Hot Potatoes

Learn How To Code

Detailed Design Document

Group Name: Hot Potatoes Inc.

Contributors:
Allant Gomez
Chris Mnich
Shahab Shekari
Steven Rengifo
Zachary Guadagno

Rutgers University Department of Computer Science CS 431 Software Engineering
Instructor: Alex Borgida
November 3, 2014

Introduction

This document includes some details of the Hot potatoes apis. What this document tries to describe is the Hot potatoes simulator, which helps kids learn how to program using a robot that is picking up potatoes and trying to get “home”.

In today’s world, learning how to program is a very vital skill that many people aren’t taught or exposed to in school. There are many reasons for this, the most frequent one being that learning how to program can be hard, especially for kids. There are many aspects of programming such as syntax and logic that are very hard for children to grasp and the process of learning these things can be very boring. That’s where Hot Potatoes comes in. Hot Potatoes is a game designed to teach kids how to program through a
series of worlds in which a hungry robot must collect a series of hot potatoes scattered throughout a world and go home to eat them. Each world will be designed to teach kids about different aspects of programming and to learn the logic of programming in a fun way. The way the game works is the student will program the robot to navigate through the world using a series of commands. Hot Potatoes also allows users to create their own worlds so they can challenge themselves and friends. Teachers can also create an account on the system that will allow them to track the progress of their students and create their own worlds as well, making Hot Potatoes a very valuable tool that makes programming fun and easy.

Structure

We’ve decided to go for an MVC (Model View Controller) architecture to address the structure of our apis. Our Javadoc and UML diagrams are both attached in this archive.

Design

- **Model**
  - **Block** - This class is responsible for keeping track of the information that a block in the game world has.
    import java.util.ArrayList;
    import java.util.List;
    - Block(int x, int y)
      Creates a new instance of a block.
    - add(BlockState state)
      Adds a blockstate to this grid. Returns the created block.
    - coordinates()
      Gets the coordinates of the grid. Returns the coordinate of the block on its grid.
    - is(BlockState state)
      Checks whether a particular blockstate is contained within this grid. Returns a boolean as to wheather the block has the specified blockstate.
    - remove(BlockState state)
      Removes a blockstate from this block. Returns the block with the specified state.
  - **BlockState** - This class is responsible for differentiating between different types of blocks. A BlockState has a enum field which can be one of the following: “BLOCKED”, “POTATO”, “ROBOT”, “START”, “TARGET”, and “UNBLOCKED”. “BLOCKED” means the user cannot control the robot to move to a block with that state. “POTATO” is a block in which a user wants to move in order to collect a potato. “ROBOT” is a block with the user’s robot. “START” is a block where the user’s robot starts a world from. “TARGET” is a block where the user must move a robot to complete a world. “UNBLOCKED” is a block that is unoccupied so a user can move his robot on freely.
  - **Code** - This class will store the CodeBlocks from the user.
    - ArrayList<CodeBlock> list
      Used to store all the codeblocks.
    - getList()
      Returns an iterator that will iterate through the arraylist.
    - add(CodeBlock codetoadd)
      Appends the specified codeblock to the end of this list.
    - remove(CodeBlock codetoremove)
      Removes the first occurrence of the specified element from this list, if it is present.
    - clear()
      Removes all of the elements from this list.
    - contains(CodeBlock code)

Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that (o==null ? e==null : o.equals(e)).

- **CodeBlock** - This class will store a block of code in text and also especify which type and where the code is position.
  - getCodetype()
    Gets the codetype of this block
  - setCodetype(CodeType codetype)
    Sets the codetype for this block
  - getCodeposition()
    Gets the position of this codeblock within the entire code structure
  - setCodeposition(int codeposition)
    Sets the codeposition of this codeblock within the entire code structure
  - getCodetext()
    Gets the text contained within this code block
  - setCodetext(String codetext)
    Sets the text for this codeblock
- **CodeType -** This class will take care of the code type using three variables: int type, int color and int format. Type will be either a while loop, if clause, condition etc. Color will depend of where the code was added. And, format will include the number of the format.
  - deleteTable()
    Checks whether this code type is deletable or not.
  - getType()
    Returns the type of this codeblock
  - setType(int type)
    Sets the type of this codeblock
  - getColor()
    Gets the color of the codeblock
  - setColor(int color)
    Sets the color of the codeblock
  - getFormat()
    Gets the format of the codeblock
  - setFormat(int format)
    Sets the format of the codeblock
- **Coordinate** - This class is responsible for keeping track of the information a Coordinate in the game world has.
  - Coordinate(int x, int y)
    Creates a new Coordinate object.
  - getX()
    Gets the x coordinate. Returns an int that is the x parameter of the specified Coordinate.

