CSSE2002/7023

Semester 2, 2021

Programming in the Large

Week 5.1: Javadoc, Specifications, and Contracts

In this Session

- Javadoc
- Procedural Abstraction
- Specifications
- Contracts
- Defensive Programming

Javadoc

```
/**
  * Do a thing
  * @param s Name of species
  * @param f Fraction of Avagadro's Number
  * @return Time required (in microseconds)
  */
public long doThing(String s, float f)
```

Javadoc comments

- Begin with /** — *note the second **
  - Javadoc generator ignores all other comments
- End with */
- Must be immediately above the thing being commented
- Use tags begining with @
- Some tags take parameters, some just text

Tags

For methods:

Tag	**Notes/comment**
@param *varname ...*	Describe a parameter for a method
@return *...*	Describe the return *value* (not its type unless that is unclear from the signature)
@throws *Exception ...*	Describe when a particular type of exception is thrown by the method
@requires *Precondition*	Soon
@ensures *Postcondition*	Soon

For classes:

- `@author` *`authorname`*
- `@copyright ...`

These are only the basic ones.

Procedural Abstraction

Methods are considered by **what** they do,

Procedural Abstraction

Methods are considered by **what** they do,

. . . not **how** they do it.

Procedural Abstraction

Procedural abstraction makes programs easier to read and maintain by keeping the size and complexity of methods small. Guidelines:

1. Methods should be decomposed according to functionality, not lines of code.
2. Each method should perform one task (can be an aggregate task).
    - Delegate additional tasks to private "helper" methods.
3. Be suspicious of:
    - Methods that are hard to name (See 2).
    - Repeated code
    - Any method which is overly long or complex.
    - Conditional statements on the types of method arguments.

NB: *Suspicious* is not the same as "it's wrong".

Specifications

Ideally specifications:

- Allow the implementation of a method to be read or written without needing to examine the implementations of any other methods.
- Allow a method to be re-implemented without requiring changes to any methods that use it.
- Should:
  - Rule out all implementations that are unacceptable (i.e. be sufficiently restrictive).
  - Not preclude acceptable implementations (i.e. be sufficiently general).
  - Be easy for users (programmers) to understand (i.e. be clear).
  - Draw attention to possible consequences of implementation decisions[1].

[1] e.g. if it will affect performance

Restrictiveness — Keep out incorrect implementations

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  */
public int search(int[] ar, int x)
```

What happens if `x` is not in `ar`?

Restrictiveness — Keep out incorrect implementations

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  */
public int search(int[] ar, int x)
```

What happens if `x` is not in `ar`?

"Well, obviously . . ."

Restrictiveness — Keep out incorrect implementations

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  */
public int search(int[] ar, int x)
```

What happens if `x` is not in `ar`?

"Well, obviously . . ."

NO!

Restrictiveness — Keep out incorrect implementations

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  */
public int search(int[] ar, int x)
```

What happens if `x` is not in `ar`?

"Well, obviously . . ."

NO!

This is tempting, but the spec doesn't say, so don't assume.
By its silence, the spec allows any value to returned. So a return of 1 could mean "It's at index 1" or "I didn't find it"

Restrictiveness

Better:

```
/** Return an index (i) of ar such that
  * ar[i] == x, if any
  * else, return -1
  */
```

Suppose `x` appears in `ar` multiple times. There is nothing in the spec which requires:

- that you will get the lowest index
- that you will get the same answer running multiple times

Generality — Allow acceptable alternative versions

```
/** Calculate the square root of a number within
  * a given error
  * @param sq Number whose square root is to be found
  * @param e The allowable error
  * @return rt such that 0 <= (rt * rt - sq) <= e
  */
public double sqrt(double sq, double e)
```

The above is overly restrictive: `-e <= (rt*rt - sq) <= e`

Generality

```
/** Examine ar[0], ar[1], ... in turn and
  * return the index of the
  * first one that is equal to x, if any,
  * else return -1
  */
public int search(int[] ar, int x)
```

Even if you think that is the most natural way to do something, that doesn't mean it should be part of the spec.

Specs should describe *what* happens, not how.

- don't include implementation details

Clarity

A specification should facilitate communication between people.

Two ways in which communication could fail:

1. Reader doesn't understand.
2. Reader only *thinks* they understand.

Clarity can be improved by:

- Being as concise as possible. Long specs are:
  - more likely to contain errors/contradictions
  - less likely to be read properly
  - more likely to be misunderstood
- Redundancy may help
  - be clear it is redundant by using e.g. or i.e.
- Making specifications more formal

Formality

Specifications of software range in formality

- informal — e.g. normal comments
- semi-formal — structured English documentation (e.g. using Javadoc tags
- formal — mathematical constraints (e.g. using Java syntax for boolean expressions)

Informal Specifications

```
/** Withdraw an amount from this account and
  * return how much is left.
  */
public int withdraw(int amount)
```

What happens when:

- `amount` is negative?
- `amount` is bigger than balance?

