101JMLSpecifications

Rodrigo Bonifácio
(rbonifacio@cic.unb.br)

101JMLSpecifications

Rodrigo Bonifácio
(rbonifacio@cic.unb.br)

A contribution to the SoTeSoLa hackaton

Motivation

A number of constraints are (informally) specified in the 101 Wiki

Description

The system must support the human resources domain in the following manner.

There are companies, departments, and employees.

- Each company has a unique *name*.
- Each company aggregates a possibly empty list of *departments*.
- Each department has a unique *name* across the company.
- Each department must have a manager.
- Each department aggregates possibly empty lists of *employees* and *sub-departments*.
- Each employee has a *name*.
- Employees have additional properties for *salary* and *address*.
- Each employee serves only in one position in one company.
- Managers are employees, too.
- All properties (names, addresses, salaries) must be not null.

This data model is interesting in so far that it immediately exercises various facets of data modeling such as properties, cardinalities, recursion, and containment. Concrete implementations may easily assume refinements of the specification, if additional facets should be covered. For instance, inheritance can be exercised by deriving managers from employees through specialization. Further, the basic model at hand essentially suggests containment relationships (i.e., tree shape), but it is easy to involve reference relationships; see, for example, Graph structure. Finally, the basic model and straightforward refinements are suitable for the illustration of major programming techniques and design patterns. For instance, the basic structure at hand may suggest application of the Composite pattern.

Add comments: - 1 company versus many - uniqueness of employees

Illustration

Here is an illustrative description of a company called "meganalysis".

We use some concrete syntax to render the structure.

Some *101companies* Implementations support (a variant of) this concrete syntax.

See 101implementation:antlrAcceptor for example.

```
company "meganalysis" {
  department "Research" {
```

101feature:Global invariant

Path: Base → 101companies → 101main → 101feature → Structural 101feature → **Global invariant**

Headline

--- A constraint on salaries within the company hierarchy ---

Description

A manager of any department or sub-department is required to receive a salary that is higher than the salaries of all employees of the department and all sub-departments. (It is clear that this constraint is not universally adopted by companies in practice.) The constraint is interesting in so far that many type systems will not be able to model this constraint directly, but instead the constraint may need to be implemented explicitly by traversing the company structure.

Citations

Features

- Attribute editing
- Precedence

Contributions

- 101implementation:emftext
- 101implementation:gwt
- 101implementation:gwtTree
- 101implementation:html5tree
- 101implementation:javaExorcism

Are those constraints being checked?

Are those constraints being checked?

We evaluate this considering the `javaComposition` implementation

Let’s run some test cases

```java
public void setUp() {
  ralf = new Employee("Ralf", "Koblenz", 10000.0);
  andrei = new Employee("Andrei", "Koblenz", 15000.0);
  rodrigo = new Employee("Rodrigo", "Brasilia", 9000.0);
  vander = new Employee("Vander", "Brasilia", -10000.0);

  research = new Department("research", ralf);
  research.addEmployee(andrei);
  research.addEmployee(rodrigo);

  company = new Company();
  company.addDepartment(research);
}

public void testNotNullNames() {
  assertNotNull(company.getName());
}

public void testManagerSalaryGreaterThanEmployeeSalaries() {
  for(Employee e : research.getEmployees()) {
     assertTrue(research.getManager().getSalary() > e.getSalary());
  }
}
```

```java
public void testTotalAndCutFeatures() {
  assertEquals(34000.0, company.total());
  company.cut();
  assertEquals(17000.0, company.total());
}

public void testAfterCutMustBeSmallerSalary() {
        double beforeCut = vander.getSalary();
        vander.cut();
        double afterCut = vander.getSalary();

        assertTrue(afterCut < beforeCut);
}
```

Java - Eclipse - /Users/jml/Documents/workspace

Debug | Java

Package Explorer | JUnit

Finished after 0.073 seconds

Runs: 4/4 Errors: 0 Failures: 3

org.softlang.tests.TestFor101Company [Runner: JUnit 3] (0.007 s)

- testNotNullNames (0.001 s)
- testManagerSalaryGreaterThanEmployeeSalaries (0.000 s)
- testTotalAndCutFeatures (0.004 s)
- testAfterCutMustBeSmallerSalary (0.002 s)

Failure Trace

We could write constraint checkers directly within the body of the methods

We could write constraint checkers directly within the body of the methods but, in this case, the constraints would not appear in the interface of the methods, ...

We could (formally) specify the mentioned constraints using JML

We could (formally) specify the mentioned constraints using JML

"JML (Java Modeling Language) is a formal behavioral interface specification language for Java"

Design by Contract with JML (Gary T Leavens and Yoonsik Cheon)

JML4C compiles the source code into byte code, but with runtime checkers

JMLUnitNG generates unit tests to check those constraints

JMLUnitNG generates unit tests to check those constraints

- a skipped test case does not satisfy either an invariant or a precondition.
- a failed test case satisfies all invariants and preconditions, but at least one postcondition does not hold.