        - getY()
          Gets the y coordinate. Returns an int that is the y parameter of the specified Coordinate.

    - **Direction**
    - **Grid** - This class is responsible for keeping track of the information that a grid in the game world holds.
      import java.util.List
        - getBlock(Coordinate coordinate)
          Given a coordinate, returns the block on the grid. Returns the block at the specified Coordinate of the grid.
        - getStartingPosition()
          Gets the starting position of the robot. Returns the block in which the robot starts at for the grid.
        - getTargetPosition()
          Gets the target position of the robot. Returns the block in which the the robot must reach to end the world on the grid.
        - numPotatoes()
          Gets the number of potatoes on the grid. Returns an int that tells the user the amount of potatoes on the grid.
        - setStartingPosition(Block block)
          Sets the starting position of the robot on the grid. Returns the grid with the changed StartingPosition.
        - setTargetPosition(Block block)
          Sets the target position of the robot. Returns the grid with the changed TargetPosition.
    - **Robot**
      This class is an abstraction of Karel, the robot, and the actions it can perform.
        - Robot(Coordinate coordinate)
          Creates a new robot.
        - move()
          Moves the robot one block in the direction its facing
        - pickup()
          Picks (if available) a potato from its current block
        - drop()
          Drops (if available in backpack) a potato onto the current block
        - setDirection(Direction direction)
          Changes the direction of the robot
        - getCoordinate()
          Gets the robot's current coordinates
        - getDirection()
          Gets the current direction the robot is facing

        - getBackpackSize()
          Gets the size of the robot's backpack
        - setCoordinate(Coordinate coordinate)
          Sets new coordinates for the robot
    - **User**
      This class is an abstract representation of a user of our system.
        - User(String username, String password, boolean isAdmin)
          Creates a new user account.
        - setUsername(String username)
          Changes the current username of this account
        - setPassword(String password)
          Changes the current password of this account
        - changeAdminStatus(boolean isAdmin)
          Changes the current admin status of this account
        - getUsername()
          Gets this user account's username
        - getPassword()
          Gets this user account's encrypted password
        - isAdmin()
          Gets this user account's admin status
        - encrypt(String password)
          Encrypts a plain text password
- **View**
    - **ActionPanelView -** Handles the display of the "Action Panel" which contains the buttons for inputting code into the CodeView
    - **AdminDashboard -** The welcome screen for an Adminstrator user. Has buttons that allows an administrator to build a world, add/delete users, and check on the scores of other users
    - **BuildPanelView -** Panel of buttons used in the buildScreen. Has buttons that allows the user to build their world.
    - **BuildScreen -** The screen where the user is able to build their own world.
    - **CodeBlockView -** Handles the display of code blocks in the code view. Making sure each block is properly formatted.
    - **CodeView -** The panel where all of the code blocks are displayed. Makes sure that they are all formatted correctly.
    - **Dashboard -** The interface for a dashboard, contains the basic layout. The StudentDashboard and AdmindDashboard extend this.
    - **GridView -** Handles the display of the grid.
    - **LevelSelect -** The screen the displays all of the availble worlds a user can choose from.
    - **LoginScreen -** The screen where the user logs in.
    - **MacroPanelView -** A panel that holds all the macros a user creates
    - **MenuView -** Handles the menu bar on the top of all the screens.

- **PlayScreen-** The screen where the user actually plays the game. Brings together the ActionPanelView, CodeView, GridView, MacroPanelView, and MenuView.
- **Registration -** Screen where a new user can be created. Accessed through the AdministrationDashboard.
- **ScoreView -** Screen that displays all of the scores for Student users, accessed through the AdminDashboard.
- **Screen -** General interface for all of the screens.
- **StudentDashboard-** Screen that is displayed once a Student is successfully logged in. Links to the LevelSelect screen and BuildScreen.

