Kate Templeton
ENGW 3302
Fall
Unit 4: Writing for a Professional Audience
Word Count of Context Memo: 697
Word Count of Remainder of Document: 1,903

Context Memo

My fictional bosses at my fictional company (call it London Software Inc.) have tasked me with creating a design document which highlights the skeleton for our new online board game. Our board game is called Santorini, and the finer details concerning the rules, wider implementation of our game are not important for this project. This project only concerns the most crucial aspect of our new online game - the board.

The reason I have chosen to invent a company and project for this assignment is because this field - the field of Software Development - is a field I am passionate about. Although I have not had many internships or real-world experience in this industry, I have spent the past 5 years examining it in depth and feel I have a solid grasp of the type of assignments I may be given at a Software firm.

My design document features three elements which would be shown to my bosses. These three elements form a presentation which I would present. Part I is the code (presented to suit the more technical bosses), Part II is the HTML documentation (presented to suit the more visual, less technically inclined bosses), Part III is the UML diagram (presented to suit the even less technically inclined bosses). The three elements found below are in this document to appease all types of bosses and are expected in most design documents at most Software companies - even fictional ones! The three parts are described below:

Part I contains the code which would be implemented by our companies technicians. This code would actually be found on an IDE (Integrated Development Environment), which is basically where you write code. You don't use Pages to write code for example, you use an IDE. For this Unit, because I cannot demonstrate my code on an IDE through a PDF, I have simply copy and pasted the code from my IDE into this

document. This code contains keywords, fields, and features which would be understood to only someone who knew Java well (Java is my programming language of choice).

`public interface Board { :` the *public* keyword signifies this code can be viewed and accessed from anywhere in the rest of the code. The *interface* is a Java (Java is the programming language I am using) keyword which signifies all classes which implement this interface must inherit certain properties and features. *Board* is the name of my interface.

The code also contains methods (which do stuff when called), and fields (which are used to pass to these methods. Finally, the code in green beginning with * are what we programmer call 'comments' are only there to explain certain parts of the code. All of these keywords and terms are explained in more detail in the Glossary - Part IV.

Part II contains the HTML documentation which aims to give the viewer a clean view into the design of my Board interface. In truth, this document is to be viewed on a browser (Chrome, Safari etc), but for the purposes of this Unit, I have just taken screenshots of what the viewer might see. As one can see, the HTML doc contains a cleaner, more condensed way to view my code design, without getting caught up in the code and it's keywords themselves.

Part III is the UML (Unified Modelling Language) tree which shows the methods my design uses, as well as it's fields and it's types. It is suited to the bosses who understand Java but may not want to dive into the code. These bosses may also just be intrigued how my code would fit into the wider game, and this UML tree shows a succinct version of this.

The combination of all of these elements form the design document I would present to my bosses. They would view Part I in my IDE during the presentation, they would have Part II shown by me in a presentation, and they would either be sent Part III or it would exist in my presentation as a complimentary feature. My fictional bosses would then asses my design in it's total and give me another task - probably more implementation or design of another part of Santorini!

I. Code

```
/**
 * This interface represents the Board for a game of Santorini.
 * <p>
 * The Board is responsible for keeping track of the spaces that make up the
board, managing the
 * actual movement of workers and the building of floors, and producing
 * a JSON representation of the status of the board for the players to see.
 * <p>
 * The Board does not enforce game rules. The board only enforces rules
inherent to the board,
 * such as when a move is requested to a spot on the board that does not
exist, or when building
 * above four floors is requested.
 */
public interface Board {

  /**
   * Constant for the player turn index that is not a valid player, and
indicates
   * the absence of a worker on a cell.
   */
  int NO_WORKER = -1;

  /**
   * Constant for the maximum number of floors that can be built on a cell.
   */
  int MAX_FLOORS = 4;

  /**
   * This function is intended for use during the setup phase of the game,
when a player
   * initially places their worker. The side effect of this function is that
it places
   * a worker at the given location if the board's constraints are satisfied.
   *
   * @param x               represents the column number of the cell that the
player wants
   *                        to place a worker on. Column numbers are counted
left-to-right
```

