



LSR

Jartege : a Tool for Random Generation of Unit Tests for Java Classes

Catherine Oriat

LSR/IMAG, Grenoble, France

(presented by Yves Ledru)



The need for automatic test generation

LSR

- Testing usually estimated to 40% of the total development cost
- Agile methods favour continuous testing

⇒ The need for a large number of tests

⇒ The need for automatic generation



Sources of test generation

LSR

- Automatic generation can be systematic:
 - From the structure of the code (white box structural testing)
 - From the structure of the specification (black box functional testing)
 - From knowledge on the input (combinatorial testing)
- Automatic generation can be random
 - Usually presented as the poorest approach for selecting test data [Myers94]
 - But cheap and able to detect a large number of errors

Conformance testing

LSR

- Testing to compare an implementation to a reference specification



- We focus on
 - Java programs
 - JML specifications

Java Modelling Language

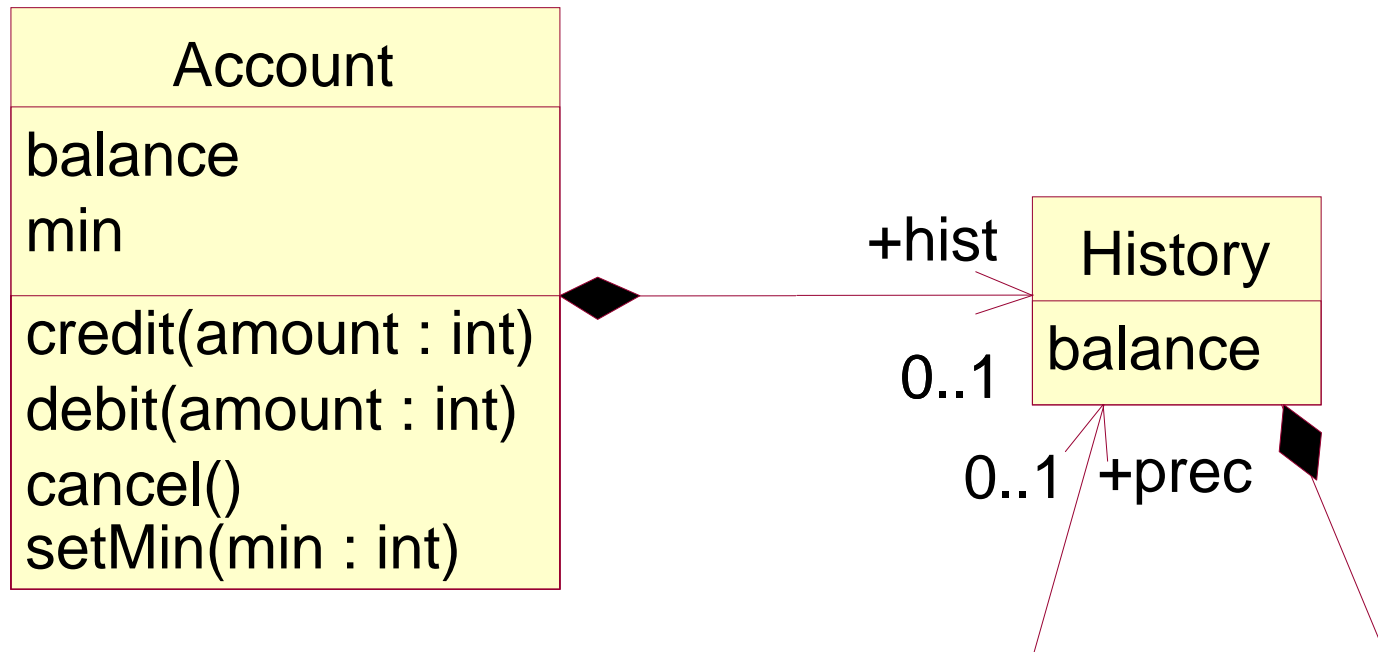


LSR

- www.jmlspecs.org
- Several verification tools (testing tools, static checkers, proof tools)
- JML specifications:
 - Based on the « Design by ContractTM » principle
 - Executable specifications
 - Automated oracle for the tests
- JML-Junit :
 - A combinatorial testing tool
 - Yoonsik Cheon and Gary T. Leavens.
A Simple and Practical Approach to Unit Testing: The JML and JUnit Way.
In ECOOP 2002 Proceedings. Vol. 2374 of LNCS, Springer, 2002.

