Programming Paradigms
159.272
Semantics

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

Readings

1. Programming With Assertions
   http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html

2. Kent Beck, Erich Gamma: JUnit Cookbook.
   http://junit.sourceforge.net/doc/cookbook/cookbook.htm

3. Kent Beck, Erich Gamma: Test Infected: Programmers Love Writing Tests. http://members.pingnet.ch/gamma/junit.htm *

4. Liskov Substitution Principle (page contains several links to other good resources)

   http://c2.com/cgi/wiki?LiskovSubstitutionPrinciple

* Note that the technical part of this article is outdated - it refers to an older version of JUnit. But this is still a valuable resource as the philosophy of unit testing is explained well in this article by the inventors of JUnit.

Overview

- contracts between methods
- pre- and postconditions
- assertions
- Liskov's Substitution Principle (LSP)
- unit testing with JUnit

The Limitations of Type Checking

- in a statically typed language, the compiler can catch many problems that would otherwise result in runtime problems
- the advantage is that this is cheaper: it is expensive to fix already deployed code
- the drawback is that programmers start to rely too much on the compiler, and develop the (dangerous) attitude that "if it compiles then it works"

Semantics

- the compiler can only reason about the **syntax** (type signature) of methods, not about their **meaning**
- semantics: the branch of logic concerned with meaning
- in particular, the compiler cannot deal with situations like this:

ok for the compiler: types match!

intension: add two numbers

```
public int add(int number1, int number2) {
        return number1 - number2;
}
```

actual semantics clashes with intention

Constraints

- how can semantics be expressed?
- one option this can be done is through **constraints** that can be checked
- if methods satisfy these constraints, they are **correct** (by definition)
- sometimes these constraints can be directly expressed within the programming language, and tools can be used to check for constraint violations

Contracts

- sometimes, the methods are constrained against each other
- a method is only correct if it is consistent with other methods, or with itself invoked with different parameters
- these constraints are usually expressed as **contract rules**
- the rules are contracts in the sense as follows: if programmers follow these rules, they can take advantage of functionality in other classes (such as sort algorithms, data structures based on hashing etc)

Contracts ctd

- in Java (and other mainstream programming languages) these rules **cannot be formalised** and checked, they are usually documented in comments
- there are hard rules and soft rules (recommendation of what typical behaviour is)
- violating these rule can lead to subtle, difficult to trace programming errors
- for an example, see the lecture on collections (map lookup)

Contract Example 1: equals

It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
It is *consistent*: for any non-null reference values x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
For any non-null reference value x, x.equals(null) should return false.

consequences:
If violated, data structures using equals may show unexpected behaviour (e.g., lookups fail).

source:
http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object)

Contract Example 2: equals + hashCode

If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

consequences:
If violated, data structures based on hashing may show unexpected behaviour (e.g., lookups fail).

source:
http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

Contract Example 3:
clone + equals + getClass + ==

The general intent is that, for any object x, the expression:
`x.clone() != x` will be true, and that the expression:
`x.clone().getClass() == x.getClass()`
will be true, but these are not absolute requirements. While it is typically the case that: `x.clone().equals(x)` will be true, this is not an absolute requirement.

consequences:
If violated, `clone()` does not work as expected.

source:
http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#clone()

Contract Example 4: compare + equals

The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. .. The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0. .. It is strongly recommended, but *not* strictly required that (x.compareTo(y)==0) == (x.equals(y))

consequences:
If violated, data structure like TreeSet and utilities like Arrays.sort and Collections.sort will not sort objects correctly.

source:
http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html

Defining the Semantics of a Method

- methods can be seen as **state changing** manipulations of objects (in particular of the callee)
- the state change of object(s) can be described as a combination of:
  - state before the method invocation
  - state after the method invocation
- this is often described by a combination of:
  - **preconditions** or **expectations** - descriptions of state before a method invocation
  - **postconditions** or **guarantees** - descriptions of state after a method invocation

Design By Contract (DbC)

- this is a method proposed by Bernhard Meyer from ETH Zuerich
- the Eiffel programming language supports DbC
- pre - and postconditions can be directly expressed in Eiffel (`requires` and `ensures` clauses)
- some extensions of Java also support DbC, such as contract4J and the Java Modelling Language (JML)
- Whiley developed by Dave Pearce from Victoria Uni / Wellington, compiles to Java bytecode

