6.102 Software Construction
Prof. Rob Miller and Max Goldman

rcm

6.102 Spring 2024 Quiz 1

You have **80** minutes to complete this quiz. There are **5** problems.

The quiz is **closed-book** and closed-notes, but you are allowed one two-sided page of notes handwritten directly 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: enter *done* 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**.

A *sliding-tile puzzle* is a square two-dimensional grid containing N *tiles* and one *hole*, initially in scrambled order. To make a move, the player slides a neighboring tile horizontally or vertically into the hole, so that the tile and the hole switch places. The puzzle is solved when all the tiles are in order (reading left to right, and top to bottom) and the hole is in the bottom-right cell.

A common sliding-tile puzzle is the *N-puzzle*, where the tiles are the numbers 1 to N, and N+1 is a perfect square. A *15-puzzle* looks like this when it is solved:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 __
```

Tiles aren't limited to numbers; for example, they may be pieces of a picture that is formed when the puzzle is solved.

Because tiles can vary, a sliding-tile puzzle should specify its tile set explicitly. For example, the sliding-tile puzzle whose tiles are the numbers 1 to 8 can be named as any of the following:

- the sliding-tile puzzle with tiles `{1...8}`
- the N-puzzle where N=8
- the 8-puzzle

... but just calling it a "sliding-tile puzzle" would be insufficient.

This quiz uses these ADTs:

- `NPuzzle` represents a mutable sliding-tile puzzle whose tiles are numbered 1...N, where N+1 is a perfect square.
- `Puzzle<T>` represents a mutable sliding-tile puzzle whose tiles come from a type `T`.

These ADTs may have operations for:

- making a new puzzle;
- sliding a tile;
- examining the grid;
- asking whether or not the puzzle is solved;

... and possibly other operations as well. Note that the problems in this quiz are independent of each other; an operation mentioned in one problem does not necessarily exist in a different problem.

Problem ⤭1.

For the rep below, write a rep invariant and abstraction function. Each answer should fit in the box without scrolling.

```
/** Mutable type representing a sliding-tile puzzle where the tiles are 1...N, and N+1 is a perfect sc
class NPuzzle {
    private grid: Array<Array<number>>;
    // for example,
    //   grid = [ [ 3, 1 ], [ -1, 2] ]
    // represents the 3-puzzle:
    //   3 1
    //   _ 2
}
```

Rep invariant:

/* squareness */ grid[r].length === grid.length for all 0 <= r < grid.length
/* legal tiles/hole */ all numbers in grid are integers in the set { -1 } union { 1, ..., grid.length^2 - 1 }
/* uniqueness */ all numbers in grid occur exactly once

Abstraction function:

AF(grid) = the N-puzzle where N = grid.length^2 - 1, where location (row,column) is the hole if grid[row][[column] = -1, otherwise contains the tile labeled grid[row][column], for 0 <= row, column < grid.length

```
/** Mutable type representing a sliding-tile puzzle where the tiles are 1...N, and N+1 is a perfect sc
class NPuzzle {
    private loc: Map<number, {r: number, c:number}>;
    // for example,
    //   loc = {
    //      3: { r: 0, c: 0},
    //      1: { r: 0, c: 1},
    //      2: { r: 1, c: 1}
    //   }
    // represents the 3-puzzle:
    //   3 1
    //   _ 2
}
```

Rep invariant:

/* squareness */ loc.size()+1 is a perfect square
/* legal tiles */ the set of keys in loc = the set of integers { 1, ..., loc.size() }
/* legal coordinates */ for all { r, c } values in loc, r and c are integers such that 0 <= r,c, < sqrt(loc.size()+1)
/* uniqueness */ every { r, c } in loc is a unique (r,c) pair

Abstraction function:

AF(loc) = the N-puzzle where N = loc.size(), where tile k is in location (loc.get(k).r, loc.get(k).c), for k in {1...N}

```
/** Mutable type representing a sliding-tile puzzle where the tiles are 1...N, and N+1 is a perfect sc
class NPuzzle {
    private row: Map<number, number>;
    private col: Map<number, number>;
    // for example,
    //   row = { 3: 0,  1: 0,  2: 1 }
    //   col = { 3: 0,  1: 1,  2: 1 }
    // represents the 3-puzzle:
    //   3 1
    //   _ 2
}
```

Rep invariant:

/* squareness */ row.size()+1 is a perfect square
/* legal tiles */ the set of keys in row = the set of keys in col = the set of integers { 1, ..., row.size() }
/* legal coordinates */ all values in row and col are integers i such that 0 <= i < sqrt(row.size()+1)
/* uniqueness */ every pair (row.get(k), col.get(k)) is unique (for all keys k in row)

Abstraction function:

AF(row,col) = the N-puzzle where N = row.size(), where tile k is in location (row.get(k), col.get(k)), for k in {1...N}

Problem ⤮2.

Louis Reasoner suggests a possible rep for an implementation of `StringPuzzle`, which has tiles which can be strings:

```
private cells: Array<string>;
private holeIndex: number;
```

For example, this rep:

```
cells = [ 'R', 'F', 'anything', 'C' ]
holeIndex = 2
```

...represents the 3-puzzle:

