Fortgeschrittene objektorientierte Programmierung (Advanced Object-Oriented Programming)

Coordinates: No. 185.211, VU, 3 ECTS
Lecturer: Franz Puntigam
Web: http://www.complang.tuwien.ac.at/franz/foop.html
Studies: Software Engineering & Internet Computing
         Computational Intelligence
         Visual Computing
Requirements: experience in object-oriented programming
Registration: in TISS until 10.03.2017
              group building: 10.03.2017 during lecture
Substitutability and Assertions
Principle of Substitutability

Type S is subtype of a type T iff each instance of S is usable where an instance of T is expected

Principle of substitutability holds if

types of input parameters are contravariant,
types of variables and through-parameters are invariant,
types of constants, results, output parameters are covariant,

and corresponding methods in S show same behaviour as those in T
Substitutability and Behaviour

Methods in subtype S show same behaviour as those in supertype T if

assertions to be fulfilled by clients (= preconditions) in S not stronger than those in T

assertions to be fulfilled by servers (= postconditions and invariants) in S not weaker than those in T

assertions to be fulfilled by clients and servers (only useful for invariants in some cases) equivalent in S and T

methods in S do not throw more exceptions than those in T (in corresponding situations)
Expressing Assertions

In theory, all assertions formally expressible (logics, algebra), relationships between assertions statically checkable.

In practice, many assertions not expressible and relationships not checkable because:

- Programming language has insufficient support (informal comments, usually ambiguous).
- Clients have not enough information about object state (data hiding in conflict with Design-by-Contract).
- Object state changes in non-predictable way (concurrency, aliasing).
Example of Using Assertions

class IntSet {
    public boolean find(int x) { ... }
    // true iff x is in set
    public void insert(int x) { ... }
    // immediately after insertion x is in set
    ...
}

...
IntSet set = new IntSet();
set.insert(1);       // now 1 is in set
boolean a = set.find(1);
dosomething_not_using_set();
boolean b = set.find(1);
Possible Problems in Example

“Immediately after” is ambiguous

Usual interpretation: Element remains in set as long as no “delete” is invoked (that could be defined in a subtype)

Constructor creating “set” could have introduced alias (different from “set”) that causes “do_something_not_using_set” to change the set
→ “b” possibly false

Concurrent thread (spawned in the constructor) could have deleted element
→ “a” and “b” possibly false
How to Solve the Problems

Invoke “find” to check if the element is in the set

  often bad solution because nothing useful to do if check fails

Prevent aliasing altogether

  bad solution because extremely expensive

In case of concurrency always ensure atomic actions

  often a good idea, but sometimes very expensive

avoid unexpected side-effects (e.g., in constructor)

  only useful alternative
  however, expectations hardly ever formally expressible
History Constraints

Advice: Use all available information, no matter how easily expressible as usual assertions (pre-, post-conditions, invariants)

Example: History constraints constrain the evolution of objects in a way hardly expressible as usual assertions

like “value of ‘counter’ can only increase one by one” (server responsible),
or “‘unlock’ invokable only after ‘lock’” (client responsible).

History constraints to be fulfilled by servers resemble invariants; they can be more restrictive in subtypes.

History constraints to be fulfilled by clients restrict invocation sequences; subtypes can support more invocation sequences than supertypes.
Suggestions about Assertions

Considering all possibilities would be too expensive; don’t try to do so.

Be predictable, avoid tricks, rely on “usual” behaviour of programmers.

Use design rules (specific to company or project).

Use all available information as assertions (including history constraints).

First rule: Avoid unnecessary dependences!

- no avoidable assertions,
- no unneeded invocations and parameters,
- no unnecessary visibility of variables and methods,
- only well-considered parameter types
- code only at lowest level in class hierarchy (= avoid inheritance)
Names
Significance of Names

Names are **abstractions** of object behaviour, method behaviour and variable properties, thereby resembling informal assertions.

Names support **intuition**;
comments necessary only if intuition insufficient
→  good programs readable even without comments

Intuitive names cause programmers to be **predictable**
(bad names → no trust → unnecessary code)

Names of types provide **semantic** information
(not only for programmers, also for machines)
Examples of Structural Types

Two structural (= anonymous) types in subtyping relation:

{ String name; String address() }
{ String name; String address(); int regNr }

Member names used as type names:

{ String name; String address(); void isPerson() }
{ String name; String address(); void isPerson();
  int regNr; void isStudent() }

In theory we mainly use structural types,
in practice we mainly use nominal (= named) types.

Structural types can simulate nominal types, but accidental coincidence possible (can be faked)
# Structural versus Nominal Types

<table>
<thead>
<tr>
<th></th>
<th>Structural</th>
<th>Nominal</th>
</tr>
</thead>
<tbody>
<tr>
<td>type equivalence</td>
<td>same structure</td>
<td>same name</td>
</tr>
<tr>
<td>subtyping</td>
<td>implicit</td>
<td>explicit (inheritance)</td>
</tr>
<tr>
<td>usability</td>
<td>simple</td>
<td>more difficult</td>
</tr>
<tr>
<td>plug &amp; play</td>
<td>easier</td>
<td>not that easy</td>
</tr>
<tr>
<td>name conflicts</td>
<td>possible</td>
<td>easily possible</td>
</tr>
<tr>
<td>accidental relations</td>
<td>possible</td>
<td>unlikely</td>
</tr>
<tr>
<td>readability</td>
<td>not so good</td>
<td>good</td>
</tr>
<tr>
<td>abstraction of behaviour</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
Nominal Types Ensure Properties

Example:

```java
class SortedList<A extends Comparable<A>> { ... }
```

Property “sorted” hardly directly checkable, except by class membership

Each instance of `SortedList<T>` initialized by a constructor in `SortedList` and its subclasses (faking almost impossible)

If `SortedList` and its subclasses ensure the property “sorted”, we can rely on it
Structural Types have Benefits

as bounds for bounded genericity

when using plug & play on software components

when introducing supertypes of already existing subtypes
Genericity with Anonymous Bounds

Java example: class SortedList<A extends Comparable<A>>

Why has A to inherit from Comparable<A>?
Only requirement: A provides methods of Comparable<A> because Comparable<A> does not imply further properties

Example in Ada showing the use of anonymous bounds:

generic
    type T is private;
    with function compare(x,y: T) returns Boolean
package SortedList
...

Kinds of Bounded Genericity

**F-bounded Genericity:** \[ S \leq T\langle S \rangle \]

can deal with binary methods, but restricted to one inheritance level

Java example: \texttt{Integer \leq Comparable<Integer>}
(subtype relation = relation on bounds)

**Higher Order Subtyping:** \[ S <\# T \text{ if } \forall U : S\langle U \rangle \leq T\langle U \rangle \]

also called **matching** (on functions over types)

supports binary methods directly, but does not provide substitutability

statically type-safe as bound in bounded genericity
(subtype relation \( \neq \) relation on bounds)