Pre- and Postconditions (informal)

- example: `add(Object object)` method in a collections that accepts duplicates (such as `List`)
- precondition: **require that** object is not null
- postcondition: **guarantee that** the old size of the container is increased by one

asserts

- `assert` statements can be used in Java to define checkpoints
- an `assert` statement checks a boolean condition
- if the condition evaluates to false, an `AssertionError` is thrown
- by default, asserts are ignored by the JVM (and therefore there is no performance penalty evaluating asserts)
- asserts must be enabled by starting the JVM with the `-ea` option, this means that the application is run in diagnostic mode
- asserts can be switched on or off for individual packages

assert Example

```
public void add(Object obj) {
      assert obj!=null;
      int s = this.size();
      // add object here
      ...
      assert this.size()==s+1;
}
```

precondition

postcondition

Liskov Substitution Principle (LSP)

- LSP (proposed by Barbara Liskov) describes constraints that should be applied to make inheritance safe
- in particular, LSP requires that when replacing (substituting) an instance of a type T by an instance of a subtype S of T, the behaviour of the program should not be changed

LSP Example

```
public class EvenNumberChecker {
        public boolean isEvenNumber(int i) {
                return i % 2 == 0;
        }
}


..
EvenNumberChecker chk = new EvenNumberChecker();
boolean isEven = chk.isEvenNumber(-2);
```

source: https://oop-examples.googlecode.com/svn/semantics/

LSP Example ctd

```
public class SafeEvenNumberChecker extends EvenNumberChecker
{
      @Override
      public boolean isEvenNumber(int i) {
            if (i < 0)
                  throw new IllegalArgumentException();
            else
                  return i % 2 == 0;
      }
}
..
EvenNumberChecker chk = new SafeEve          ();
boolean isEven = chk.isEvenNumber(-2);
```

unsafe substitution: this suddenly fails with a runtime exception!

LSP Example ctd

- the unsafe behaviour of the overriding method can be expressed in two ways:
- the overriding method **expects more**: the arguments should not be negative
- i.e., the **preconditions are strengthened**
- the overridden method **guarantees less**: while the overridden method (in `EvenNumberChecker`) can guarantee that it will not throw a runtime exception, the overriding method can make no such guarantee
- i.e., the **postconditions are weakened**

LSP ctd

- to safeguard inheritance, the following rules should be applied when overriding methods:

  1. **preconditions should not be strengthened**
  2. **postconditions should not be weakened**

in some situation where pre- and postconditions can be expressed through type signatures, the compiler can enforce these rules !

Covariant Return Types

```
public class A {
        public java.io.OutputStream getStream() {
                return null;
        }
}
public class B extends A {
        @Override
        public java.io.FileOutputStream getStream() {
                return null;
        }
}
```

note: `java.io.FileOutputStream extends java.io.OutputStream`

Covariant Return Types (ctd)

```
public class A {
        public java.io.OutputStream getStream() {
                return null;
        }
}
public class B extends A {
        @Override
        public java.io.FileOutputStream getStream() {
                return null;
        }
}
```

- the compiler **accepts** this
- the postcondition to return an `OutputStream` is strengthened as a particular kind of `OutputStream` is returned

Covariant Return Types ctd

```
public class A {
	public java.io.FileOutputStream getStream() {
		return null;
	}
}
public class B extends A {
	@Override
	public java.io.OutputStream getStream() {
		return null;
	}
}
```

note: `java.io.FileOutputStream extends java.io.OutputStream`

Covariant Return Types ctd

```
public class A {
        public java.io.FileOutputStream getStream() {
                return null;
        }
}
public class B extends A {
        @Override
        public java.io.OutputStream getStream() {
                return null;
        }
}
```

- the compiler **rejects** this
- the overriding methods can no longer guarantee that an instance of `FileOutputStream` is returned: the post condition is weakened

Changing Visibility when Overriding

```
public class A {
      void foo() {}
}
public class B extends A {
      @Override
      public void foo()   {}
}
```

Changing Visibility when Overriding

```
public class A {
      void foo() {}
}
public class B extends A {
      @Override
      public void foo()  {}
}
```

- the compiler **accepts** this
- this can be seen as weakening preconditions (expectations): the caller does not have to be within the same package anymore

