MIT
6.031 Software Construction
Prof. Rob Miller and Max Goldman

6.031 Spring 2021 Quiz 1

You have 50 minutes to complete this quiz. There are 5 problems. The quiz is open-book: you may access any 6.031 or other resources, but you may not communicate with anyone except the course staff.

This page automatically saves your answers as you work. Saved answers are marked with a green cloud-with-up-arrow icon. 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. There is no 'save' or 'submit' button.

If you want to ask a clarification question, visit whoosh.mit.edu/6.031 and click "raise hand" to talk to a staff member.

Good luck!

When the quiz starts, **before you begin**, please sign this honor statement.

> I affirm that I will not communicate with classmates or anyone else (other than 6.031 staff members) about anything related to this quiz until the solutions are officially released.

By entering your full name below (first and last name), you agree to this honor statement.

Problem	Points
1	24
2	16
3	16
4	26
5	18

In this quiz you will work with an abstract datatype **Duck** and some other associated ADTs. For now, `Duck` declares a single instance method:

```
public enum Squeak { LOUD, QUIET }

// A rubber duck that can squeak loudly a certain limited number of times.
public interface Duck {
    // If this duck has loud squeaks remaining,
    //   returns LOUD and modifies this duck to reduce that amount by one.
    // Otherwise, returns QUIET.
    public Squeak squeak();
}
```

For brevity, the quiz will use `// comments like this` instead of `/** JavaDoc comments */` for specifications.

Problem ⤭1. (24 points)

(⤭a) With respect to `Duck`, what **two** kinds of ADT operation is `squeak()`? *Answer with two words.*

Consider this `Duck` whose rep is a `List` of booleans:

```
public class ArrayDuck implements Duck {
    private List<Boolean> squeakiness;

    // Creates a new rubber duck that can squeak loudly 4 times.
    public ArrayDuck() { ... }

    @Override public Squeak squeak() { ... }
}
```

(⤭b) Alyssa proposes the following rep invariant:

-Either- `squeakiness contains only `true` values`
-or- `squeakiness contains only `false` values`

Write an abstraction function given this rep invariant:

And implement the `ArrayDuck()` constructor to satisfy its spec:

```
public ArrayDuck() {
```

```
}
```

(⤭c) Ben proposes a different rep invariant, using the same `List`-of-booleans rep:

```
squeakiness has length 3
```

Write an abstraction function given this rep invariant: *(Feeling stuck? Try counting in binary.)*

And implement the `ArrayDuck()` constructor to satisfy its spec:

```
public ArrayDuck() {
```

```
}
```

(⤭d) Write a safety from rep exposure argument that will hold for either of those rep invariants and any correct implementations of the constructor and `squeak()`:

Problem ⤭2. (16 points)

(⤭a) Louis creates this `Duck`:

```
// A rubber duck that can squeak loudly zero times.
public class QuietDuck implements Duck {
    @Override public Squeak squeak() { return Squeak.QUIET; }
}
```

And suggests that the entire program can share the following global value:

```
public static final Duck QUIET_DUCK = new QuietDuck();
```

... but receives the cryptic code review: "not RFC, future additions to `Duck` interface."

Specifically, what kind of future additions to the `Duck` interface is the reviewer most worried about?

And what is the most serious problem they pose for the shared `QUIET_DUCK`?

(⤭b) Chastened, Louis creates this `Duck` instead:

```
// A rubber duck that can squeak loudly infinitely many times.
public class LoudDuck implements Duck {
    @Override public Squeak squeak() { return Squeak.LOUD; }
}
```

... but receives the cryptic code review: "not SFB, spec!"

Specifically, what about `LoudDuck` as specified is a bug?

And give an example of how a correct client of `Duck` could be broken as a result:

Problem ⤭3. (16 points)

(⤭a) Charlie wishes to add a new operation to `Duck`:

-Either-

```
// Modifies this duck by setting
//   new number of loud squeaks remaining = base^(old number of loud squeaks remaining).
// Requires positive base.
public void powerUp(int base)
```

Charlie has already come up with a partition on `base`.

Write an excellent input space partition, with **3 or more subdomains**, that is **not** a partition on the value of `base`:

-or-

```
// Modifies this duck by setting
//   new number of loud squeaks remaining = (old number of loud squeaks remaining)^expo.
// Requires positive expo.
public void powerUp(int expo)
```

Charlie has already come up with a partition on `expo`.

Write an excellent input space partition, with **3 or more subdomains**, that is **not** a partition on the value of `expo`:

(⤭b) And one more:

```
// Requires nonempty row sorted in ascending order of number of loud squeaks remaining.
// Returns i such that row[i] is the first duck with more loud squeaks remaining than
//   this duck, or the length of row if there is no such duck.
public int placeInRow(List<Duck> row)
```

Charlie has already come up with a partition on the length of `row`.

Write an excellent input space partition, with **3 or more subdomains**, that is **not** a partition on the length of `row`:

Problem ⤭4. (26 points)

Yolanda is also building a new feature: rubber duck debugging.