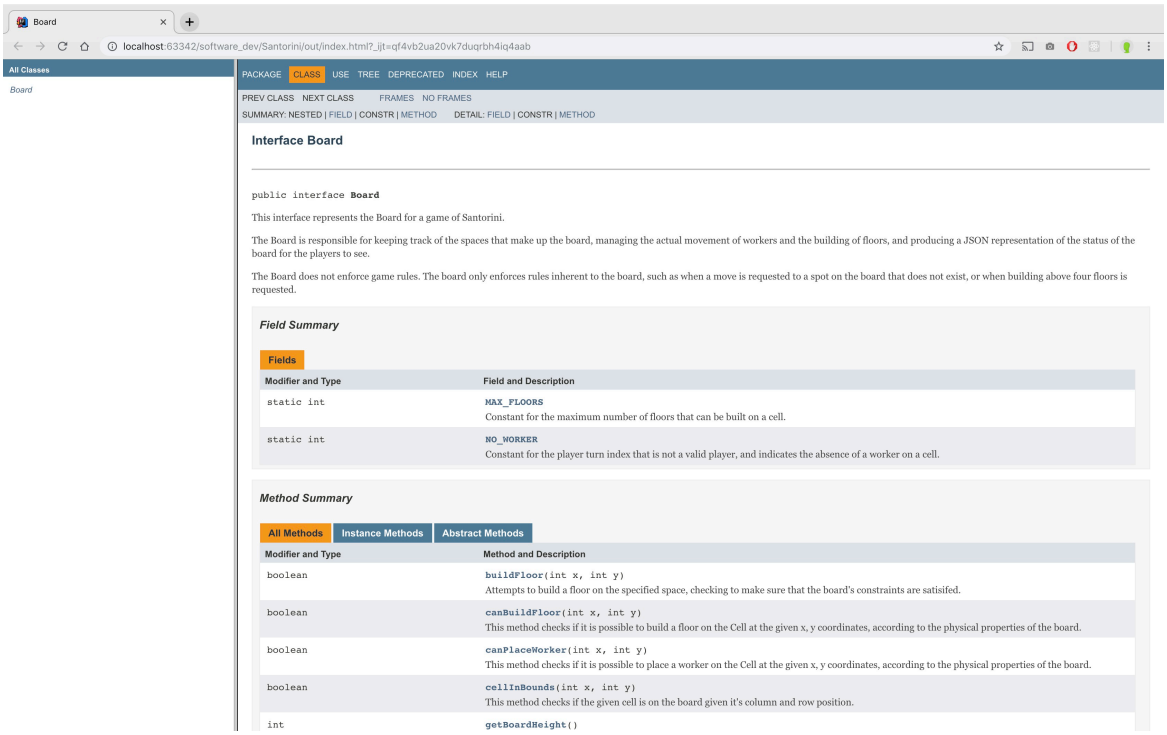
```
   *                         starting from 0.
   * @param y                represents the row number of the cell that the
player
   *                         wants to place a worker on. Row numbers are
counted top-to-bottom
   *                         starting from 0.
   * @param playerTurnIndex represents the index (starts from 0) position in
turn
   *                         order of the player who is placing a worker. It
is a
   *                         unique identifier for the player.
   * @return true if placing a worker was possible at the given location and
false if it was not.
   * Cases that will return false are: (a) the given location was outside the
bounds of
   * the board, (b) the given location is occupied.
   */
  boolean placeWorker(int x, int y, int playerTurnIndex);

  /**
   * Attempts to build a floor on the specified space, checking to make sure
that the board's
   * constraints are satisifed. The side effect of this function is that it
adds a floor to the
   * given location if the constraints are satisfied.
   *
   * @param x represents the column number of the cell that is being built
on. Column numbers are
   *          counted left-to-right starting from 0.
   * @param y represents the row number of the cell that is being built on.
Row numbers are counted
   *          top-to-bottom starting from 0.
   * @return true if building a floor was possible at the given location and
false if it was not.
   * Cases that will return false: (a) the given location was outside the
   * bounds of the board, (b) the cell has the maximum number of floors (4)
and another
   * could not be added.
   */
  boolean buildFloor(int x, int y);

  /**
   * Attempts to move a worker from the specified cell to the "to" specified
cell, checking to
   * make sure that the board's constraints are satisfied. The side effect of
this function is
   * that it removes the worker from the cell at the given location (fromX,
fromY) and adds the
   * worker to the given "to" location (toX, toY).
   *
   * @param fromX represents the column number of the cell from which the
player wants to move
   *              the worker. Column numbers are counted left-to-right
starting from 0.
   * @param fromY represents the row number of the cell from which the player
wants to move
   *              the worker. Row numbers are counted top-to-bottom starting
from 0.
```

