MIT
6.102 Software Construction
Prof. Rob Miller

6.102 Spring 2023 Quiz 1

You have **50** minutes to complete this quiz. There are **5** problems, each worth approximately an equal share of points.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes on paper, and you may use blank scratch paper. You may not open or use anything else on your computer: no 6.102 website or readings; no VS Code, TypeScript compiler, or programming tools; no web search or discussion with other people.

Before you begin: you must *check in* by having the course staff scan the QR code at the top of the page.

To leave the quiz early: click the *done* button at the very bottom of the page and show your screen with the check-out code to a staff member.

This page automatically saves your answers as you work. If you see a stuck yellow spinner, red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz.

If you feel the need to write a note to the grader, you can click the gray pencil icon to the right of the answer.

Good luck!

You can **open this introduction in a separate tab**.

Connect-K (for some `k`) is a game with a grid of cells that stands upright on a table, `width` cells wide and `height` cells tall.

The columns are numbered left to right, 0 to `width-1`, and the rows are numbered bottom to top, 0 to `height-1`.

The game has two players: one player uses red pieces, and the other uses yellow pieces.

Players take turns choosing a grid column and adding one of their pieces to it, by dropping it into the column from above. The piece falls down the column to occupy the lowest empty cell, on top of any pieces that might already be in that column. A column that is completely full cannot be chosen for a move.

The first player to create a contiguous run of `k` of their own pieces (horizontally, vertically, or diagonally) wins the game.

If the grid is completely full of pieces with no winner, then the game is a draw.

In the rest of this quiz, `ConnectK` is an abstract data type that represents and plays games of Connect-K. This ADT has operations for:

- making a new game;
- making a move;
- asking whether or not the game is over, the winner of the game, and whether the game has reached a draw;

... and possibly other operations as well.

Problem ⤫1.

For one proposed design of `ConnectK`, here is a snippet of code:

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } );
game.play(CKPlayer.YELLOW, 3);
game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

Classify each of the distinct `ConnectK` operations shown in this code, using the names shown in the code. There may be more boxes than you need.

operation name	kind of operation
new ConnectK()	creator
play()	mutator
isGameOver()	observer

The precondition for the constructor includes the requirement that `k`, `width`, and `height` must be positive integers.

Suppose one `ConnectK` implementation, when the constructor is called with parameters that are not positive integers, returns a `ConnectK` object for which every other operation returns immediately with a constant value (violating the specs of those other operations). Say whether this implementation is breaking the `ConnectK` specification, and why or why not. Your answer should fit in the box without scrolling.

The implementation is not breaking the ConnectK spec. The client violated the precondition of the constructor, so the implementation does not have to satisfy the postcondition -- which means it is allowed to create and return a nonfunctional ConnectK object.

Regardless of the answer to the previous question, what principle is that implementation violating? Use at most 5 words, and be as specific as possible.

fail fast

Write TypeScript code for the `CKPlayer` data type that is consistent with the code above and that statically allows only two-player games. Your answer should fit in the box without scrolling.

enum CKPlayer { YELLOW, RED };

Problem ⤨2.

Ben Bitdiddle proposes this field as part of the representation for the `ConnectK` data type:

```
private grid: Array<Array<CKPlayer>>;
```

Ben observes that the empty cells of a `ConnectK` grid are always at the top of a column, and the pieces are always at the bottom, so `grid` can be arranged so that it only contains the pieces and doesn't need to contain empty cells.

Add enough fields to this representation to be able to implement the other desired operations of `ConnectK`. The representation should have as little redundant information as possible.

```
private readonly k: number;
private readonly height: number;
// width is not necessary; grid.length will store the width
// a field like "nextTurn: CKPlayer|undefined" is also be necessary if play() is required to verify that the player is allowed to play
```

Write a rep invariant for your representation. Your answer should fit in the box without scrolling.

grid[i].length <= height for all valid indexes i in grid
k and height are positive integers
(optional) at most one player has k in a row
(optional) | (# RED) - (#YELLOW) | <= 1

Write an abstraction function for your representation. Your answer should fit in the box without scrolling.

AF(grid, k, height) = the ConnectK game whose grid has width `grid.length` and height `height` and where the cell at row r, column c contains the piece grid[c][r] if r < grid[c].length or is empty otherwise, and a player needs `k` in a row to win the game.

Problem ⤫3.

Here are sketches for two operations for `ConnectK`:

- `isGameOver(): boolean` returns true if and only if the game is over
- `getWinner(): CKPlayer|undefined` returns the winner of the game, or `undefined` if the game is a draw

Complete the sketch for `getWinner` with two possible specs. One spec should add a precondition, and the other should not. One spec should be stronger than the other.

stronger `getWinner` spec:

requires: true
effects: returns the winner of the game, or undefined if the game is a draw or not over yet

weaker `getWinner` spec:

requires: game is over
effects: returns the winner of the game, or undefined if the game is a draw

Suppose you plan to start with one of these two specs; then implement and publish the `ConnectK` type so that clients start using it; and then switch to the other spec in the next release. Which spec should you start with, so that you have the freedom to switch to the other spec without affecting clients?

stronger `getWinner` spec [✓ weaker `getWinner` spec]

After choosing that spec and implementing it, Ben Bitdiddle points out that `isGameOver` and `getWinner` both contain code that scans the game grid looking for `k` in a row, which is wasteful.