- **Controller**
    - **ActionPanelController -** This class controls the ActionPanelView and determines which action buttons are visible to the user while making a program.
      <u>import view.ActionPanelView</u>
      <u>import view.CodeView</u>
        - <u>ActionPanelController(ActionPanelView actionPanel, CodeView codeView)</u>
          Creates a new Action Panel Controller
        - <u>update()</u>
          Updates the Action Panel
    - **BuildPanelController -** This class controls the BuildPanelView and determines which build buttons are visible to the user while creating a world.
      <u>import view.BuildPanelView</u>
      <u>import view.GridView</u>
        - <u>BuildPanelController(BuildPanelView buildPanel, GridView grid)</u>
          Creates a new Build Panel Controller
        - <u>updateGV()</u>
          Updates the Grid View
        - <u>updateBP()</u>
          Updates the Build Panel View
    - **CodeController** - This class will take care of controlling the code class and codeview class.
      <u>import src.model.Code</u>
      <u>import src.view.CodeView</u>
        - <u>run()</u>
          Main class that will control the code and view
        - <u>editCodeBlock(CodeBlock code)</u>
          Edit parts of the CodeBlock
        - <u>removeCodeBlock(CodeBlock code)</u>
          Remove entire code block from Code
        - <u>addCodeBlock(CodeBlock code)</u>
          Add CodeBlock to the end of the code

- **DatabaseController**
  This class is responsible for loading/saving user/game data from our data store.
  import model.User
  import java.util.HashMap
  import java.util.Map
    - loadUserAccounts()
      Loads all available user accounts (username/user obj)
    - saveUserAccounts(Map<String, User> userAccounts)
      Saves a new file containing <username, user obj> entries. This happens whenever we add/remove a user account.
    - load(User user)
      Loads (from file) the GameController of a given user
    - save(GameController gameController)
      Saves (to file) the GameController of a given user

- **GameController**
  This class will call many methods throughout the package to execute the code the user has inserted into the CodeView and makes calls to update the display in conjunction with whatever code is being executed.
  Imports:
    - (from Model)
        - Grid
        - Robot
        - CodeBlock
        - User
    - Serializable

  Methods:
    - addCodeBlock(CodeBlock code)
      Adds the designated codeBlock into the codeView
    - editCodeBlock(CodeBlock target)
      Allows the user to edit a block of code they inserted into the code view
    - load()
      Loads the previous state of the level
    - removeCodeBlock(CodeBlock target)
      Removes a designated CodeBlock from the code view
    - run()
      Runs the current code the user has inserted into the code view by stepping through the codeBlock data structure and executing statements accordingly
    - save()
      Saves the current state of the level

        - update()
          Updates the display of the CodeView, ActionPanel, and Grid
- **GridController**
  This class deals with updating the grid in the game world based on the actions of the user.
  GridController(Grid grid)
  Creates a new grid controller
  update()
  Updates the grid controller. Returns the GridController used for the update.
- **MenuController -** This class controls the MenuView and contains actions that the user can perform from the Menu bar.
  import model.User
  import view.MenuView
  import model.Grid
    - MenuController(MenuView menu)
      Creates a new Menu Controller
    - saveState()
      Saves the state of the program
    - saveWorld(Grid grid)
      Saves the world
    - exit()
      Exits the program
    - back()
      Takes the program to the previous screen
    - logout(User user)
      Logs out the current user
    - help()
      Brings up a help menu
- **RobotController**
  This class deals with actions that can be executed by the robot, including picking up, droping and changing directions.
  import model.Coordinate
  import model.Direction
  import model.Robot
    - RobotController(Robot robot)
      Creates a new Robot Controller
    - RobotController pickup()
      Picks up a potato from the current block
    - RobotController drop()
      Drops a potato onto the current block
    - RobotController move()
      Moves the robot by one spot towards the direction its facing

        - isMoveValid(Coordinate coordinate)
          Checks whether a given move is valid
        - changeDirection(Direction direction)
          Changes the direction of the robot
    - **UserController**
      This Class is responsible for user level actions, including logging out, logging in, adding a user, and removing a user.
      import model.User
        - UserController(User user)
          Creates a new user controller
        - boolean logout()
          Logs out a user (saves the user file to disk).
        - GameController login(String username, String password)
          Attempts to login a user. Calls DatabaseController and loads the list of existing user accounts. If the username/password is valid, it'll get the user's GameController.
        - boolean addUser(String username, String password, boolean isAdmin)
          Adds a new user.
        - boolean removeUser(String username)
          Removes a user and his/her corresponding files
          username is the username of the user to be deleted

Acknowledgements

Allant Gomez: Javadoc, CodeController, CodeView, Code, CodeBlock, CodeBlock, CodeType, CodeBlockView, CodeView
Chris Mnich: Javadoc, GameController, All Screens & Dashboards
Shahab Shekari: Javadoc, UML, UserController, RobotController, User, Robot, DBController
Steven Rengifo: Javadoc, ActionPanelController, BuildPanelController, MenuController, ActionPanelView, BuildPanelView, MacoPanelView, MenuView
Zachary Guadagno: Javadoc, Block, BlockState, Coordinate, Grid, GridView, GridController