6 Understanding Mutation

Our lives just got to be more difficult. Many Java programmers cannot imagine living without `void` methods and the corresponding mutation. However, this makes the job of designing test for the methods much harder.

Do the work for the lab we have used with our students in the Spring 2011. (Do only the first part of the lab. The second one deals with equality, the topic we have covered in our Lab 3.

Look for the lessons learned:

- We need to go through three or four steps for every test: initialize the data, invoke the method, test the effects, and possibly reset the data.
- In general, we want to make sure that the tests can be run in any order. That means we need to reset the data after each test, or initialize the data before each test.
- To tests some methods more extensively we may need to build a complex test case: initialize data, invoke the method, test the effects, then invoke the method again and test the effects, repeating the last two steps several times.
- If our tests just verify that the value of a specific field has changed to the desired value, there is no guarantee that the program did not make other changes, that went undetected. For example, if we only check that the balance in the account has been recorded correctly, we may not find out that the malicious programmer changed the account number during the transaction.

Typically, our design for the tests in the presence of mutation starts by defining one or more *reset* methods that return the value of the test data to the original state. Each test case then can start with invoking the needed *reset* methods, the method invocation, the test evaluation, and often with another invocation of the *reset* methods.

Practice

Rewrite the game you have designed earlier using the mutation style. Instead of importing the *draw* library import the *isdraw* library and the *tunes* library. We may use the *tunes* library to add sounds and music to our games (and explore how to design tests in this context.)