A case study : bank accounts

LSR



- The balance must always be greater than the minimum
- The history is the list of successive balances
- The minimum may be changed



JML specification of accounts

LSR

The invariant is a property that must be true on entry and exit of all methods of the class

```
public class Account {
  /*@ public invariant getBalance( ) >= getMin( ); */
  private int balance; // The balance of this account
  private int min; // The minimum balance
  private History hist; // The history list of this
                        // account

  /* The balance of this account. */
  public /*@ pure */ int getBalance( )
    { return balance; }

  /* The history list of this account. */
  public /*@ pure */ History getHist( )
    { return hist; }

  /* The minimum balance of this account. */
  public /*@ pure */ int getMin( )
    { return min; }
```

Pure methods may not modify the object



JML specification of accounts (2)

LSR

```
/* Constructs an account with the specified balance and  
 * minimum balance. */
```

```
/*@ requires balance >= min; */  
public Account (int balance, int min) {  
    this.balance = balance;  
    this.min = min;  
    this.hist = null;  
}
```

Requires expresses a pre-condition, i.e. a condition that must be true at the entry of the method

If the precondition is false, the method should not be called

```
/* Sets the minimum balance to the specified value. */  
/*@ requires getBalance ( ) >= min; */  
public void setMin (int min) {  
    this.min = min;  
}
```

Credit method

LSR

```
/* Credits this account with the specified amount. */  
/*@ requires amount >= 0;
```

You may only credit positive amounts

```
/*@ ensures  
/*@ getBalance() == \old (getBalance()) + amount &&  
/*@ \fresh (getHist()) &&
```

The balance is updated

Ensures expr \old(expr) refers to the value of « expr »
i.e. a predicate at the start of the operation

The history is updated

This operation may not
raise exceptions

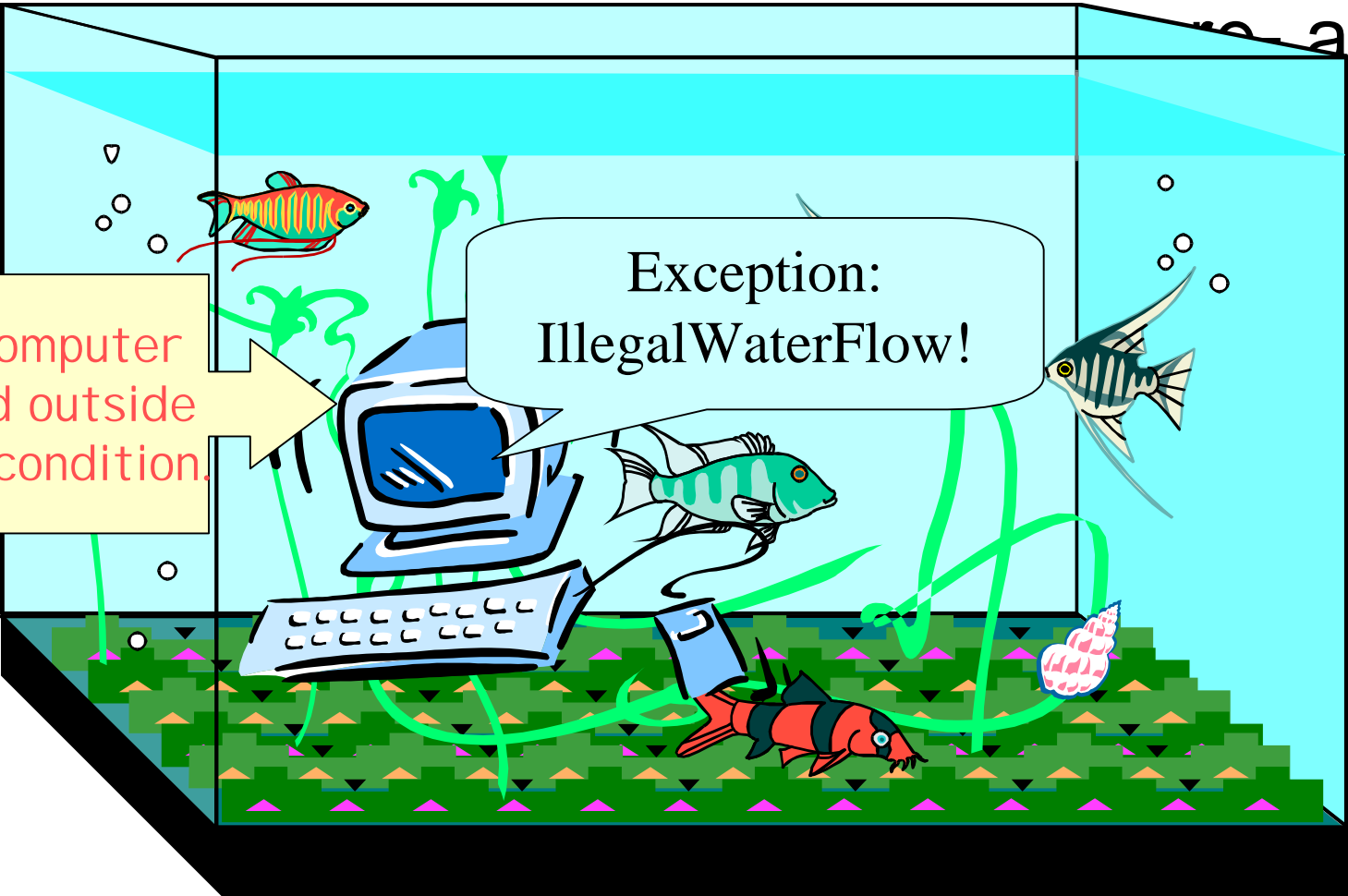
```
/*@ signals (Exception e) false;  
*/  
public void credit(int amount) {  
    hist = new History (balance, getHist ( ));  
    balance = balance + amount;  
}
```

