
AppModel	Used for establishing connection with the database and getting information about puzzles.

3.2 Puzzle

Puzzle is the gameplay handler of our program. It uses MVC design pattern and implements 3 classes: PuzzleModel, Puzzle(Controller) and PuzzleView. This package is responsible for everything about the puzzles such as displaying them, uploading current puzzle state to database, and interacting with user.

3.2.1 Detailed Design Diagram



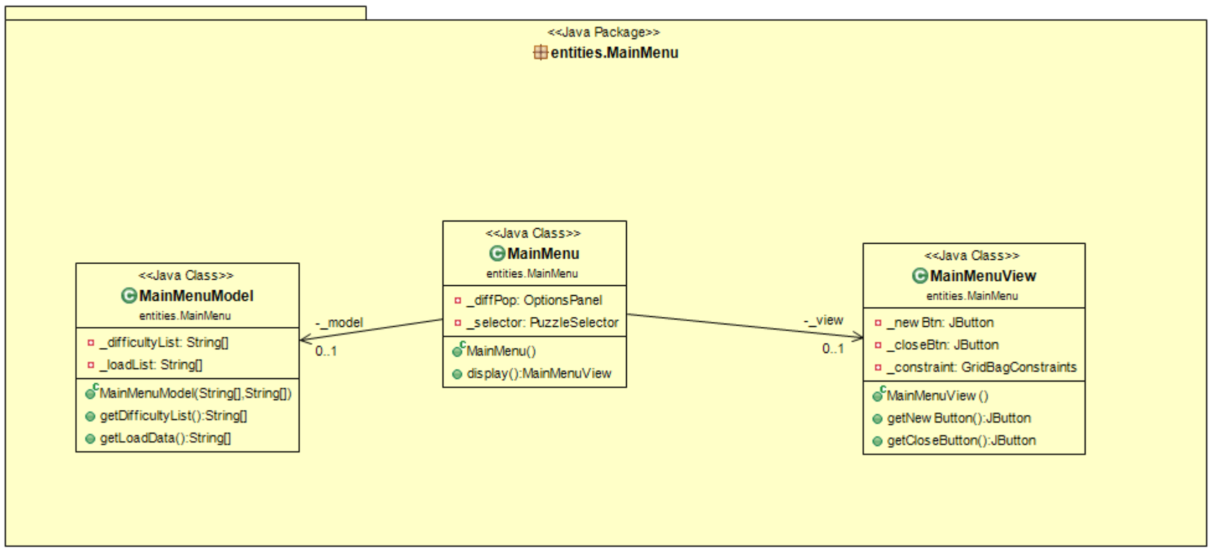
3.2.2 Units Description

Puzzle	Main class that plays role of controller. Accesses PuzzleModel to change PuzzleView.
PuzzleModel	Model class. Uses AppModel to connect to the database and get information about puzzles and uses this information to check them or use a save from previous session.
PuzzleView	View class. Builds the frame with the puzzle into AppModel using information about puzzle from PuzzleModel.

3.3 Main Menu

MainMenu is the menu layer of our program. It uses MVC design pattern and implements 3 classes: MainMenuModel, MainMenu(Controller), and MainMenuView. MainMenu gives user options to start game, choose difficulty or exit the application.