Why is this wasteful from a code-style point of view? In at most 5 words, state what principle is being violated. Be as specific as possible.

don't repeat yourself

How can you address the problem in the implementations of these two operations, without changing the spec or rep of `ConnectK`? Your answer should fit in the box without scrolling.

move the grid-scanning into a helper method called by both functions

This is also wasteful from a performance point of view (program running time). How can you address the problem in the implementations of these two operations by changing the rep but **not** changing the spec or any other operation implementations of `ConnectK`? Your answer should fit in the box without scrolling.

cache the game-ending status (whether the game is over, who won, or whether it's a draw) as fields in the rep

Louis Reasoner complains that `isGameOver` and `getWinner` are observer operations, so they shouldn't mutate the rep. Ben Bitdiddle answers with a technical term from the theory of abstract data types. In 5 words or less, what idea are you using when you implement observers in this way? Be as specific as possible.

benevolent side effect

Problem ⤮4.

Your job is to develop a testing strategy for the play operation of ConnectK. You don't have an implementation of ConnectK yet. All you have is the description of the ConnectK game given in the introduction to this quiz, and the example code:

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } );
game.play(CKPlayer.YELLOW, 3);
game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

For each of the following proposed partitions, state a problem. Your answer should fit in the box without scrolling.

grid is empty, 50% full, or completely full

partition is not complete; "50% full" is ill-defined when grid is odd size; "completely full" may be an empty subdomain if play()'s precondition excludes making illegal moves

chosen column is negative, zero, or positive

"negative column" may be an empty subdomain if play()'s precondition excludes it; "positive" includes a boundary value (width)

player is a value of `CKPlayer`, or player is not a value of `CKPlayer`

"not a value of CKPlayer" is excluded by static type checking if play() has a reasonable type signature

Suppose you are now debugging a `play` implementation using the example code. Your debugger is currently paused at the start of the line containing the first `play` call, as shown on the left. You want to make the debugger reach the start of the next line, as shown on the right.

before

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } );
▶game.play(CKPlayer.YELLOW, 3);
game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

after

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } )
game.play(CKPlayer.YELLOW, 3);
▶game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

Describe two different ways to do this, using specific debugger commands. Each way should start with a different command.

step over

set a breakpoint on the second play() line, then continue to that breakpoint

Problem ⋊5.

Suppose you create an immutable version of Connect-K, called `ImConnectK`, whose specification is as similar as possible to `ConnectK`.

For this example code:

```
const game = new ConnectK( { k: 4, width: 10, height: 5 } );
game.play(CKPlayer.YELLOW, 3);
game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

What edits are necessary to make this code use `ImConnectK` instead of `ConnectK`? You can either describe the specific edits, or rewrite the code if you prefer. Your answer should fit in the box without scrolling.

```
let game = new ImConnectK({ k: 4, width: 10, height: 5 });
game = game.play(CKPlayer.YELLOW, 3);
game = game.play(CKPlayer.RED, 5);
assert( ! game.isGameOver() );
```

For this code:

```
g1 = new ConnectK( { k: 4, width: 10, height: 5 } );
g2 = new ConnectK( { k: 4, width: 10, height: 5 } );
g3 = new ImConnectK( { k: 4, width: 10, height: 5 } );
g4 = new ImConnectK( { k: 4, width: 10, height: 5 } );
```

Group g1, g2, g3, g4 according to observational and behavioral equivalence. Place every value in a box, such that equivalent values are in the same box, and non-equivalent values are in different boxes. There may be more boxes than you need.

Observational equivalence of g1, g2, g3, g4:

g1, g2	g3, g4		

Behavioral equivalence of g1, g2, g3, g4:

g1	g2	g3, g4	

Suppose `ImConnectK` is still using this field in its rep:

```
private grid: Array<Array<CKPlayer>>;
```

... as its primary representation of the game grid. Ben Bitdiddle points out that this allows for a lot of sharing of data between different `ImConnectK` instances, because each move of the game only affects one column. The unchanged columns can be shared between the `ImConnectK` instances representing the game before and after the move.

Louis Reasoner complains that Ben's idea (sharing data between different `ImConnectK` instances) leads to *aliasing* and *rep exposure*. In what sense is Louis right, and in what sense is he wrong? Address each of the terms he mentioned in the corresponding box. Your answers should fit in the boxes without scrolling.

aliasing

Louis is right that aliasing *does* exist -- the reps of different ImConnect instances are pointing to some of the same column arrays. But ImConnect objects do not mutate their grids once created, so as long as play() is careful to copy the changed column, no aliased columns will be mutated.

rep exposure

Louis is wrong that this is rep exposure -- the sharing is not between client and implementation, but within the implementation.