The contract

LSR

This computer
is used outside
its precondition.

Exception:
IllegalWaterFlow!



- But if the program is called outside its precondition, anything may happen...

Cancel method

LSR

```
/* Cancels the last credit or debit operation. */  
/*@ requires getHist() != null;
```

Cancel only makes sense if there is some history

The last record is deleted from the history

```
/*@ ensures  
/*@ getHist() == \old (getHist().getPrec()) &&  
/*@ getBalance() == \old (getHist().getBalance());
```

The previous balance is restored

```
/*@ signals (Exception e) false;  
*/  
  
public void cancel ( ) { balance = hist.getBalance ( );  
hist = hist.getPrec ( );  
}  
} // End of class Account
```



History

LSR

```
public class History {
    private int balance; // The balance of this history.
    private History prec; // The preceding history.

    /* Constructs a history with the specified balance and
       preceding history. */
    public History (int balance, History prec) {
        this.balance = balance;
        this.prec = prec;
    }

    /* The balance of this history. */
    public /*@ pure */ int getBalance ( )
        { return balance; }

    /* The preceding history. */
    public /*@ pure */ History getPrec ( )
        { return prec; }
}
```

No JML assertion here!



Jartege

LSR

- Java framework for random test generation
- Mainly unit tests
- Principles
 - Discovers the methods of the class using Java introspection/reflection
 - Randomly generates objects and parameters for the method
 - Builds sequences of calls
 - Takes advantage of the JML specification:
 - Pre-conditions filter irrelevant calls
 - Invariant and post-conditions as test oracle



Jartege in practice

LSR

```
/** Jartege test cases generator for classes Account and
    History. */
class TestGen { public static void main (String[] args){

    ClassTester t = new ClassTester();
                // Creates a class tester

    t.addClass ("Account"); // Adds the specified classes
    t.addClass ("History"); // to the set of classes
                            // under test

    // Generates a test class TestBank,
    // made of 100 test cases.
    // For each test case,
    // the tool tries to generate 50 method calls.
    t.generate ("TestBank", 100, 50);
}}
```



A typical Jartege test case

LSR

```
// Test case number 1
public void test1 ( ) throws Exception {
    try {
        Account ob1 = new Account (1023296578, 223978640);
        ob1.debit (152022897);
        History ob2 = new History(1661966075,(History)null);
        History ob3 = new History (-350589348, ob2);
        History ob4 = ob2.getPreac ( );
        int ob5 = ob3.getBal
        ob1.cancel ( );
// ...
    } catch (Throwable except) { error ( except, 1);}
}
```

Here cancel appears
when its precondition is true!

