CSE 331

Software Specifications

slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia

http://www.cs.washington.edu/331/

Specifications

- **specification**: A set of requirements agreed to by the user and the manufacturer of a software unit or product.
  - Describes the client and implementer's expectations of each other.

- In object-oriented design, a class's spec describes all publicly visible behavior or attributes of that class.
  - the class's superclass and interfaces implemented (if any)
  - constructors
  - methods
  - public constants or fields (if any)
  - nested / inner types
  - any assumptions or guarantees made by the class

Benefits of specs

- Specs provide *abstraction*:
  - **procedural abstraction** (describe methods' behavior, not code)
  - **data abstraction** (describe classes' functionality, not implementation)

- Specs facilitate simplicity by *two-way* isolation:
  - Isolate client from implementation details
  - Isolate implementer from how the part is used
  - Discourages implicit, unwritten expectations

- Specs facilitate change:
  - The spec, rather than the code, gets "frozen" over time.

How is a spec written down and documented?

Code as spec (bad)

- The class's author might say, "To understand how my class works, just look at its code." What's wrong with this?

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

  - Too much detail! Client only cares *what* it does, not *how* it does it.

Interface as spec (bad)

- The class's author might say, "To understand how my class works, just look at its public interface." Is this good or bad?

```
public interface List<E> {
    public int get(int index);
    public void set(int index, E value);
    public void add(E value);
    public void add(int index, E value);
    ...
    public boolean subList(List<E> src, List<E> part);
}
```

  - Not enough detail! Interface describes only the *syntax*, but the client also needs to understand in detail the *semantics* (behavior).

Comments as spec

- Comments are *essential* to properly specifying behavior.
  - But many comments are informal and incomplete:

```
// checks to see if part appears within src
public boolean subList(List<String> src, List<String> part) {
```

- In what ways are the above comments inadequate?
  - Must `part`'s elements appear consecutively, in the same order?
  - What if `src` is null? What if `part` is null?
  - What if either list is empty? What if both are empty?
  - What is the expected runtime of the method?
  - What value does it return if `part` is found, versus if it isn't? (arguably obvious, but not stated very clearly in the comments)

What is a better comment?

- If the previous comment is inadequate, is this one a better choice?

```
// This method has a for loop that scans the "src" list from
// beginning to end, building up a match for "part", and
// resetting that match every time that a non-matching
// element is found.  At the end, it returns false if ...
public boolean subList(List<E> src, List<E> part) {
```

  - The above comments describe too many implementation details.
  - It is possible to describe behavior thoroughly without describing every detail of the code used to implement that behavior.

Spec by documentation

- The following comment header describes the behavior in detail:

```
/** Returns whether all elements of part appear
 * consecutively within src in the same order.
 * (If so, returns true;  otherwise, returns false.)
 * src and part cannot be null.
 * If src is an empty list, always returns false.
 * Otherwise, if part is an empty list, always returns true.
 * ... */
public boolean subList(List<String> src, List<String> part) {
```

- Note that it does not describe the code inside the method.
  - Only describes what the method's externally visible behavior (return value) will be, based on its externally supplied parameters.

Spec exercise

- Suppose a method *M* takes an integer *arg* as an argument
  - Spec 1: "returns an integer equal to its argument"
  - Spec 2: "returns a non-negative integer equal to its argument"
  - Spec 3: "returns an integer ≥ its argument"
  - Spec 4: "returns an integer that is divisible by its argument"
  - Spec 5: "returns its argument plus 1"
- Which code meets which spec(s)?
  - Code 1: return arg;
  - Code 2: return arg + arg;
  - Code 3: return Math.abs(arg);
  - Code 4: return arg++;
  - Code 5: return arg * arg;
  - Code 6: return Integer.MAX_VALUE;
    - ignore int overflow for all five.

Spec1	Spec2	Spec3	Spec4	Spec5

Good documentation

- Good documentation comments describe the following:
  - the method's overall core behavior or purpose
  - preconditions (what the method *requires*)
  - postconditions (what the method promises)
    - *modifies*: What objects may be affected by a call to this method?
      - (Any object not listed here is assumed to be untouched afterward.)
    - *throws*: What errors or exceptions might occur?
    - *effects*: Guarantees on the final state of any modified objects.
    - *returns*: What values will the value return under what circumstances?

Spec strength

- A **weaker spec** is one that requires more and/or promises less.
  - less work for the implementer of the code; more for the client
  - *examples*: doesn't work for negatives; requires sorted input; undefined results if the list contains duplicates; strings must be in valid format

- A **stronger spec** is one that requires less and/or promises more.
  - less work for the client, but harder to implement
  - *examples*: guaranteed to find a match; uses a default if a bad value is supplied; specifies behavior for entire range of input; runtime bounds

- If a spec S_2 is stronger than S_1, then for any implementation I,
  - I satisfies S_2 $\Rightarrow$ I satisfies S_1
    - Which kind of spec is better? (It depends.)