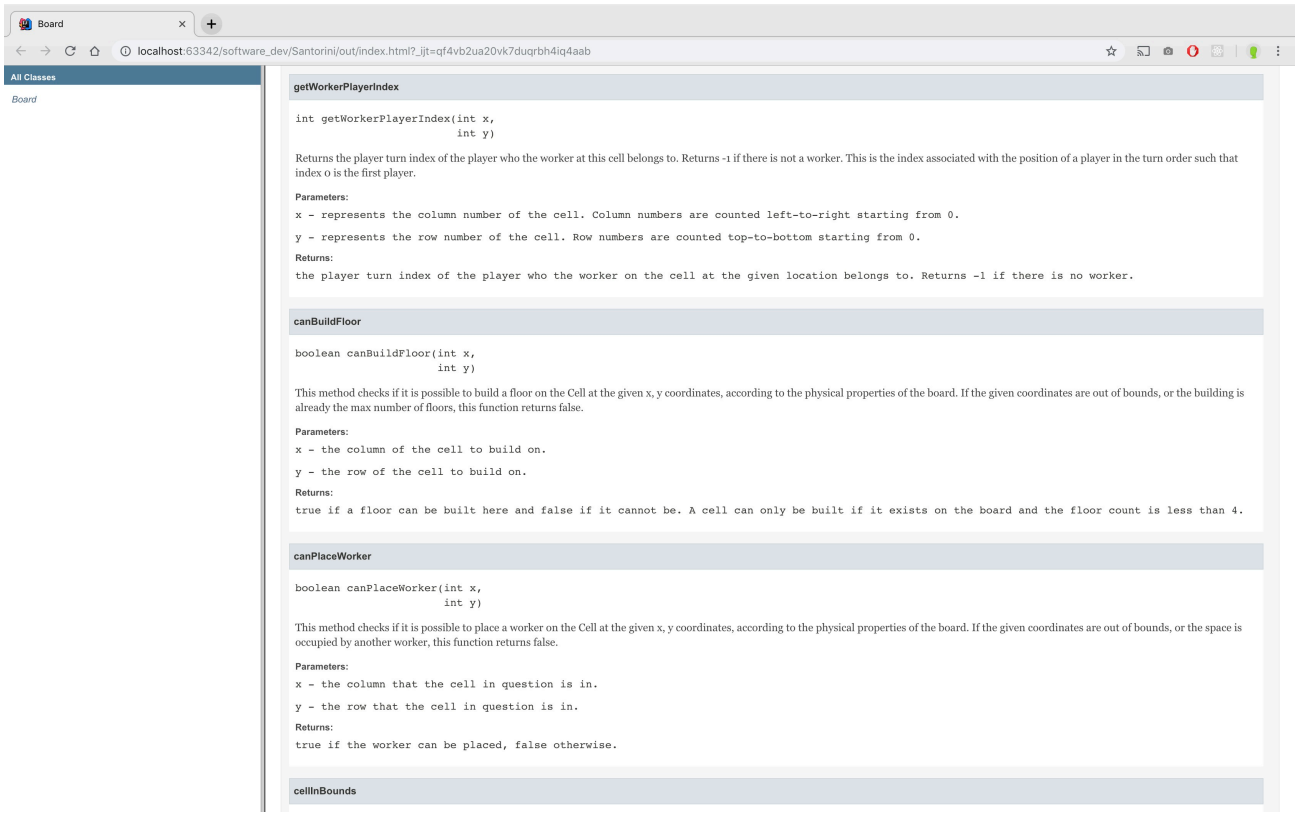
```
   * @param toX   represents the column number of the cell to which the
player wants to move
   *              the worker. Column numbers are counted left-to-right
starting from 0.
   * @param toY   represents the row number of the cell to which the player
wants to move
   *              the worker. Row numbers are counted top-to-bottom starting
from 0.
   * @return true if placing a worker was possible at the given location and
false if it was not.
   * Cases where it should return false are: (a) the given from location was
not the location
   * of a cell with a worker to move, (b) the given to location was not the
location of a cell on
   * the board, or (c) the given to location is occupied.
   */
  boolean moveWorker(int fromX, int fromY, int toX, int toY);

  /**
   * Returns the number of floors of the building at the given location.
   *
   * @param x represents the column number of the cell. Column numbers are
counted left-to-right
   *          starting from 0.
   * @param y represents the row number of the cell. Row numbers are counted
top-to-bottom starting
   *          from 0.
   * @return the number of floors of building on the cell at the given
location.
   */
  int getNumFloors(int x, int y);

  /**
   * Returns the player turn index of the player who the worker at this cell
belongs to. Returns -1
   * if there is not a worker. This is the index associated with the position
of a player in the
   * turn order such that index 0 is the first player.
   *
   * @param x represents the column number of the cell. Column numbers are
counted left-to-right
   *          starting from 0.
   * @param y represents the row number of the cell. Row numbers are counted
top-to-bottom starting
   *          from 0.
   * @return the player turn index of the player who the worker on the cell
at the given location
   * belongs to. Returns -1 if there is no worker.
   */
  int getWorkerPlayerIndex(int x, int y);

  /**
   * This method checks if it is possible to build a floor on the Cell at the
given x, y
   * coordinates, according to the physical properties of the board. If the
given
   * coordinates are out of bounds, or the building is already the max number
of floors,
```

