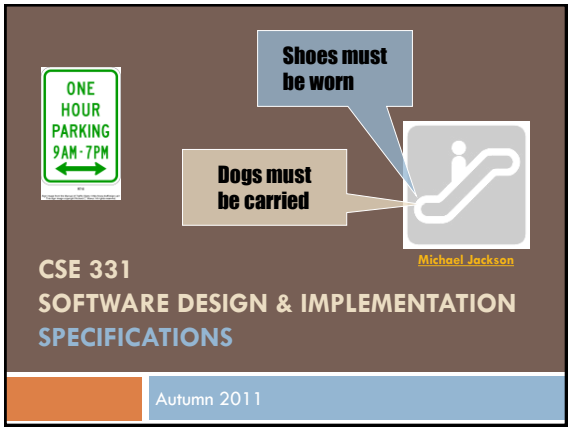
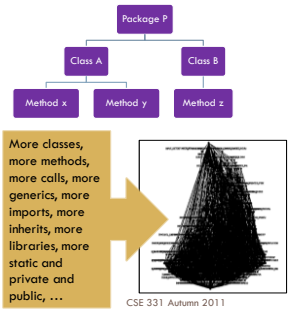


CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
SPECIFICATIONS

Autumn 2011

The challenge of scaling software

- Small programs tend to be simple and malleable: relatively easy to write and to change
- Big programs tend to be complex and inflexible: harder to write and (much) harder to change
- Why? In large part because interactions become harder to understand and to manage
- We will try to reduce this challenge by using *specifications* to simplify and manage these interactions

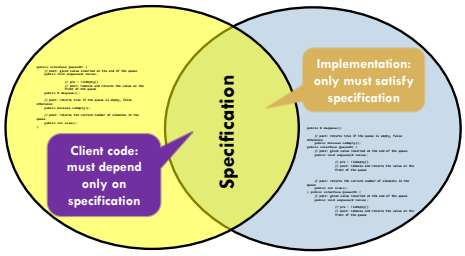


More classes, more methods, more calls, more generics, more imports, more inherits, more libraries, more static and private and public, …

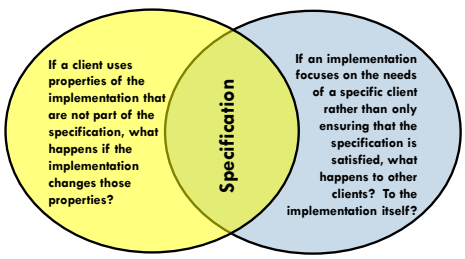
A *specification* is a contract

- A set of obligations agreed to by the user (client) and the manufacturer (implementer) of the product
- Facilitates simplicity by two-way isolation
  - Isolate client from implementation details
  - Isolate implementer from how the part is used
  - Discourages implicit, unwritten expectations
- Facilitates change
  - Allows either side to make changes that respect the specification
  - An effective specification changes very little (at most), allowing the code (on both sides) to be more malleable

Different but dualistic roles
Implementers vs. Clients



Respecting the specification has value



You play both roles

- Not only in 331, but commonly in your career
- By reducing how much you and your dualistic "alter ego" know about each others' view, the interactions can be kept cleaner
- This is hard!

Leading towards "Truth, Justice and the 331 Way"

Isn't the interface a specification?

7

- Java (and most languages) allow programs to define interfaces as a boundary between the implementations and the clients

```
public interface List<E> {
  public int get(int);
  public void set(int, E);
  public void add(E);
  public void add(int, E);
  …
  public static boolean sub(List<T>, List<T>);
}
```

- The interface is a weak kind of specification that provides the syntax, but nothing about the behavior and the effects
- This kind of contract says, "I'll give you this and you'll give me that, but 'this' and 'that' aren't carefully defined"

Why not just read code?