3.3.1 Detailed Design Diagram



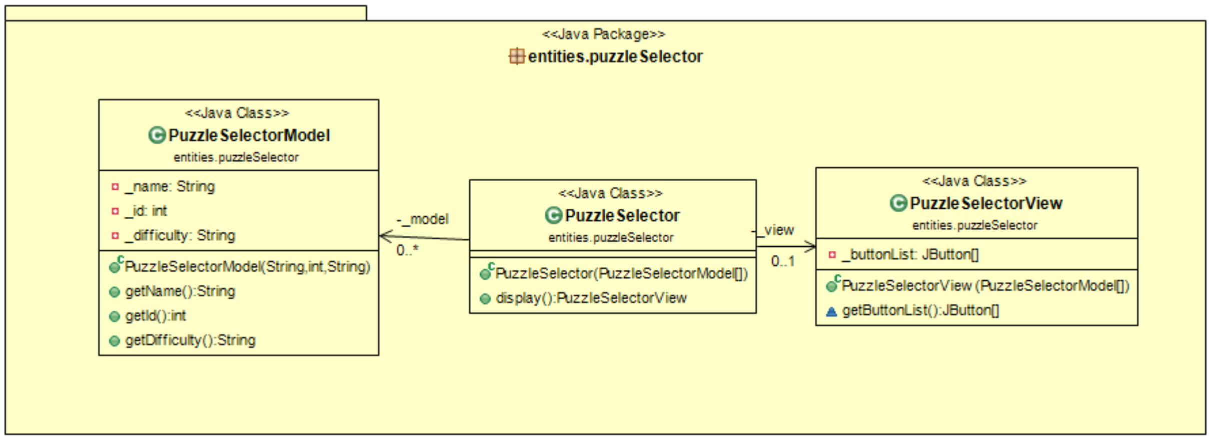
3.3.2 Units Description

MainMenu	Main class that plays role of controller. Accesses MainMenuModel to change MainMenuView.Also includes PuzzleSelector to display available puzzles after difficulty is chosen.
MainMenuModel	Model class. Uses AppModel to connect to the database and get information about state of MainMenu and difficulty list.
MainMenuView	View class. Builds the frame with the puzzle into AppModel using information about state of MainMenu such as popups from MainMenuModel.

3.4 PuzzleSelector

PuzzleSelector is the selector for the available puzzles in the application. It uses MVC design pattern and implements 3 classes: PuzzleSelectorModel, PuzzleSelector(Controller), and PuzzleSelectorView. This selector is an necessary part of starting game use case as it will load a selection of available puzzles and after user selection will pass state of chosen puzzle to Puzzle class so it can be played.

3.4.1 Detailed Design Diagram



3.4.2 Units Description

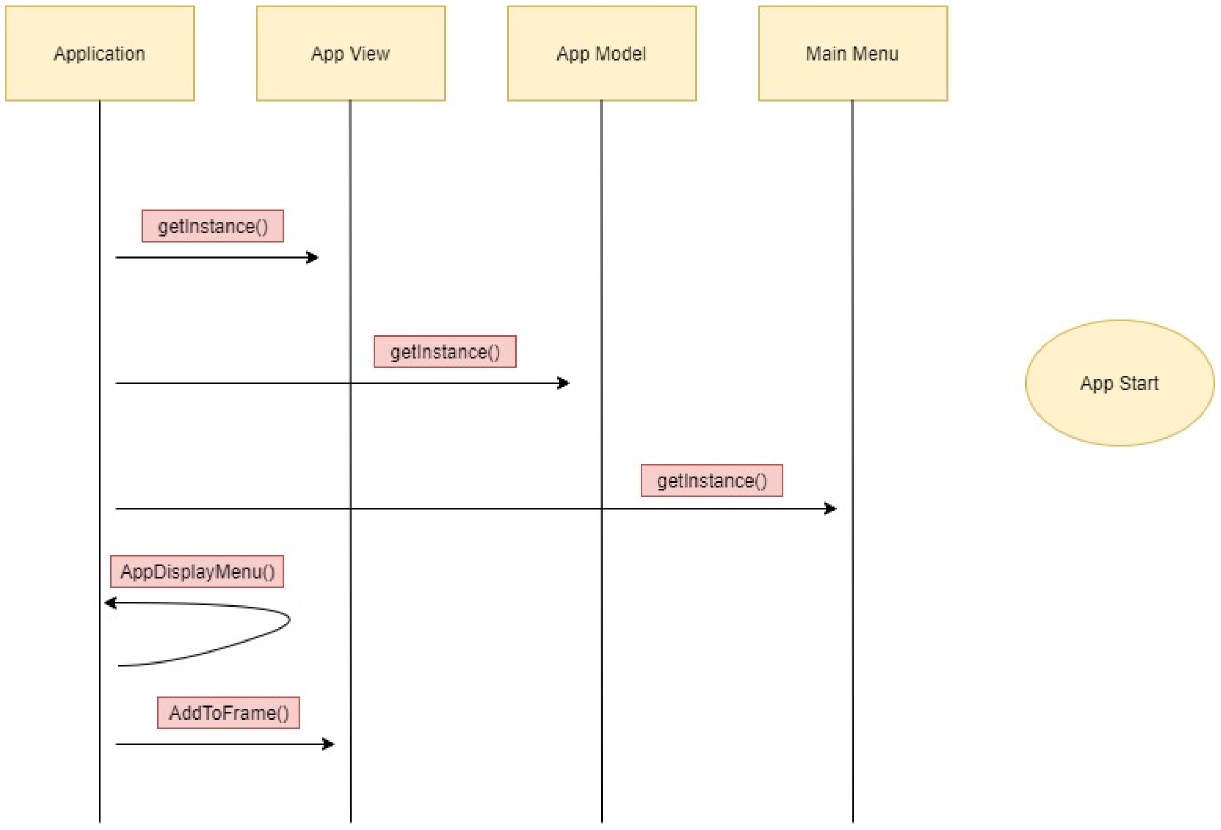
PuzzleSelector	Main class that plays role of controller. Accesses PuzzleSelectorModel to change PuzzleSelectorView.
PuzzleSelectorModel	Model class. Uses AppModel to connect to the database and get information about puzzles available to the user.
PuzzleSelectorView	View class. Builds the frame with the puzzle into AppModel using information about puzzle from PuzzleSelectorModel and displaying all the available puzzles.