Suppose that **`Problem`** and **`Solution`** are immutable ADTs. Consider this `Duck`:

```
// A debugging rubber duck that can remember its solutions to problems.
public class DebugDuck implements Duck {

    // Creates a new debugging rubber duck that can squeak loudly n times.
    // Requires nonnegative n.
    public DebugDuck(int n) { ... }

    @Override public Squeak squeak() { ... }

    // If this duck has returned at least one solution to the given problem before,
    //   or if this duck has loud squeaks remaining,
    //   returns a nonempty list of solutions to the problem.
    // Otherwise, returns null.
    public List<Solution> debug(Problem problem) { ... }
}
```

(⤭a) The fact that `DebugDucks` can, under the right conditions, return `Solutions` to any `Problem` is amazing! But the spec of `debug(..)` could be improved. Critique that spec: what would be your first and most important piece of code review feedback?

(⤭b) State two **good** different ways to revise the spec to solve this problem, giving rewritten versions of the relevant line(s) in the spec. Do **not** add new preconditions. Your revisions must retain the same essential functionality, but achieve it in a different way.

(⤭c) Compare the original spec to this shorter alternative:

```
// Returns a nonempty list of solutions to the given problem if this duck
//   has loud squeaks remaining. Otherwise, returns null.
```

This alternative **pre**condition is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

Given the preconditions, this alternative **post**condition is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

This alternative **specification** is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

(⤭d) Compare the original spec to this shorter alternative:

```
// This duck must have loud squeaks remaining.
// Returns a nonempty list of solutions to the given problem.
```

This alternative **pre**condition is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

Given the preconditions, this alternative **post**condition is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

This alternative **specification** is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

(⤭e) Compare the original spec to this alternative, identical except for the 3rd line:

```
// ... first 2 lines unchanged ...
//   returns a nonempty list of distinct solutions to the problem.
// ... last line unchanged ...
```

This alternative **pre**condition is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

Given the preconditions, this alternative **post**condition is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

This alternative **specification** is:

- ○ stronger
- ○ weaker
- ○ identical
- ○ incomparable

Problem ⤭5. (18 points)

Zach loves the idea of rubber duck debugging, but does not want mutable ducks and makes a new `ImDuck` interface.

He has also been spending quarantine with only rubber ducks to talk to. (That may not have been healthy, but he does have a more realistic estimate of their debugging capabilities than Yolanda.)

```
// Immutable debugging rubber duck, with a call sign.
public interface ImDuck {
    // Make a new duck with the given call sign (in the squeak language of rubber ducks).
    public static ImDuck make(List<Squeak> callsign) { ... }

    // Returns the call sign of this duck (in the squeak language of rubber ducks).
    public List<Squeak> callsign();

    // Returns LOUD if and only if problem contains this duck's call sign as a subsequence.
    public Squeak debug(List<Squeak> problem);
}
```

(⤭a) With respect to `ImDuck`, what kind(s) of ADT operation is `make(..)`? *Answer with one or more words.*

(⤭b) With respect to `ImDuck`, what kind(s) of ADT operation is `callsign()`? *Answer with one or more words.*

(⤭c) Really quite worried about mutability, Zach tries to implement `StringImDuck` as follows:

```
1   public class StringImDuck implements ImDuck {
2       private final String myCallsign;
3       // Make a new duck with the given call sign.
4       public StringImDuck(List<Squeak> callsign) {
5           this.myCallsign = callsign.remove(0).name();
6           while (callsign.size() > 0) {
7               this.myCallsign += "," + callsign.remove(0).name();
            }
        }
        // ... other operations ...
```

Note two things that work as Zach seems to expect:
– `remove(..)` returns the removed `List` item,
– enum instance method `name()` returns the name of that enum constant as a `String` (in this case, `"LOUD"` or `"QUIET"`).

Other than that, this code for `StringImDuck()` is pretty buggy. Write three distinct reasons why the code is wrong. Do **not** make assumptions about the RI or AF. Do **not** state different instances of the same conceptual flaw: state three different problems, one in each box.

(⤭d) Suddenly concerned about space efficiency, Zach tries to implement `EncodedImDuck` as follows:

```
public class EncodedImDuck implements ImDuck {
    private final List<Integer> myCallsign;
    // Make a new duck with the given call sign.
    public EncodedImDuck(List<Squeak> callsign) {
        // encode alternating numbers of consecutive squeaks of the same kind,
        //   starting with LOUD
        // for example, callsign = [ QUIET, LOUD, LOUD, LOUD, QUIET ]
        //   becomes this.myCallsign = [ 0, 1, 3, 1 ]
    }
    // ... other operations ...
```

The idea is to use a *run-length encoding* in the rep: every subsequence of *n* repeated LOUD or QUIET squeaks is stored as just the integer *n*.

Assume the abstraction function works in the most straightforward way to reverse this encoding: if `myCallsign[i]` = *n*, that represents a length-*n* run of `LOUD` (if `i` is even) or `QUIET` (if `i` is odd) squeaks in the duck's call sign.

Write a rep invariant for `EncodedImDuck` such that every abstract `ImDuck` can be represented in exactly one way:
(Feeling stuck? Start with any RI, then strengthen.)