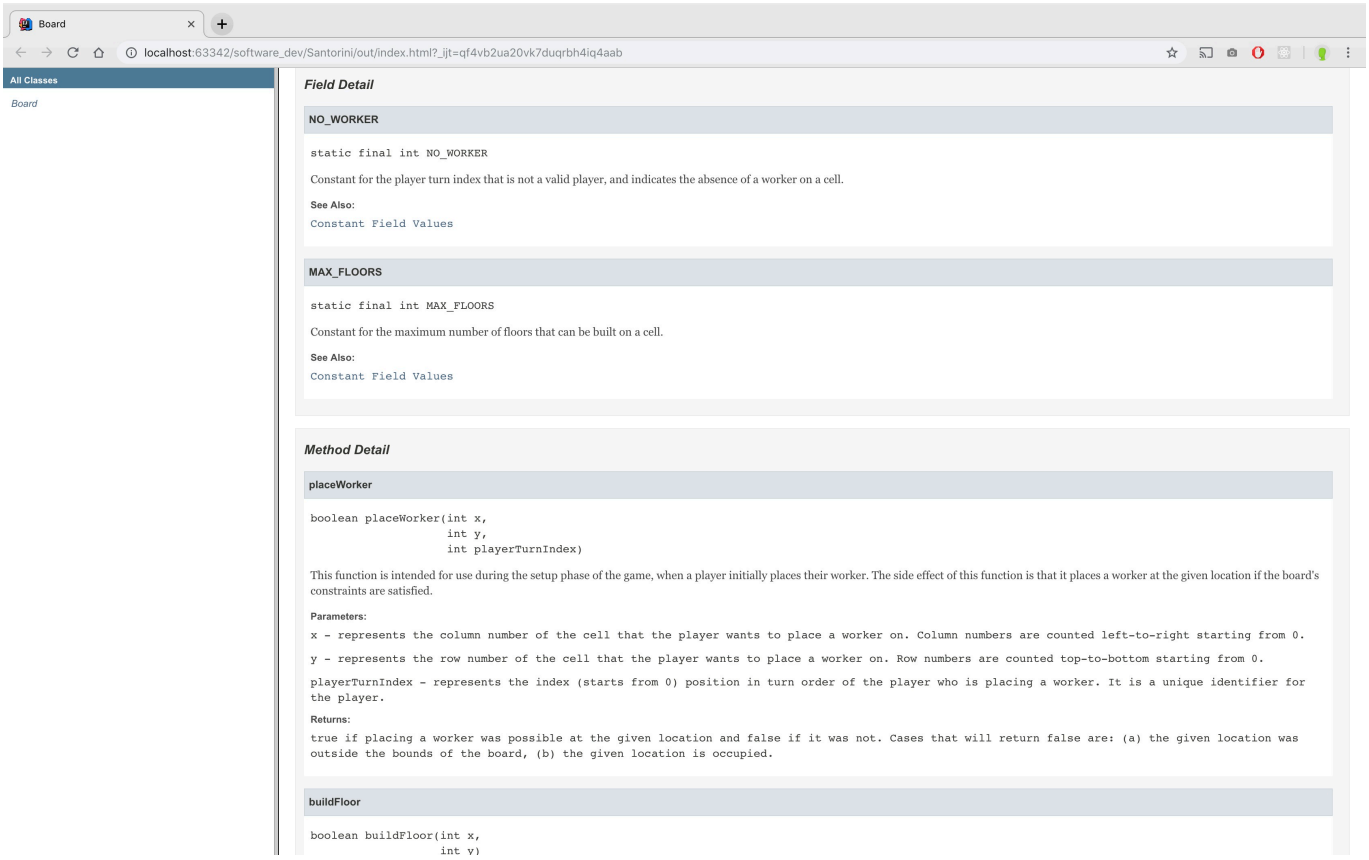
```
   * this function returns false.
   *
   * @param x the column of the cell to build on.
   * @param y the row of the cell to build on.
   * @return true if a floor can be built here and false if it cannot be. A
cell can only be built
   * if it exists on the board and the floor count is less than 4.
   */
  boolean canBuildFloor(int x, int y);

  /**
   * This method checks if it is possible to place a worker on the Cell at
the given x, y
   * coordinates, according to the physical properties of the board. If the
given
   * coordinates are out of bounds, or the space is occupied by another
worker,
   * this function returns false.
   *
   * @param x the column that the cell in question is in.
   * @param y the row that the cell in question is in.
   * @return true if the worker can be placed, false otherwise.
   */
  boolean canPlaceWorker(int x, int y);

  /**
   * This method checks if the given cell is on the board given it's column
and row position.
   *
   * @param x is the column that the cell in question is in.
   * @param y is the row that the cell in question is in.
   * @return true if the given cell position is on the game board.
   */
  boolean cellInBounds(int x, int y);

  /**
   * Checks if the two given cells are adjacent on the board, either
vertically,
   * horizontally, or diagonally.
   *
   * @param x1 is the column of the first cell.
   * @param y1 is the row of the first cell.
   * @param x2 is the column of the second cell.
   * @param y2 is the row of the second cell.
   * @return true if they are adjacent, false if not.
   */
  boolean isAdjacent(int x1, int y1, int x2, int y2);

  /**
   * Returns the height of the board space. (The number of rows.)
   *
   * @return the integer value that represents the height of the board (in
cells).
   */
  int getBoardHeight();

  /**
   * Returns the width of the board space. (The number of columns.)
```

