Prof. Dr. A. Poetzsch-Heffter
Dipl.-Inform. Kathrin Geilmann

University of Kaiserslautern
Department of Computer Science
Software Technology Group

Advanced Aspects of Object-Oriented Programming (SS 2010)

Practice Sheet 1 (Hints and Comments)

Exercise 1 Lecture concepts

Encapsulation & Information Hiding is used to restrict the access to an object's components. Objects hide their internal structure by using get- and set-methods, making methods and attributes private etc. The reason for the usage of it is, that the implementation can change but it should have no effects to the user of a component or object.

Polymorphism & Subtyping Polymorphism is the ability of a type A, to appear as a type B in another context. Subtyping is a form of polymorphism, where the polymorph types are related to each other by the subtype relation and a notion of substitutability, meaning that a subtype can be used everywhere, where the the supertype can be used.

Declarative & Model-based Specifications Declarative specifications just specify which properties hold, they do not say anything about how the holding is achieved. In model-based specifications a model of the implementation is used and operations, properties etc are specified in that model. The implementation then has to be a concretization of the model.

Functional & Non-functional Properties Functional properties describe the action of a program, where as non-functional properties describe anything else. e.g. ressources which can be used, ...

Exercise 2 Java Programming

Straightforward implementation.

Exercise 3 Identity vs Equality of Objects

a) String-Objects in Java are interned. That is, string literals create a string object, but they create the same string object for the same literal. (JLS 3.10.5). This result is, that `a1` and `a2` reference the same object, whereas `b1` and `b2` both create new string objects at runtime. The two strings have the same value (namely "B") i.e. they are equal, but they have not the same identity.

b) Explicitly internalize the strings `b1` and `b2`.

c) The strings have to be immutable, otherwise a change on a shared (interned) string via a variable x would change the value of other variables as well.

Exercise 4 The `equals` Method

a) In the following `d1` and `d2` reference instances of the class `Date` and `d3` and `d4` reference instances of the class `NamedDate`. We have to check calls to equals with all combinations of types.

- *Reflexivity* `d1.equals(d1)` ok; `d3.equals(d3)` ok; the given implementation is reflexiv.
- *Symmetry* `d1.equals(d2)` ⇒ `d2.equals(d1)` ok
  `d3.equals(d4)` ⇒ `d4.equals(d3)` ok
  `d1.equals(d3)` ⇒ `d3.equals(d1)` ok
  `d3.equals(d1)` ⇒ `d1.equals(d3)` ok

  The given implementation is symmetric.
- *Transitivity* Counterexample:

```
d3 = new NamedDate(2010, 5, 7, ''A'');
d4 = new NamedDate(2010, 5, 7, ''B'');
d1 = new Date(2010, 5, 7);
```

`d3.equals(d1)` and `d1.equals(d4)` are true but `d3.equals(d4)`; is false

The given implementation is not transitive.

- *Consistency* ok
- *Non-Null* ok

b) The contract is fulfilled except for the call with null (fix it, by adding a check) but this solution breaks Liskov's substitution principle. The principle states that whenever a property is true for an object of type T it should be true for objects of type S as well, where S is a subtype of T.

In other words, whenever a type T is used it should be safe, i.e. leading to the same results, to use a subtype of T. This is not the case with the given implementation. Consider the following code:

```
HashMap<Date, String> m = new HashMap<Date, String>();
m.put(new NamedDate(2010,4,5, ''A''), ''rainy'');
// the following calls may give different results (depending on
// which object equals is called and which is used as parameter),
// where the substitution principle expects the same
m.containsKey(new Date(2010,4,5));
m.containsKey(new NamedDate(2010,4,5,''A''));
```

Breaking the substitution principle often results in unexpected behavior and should be avoided if possible.