Assertions, Specifications, and Design-by-Contract

Lecture 19

Wider vs Narrower Interfaces

- Recall behavioral subtyping
- Substitution principle:
  - If a client is correct wrt a "wide" type, that same client is still correct wrt a "narrower" one
- Question: When designing an interface, how wide/narrow should it be?

Design Issue #1: Which is Better?

- Answer: It depends!
- A wider spec:
  - Demanding on inputs, tolerant on outputs
  - Easier to implement
  - Harder to use
  - Less powerful
- A narrower spec:
  - Tolerant on inputs, demanding on outputs
  - Harder to implement
  - Easier to use
  - More powerful
- High-level tradeoff
  - Generality/flexibility, vs power/performance

To Ponder

- Consider the following two phrases:
  - "Congratulations! You've just won a ___"
  - "In Lab 8, you are asked to build a ___"
- Fill in the blanks with one of:
  A. vehicle
  B. 4-wheeled vehicle
  C. car
  D. Volvo S60

Wider vs Narrower Methods

- Consider a method selectTransport
  - Return value: Vehicle or Bicycle?
  - Argument: Person or Student?
  
  \[
  \text{Vehicle} \quad \text{Person} \\
  \text{? selectTransport( ? a)} \\
  \text{Bicycle} \quad \text{Student}
  \]

  Vehicle is wider than Bicycle
  Person is wider than Student

Wider vs Narrower Methods

- Vehicle m(Student a)
- Bicycle m(Student a)
- Vehicle m(Person a)
- Bicycle m(Person a)
**Good Practice: Which Declared Type?**

- How specific should the declared type of an argument / return value be?
  - Vehicle selectTransport(Person a)
  - Bicycle selectTransport(Student a)

- Typical advice:
  - "As specific as possible, without revealing implementation details"*
  - "As general as possible, while still being useful to client"

- The right way to think about it:
  - The type is dictated by the mathematical (abstract, client-side) model

---

**Requires Clause**

- Obligation on client
  - if client satisfies this obligation, component method must terminate without an exception, satisfying ensures
  - If requires is not satisfied, method could do anything, including:
    - Terminate in whatever state it wants
    - Not terminate
    - Throw an exception
  - This last case, though, should be included in specification
    - Document the "exceptional requires clause"
    - Condition under which method throws exception
    - Also document this case's ensures clause

---

**Requires and Throws**

```java
@requires n is even
void f(int n) { ... }

@requires n is even
@throws IllegalArgumentException if n is odd
void g(int n) { ... }
```

- f(5): //anything could happen
- g(7): //must throw exception

---

**Design Issue #2: Violated Requires**

- How should a violation of the requires clause be handled?
- What to include in "exceptional requires"?
  - Answer: Use checked exceptions when
    - Client can not unilaterally guarantee that the requires holds (lack of control)
    - It is likely to be prohibitively expensive for the client to check whether the requires holds
  - Recall example of lack of control
    - Guaranteeing existence of a file
  - Wrong answer:
    - Include everything outside of requires clause
    - Exceptional requires clause is @requires

---

**Example: BigNatural Constructors**

- BigNatural has 2 constructors
  - //requires v >= 0
    - SlowBigNatural(int v)
  - //requires s is a well-formed representation of natural number with no leading 0's
    - SlowBigNatural(String s)

- Checking first requires is easy for client
  - So, do NOT use an exception for negative argument
- Checking second requires is hard for client
  - So, CAN use an exception for malformed argument
- Or, another design:
  - Provide a (static) boolean method that returns whether or not a String is well-formed
  - Burden now back on client to check that the requires holds, presumably by using this method
- Performance cost for checking twice?