Constraints on employees' state

- name, address, and salary must be not null
- salary must be greater than zero
- before cut salary = (2 * after cut salary)

```java
public class Employee implements Serializable {

    private static final Double MIN_SALARY = 50.0;
    private static final long serialVersionUID = -2108895926771652 50L;
    private /*@ spec_public non_null @*/ String name;
    private /*@ spec_public non_null @*/ String address;
    private /*@ spec_public @*/double salary;

    //@public invariant salary > 0;

    public Employee() {
        name = "";
        address = "";
        salary = MIN_SALARY;
    }
    /*@
      @ requires pname != null && paddress != null && psalary > 0.0;
      @ ensures name == pname
      @       && address == paddress
      @       && salary == psalary;
      @*/
    public Employee(String pname, String paddress, double psalary) {
        name = pname;
        address = paddress;
        salary = psalary;
    }
```

```java
/*@
  @ requires paddress != null && !paddress.trim().equals("");
  @*/
public void setAddress(String paddress) {
    this.address = paddress;
}

/*@
  @ requires psalary > 0;
  @*/
public void setSalary(double psalary) {
    this.salary = psalary;
}

/*@
  @ requires salary > 0;
  @ ensures \old(salary) == salary * 2;
  @*/
public void cut() {
    setSalary(getSalary() / 2);
}
```