8

```
T boolean sub(List<T> src, List<T> part) {
  int part_index = 0;
  for (T elt : src) {
    if (elt.equals(part.get(part_index))) {
      part_index++;
      if (part_index == part.size()) {
        return true;
      }
    }
    else {
      part_index = 0;
    }
  }
  return false;
}
```

In small groups, spend 1-2 minutes listing reasons why reading code would be a poor substitute for having a specification

Code is complicated

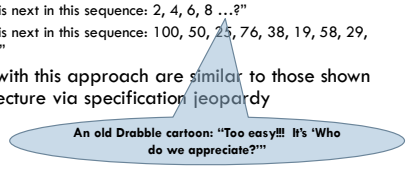
9

- Much detail not needed by client – understanding every line of code is excessive and impractical
  - Ex: Read all source code of Java libraries before using them?
- Client should care only what is in the specification, not what is in the code
  - When a client sees the implementation code, subconscious dependencies arise and may be exploited
  - Why is this bad?
  - Why should you be especially concerned about this?

Why not just run code?

10

- The client depends on what the implementation computes – what better way to find out than by seeing what it computes?
- If you run enough test inputs, you are forming a partial specification
  - Ex: from many standardized tests
    - "What is next in this sequence: 2, 4, 6, 8 …?"
    - "What is next in this sequence: 100, 50, 25, 76, 38, 19, 58, 29, 88, …?"
- Problems with this approach are similar to those shown in the 1st lecture via specification jeopardy



Which code details are essential?

11

- A lot of choices are made in writing code – some are essential while others are incidental – but which is which?
  - Internal variable names? Algorithms used? Resource consumption (time, space, bandwidth, etc.)? Documentation? Etc.?
- Code invariably gets rewritten, making the distinction between essential and incidental crucial
  - What properties can the client rely on over time? Which properties must the implementer preserve for the client's code to work? Future optimizations, improved algorithms, bug fixes, etc.?
  - Alternatively, what properties might the implementation change that would break the client code?
- There is no simple definition of this distinction, but it is captured in practice in every specification – again, your sensibilities about this issue with grow over time

Comments

With more comments on comments later on

12

- Comments can, and do, provide value if and when written carefully – and when kept up-to-date
- Many comments convey only an informal, general idea of what that the code does

```
// This method checks if "part" appears as a
// sub-sequence in "src"
   boolean sub(List<?> src, List<?> part) {
     ...
  }
```

- This usually leaves ambiguity – for example, what if `src` and `part` are both empty lists?

Improving the spec of `sub()`

13

```
// Check whether "part" appears as a sub-sequence in "src"
```

- Needs additional clarification

```
// a) src and part cannot be null
// b) If src is empty list, always returns false
// c) Results may be unexpected if partial matches can happen
//    right before a real match; e.g., list (1,2,1,3) will not
//    be identified as a sub sequence of (1,2,1,2,1,3)
```

- Or needs to be replaced with a more detailed description

```
// This method scans the "src" list from beginning to end,
// building up a match for "part", and resetting that match
// every time that...
```


Further improving the spec of `sub()`

14

- A complicated description suggests poor design and rarely clarifies a specification
- Try to simplify rather than describe complexity
  - Perlis: Simplicity does not precede complexity, but follows it."
- Rewrite the specification of `sub()` more clearly and sensibly

```
// returns true iff sequences A, B exist such that
// src = A : part : B [":" is sequence concatenation]
```

- The "declarative" style of this specification is important
  - Contrast to an operational style such as "`This method scans the "src" list from beginning to end`…"
  - The mathematical flavor is not necessary, but it can help reduce ambiguity

Examples of specifications

15

- Javadoc "is a tool for generating API documentation in HTML format from doc comments in source code."
  - Get used to using it
- Javadoc conventions expect programs to provide
  - method prototype – basically, the name of the method and the types of the parameters and of the return
  - text description of method
  - *@param*: description of what gets passed in
  - *@returns*: description of what gets returned
  - *@throws*: list of exceptions that may occur

Example: Javadoc for String.contains

16

- *tags* in Java comments
- These are parsed and formatted by Javadoc
- Viewable in web browsers

```
/**
* Returns true if and only
* sequence of char values.
*
* @param s the sequence to
* @return true if this str
* @throws NullPointerExcep
* @since 1.5
*/
public boolean contains(Ch
  return indexOf(s.toStrin
}
```

public boolean contains(CharSequence s)
Returns true if and only if this string contains the specified sequence of char values.
Parameters:
s- the sequence to search for
Returns:
true if this string contains s, false otherwise
Throws:
NullPointerException
Since:
1.5

CSE 331 specifications

(Javadoc is extensible)