LSP and the Java Compiler

- the Java compiler supports covariant parameter types
- a similar concept is **contravariant parameter types** - this is **not** supported by the current Java compiler
- the Java compiler also supports if declared exception types are replaced by subclasses when overriding methods: this also strengthens postconditions (guarantees)

Unit Testing

- it is difficult check constraints in general
- asserts are useful, but may impose undesirable overhead at runtime
- we could try to check constraints for **selected objects** only
- this is the idea behind **unit testing**
- unit testing supports checking pre- and postconditions for method invocations
- the focus is on checking postconditions

Unit Testing History

- family of XUnit tools with support for most (X) programming languages
- started emerging in the late 90ties first for Smalltalk (dynamically typed!) (SUnit), then ported to Java (JUnit) and many other languages
- invented by Kent Beck and Erich Gamma
- corner stone of a software development philosophy: test-driven development (TDD)
- TDD: write tests first as specifications, and then implement code until all tests succeed
- when errors occur, write another tests case that fails, and implement code until all tests (including the new one) succeed

Inside a JUnit4 Test Case

```
public class SimpleAddToSetTest {
        private Set set = null;
        @Before public void setUp() {
                set = new HashSet();
    }
        @After public void tearDown() {
                set = null;
    }
        @Test public void testAddOne() {
                set.add("one");
                assertEquals(1,set.size());
    }
}
```

the object to be tested

set up test

clean up after test

the actual test

code to be tested

postcondition

JUnit 4

- JUnit 4 is **annotation-based** (while earlier versions of JUnit are not)
- the annotations `@Test`, `@Before` and `@After` are defined in the `org.junit` package
- each (public) method annotated with `@Test` is an actual test (a JUnit **test runner** will be able to run this method as a test)
- the `@Before` and `@After` methods are executed before/after each test to set up / clean up the test environment
- setting up the set of objects to be tested is also called a **test fixture**

JUnit Eclipse Integration

- JUnit is not part of the JRE/JDK
- this means that the JUnit library must be added to the classpath of a project
- in Eclipse, this is called the **build path**
- JUnit can be added as follows:
  Project Properties > Build Path > Libraries >
  Add Library
- to add a new test case
  Add New > New JUnit Test Case
- to execute a test case
  Run As > JUnit Test Case

asserts

- postconditions are written using the heavily overloaded **assert** methods
- these methods are defined as static methods in `org.junit.Assert`
- usually these methods can be made available using **static imports**:
  `import static org.junit.Assert.*;`
- if an assert fails, the test is marked as failed
- running a test can have three possible outcomes: success, failure through a failing assertion, or failure through an uncaught exception or error that occurs when the test is executed

Selected assert methods

- `assertTrue(boolean value)` - test whether the value is true
- `assertEquals(Object expected,Object actual)` - test whether the computed (actual) object is equal to the expected object
- `assertEquals(long expected, long actual)` - test whether the computed (actual) value is equal to the expected value
- `assertEquals(double expected, double actual, double delta)` - test whether two doubles or floats are equal to **within a positive delta**.
- a full list of methods can be found here: http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Example: Tests for Specifying a List

```
@Test public void testEmpty() {
        assertEquals(0,list.size());
}
@Test public void testAddOne() {
        list.add("one");
    assertEquals(1,list.size());
}
@Test public void testAddMany() {
        for (int i=0;i<100;i++) {
                list.add("element"+i);
        }
        assertEquals(100,list.size());
}
```

Testing Exceptions

```
@Test
public void readFromNonExistingFile() {
     try {
          new FileReader(nul
     assertTrue(false);
   }
   catch (FileNotFoundExcepti
     assertTrue(true);
   }
}
```

this should fail: the attempt to read data from null must trigger a FileNotFoundException

this should succeed

Testing Exceptions ctd

```
@Test(expected=FileNotFoundException.class)
public void readFromNonExistingFile() {
      new FileReader(null);
}
```

JUnit4 has built-in support for exception testing

Unit Testing ctd

- there are many extensions for JUnit to facilitate testing in particular areas (DB, UI, web applications etc)
- even if all tests succeed, this is no proof that the code is correct - there might be tests missing
- there are metrics that can be used to measure how well tested code is
- test-driven development and unit testing are discussed in more details in the software engineering papers, in particular 159.251