---

**Comparison**

```java
if (v >= 0) { //sometimes safe to omit
    b = new SlowBigNatural(v);
    ... }
```

```java
try {
    b = new SlowBigNatural(v);
    ... }
catch (NegativeArgumentException e) {
    ... //some code to recover?
}
```
Disjoint Normal/Exception'l Requires

- Prefer mutually exclusive requires and exceptional requires clauses

```
class Collections {
    /**
     * Copies all of the elements from one list into another. After the operation, the index of each
     * copied element in the destination list will be identical to its index in the source list. The
     * destination list must be at least as long as the source list. If it is longer, the remaining elements
     * are unaffected.
     *
     * @param dest The destination list.
     * @param src The source list.
     * @throws IndexOutOfBoundsExceptionif the destination
     * list is too small to contain the entire source List.
     */
    static <T> void copy (List<T> dest, List<T> src)
}
```

Good Practice: Doc Exceptions

- Document every checked exception
  - @throws clause for each, giving exceptional requires
- Throw (and document) exceptions at the right level of abstraction
  - Avoid revealing implementation specifics
    - eg IndexOutOfBoundsException vs ArrayIndexOutOfBoundsException
- Document “some” runtime exceptions
  - The ones the client should reasonably care about (?)
  - Never include these in method signature
  - Danger: no real enforcement mechanism
  - Consistency within project? Client attention?
- Parent’s @throws for unchecked exceptions not inherited
- Documentation for checked exceptions is inherited (if child declares)

Implications for JUnit

- Throwing exceptions is part of promised behavior
  - JUnit test cases should exercise this behavior
  - Seeing exception is a “pass” for test case
- @Test annotation with “expected” parameter
  ```
  @Test(expected=IndexOutOfBoundsException.class)
  public void empty() {
      (new ArrayList<Object>()).get(0);
  }
  ```

Assertions

- An assertion is a statement that should always evaluate to true
- Keyword: assert
  ```
  assert eval-expr [: detail-expr];
  assert tail.next == null : "No list end";
  ```
- If the eval-expr does not evaluate to true, an AssertionError is thrown
  - An error (ie extends Error) since an assertion violation is unrecoverable
  - detail-expr can be either
    - A String (becomes the informal description)
    - A Throwable (gets chained as the cause)

Roles of Assertions

- Checking convention (ie representation invariant)
  - At the end of the constructor
  - At the end of every (mutator) method
- Checking requires
  - Defensive programming: check assumptions
- Checking ensures
  - Verify implementation has delivered promised behavior
- Checking flow-of-control
  - Example: "assert false" at a point that should never be reached
  - Style note: “throw (new AssertionError())” usually preferred to “assert false”
- Checking loop invariants
Turning Assertions On (and Off)

- Assertions are disabled by default
  - Enabled with a command-line argument
    - $ java MyProg -enableassertions
  - Class-level and package-level control
    - -ea (-da) to enable (disable) all assertions
    - -ea:edu.osu.Tester to enable only in class Tester
    - -ea:edu.osu..., to enable only in package edu.osu
  - In Eclipse, use “VM arguments”
    - Java > Installed JREs > Edit > Default VM Args
    - (Or use Run Configurations for finer control)
  - Never use assertions with side-effects
  - Example: assert i++ < max;
- Resist temptation to disable assertions for performance
  - Benefit is likely to be negligible
  - Robustness always outweighs speed

Good Practice: Public Methods

- Widely-accepted Java coding practice:
  - Never use assert to check requires of public methods
  - Prefer a RuntimeException (eg IllegalArgumentException)
  - OK for requires of private methods
  - OK for ensures of all methods (private and public)
  - But a violation of requires clause is not recoverable (by client), so it should be an Error, not an Exception!
  - Really, these contract checks belong in a separate component (a checking wrapper)
  - But without better linguistic support for such things, assertions will have to do
- Contrary to Sun recommendations, use asserts liberally, even for public methods
  - assert (requires || exceptional-requires)

Summary

- Interface design: How wide should a specification be?
  - Trade-off: Generality vs power
- Interface design: How should a violation of requires be handled?
  - Exceptions when client lacks enough control
  - Exceptions when check is too expensive for client
- Exceptions
  - Part of component’s interface (visible)
  - Requires vs exceptional requires clauses
- Testing exceptions with JUnit
- Assertions
  - Can be turned on/off at execution time