Software Engineering — Lab 4

Uppsala University

Spring 2013

Using code contracts we're now going to see how we can "guard" behavior and state. And also how to ensure that subclasses respect the restrictions set by their superclass.

For a more comprehensive resource on code contracts, please refer to Microsoft Research's user manual.

1 State-space, behavior and Code Contracts

In class you've discussed an example with vehicles with different weights. We're going to implement that example and attempt to enforce behaviour and state-space through the use of contracts. Let's start off with some basic classes:

```
class RoadVehicle
{
    public double Weight { get; protected set; }

    public RoadVehicle() { }
    public RoadVehicle(double weight)
    {
        if (weight < 0.5 || weight > 10.0)
             throw new ArgumentException();

        Weight = weight;
    }
}

class Automobile : RoadVehicle
{
    public Automobile(double weight)
```

```
        {
            if (0.2 < weight || weight > 13.0)
                throw new ArgumentException();

            Weight = weight;
        }
    }
```

Think back to class, and try to realize the problem of the above code. Hint: the subclass ignores the restrictions posed by the superclass's constructor by simply not calling it. A mistake of bad design, perhaps, but an easy mistake to make.

1.1 Setting up Visual Studio to use Code Contracts

To enable Visual Studios static and runtime contract checking, right-click your project in the `Solution Explorer` and choose `Properties`. Select the `Code Contracts` tab and make sure to check `Perform Runtime Contract Checking`, `Perform Static Contract Checking` and also to set the `Warning Level` to `High`. Save the configuration file.

Whenever you now build the project, the `Background contract analysis` will be run, and you can track its progress in the `Output`-pane.

1.2 Using invariants to enforce the weight

Now, let's make use of invariants instead of exceptions as to force any subclass to follow the restrictions on weight. Add the following method to the RoadVehicle class:

```
/* RoadVehicle */
[ContractInvariantMethod]
private void ObjectInvariant()
{
    Contract.Invariant(Weight > 0.5 && Weight < 10.0);
}
```

Invariants in C# are added through creating a (void-returning) private method that is decorated with `[ContractInvariantMethod]`. The method must never be called from anywhere by the user code.

1.2.1 Repeat for the Automobile class

Do the same thing for the Automobile class but using the weight restrictions of the automobile.

1.2.2 Remove the manual exception throws

Since we're moving over to using invariants. Remember to *remove* the now unnecessary `ArgumentException` checks.

1.2.3 Build the project

If you've progressed correctly, build the project, and then wait a couple of minutes for the static checker to run. You'll notice a couple of "squigglies" show up – indicating paths that will lead to erroneous states. What are the states and why do they happen? Discuss with a partner.

Eliminate the errors through (1) removing the parameter-free constructor in the base class, and (2) calling the base constructor with a parameter from the subclass's constructor. As follows:

```
class RoadVehicle
{
    public double Weight { get; protected set; }

    public RoadVehicle(double weight)
    {
        Weight = weight;
    }
    ...
}

class Automobile : RoadVehicle
{
    public Automobile(double weight) : base(weight)
    { }
    ...
}
```

The project should now be free of static errors, and we have increased the level of "correctness" in the code as it is now significantly difficult to create a subclass that breaks the invariant rules of the superclass.

Let's have a look at how code contracts can help us realize there are problems with our code. There is still a major problem with our code, and we will find it when we try instantiating an automobile with edge case weight. Put the following code in the main entry point of the application, run the program and take note of the exception that arise.

```
Automobile car = new Automobile(12.0);
```

We should now realize that the conceptual problem with our code is that we're actually violating Liskov's substitution principle. Discuss with a partner why that is.

1.3 Rethinking the automobile class

Now that we've realized that an automobile cannot possible be a ground vehicle if the weight of the automobile is greater than the maximum allowed weight for anything that is a ground vehicle, let's rethink. Instead, rewrite the invariant of the automobile so that the invariant enforce:

```
Weight > 1 && Weight < 7.5
```

1.4 Pre- and post-conditions

Now, we'll leave the invariants and talk about pre- and post-conditions. Let's introduce the concept of passengers to the automobile class. Implement the following method and it's required members:

```
/* Automobile */
public void SetPassengers(int n)
{
  if (n > 5)
    throw new ArgumentException("Could not fit all passengers");

  _currentPassengers = n;
}
```

1.4.1 Pre–

Let's replace the exception-throwing code above with a pre-condition. You can think of pre-conditions as a way to make sure that a method is only allowed to be called with valid input. Rewrite the method like so:

```
public void SetPassengers(int n)
{
  Contract.Requires(n <= 5);
  _currentPassengers = n;
}
```

Now, to test whether or not this works, put the following code in the main entry of the application:

```
Automobile am = new Automobile(4);
am.SetPassengers(2); // Should be ok
am.SetPassengers(6); // Should not be ok
```

Rebuild the project and notice the errors that the static checker finds. Now we need not even run the application before the checker realize our mistakes.

Task: Experiment with how the static checker behaves when we are using variables instead of primitives. Use a variables instead of a hard-coded number.

1.4.2 Post–

Now let's instead focus on post-conditions. Quite intuitively, if pre-conditions ensure correctness of state upon entry in a method, post-conditions ensure correctness of state upon exit out of a method. Using post-conditions it is often possible to mathematically and specifically describe what the method "actually do". To write post-conditions we use the ensures-method:

```
Contract.Ensures(statement);
// Where the statement evaluates to true or false
```

Task Write a post-condition that enforce the implementation of the SetPassenger-method. Run the application to see if your post-condition passes. To verify that you've written a correct post-condition, "break" the code by changing the implementation to an incorrect one, such as:

```
_currentPassengers = n + 1;
```

1.5 Extend the classes – optional

If you've completed all of the above. Think of some new behavior that could be added to either the sub- and/or the superclass. Implement that behavior and then write invariants, pre- and/or post-conditions to "protect" it.

Good luck!