4 Dynamic Design Scenarios

The three most important execution scenarios of this are start game scenario, save and exit scenario, and input validation scenario. We chose the following 3 execution scenarios because they ensure the proper functionality of the Kakuro game demonstrates how the various subsystems and units are interacting to achieve a system-level service. The Start game scenario is where the system builds the main menu and starts the application. The save and exit scenario is where the application asks the user if they would like to save their unfinished puzzle before exiting the program. The input validation scenario is where the application validates the inputs on the puzzle once the user clicks the check button.

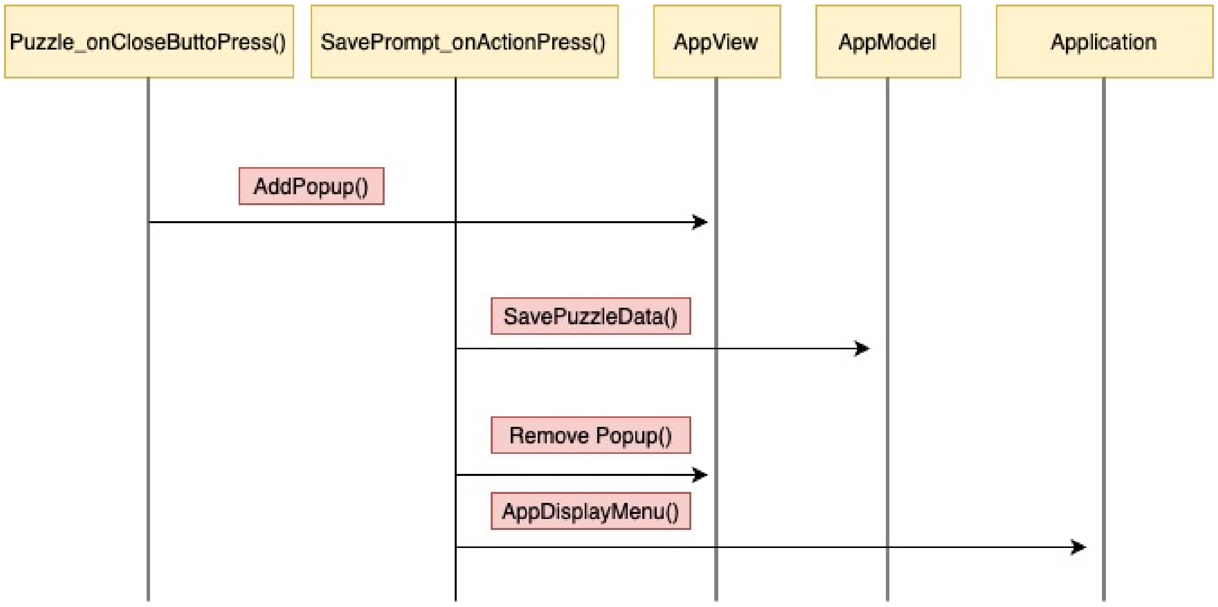
4.1 Start Game Scenario

This is the application start up scenario that initializes core systems and displays the menu prompting the first user interaction. Application Singleton is initialized through the getInstance() call. Since no instance exists yet it creates the Applications view and model and a main menu object. Once all the initial systems and main menu object are created the application calls its own AppDisplayMenu function which adds the mainMenu to the AppView through the AddToFrame method.



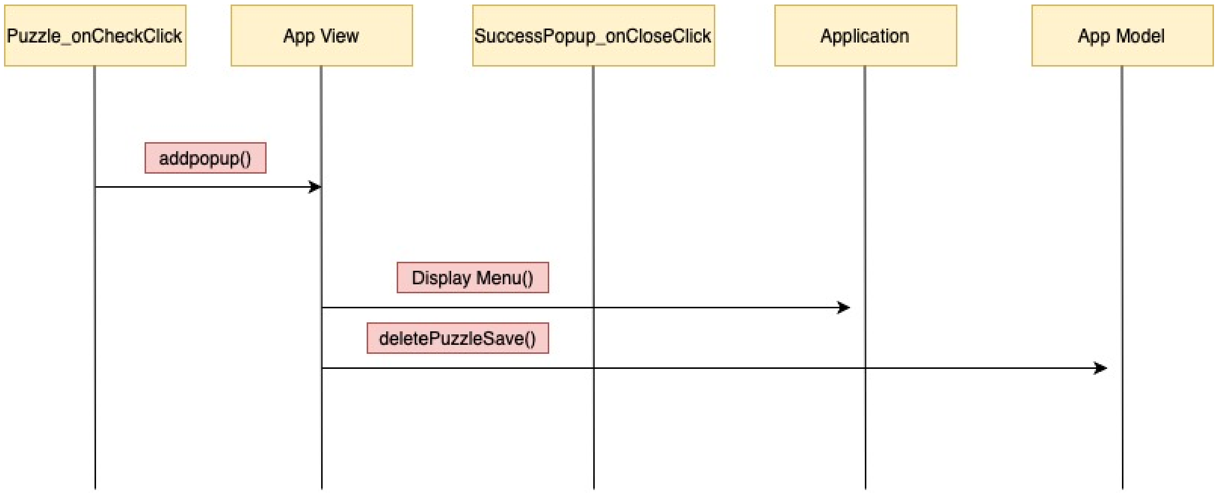
4.2 Save and Exit Scenario