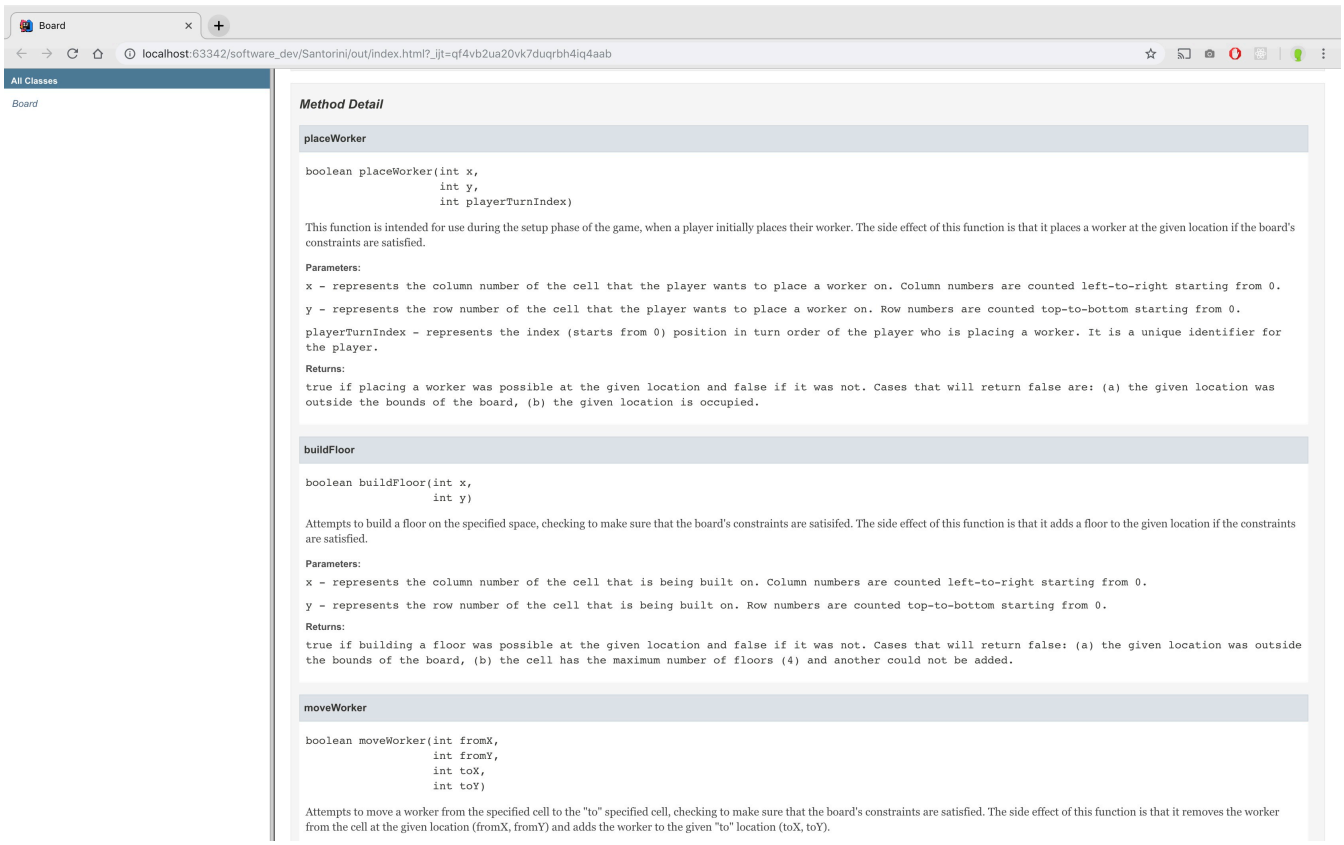
```
   *
   * @return the integer value that represents the width of the board (in
cells).
   */
  int getBoardWidth();

  /**
   * Copies this board into a new board and returns it.
   *
   * @return a new copy of this board.
   */
  Board getCopy();

  /**
   * Returns a String that conforms to the specified JSON representation of a
board.
   *
   * @return the JSON representation of the board according to the following
form:
   * [[[I, N] ...] ...] where [I, N] represents a cell, I is an integer
representing the player
   * turn index of the worker on this tile (if there is no worker the value
should be -1), and N
   * is a natural number representing the number of building floors on this
cell. This
   * representation is column-major (columns on the outside) and top-down,
meaning the first
   * cell listed (cell [0,0]) is in the top left corner of the board.
   */
  String toJSON();
}
```