```
R F
_ C
```

As a further example, Louis shows that sliding the `R`-tile down could then update the rep to this:

```
cells = [ 'R', 'F', 'R', 'C' ]
holeIndex = 0
```

...which represents the new state of the puzzle with the `R`-tile moved:

```
_ F
R C
```

Alyssa Hacker observes that Louis must be thinking about a particular abstraction function, even though he hasn't said what it is. Based on Louis's description, give two different rep values that map to the same abstract value, or say why this isn't possible. Your answers should fit in the boxes without scrolling.

cells = ['a', 'b', 'c', 'd']
holeIndex = 0

cells = ['Z', 'b', 'c', 'd']
holeIndex = 0

Louis has also written `equalValue()` for his rep. His implementation looks like this:

```
/**
 * @param that  value to compare `this` with
 * @returns true iff this and that represent the same abstract value.
 */
public equalValue(that: StringPuzzle): boolean {
    return this.holeIndex === that.holeIndex && this.cells === that.cells;
}
```

In code review, Alyssa raises two issues about Louis's implementation. Fill in the blank in each of her comments. Your answers should fit in the boxes without scrolling.

"This implementation of `equalValue` is not consistent with the way TypeScript's built-in equality operations work, because ____________________."

... the === operator is referential equality on arrays, but Louis needs it to be value equality

"This implementation of `equalValue` is not consistent with your abstraction function, because ____________________."

... it should not require that this.cells[this.holeIndex] is equal to that.cells[that.holeIndex], because the abstraction function ignores that cell

Alyssa continues: (fill in the blank, fitting in the box without scrolling).

"But at least this implementation of `equalValue` is *symmetric*, because ______________."

=== is symmetric, so a.holeIndex === b.holeIndex and a.cells === b.cells have the same results as b.holeIndex === a.holeIndex and b.cells === a.cells (respectively) no matter what values are used for `a` and `b`

Alyssa continues: (fill in the blank, fitting in the box without scrolling).

"But at least this implementation of `equalValue` is *reflexive*, because ______________."

=== is reflexive, so a.holeIndex === a.holeIndex && a.cells === a.cells is always true no matter what value `a` is used for `this`

Alyssa recommends rewriting Louis's `equalValue()` using functional programming. Here are the library functions Alyssa uses:

```
// filters iterable for only those elements for which f() returns true, when f is given both the eleme
function filterWithIndex(iterable:Iterable<T>, f:(e:T,index:number)=>boolean):Iterable<T>;

// applies binary function f() to corresponding elements of iterable1 and iterable2, and returns seque
function map2(iterable1:Iterable<T1>, iterable2:Iterable<T2>, f:(e1:T1,e2:T2)=>U):Iterable<U>;

// returns true if and only if all elements of iterable are true
function all(iterable:Iterable<boolean>):boolean;
```

And here is her implementation, with some blanks to fill in:

```
/**
 * @param that  value to compare `this` with
 * @returns true iff this and that represent the same abstract value.
 */
public equalValue(that: StringPuzzle): boolean {
  const exceptHole = _____________;
  const compareTiles = ______________;
  return this.holeIndex === that.holeIndex
      && this.cells.length === that.cells.length
      && all( map2( filterWithIndex(this.cells, exceptHole),
                    filterWithIndex(that.cells, exceptHole),
                    compareTiles ) );
}
```

Fill in the blanks below. Your answers should fit in the boxes without scrolling.

`exceptHole`

(cell:string, index:number): boolean => index !== this.holeIndex

`compareTiles`

(tile1:string, tile2: string): boolean => tile1 === tile2

Problem ⤨3.

Consider this operation on `NPuzzle`:

```
/** Mutable type representing a sliding-tile puzzle where the tiles are 1...N, and N+1 is a perfect sc
class NPuzzle {
    ...

    /**
     * @param tile must be an integer in {1...N}
     * @returns true if and only if the tile numbered `tile` in this puzzle
     *          is horizontally or vertically adjacent to the hole
     */
    public canSlide(tile: number): boolean
}
```

Critique the following proposed partition of `canSlide`. Your answer should fit in the box without scrolling.

`tile < N ;   tile = N ;   tile > N`

Includes a subdomain that contains no possible test cases: tile cannot be > N.

hole is in a corner; hole is in middle of grid

Not complete: no subdomain covers cases where the hole is on the edge, for example.

Propose a good partition for this function, substantially different from the partition above. Your answer should fit in the box without scrolling.

result is true (tile is next to the hole) or false (tile is not next to the hole).

Problem ⤨4.

Consider this operation, whose spec is incomplete:

```
/** Mutable type representing a sliding-tile puzzle where the tiles are 1...N, and N+1 is a perfect sc
class NPuzzle {
    ...