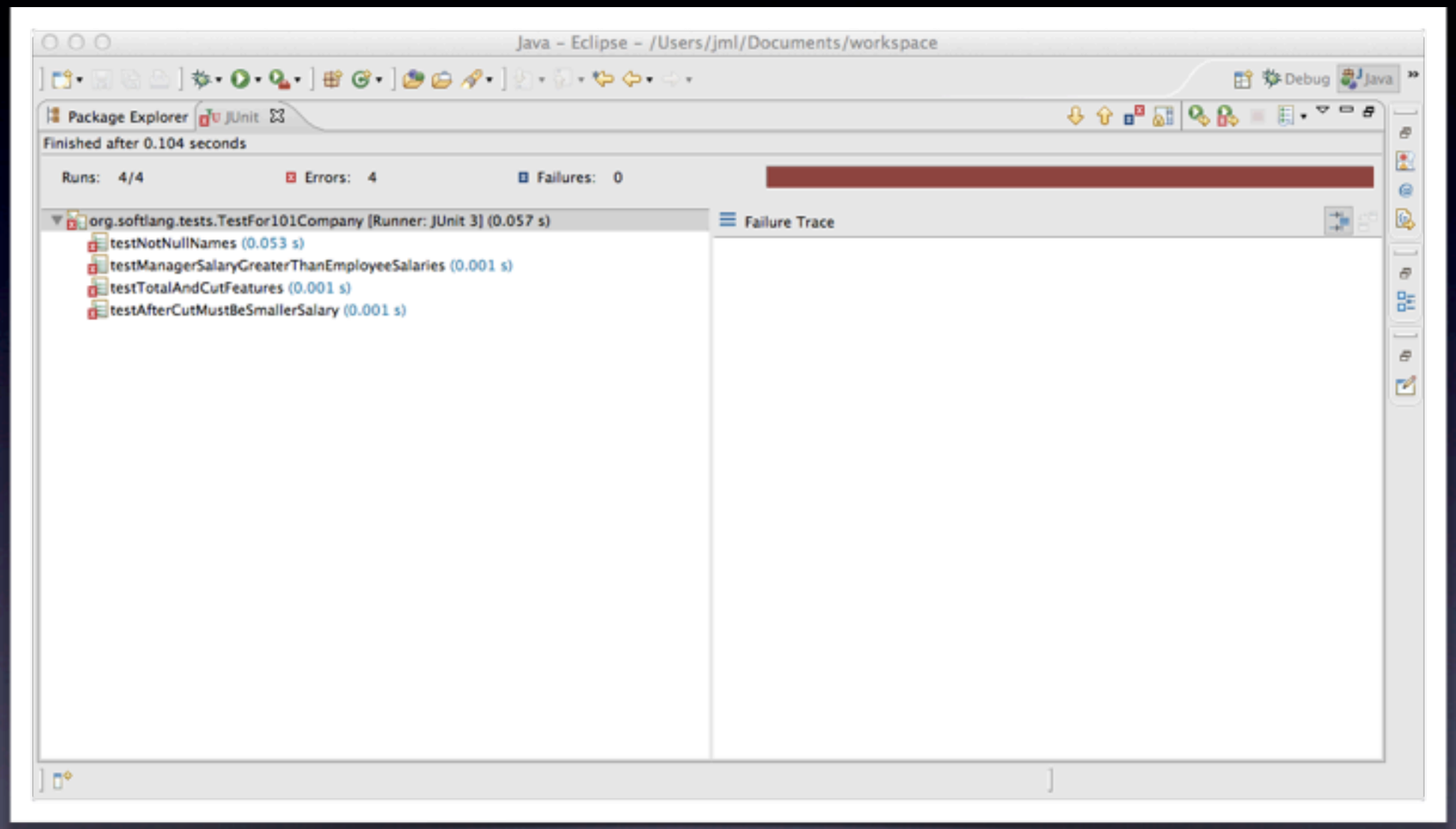
Constraints on department's state

- Name must be not null
- total before cut = (2 * total after cut)
- Manager must have a salary that is higher than the salary of any other employee of the department

```java
public class Department implements Serializable {

    private static final long serialVersionUID = -2008895922177165250L;

    private /*@ spec_public non_null @*/ String name;
    private /*@ spec_public non_null @*/ Employee manager;
    private /*@ spec_public non_null @*/ List<Department> subdepts = new LinkedList<Department>();
    private /*@ spec_public non_null @*/ List<Employee> employees = new LinkedList<Employee>();

    //@ public invariant !name.trim().equals("");

    public Department() {
        name = "";
        manager = new Employee();
    }

    /*@
      @ requires pname != null;
      @ requires pmanager != null;
      @ ensures name == pname;
      @ ensures pmanager == manager;
      @*/
    public Department(String pname, Employee pmanager) {
        this.name = pname;
        this.manager = pmanager;
    }
```

```java
/*@ requires manager != null && employee != null;
  @ requires manager.getSalary() > employee.getSalary();
  @ requires (\forall Employee e;
  @                   employees.contains(e);
  @                   employee != e);
  @*/
public void addEmployee(Employee employee) {
    employees.add(employee);
}

public /*@ pure @*/ double total() {
    double total = getManager().getSalary();
    for (Department s : getSubdepts())
        total += s.total();
    for (Employee e : getEmployees())
        total += e.getSalary();
    return total;
}

/*@
  @ requires manager != null && subdepts != null && employees != null;
  @ ensures \old(total()) == 2 * total();
  @*/
public void cut() {
    getManager().cut();
    for (Department s : getSubdepts())
        s.cut();
    for (Employee e : getEmployees())
        e.cut();
}
```

Let’s run our test cases using the JML compiled version


Java - Eclipse - /Users/jml/Documents/workspace
Debug
Java
Package Explorer
JUnit
Finished after 0.104 seconds
Runs: 4/4
Errors: 4
Failures: 0
org.softlang.tests.TestFor101Company [Runner: JUnit 3] (0.057 s)
testNotNullNames (0.053 s)
testManagerSalaryGreaterThanEmployeeSalaries (0.001 s)
testTotalAndCutFeatures (0.001 s)
testAfterCutMustBeSmallerSalary (0.001 s)
Failure Trace

Considering all specified constraints,
JMLUnitNG generates
more than 100 test cases

Test results

```
bash-3.2$ make runTests
ant -f runTestNG.xml
Buildfile: /Users/jml/workspace/101JMLSpecifications/runTestNG.xml

test:
   [testng] [TestNG] Running:
   [testng]   Ant suite
   [testng]
   [testng] ===================================================
   [testng] Ant suite
   [testng] Total tests run: 66, Failures: 4, Skips: 47
   [testng] ===================================================
   [testng]
   [testng] The tests failed.

BUILD SUCCESSFUL
Total time: 5 seconds
```

When we run the generated test suit against the original implementation, no test skips!

Are those constraints being checked?

Are these constraints being checked?

No!

In this way, we could not rely on simple assumptions

In this way, we could not rely on simple assumptions

```
@Test
public void testNegativeSalary() {
    Employee emp = new Employee("RBonifacio", "Brasilia", SALARY);
    double before = emp.getSalary();
    emp.cut();
    double after = emp.getSalary();
    assertTrue(before > after);
}
```

Summary

Summary

- We (manually) translate informal specifications of 101Companies available at the Wiki into formal JML specifications (reverse engineering)

Summary

- We (manually) translate informal specifications of 101Companies available at the Wiki into formal JML specifications (reverse engineering)
- From the formal specification, we automatically generate code to check invariants, preconditions, postconditions, and unit testes (reengineering?).

Summary

- We (manually) translate informal specifications of 101Companies available at the Wiki into formal JML specifications (reverse engineering)
- From the formal specification, we automatically generate code to check invariants, preconditions, postconditions, and unit testes (reengineering?).
- Findings: must constraints that are informally specified in the Wiki are not considered in the Java Implementations.

Next step

- Use 101Companies to compare languages that could be used to specify similar constraints (Haskell Quick Checker, Alloy, Spec#, Eiffel, ...).

101JMLSpecifications

Rodrigo Bonifácio
(rbonifacio@cic.unb.br)