Discovers the methods of the class

Randomly generates objects and parameters

f

Takes advantage of the JML specification:

- Pre-conditions filter irrelevant calls
- Invariant and post-conditions as test oracle

Execution of the test suite

LSR

During test case number 2

The invariant was broken

1) Error detected in class TestBank by method test2:
JMLInvariantError:

By method "credit@posthAccount.java:79:18i" of class
"Account"

for assertions specified at Account.java:11:32 [...]
at TestBank.test2(TestBank.java:138)

[...]

Number of tests: 100

Number of errors: 71

Number of inconclusive tests: 0

At the exit of method « credit »

- The test suite includes 100 test cases
- 71 tests ended with an error



Controlling random generation

LSR

« if we leave everything to chance, Jartege might not produce interesting sequences of calls »

A typical problem: how to handle strong preconditions?

e.g. a random debit will not satisfy the precondition if balance is close to min.

Jartege features several mechanisms to define an « *operational profile* »



Controlling the creation of objects

LSR

- When a method call needs an object, we can
 - Either create a new one
 - Or reuse an existing one
- A creation probability function controls the creation of objects:
 - $F(0) = 1$
 - $F(n) \in [0,1] \forall n > 0$
- For example:
 - It does not make sense to create multiple bank accounts in our case study
 - $F(0) = 1, F(n) = 0 \forall n > 0$

```
t.changeCreationProbability("Account",  
                             new ThresholdProbability(1));
```

Parameter generation

LSR

- Instead of using the full range of a parameter, we can provide our own generation function.

```
public class JRT_Account {
private Account theAccount; // The current account
/* Constructor. */
public JRT_Account (Account theAccount) {
    this.theAccount = theAccount; }

/** Generator for the first parameter
    of operation debit (int). */
public int JRT_debit_int_1 ( ) {
    return RandomValue.intValue (0,
        theAccount.getBalance() - theAccount.getMin());
}}
```

The parameter of debit is generated with respect to the **current** values of balance and min.

It is more likely to meet the precondition!



Other features

LSR

- Weights
 - On the choice of classes
 - On the choice of methods
 - (allows to forbid the test of a given method)
- Test fixtures (like JUnit)
 - Additional attributes for the test class
 - setUp and tearDown methods



Errors found on the case study

LSR

- 71 test sequences ended with a failure.
- Analysis of these failures leads to **two basic errors**
- The test cases have been reduced manually to the minimal sequence that reveals the error
- The errors were unknown from us.
- First error:

```
public void test1 ( ) {  
Account ob1 = new Account (250000000, 0);  
ob1.credit (2000000000); // Produces a negative balance,  
} // below the minimum balance.
```

Credit produces an **overflow** of balance which becomes a negative value under min



Error #2

LSR

```
public void test11 ( ) {  
    Account ob1 = new Account (-50, -100);  
    ob1.credit (100);  
    ob1.setMin (0);  
    ob1.cancel ( ); // Restores the balance to a value  
}                // inferior to the minimum balance.
```

- The precondition of setMin only takes into account the current balance.
- Restoring a previous balance with cancel may lead to it to be under the new minimum



Conclusion

LSR

- Summary:
 - A framework for random testing
 - Of java programs
 - Specified in JML
 - Using operational profiles
- The tool has been applied to several case studies:
 - Small banking application from Gemplus (ASE'04)
 - Jartege itself (use your own medicine!)
- Experiments confirmed the expected advantages of the tool:
 - Low cost of test generation
 - The tool helps findings errors



Future work

LSR

- Generation of method parameters
 - Currently done by hand
 - Could be generated from preconditions
 - Corresponds to the extraction of range constraints
- Once a test case reveals a failure
 - Identification of the smallest sequence that leads to an error
 - Currently manual, could be automated
- Operational profiles may help to evaluate reliability of a given software.