    /**
     * Slides the tile at location `r`,`c` in the grid horizontally or vertically.
     * TBD
     */
    public slideFrom(r: number, c: number): void
}
```

Write a spec, in TypeDoc syntax that can be substituted in place of TBD in the comment above, that includes explicit preconditions for anything that might otherwise be implicit, so that the implementer can always do what the first sentence of the spec comment says.

@param r   row, must be integer in [0, sqrt(N+1))
@param c column, must be integer in [0, sqrt(N+1)). Hole must be adjacent to the tile at location `r`,`c`.

We'll call the spec you just wrote above **spec A**.

Now write a different spec, **spec B**, also in TypeDoc syntax as a substitution for TBD, that replaces all those explicit additional preconditions with postconditions. Your answer should fit in the box without scrolling.

@throws error if either r or c is not an integer in [0, sqrt(N+1)), or if the hole is not adjacent to the tile at location `r`,`c`.

Spec B still has a precondition, that was part of the original description of `slideFrom()` above. What is that precondition?

r and c are numbers

Now write a different spec, **spec C**, also in TypeDoc syntax as a substition for TBD, that expects the implementer to slide other tiles if needed to move the specified tile. Spec C must be **underdetermined**. Your answer should fit in the box without scrolling.

Moves as few other tiles as possible so that the specified tile can be moved.
@throws error if either r or c is not an integer in [0, sqrt(N+1))

Is your spec A stronger than, weaker than, or incomparable to your spec B? Explain why. Your answer should fit in the box without scrolling.

Weaker, because A's precondition is stronger than B's precondition, and for inputs that satisfy both preconditions, the postconditions of A and B are the same.

Is your spec A stronger than, weaker than, or incomparable to your spec C? Explain why. Your answer should fit in the box without scrolling.

Weaker, because A's precondition is stronger than C's precondition, and for inputs that satisfy both preconditions, the postconditions of A and C are the same.

Is your spec B stronger than, weaker than, or incomparable to your spec C? Explain why. Your answer should fit in the box without scrolling.

Incomparable, because the postconditions of B and C require different results when the hole is not adjacent to the specified tile (B requires throwing an error, while C requires moving other tiles so that the specified tile can be moved).

Problem ⤭5.

Here is some more detail of `NPuzzle` and `Puzzle<T>`:

```
/** Mutable type representing a sliding-tile puzzle where the tiles are 1...N, and N+1 is a perfect sc
class NPuzzle {
    /** Makes a new randomly-shuffled N-puzzle (N+1 must be a perfect square). */
    public constructor(N: number) { ... }

    ... // assume rest of NPuzzle is implemented correctly
}

/** Mutable type representing a sliding-tile puzzle where the tiles have type T,
    where T uses === for equality. */
interface Puzzle<T> {
    /** Slide a tile horizontally or vertically. ... */
    public slideTile(tile: T): void;

    /** true iff puzzle is solved */
    public isSolved(): boolean;

}
```

In this problem, sliding-tile puzzles have **only** these operations. Do not implement or use operations that were mentioned in previous problems.

Rewrite the first line of the `NPuzzle` class definition so that it is declared to be a subtype of `Puzzle<T>`.

class NPuzzle implements Puzzle<number> {

Assume this change to `NPuzzle` has been made and the rest of `NPuzzle` is now implemented correctly, again with **only** the operations from this problem, not from previous problems.

Alyssa Hacker suggests using `NPuzzle` in the rep for puzzles that use an arbitrary tile type `T`. Here is her rep, with an example:

```
    private puzzle: NPuzzle;
    private solutionOrder: Array<T>;
    // for example, if T is string, then
    //    puzzle =  2 _
    //              1 3
    //    solutionOrder = [ 'S', 'F', 'B' ]
    // represents the puzzle state:
    //        F _
    //        S B
```

Write an implementation of the `Puzzle<T>` interface using Alyssa's suggestion, which:

- uses Alyssa's rep exactly as is – just write **REP** in your code where the rep should go
- **omits all comments** (like specs and AF/RI), omits blank lines, and omits `checkRep()`
- offers some way to create an instance of `Puzzle<T>`
- has rep independence
- has no rep exposure

You may assume the strongest or weakest spec you want for the given operations, as long as they are still useful.

Your answer should be a complete TypeScript class definition, and should fit in the box without scrolling. (Remember to omit all comments and all blank lines.)

```
class AlyssaPuzzle<T> implements Puzzle<T> {
  REP
  constructor(solutionOrder: Array<T>) {
    this.puzzle = new NPuzzle(solutionOrder.length);
    this.solutionOrder = solutionOrder.slice(0);
    // or [...solutionOrder] or other ways to defensively-copy array
  }
  public slideTile(tile: T): void {
    this.puzzle.slideTile(this.solutionOrder.indexOf(tile) + 1);
  }
  public isSolved(): boolean {
    return this.puzzle.isSolved();
  }
}
```

Classify each of the operations of your implementation. There may be more boxes than you need.

operation name	kind of operation
constructor	creator
slideTile	mutator
isSolved	observer

Louis Reasoner wants to make a 3-puzzle (a puzzle whose tiles are the numbers 1, 2, 3):

```
const my3Puzzle: Puzzle<number> = _________________ ; // make a 3-puzzle
```

Based on the code you wrote above, write two different ways to fill in the blank, which have **different dynamic types**:

new NPuzzle(3)

new AlyssaPuzzle<number>([1,2,3])