Assignment 2 (Solution)

Exercise 1

We use the notation `requires: expr` to specify the precondition and `ensures: expr` for the postcondition:

```
public class Player {
    private int x, y;
    public final int n;

    // invariant: 0 <= x && x < n && 0 <= y && y < n

    // requires: 0 <= x && x < n
    // requires: 0 <= y && y < n
    public Player(int x, int y, int n) {
       this.x = x;
       this.y = y;
       this.n = n;
    }

    public int getX() {
      return x;
    }

    public int getY() {
      return y;
    }

    // ensures: old(getX()) == 0 => getX() == n - 1
    // ensures: old(get(X)) > 0 => getX() == old(getX()) - 1
    // ensures: getY() == old(getY())
    public void moveLeft() {
       x = x - 1;
       if (x < 0) {
           x = n - 1;
       }
    }
    ...
}
```

Exercise 2

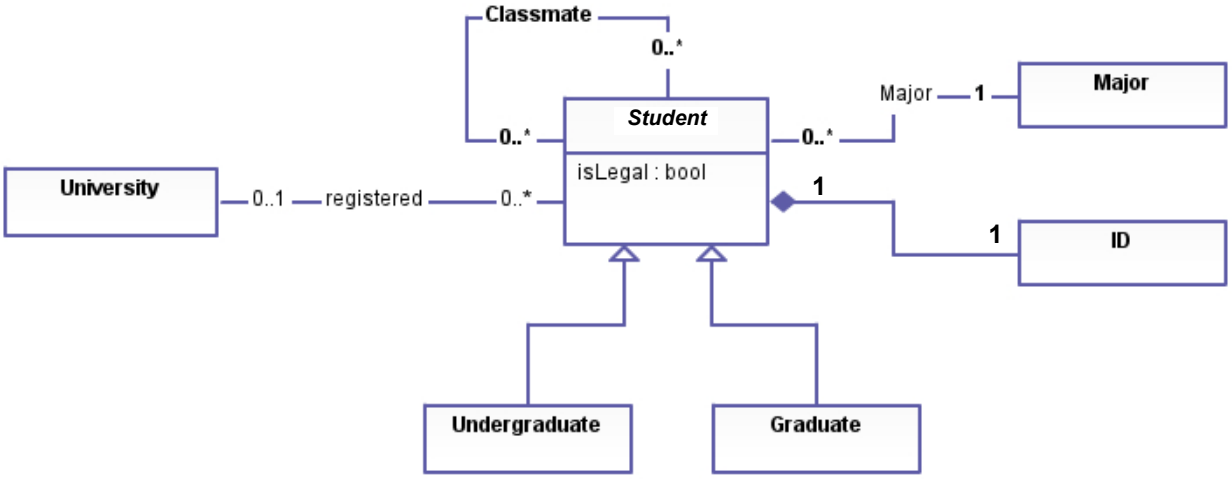
Main insight: `elems.Length` must be strictly greater than 0 so that `Add` works, therefore we get the preconditions `initialElements.length > 0` and `howMany > 0` in the constructors.

```
public class Bag {
    private int[] elems;
    private int count;

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(elems != null);
        Contract.Invariant(0 < elems.Length);
        Contract.Invariant(0 <= count && count <= elems.Length);
    }
    public Bag(int[] initialElements) {
        Contract.Requires(initialElements != null);
        Contract.Requires(0 < initialElements.Length);
        ...
    }
    public Bag(int[] initialElements, int start, int howMany) {
        Contract.Requires(0 <= start);
        Contract.Requires(0 < howMany);
        Contract.Requires(initialElements != null);
        Contract.Requires(start + howMany <= initialElements.Length);
        ....
    }
    [Pure]
    public int Count() {
        ....
    }
    [Pure]
     public int[] GetElements() {
        ...
    }

    public int RemoveMin() {
        Contract.Requires(0 < Count());
        ....
    }
    public void Add(int x) {
        Contract.Ensures(Count() == Contract.OldValue(Count()) + 1);
        Contract.Ensures(GetElements()[Contract.OldValue(Count())] == x)
        Contract.Ensures(!Contract.OldValue(Count() == GetElements().Length) ||
          GetElements().Length == 2*Contract.OldValue(GetElements().Length))
        Contract.Ensures(Contract.ForAll(0, Count() - 1, i =>
          GetElements()[i] == Contract.OldValue(GetElements()[i])))
        ....
```

```
    }
}
```

Exercise 3

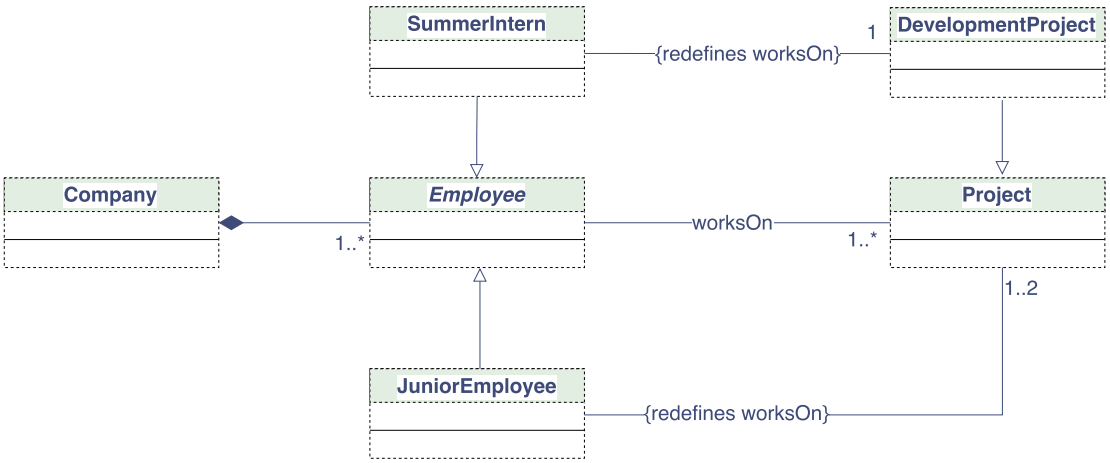
1. Student Class diagram



2. (a) Classmates have the same major
   (b) A student is legal iff he/she is registered

Exercise 4

1. Company Class diagram



2. A possible solution would be to declare a field `list of projects` in the abstract class `Employee` which will be inherited by both subclasses. Then the class `JuniorEmployee` would have to make sure that this list has maximum 2 elements, and any attempt to add more projects will result in an exception. In this implementation, the class `SummerIntern` would have to check that the assigned project is an instance of `DevelopmentProject`.

   A better alternative is shown below:

```
public interface Employee {
   public List<Project> getProjects();
}

public class JuniorEmployee implements Employee {
   private Project project1;
   private Project project2;

   @Override
   public List<Project> getProjects() {
      List<Project> projects = new ArrayList<Project>();
      projects.add(project1);
      if (project2 != null) {
          projects.add(project2);
      }
     return projects;
   }
   ...
}

public class SummerIntern implements Employee {
   private DevelopmentProject project;

   @Override
   public List<Project> getProjects() {
      List<Project> projects = new ArrayList<Project>();
      projects.add(project);
      return projects;
   }
   ...
}
```