In this scenario the player will close and save the puzzle that they are currently playing. When the player clicks the close button the closeButtonActionListener lambda defined in Puzzle calls AppView's addPopup method to add a save prompt to the window. Then the player opts to save by clicking on the ActionButton (the save option) which calls another lambda inside puzzle. In this lambda method we make a call to the AppModel's SavePuzzleData method which stores current puzzle data in the database. The same lambda also requests AppView to RemovePopup() and tells the application to display the main menu with AppDisplayMenu() which will clean up the puzzle nullifying and garbage collecting it.



4.3 Input Validation Scenario

In this Scenario the player will have completed the puzzle and wish to see if their solution is valid, which in this case it is. When the player clicks on the validate button Puzzle CheckButtonActionListener lambda method gets called. This lambda calls gridPanel's isSolved() which returns if the puzzle is solved or not. If it's solved we add a Success Popup. Next we check to see if the player loaded a previous session. If so we consider this session completed and can delete the existing save by calling AppModels deletePuzzleSave() method. When the successPopup's closePopup button is clicked it calls another lambda method. In this second Lambda we call Applications AppDisplayMenu() to return to menu.


Puzzle_onCheckClick
App View
SuccessPopup_onCloseClick
Application
App Model
addpopup()
Display Menu()
deletePuzzleSave()