II. HTML Javadoc

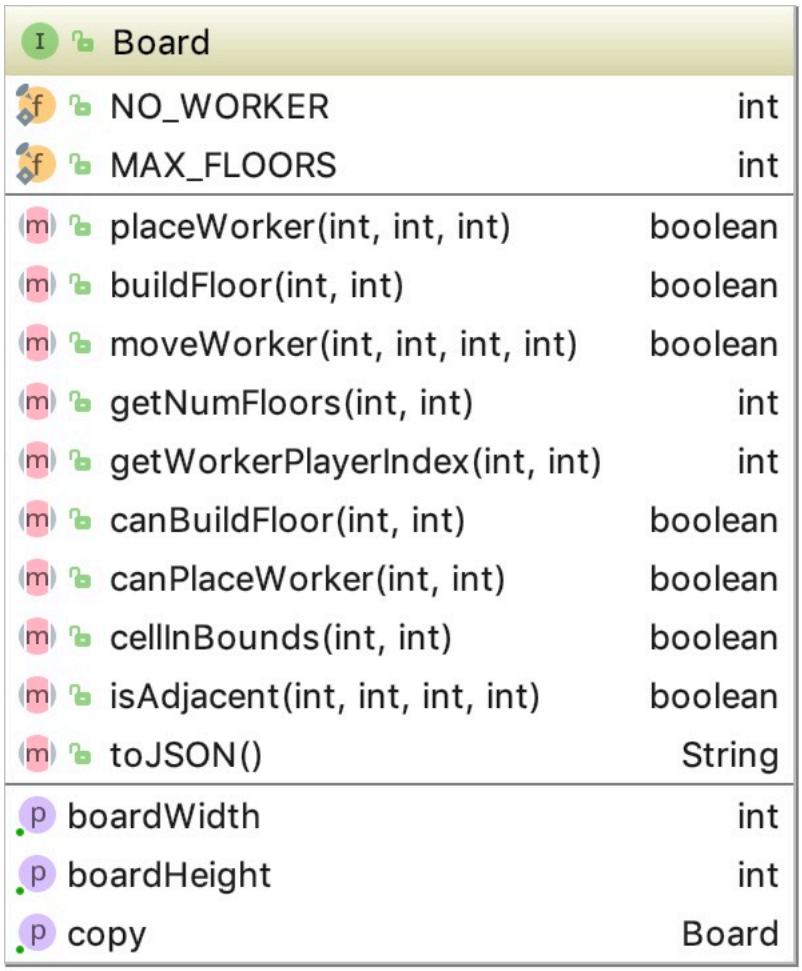





Board
localhost:63342/software_dev/Santorini/out/index.html?_ijt=qf4vb2ua20vk7duqrbh4iq4aab
All Classes
Board
Field Detail
NO_WORKER
static final int NO_WORKER
Constant for the player turn index that is not a valid player, and indicates the absence of a worker on a cell.
See Also:
Constant Field Values
MAX_FLOORS
static final int MAX_FLOORS
Constant for the maximum number of floors that can be built on a cell.
See Also:
Constant Field Values
Method Detail
placeWorker
boolean placeWorker(int x,
int y,
int playerTurnIndex)
This function is intended for use during the setup phase of the game, when a player initially places their worker. The side effect of this function is that it places a worker at the given location if the board's constraints are satisfied.
Parameters:
x - represents the column number of the cell that the player wants to place a worker on. Column numbers are counted left-to-right starting from 0.
y - represents the row number of the cell that the player wants to place a worker on. Row numbers are counted top-to-bottom starting from 0.
playerTurnIndex - represents the index (starts from 0) position in turn order of the player who is placing a worker. It is a unique identifier for the player.
Returns:
true if placing a worker was possible at the given location and false if it was not. Cases that will return false are: (a) the given location was outside the bounds of the board, (b) the given location is occupied.
buildFloor
boolean buildFloor(int x,
int y)