Is the balance changed when there is a failure?

Semi-Formal Specfications

```
/** Withdraw an amount from this account.
  * @param amount The amount to withdraw
  * @return Balance of this account after
  *         successful withdrawal
  */
public int withdraw(int amount)
```

Clearer but many of the same questions apply.

Formal Specifications

```
/** Withdraw an amount from this account.
  * @require amount >= 0 && amount <= getBalance()
  * @ensure getBalance() == \old(getBalance()) - amount
        && \result == getBalance()
  */
public int withdraw(int amount)
```

Document using:

```
javadoc -tag require -tag ensure Thing.java
```

Contracts

Formal specifications can be written using contracts

```
/** Description of method's behaviour
* @require precondition
* @ensure postcondition
*/
```

If the code calling the method satisfies the *precondition*, then the method guarantees[2] to satisfy the *postcondition*.

If the code calling the method does not satisfy the precondition, then the method guarantees nothing (any behaviour is allowed).

Note that this doesn't make all the questions go away.

[2] Watch out for *Exceptions* here

Programming without Contracts

```
/** Withdraw an amount from this account.
  *  @param amount The amount to withdraw.
  *  @return The balance of this account after
  *          the withdrawal successfully occurs,
  *          or -1 if it fails.
  */
public int withdraw(int amount) {
    if (amount < 0) {
        return -1;
    }
    if (balance < amount) {
        return -1;
    }
    ...
}
```

Calling code:

```
b = myAccount.withdraw(a);
if (b == -1) ...
```

Programming with Contracts

```
/** Withdraw an amount from this account.
 * @require amount >= 0 && amount <= getBalance()
 * @ensure getBalance == \old(getBalance()) - amount
 *         && \result == getBalance()
 */
public int withdraw(int amount) {
    //all conditionals are gone!
    ...
}
```

Calling code:

```
b = myAccount.withdraw(a);
```

It is up to the calling code to make sure it fulfils its side of the contract.

- i.e. `0 <= a <= myAccount.getBalance()`

More Complex Specifications

Java syntax for boolean expressions plus. . .

`\result`	return value of method
`a ==> b`	a implies b (if a then b)
`a <==> b`	a if and only if b
`\old(x)`	value of x before method occurs
`\forall C c;`	for all objects c of class C
`\exists C c;`	there exists an object c of class C

... or simply use natural language (English)

Specifications using the above constructs are supported by compile and runtime checking tools for JML (Java Modelling Language)
`http://www.eecs.ucf.edu/~leavens/JML/index.shtml`

Assertions

Assertions can be used to check formal specifications

```
/** Withdraw an amount from this account.
 * @require amount >= 0 && amount <= getBalance
...
 */
public int withdraw(int amount) {
    assert amount >= 0 && amount <= balance :
        "precondition of withdraw violated";
...
}
```

To check at runtime use: `java -ea ClassName`

Assertions

Never use `assert` as a replacement for if statements.

Many languages / systems disable all assertions in "release builds".

Assertions should be used

- to aid debugging
  - I think this should be true here
- as part of comprehensive test strategies
  - check to see if "the impossible" happens

Defensive Programming — Because “people are awful”

Defensive programming — explicitly checking for invalid inputs and bad situations, ensuring the the software does not behave dangerously regardless of input.

What if someone calls the method without checking the precondition?

Defensive Programming — Because “people are awful”

Defensive programming — explicitly checking for invalid inputs and bad situations, ensuring the the software does not behave dangerously regardless of input.

What if someone calls the method without checking the precondition?
Because you know they will.

Particularly when dealing with external input sources or when guarding critical resources, it may be better to be defensive outside the wall and use contracts inside it.

Problem of `null`

Lots of programs (and programmers) have problems with `NullPointerException`.

Root causes (contract programming):

- `null` not covered by contract – everyone is confused
- `null` is covered – programmer not paying attention
- unexpected `null`s propagate – hard to track down

Best practice for API design:

- default is that `null` is NOT a valid argument / result
- specify when `null` is allowed / expected / returned *and* document its meaning

Problem of `null`

Best practice for implementation:

- check for `null`, implicitly or explicitly
- check early
- `NullPointerException` or `IllegalArgumentException`

Example:

```
/**
...
     @param name a non-empty string
*/
public void setName(String name) {
     if (name.length() == 0) { // throws NPE if name is null
          throw new IllegalArgumentException("name empty");
     }
     this.name = name;
}
```