

# CS:5810 Formal Methods in Software Engineering

## Fall 2025

### Mini Project 3

**Due:** Tuesday, December 9, by 11:59pm

Download the accompanying zip file `code.zip`, expand it, and work on the files in the folder `code`. After completing them as specified below, type your name(s) as indicated in the files, zip the folder, and submit the zip file on ICON. *Only one team member should submit the zip file*, but make sure to write down the name of both team members in the source files.

This project has four parts, with Part D optional, for extra credit. The parts can be done independently from each other, with the exception that Part C should be done after Part A.

The provided code has been written modularly using Dafny's modules. You do not need any deep knowledge of modules for this assignment, however. A cursory reading of Chapter 9 of the textbook should suffice.

The project is meant to test your ability to understand programs in the Dafny language, annotate them with specs, and check their correctness. Use Dafny to verify that your programs are correct with respect to their specification. The implementation effort in this project requires only very basic knowledge of object-oriented programming. *You are strongly advised to do all the assigned readings on Dafny relevant to each part of the project before attempting to do it.*

**Cheating policy:** Make sure to review the syllabus for details about this course's cheating policy.

**Use of AI tools:** Make sure you read the document on ICON explaining our policy on the use of generative AI tools in homework assignments, and the policy's reporting requirements.

## Part A

File `Map.dfy` contains a Dafny module that defines a typical map data structure, available in most programming language libraries. In this case, however, maps are *immutable* in the same sense as lists: modifications are achieved by generating from the given map a new one with the desired changes. The provided implementation relies on the `List` and `Option` data types, each defined in its own module. The `List` module contains various functions over lists similar to those seen in class. The `Option` module contains simply the definition of the `Option` data type. Values of this type can be returned by functions that might not always have something to return. Such functions can return the `Option` value `None` to indicate that they have nothing to return. Otherwise, they return

`Some(v)` where `v` is the intended return value. Callers to such functions can then extract `v` using pattern matching.

In `Map.dfy`, are described at the abstract level as sets of key-value pairs, or *entries*. For simplicity, they are not fully generic in the key and the value types. Instead, keys are limited to integer numbers and values are restricted to non-reference types.<sup>1</sup> This restriction is expressed by adding the `!new` constraints to type parameters.

Since abstractly a map is just a (finite) set of entries, it is just a binary relation which, however, is required to be functional: every key is associated with at most one value by the map. Concretely, the provided code implements a map as a *list* of entries with no repetitions in which every key is associated with exactly one value. At the specification level, this is enforced through the ghost predicate `isValid` which is supposed to capture the properties above and be invariant for every function that takes a map as inputs or produce a (new) map as output. For functions that take a map as input but do not produce a map as output, `isValid` is useful as a requirement on the input map. The module defines several such functions.

You are to add a full specification for each of the provided map functions that captures the English specification in comments. Make sure you *write each contract in terms of the map's set of entries, keys and values, as appropriate*.<sup>2</sup> You can refer to these sets by using the provided ghost functions `entries`, `keys`, and `values`, respectively.

Currently, the predicate `isValid` has a dummy definition. Provide one that captures the restriction above on the `Map` implementation and connects the abstract and the concrete view. You are encouraged to define auxiliary ghost predicates to keep the definition of `isValid` simple. Just provide the definition, you do not need to add a contract too.

For full credit, the contracts you add for the various functions must be verified by Dafny. For this purpose, you can annotate the functions with `decreases` and `assert` clauses or lemmas as needed. Be aware though that if your definition of `isValid` and the various contracts capture the English specification accurately, you should not need to provide any further help to Dafny for the proofs. This mean that, if Dafny is not able to verify a contract by itself, it is likely that the contract and possibly your definition of `isValid` are incorrect or not strong enough.

## Part B

File `Mail.dfy` contains a prototype of an email client application very similar to the one we modeled in Alloy in a previous assignment. The functionality meant to be provided by the prototype is fully implemented in the classes `MailApp` and `Mailbox` as specified by the English text in comments in those classes. However, it lacks any formal specification. Add one, in terms or method contract and class invariants so as to fully capture the English specification.