17

- The *precondition*: constraints that hold before the method is called
  - *requires*: spells out any obligations on client (if *requires* is not satisfied by a client, the implementation is unconstrained)
- The *postcondition*: constraints that hold after the method is called (if the precondition held)
  - *modifies*: lists objects that may be affected by method; any object not listed is guaranteed to be untouched
  - *throws*: lists possible exceptions
  - *effects*: gives guarantees on the final state of modified objects
  - *returns*: describes return value

Ex 1: Spec and an implementation

18

```
static int test(List<T> lst, T oldelt, T newelt)
  requires lst, oldelt, and newelt are non-null
           oldelt occurs in lst
  modifies lst
  effects  change the first occurrence of oldelt in lst to newelt
           no other changes to lst
  returns  position of element in lst that was oldelt and is now newelt
```

```
static int test(List<T> lst, T oldelt, T newelt) {
  int i = 0;
  for (T curr : lst) {
    if (curr == oldelt) {
      lst.set(newelt, i);
      return i;
    }
    i = i + 1;
  }
  return -1;
}
```

Ex 2: Spec and an implementation

19

```
static List<Integer> listAdd(List<Integer> lst1, List<Integer> lst2)
  requires lst1 and lst2 are non-null
           lst1 and lst2 are the same size
  modifies none
  effects  none
  returns  a list of same size where the ith element is the sum of the
             ith elements of lst1 and lst2
```

```
static List<Integer> listAdd(List<Integer> lst1,
                             List<Integer> lst2) {
  List<Integer> res = new ArrayList<Integer>();
  for(int i = 0; i < lst1.size(); i++) {
    res.add(lst1.get(i) + lst2.get(i));
  }
  return res;
}
```


Ex 3: Spec and an implementation

20

```
static void listAdd2(List<Integer> lst1, List<Integer> lst2)
  requires lst1 and lst2 are non-null
           lst1 and lst2 are the same size
  modifies lst1
  effects  ith element of lst2 is added to the ith element of lst1
  returns  none
```

```
static void listAdd2(List<Integer> lst1, List<Integer> lst2) {
  for (int i = 0; i < lst1.size(); i++) {
    lst1.set(i, lst1.get(i) + lst2.get(i));
  }
}
```


Ex 4: Spec and an implementation

21

```
static void uniquify(List<Integer> lst)
  requires ???
  modifies ???
  effects  ???
  returns  ???
```

```
static void uniquify(List<Integer> lst) {
  for (int i=0; i < lst.size()-1; i++)
    if (lst.get(i) == lst.get(i+1))
      lst.remove(i);
}
```

In small groups, spend 1-2 minutes filling in the ??? in the specification above

Ex: java.util.Arrays.binarySearch

22

public static int binarySearch(int[] a,int key)

Searches the specified array of ints for the specified value using the binary search algorithm. The array must be sorted (as by the sort method, above) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.
Parameters:
a- the array to be searched.
key- the value to be searched for.
Returns:
index of the search key, if it is contained in the list; otherwise, (-(insertion point) - 1).

[…Long description…]

Improved specification

23

```
public static int binarySearch(int[] a,int key)

requires: a is sorted in ascending order
returns:
     some i such that a[i] = key if such an i exists,
     otherwise -1
```

- Returning `(-(insertion point)-1)` is an invitation to bugs and confusion
  - Consider: The designers had a reason; what was it, and what are the alternatives?
- We'll return to the topic of exceptions and special values in a later lecture

Summary

24

- Properties of a specification
  - The client *relies only* on the specification and on nothing (else) from the implementation
  - The implementer *provides everything* in the specification and is otherwise unconstrained
- Overall, effective use of specifications leads to simpler and more flexible programs that have fewer bugs and cleaner dependencies

Next steps

- Assignment 0
  - Due today 11:59PM
- Assignment 1
  - out later today
  - due Wednesday (10/5) 11:59PM
- Assignment 2
  - out Wednesday (10/5)
    - due in two parts
    - part A on Friday (10/7) 11:59PM
    - part B the following Wednesday (10/12) at 11:59PM
- Lectures
  - Testing and Junit (M)
  - Equality (W)
  - Abstract data types I (F)
  - Abstract data types II (M)