Board
localhost:63342/software_dev/Santorini/out/index.html?_ijt=qf4vb2ua20vk7duqrbh4iq4aab
All Classes
Board
Method Detail
placeWorker
boolean placeWorker(int x,
int y,
int playerTurnIndex)
This function is intended for use during the setup phase of the game, when a player initially places their worker. The side effect of this function is that it places a worker at the given location if the board's constraints are satisfied.
Parameters:
x - represents the column number of the cell that the player wants to place a worker on. Column numbers are counted left-to-right starting from 0.
y - represents the row number of the cell that the player wants to place a worker on. Row numbers are counted top-to-bottom starting from 0.
playerTurnIndex - represents the index (starts from 0) position in turn order of the player who is placing a worker. It is a unique identifier for the player.
Returns:
true if placing a worker was possible at the given location and false if it was not. Cases that will return false are: (a) the given location was outside the bounds of the board, (b) the given location is occupied.
buildFloor
boolean buildFloor(int x,
int y)
Attempts to build a floor on the specified space, checking to make sure that the board's constraints are satisifed. The side effect of this function is that it adds a floor to the given location if the constraints are satisfied.
Parameters:
x - represents the column number of the cell that is being built on. Column numbers are counted left-to-right starting from 0.
y - represents the row number of the cell that is being built on. Row numbers are counted top-to-bottom starting from 0.
Returns:
true if building a floor was possible at the given location and false if it was not. Cases that will return false: (a) the given location was outside the bounds of the board, (b) the cell has the maximum number of floors (4) and another could not be added.
moveWorker
boolean moveWorker(int fromX,
int fromY,
int toX,
int toY)
Attempts to move a worker from the specified cell to the "to" specified cell, checking to make sure that the board's constraints are satisfied. The side effect of this function is that it removes the worker from the cell at the given location (fromX, fromY) and adds the worker to the given "to" location (toX, toY).

Part III. UML Diagram



IV. Glossary

interface : A Java interface is a bit like a class, except a Java interface can only contain method signatures and fields. A Java interface cannot contain an implementation of the methods, only the signature (name, parameters and exceptions) of the method. An *Animal* interface may have classes which inherit it, such as a *Cat* or a *Dog*. Interfaces tend to be very general. Classes more specific and contains method implementations.

class : A basic unit of Object Oriented programming (which is style of programming). Symbolises a single object which would implement an interface and posses fields and methods. A *Cat* may be a class which implements an *Animal* interface.

method : Similar to a function. Does something, or operates on something. It is refereed to by name by other pieces of code and can be called (invoked) at any point in a program simply by utilising the method's name. A method is a subprogram that acts on data and often returns a value. A method for the interface *Animal* may be *eat(foodToBeEaten)*. The interface only declares this method, the class implements it (contains the logic of what might happen when you call *Cat.eat(broccoli)*)

public : Describes the visibility of the code that succeeds this keyword.

int : Integer.

boolean : True or false binary expression.

static : Single, immutable unit of a Java object. Only one of these exists in the entire program.

final : The object that succeeds this keyword cannot be changed after code compilation.

HTML : Hyper Text Markup Language is the standard programming language which browsers such as Chrome and Safari understand and show. *All* websites are made out of HTML code.

UML : Unified Modelling Language is the language which makes these tables. The tables are intended to provide a birds-eye view of the system or program in question.