Make sure you *express method contracts strictly in terms of the abstract state of each class*. For simplicity, in `Mailbox` the concrete and the abstract state are the same— hence the absence of ghost fields. In contrast, in `MailApp` the abstract state consists of the ghost field `userBoxes` and four non-ghost fields: `inbox`, `drafts`, `trash`, and `sent`. Field `userBoxes` is implemented in concrete with the

---

<sup>1</sup>In Dafny, reference types are object and array types. Recall that non-reference, or *value*, types are the basic types, algebraic data types, and a few built in types like `seq`, `set` and `multiset`.

<sup>2</sup>In other words, the contracts should not expose the fact that the map is implemented as a list.

aid of non-ghost field `userboxList`. Define the predicate `isValid` as instructed in the code to express class invariants, including the connection between abstract and concrete state. Then add ghost code as needed to the body of `MailApp`'s methods to maintain that connection. *Do not otherwise modify the provided method implementations.*

Also annotate the code with `reads`, `modifies` and `decreases` clauses as needed for Dafny to be able to prove the correctness of the implementation with respect to your specification.

`MailApp` and `Mailbox` rely on a few other classes and auxiliary functions. Among these, the class `Message` has a formal specification but no implementation for its methods. Provide one yourself according to the formal spec, making sure that Dafny can verify its correctness.

## Part C

In this part we go back to maps but this time consider *updatable* maps, that is, maps that can be modified. We will use an object-based implementation for that. File `UMap.dfy` contains a class `UMap` whose methods provide a functionality similar to that of immutable maps of Part A, except for method `put` and `remove` which modify the map object.

You are to provide a full contract for all the methods that are given a specification in comments. Analogously to Part A, make sure you *express method contracts strictly in terms of the abstract state of the UMap class*, represented by the set of entries in the map and the related sets of keys and values. You can refer to these sets by using the provided ghost functions `entrySet`, `keySet`, and `valueSet`, respectively. In addition, implement those methods. You can implement them as you please but consider taking advantage of the `Map` module from Part A. Concretely, you can store map's entries in a mutable field `entries` of type `Map<T>` and use the functions in the `Map` module to query or modify those entries.

As in Part B, annotate your code with `reads`, `modifies` and `decreases` clauses as needed for Dafny to be able to prove the correctness of your implementation with respect to your specification. Note that if you use functions from module `Map` in your implementation of a method, you may have to provide a full contract to those functions before Dafny can prove your contract for the method.

## Part D (optional, extra credit)

Annotate the implementation of the in-place bubble sort algorithm provided in file of `BubbleSort.dfy` with `decreases` clauses and loop invariants as needed to prove the provided contract. The code uses multisets as a way to express that the resulting array is a permutation of the input array. You can check the tutorial at <https://dafny.org/dafny/OnlineTutorial/ValueTypes> for basic information on multisets.

## General Hints

1. If you use Dafny within Visual Studio Code, open the `code folder`, not individual files. However, do concentrate on one file at a time.
2. When working on a method, it might be useful to temporarily comment any other *unrelated* methods.

3. Recall that, since Dafny checks `requires`, `ensures`, and `invariant` clauses incrementally, the order in which you write them sometimes matters, even if it should not from a logical point of view. So for example, write something like

```
invariant 0 <= i < a.Length;  
invariant a[i] > 0;
```

instead of

```
invariant a[i] > 0;  
invariant 0 <= i < a.Length;
```

so as to establish that `i` is within bounds before it is used in `a[i]`.

## Grading Policy

Your submission will be reviewed for:

- The clarity of your implementation and annotations. *Keep both short and readable.* Submissions with complicated, lengthy, redundant, or unused code or specs may be rejected.
- The correctness of your code with respect to English specification.
- The correctness of your annotations.

For each part, *your Dafny code should be free of syntax and typing errors.* You may get no credit for that part otherwise. Submission with verification errors or warnings will receive only partial credit.

Recall that while only one team member should submit your solution, all members are required to submit a team evaluation, as specified on Piazza.