# Construction and Verification of Software – 2017/2018

Open Answer Questions June 19, 2018

---

**Q-1** Complete the code below with the strongest post-conditions, the weakest pre-conditions possible, and the needed invariants so that Dafny verifies the code without errors.

```
method IsPalindrome(a:array<int>, n:int) returns (b:bool)
  requires [                                                    ]
  ensures [                                                     ]
{
  var i := 0;
  while i < n
    decreases n-i
    invariant [                                                 ]
    invariant [                                                 ]
  {
    if a[i] != a[n-i-1]
    { return false; }
    i := i + 1;
  }
  return true;
}
```

**Q-2** Consider an ADT representing the control mechanism for a 3D printer head. It controls the movement of the printer head in a 2D space depositing a thread of melted plastic.

For the safety of the hardware during transport, the printer head has a state called `Parked`, which can be reached by method `Park` and a state called `Online` which can be reached by method `Init`. The remaining operations can only be called in the state `Online`.

The printer head can only start melting plastic if it is not moving and the temperature is above a given threshold. The temperature control is not accessible through this interface.

The maximum distance that can be reached when melting is also bound by a Maximum amount (consider some constant in the code). No maximum distance is set for non-melting movements.

Complete the specification of the class by adding field declarations and functions that help define all needed TypeStates and conditions. You do not need to implement the methods.

```
class PrinterHead {

  constructor()
    requires [                                                    ]
    ensures [                                                     ]
  { ... }

  method Init()
    requires [                                                    ]
    ensures [                                                     ]
  { ... }

  method Park()
    requires [                                                    ]
    ensures [                                                     ]
  { ... }

  method MoveTo(x:int, y:int)
    requires [                                                    ]
    ensures [                                                     ]
  { ... }

  method StartMelting()
    requires [                                                    ]
    ensures [                                                     ]
  { ... }

  method StopMelting()
    requires [                                                    ]
    ensures [                                                     ]
  { ... }

}
```

**Q-3 [5 points]**

Consider an ADT that representing a local replica of a user directory with the following interface.

```
interface Directory {
  User getUserByName(String name);
  //@ requires DirInv(this) &*& name != null;
  //@ ensures DirInv(this);

  User[] findUsers(String query);
  //@ requires DirInv(this) &*& query != null;
  //@ ensures DirInv(this)

  int addUser(String name, String email);
  //@ requires InSync(this) &*& name != null &*& email != null;
  //@ ensures DirtyInv(this) &*& result > 0;

  void sync();
  //@ requires DirtyInv(this);
  //@ ensures InSync();

  boolean isDirty():
  //@ requires true;
  //@ ensures result ? DirtyInv(this) : InSync(this);
}
```

Notice that predicates `DirtyInv` and `InSync` both imply the ADT invariant `DirInv`. **Implement a concurrent ADT** that uses an instance of the (sequential) ADT interface `Directory` and uses a monitor and related conditions to establish the preconditions of the operations above.
*Note: You do not need to write verifast* close *and* open *operations, but should state what is the shared state and the predicates ensured by each condition.*

**Q-4 [3 points]**

Consider the following implementation for function `maxEven`

```
public static int maxEven(int[] a) throws IllegalArgumentException {
    if (a == null || a.length == 0)
        throw new IllegalArgumentException();
    boolean first = true;
    int max = 0;
    for (int i = 0; i < a.length; i++)
        if (a[i] % 2 == 0) {
            if (first) { max = a[i]; first = false; }
            else max = Math.max(max, a[i]);
        }
    if ( first ) throw new IllegalArgumentException();
    return max;
}
```

1. **Present the control flow graph** of the function that supports the design of glass-box tests (unrolling loops).
2. **Identify what are the paths** that should be tested in a glass-box testing approach.
3. **Produce a test** for each identified path