

# Class Specification and Implementation Review, part 2

## Purpose:

In a previous lab you wrote the specification of the class *NestedRectangles* and generated a browsable specification document for this class. In this lab, you will complete the implementation and testing of this class.

## Implement the class *NestedRectangles*:

### Declare instance variables:

An instance of *NestedRectangles* will use two instances of the class *Rectangle*.

- Write the declaration of the two *Rectangle* instance variables.
- Explain why instance variables should be private.

### Implement constructors:

- Implement the three constructors. Note that two of the constructors must create *Rectangle* instances, while the third does not. Where possible, implement a constructor by having it invoke another *Rectangle* constructor.

Recall that an class invariant requires that the dimensions of the inner rectangle must be strictly less than the corresponding dimensions of the outer rectangle. This invariant places requirements on the constructor arguments provided by the client.

- Are the constructor preconditions adequate to ensure the class invariant will be satisfied when a *NestedRectangles* instance is created?
- What is required of the constructor regarding the invariant?
- How would you make sure the invariant is satisfied when creating a *NestedRectangles* instance? What happens if the invariant is not satisfied in this case?
- What happens if the invariant is not satisfied and you do not check at construction time?

### Implement queries:

- Implement the queries. Notice that the algorithms in these cases are very simple expressions involving one or both of the component *Rectangle* instances.

### Implement commands:

- Implement the commands. This is very straightforward as well.
- Again how do you make sure that the client of this class has satisfied preconditions for a command? Notice that if the preconditions are not met, the invariant will become invalid. But an invariant must be maintained as long as the instance is active! Well, one simple thing to do

to guarantee that things do not continue with an invalid invariant is to use the *Require.condition* library unit.

### **Analysis of the client use of an instance of *NestedRectangles*:**

How does a client make sure preconditions are satisfied, for instance when invoking the command

```
/**
 * Set the width of the inner Rectangle to the specified value.
 *   require:
 *     0 < value && value < this.outerWidth()
 *   ensure:
 *     this.innerWidth() == value
 */
public void setInnerWidth (int value) { ... }
```

The client must to know the current width of the outer rectangle, and that the argument is positive and less than that width. So, since the width can change at any time, the client should write code like the following to change the outer width:

```
if (myValue > 0 && myValue < nestedRec.outerWidth()) {
    nestedRec.setInnerWidth(myValue);
    ...
}
```

The client may sometimes be able to assert that one or both conditions are satisfied by the argument. For instance:

```
int myValue = nestedRec.outerWidth() - 10;
// assert: myValue < outerWidth()
if (myValue > 0) {
    nestedRec.setInnerWidth(myValue);
    ...
}
```

This type of code must be written by the client to ensure that preconditions are not violated. If the client is a user interface, then this kind of code will appear in all places where the user interface wants to change the size of one of the rectangles. **The client must guarantee that preconditions are satisfied, and this generally means wrapping changes in conditionals.**

The problem is that this kind of code is bound to create maintenance headaches!

Supposed you write a user interface and when involving commands you write them using the code suggested above. The code is fine and the user interface will work as expected, because it does not break any preconditions. In other words, every time the user interface wants to issue a command, it explicitly checks the preconditions before doing it. “Fine code” you might say.

The headaches come when you must modify *NestedRectangles*. Suppose you must modify the application to accommodate cases where the inner rectangle can have the same dimensions as the outer rectangle. The changes required in the definition of *NestedRectangles* are minimal. For example, if *setInnerWidth* is written as

```
/**
```

```

* Set the width of the inner Rectangle to the specified value.
*   require:
*     0 < value && value < this.outerWidth()
*   ensure:
*     this.innerWidth() == value
*/
public void setInnerWidth (int value) {
    Require.condition(0 < value && value < this.outerWidth());
    innerRec.setWidth(value);
}

```

the changes are straightforward:

```

/**
* Set the width of the inner Rectangle to the specified value.
*   require:
*     0 < value && value <= this.outerWidth()
*   ensure:
*     this.innerWidth() == value
*/
public void setInnerWidth (int value) {
    Require.condition(0 < value && value <= this.outerWidth());
    innerRec.setWidth(value);
}

```

This is fine and relatively simple, and the changes are limited to the definition of *NestedRectangles*.

Here is the problem: what about the clients of *NestedRectangles*? Each is checking that the dimensions of the inner rectangle must be less than the corresponding dimensions of the outer rectangle! All that code needs to be maintained.

- How many clients can a class have?
- For a given client, in how many places must code be changed?

A server does not know the number of clients that it will have, nor should it care. And for a given client, we cannot tell where commands are used without a careful examination of the code. It all depends on the design of the client. The fact is that changing *NestedRectangles* creates problems for every client.

### **A solution:**

The cause of the problem is that it was left up to the *clients* of *NestedRectangles* to explicitly check the validity conditions regarding the relationship between the inner and outer rectangles.

The actual code to check the validity of this relationship should be provided by *NestedRectangles* itself because it is information about the *NestedRectangles* instance. *NestedRectangles* should provide queries like the following:

```

/**
* The specified value is a legal width for the inner rectangle.
*/
public boolean isValidInnerWidth (int value) { ... }

```

The query will implement the current policy for the width of the inner rectangle: less than the outer width, less or equal to the outer width, less than but no less than 1/5 of the outer width, less than the outer width and less than or equal to the inner height, *etc.*

Now rather than including code for that implements the current *NestedRectangles* policy, the client contains code like the following:

```
if (nestedRec.isValidInnerWidth(myValue)) {
    nestedRec.setInnerWidth(myValue);
    ...
}
```

This code will not break if the condition is changed. Only the code of the server, *NestedRectangles*, will need to change.

- Specify and implement the query above using the policy that the inner rectangle must be strictly contained in the outer one.
- Specify and implement the three other queries, *isValidInnerHeight*, *isValidOuterWidth*, and *isValidOuterHeight*, needed to support the checks required when the client changes the size of the rectangles. (Note that preconditions for the commands to change rectangle dimensions should be expressed in terms of these queries.)

The same situation exists with regard to changing colors.

- The current policy regarding changing colors simply requires a non-null *Color*. Suggest three alternative policies.
- Specify and implement the two queries named *isValidForeground* and *isValidBackground* needed to support a client being able to change colors.

Note that the method *swapColors* does not have any preconditions.

- Make sure that your implementation of *NestedRectangles* is syntactically correct.

A complete specification of *NestedRectangles* can found *here*.

## Testing and test plans:

We now need to design a test plan to test *NestedRectangles*.

- Write a test plan to test the class *NestedRectangles*.
- Run the graphical user interface *Lab21.flipper.RecFlipperGUIStart*.
- Is this GUI adequate for testing the class *NestedRectangles*? Be specific.
- Write a text based user interface that can be used to thoroughly test the class *NestedRectangles*.
- Write the class containing the *main* method used to start your user interface.
- Using your user interface, execute your test plan using *script* to collect your execution results in a file.

- Analyze the execution of the test plan. Did it perform as expected? What problems were detected? Did you fix any of them? (You do not have to for this lab).

**Post-lab:**

Submit the following:

- test plan;
- text base user interface class definition;
- results of execution of the test plan;
- analysis of the execution;
- answers to questions posed in the lab.