Jak łatwo zaliczyć programowanie

Na pewnym małopolskim uniwersytecie kurs programowania prowadzony jest przez bardzo nierozgarniętego wykładowcę. Rzadko sprawdza on kody napisane przez studentów, jeśli już sprawdza, to tylko kompiluje, jeśli kompiluje i uruchamia, to tylko na prostych testach, jeśli kompiluje, uruchamia i ma wymagające testy to najczęściej nie interesują go wyniki.

Wychodząc z tego założenia studenci postanowili zaliczyć cały kurs przy pomocy jednej klasy o nazwie SmartFactory. Klasa wygląda następująco:

```
public class SmartFactory {
      public static class HellNoException extends RuntimeException{ ... }
      public static <T> T fixIt(Class<T> cl, Object obj){ ... }
}
```

i implementuje tylko jedną funkcjonalność. Metodzie fixIt podaje się obiekt typu Class reprezentujący interfejs (zawsze!) i Object obj który ma ten interfejs udawać. Metoda zwraca obiekt który nieuważnemu wykładowcy wyda się poprawną implementacją interfejsu, a studentowi pozwoli zaliczyć kurs. Na rzecz obiektu zwróconego przez `fixIt` wołane będą metody z interfejsu `cl` i musi on je jakoś obsłużyć.

Może się zdarzyć że pewne funkcjonalności interfejsu student już zaimplementował w `obj`. Te funkcjonalności obiekt zwrócony z `fixIt` powinien delegować od `obj`; to znaczy, że jeśli kod woła metodę na obiekcie otrzymanym z fixIt, a `obj` implementuje metodę

- o tej samej nazwie,
- o wyniku i parametrach które mogą być odpowiednio rzutowane,
- liście wyjątków które **gwarantują że gdziekolwiek użyta jest metoda interfejsu to również metoda obj może być użyta**

to wynik wykonania tej metody powinien być **identyczny** jak wynik wykonania metody na rzecz obj. Jeśli istnieją dwie lub więcej pasujących metod, wyrzucany jest wyjątek `SmartFactory.HellNoException`. Na tym kończy się **faza pierwsza**.

Jeśli odpowiednia metoda w obiekcie `obj` nie istnieje, to obiekt zwrócony z fixIt przechodzi w **fazę drugą**. W tej fazie podejmowana jest próba skonstruowania obiektu który można zwrócić z wywołanej metody. Obiekt konstruowany jest pod następującymi warunkami.

- pola o typach klas (tzn. nie przymitywne i nie tablicowe) powinny zawierać poprawne referencje do istniejących obiektów;
- jeśli takie pole zawiera wartość różną od `null`, to jej nie zmieniamy
- jeśli zawiera `null` to wytwarzamy odpowiednie obiekty przy użyciu konstruktorów bezargumentowych (rekurencyjnie);
- powinno być to przeprowadzone tak, aby ilość wytworzonych obiektów była **najmniejsza możliwa**

Jeśli ta konstrukcja się nie powiedzie metoda powinna wyrzucić `SmartFactory.HellNoException`.

Na przykład:

```
import java.util.Collection;

public class Test0 {
      public static void main(String[] args) {
              String ala = "Ala has a cat";
              //Warning, raw type
              Collection col = SmartFactory.fixIt(Collection.class, ala);
              System.out.println(col);
              try {
                      col.iterator();
              }catch(SmartFactory.HellNoException e){
                      System.out.println("Well.... doesn't work.");
              }
      }
}
```

powoduje wypisanie

```
Ala has a cat
Well.... doesn't work.
```

A poniższy test unitarny przechodzi:

```
import org.junit.Test;
import static org.junit.Assert.*;

public class Phase2Test0 {
    public static class A {}
    public static class B {
        public String ala;
        public CharSequence ma;
    }
    public static class C {
        public static int i;
        public final int id = i++;
        public C other;
        public C yetAnother;
        public Object anObject;
    }
    public interface I {
        A test();
    }
    public interface J {
        B test();
    }
    public interface K {
        C test();
    }
    @Test
    public void testBasic() {
        assertNotNull(SmartFactory.fixIt(I.class,null).test());
        B b = SmartFactory.fixIt(J.class,null).test();
```

```
        assertEquals("", b.ala);
        assertEquals("", b.ma);
        assertTrue(b.ala == b.ma);
    }

    @Test
    public void testNumber(){
        C c = SmartFactory.fixIt(K.class,null).test();
        assertEquals(1, C.i);
        assertTrue(c == c.other);
        assertTrue(c == c.yetAnother);
        assertTrue(c == c.anObject);
        c = SmartFactory.fixIt(K.class,null).test();
        assertEquals(2, C.i);
        assertTrue(c == c.other);
        assertTrue(c == c.yetAnother);
        assertTrue(c == c.anObject);
    }
}
```

Jako że wykładowca jest (nie wiem czy o tym już pisałem...) nie bardzo rozgarnięty, brak zrozumienia typów uogólnionych zapewne uniemożliwi mu testowanie implementacji przy ich użyciu.