Class as an ADT

- **abstract data type** (ADT): A description of a type in terms of the operations that can be performed on a given set of data.
  - abstracts from the details of data representation
  - a spec mechanism; a way of thinking about programs and designs

- Start your design by designing data structures
  - Write code to access and manipulate data

ADT implementation

- **abstract data type (ADT)**: A description of a type in terms of the operations that can be performed on a given set of data.

```
public class Point {
    private double x;
    private double y;
    ...
}
```

```
public class Point {
    private double r;
    private double theta;
    ...
}
```

- Are the two above classes the same or different?
  - *different:* can't replace one with the other
  - *same:* both classes implement the concept "2-d point"
- Goal of ADT methodology is to express the sameness:
  - Clients depend only on the concept "2-d point". This is good.
  - Delays decisions; fixes bugs; allows performance optimizations.

2-D point as ADT

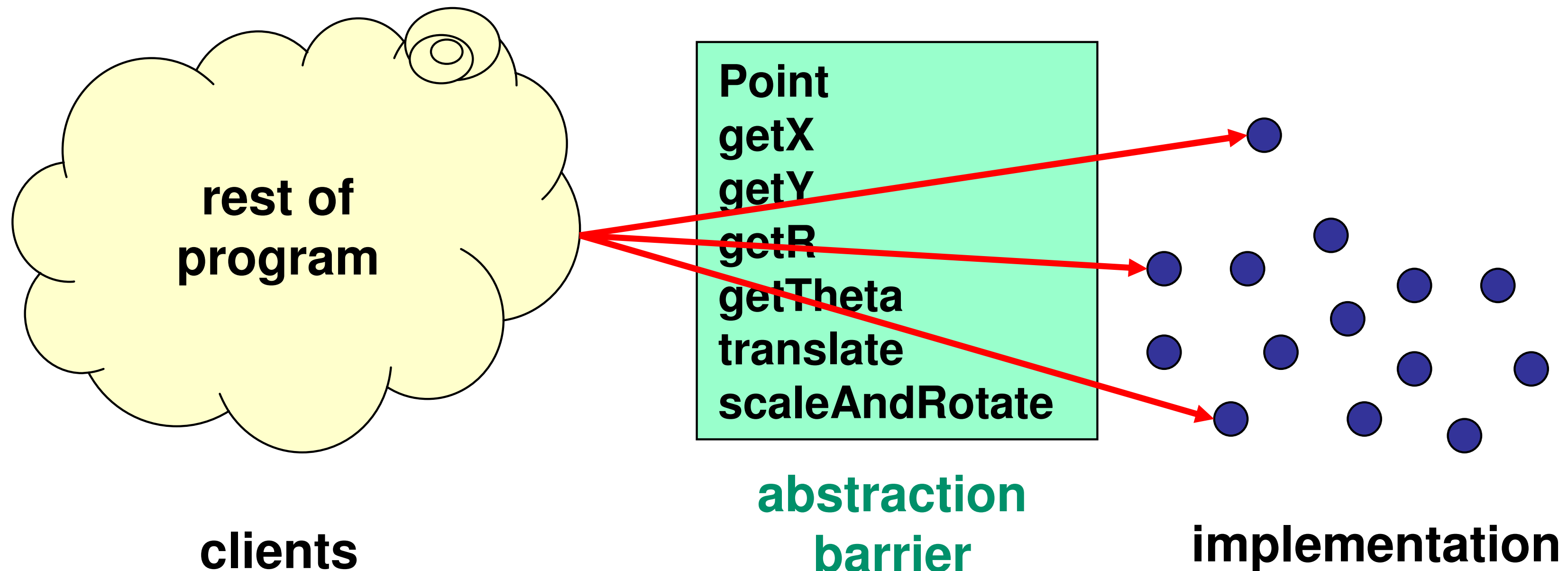
```
public class Point {
    // A 2-d point exists somewhere in the plane, ...
    public double getX()
    public double getY()
    public double getR()
    public double getTheta()

    // can be created
    public Point()                          // new point at (0, 0)

    // can be modified
    public void translate(double dx, double dy)
    public void scaleAndRotate(double dr, double dtheta)

    ...
}
```

Abstraction barriers



- The implementation is hidden.
- The only operations on objects of the type are those that are provided by the abstraction.

Categories of methods

- **accessor** or **observer**: Provides information about the callee.
  - Never *modifies* the object's visible state (its "abstract value")
- **creator**: Makes a new object (constructors, factory methods).
  - Not part of pre-state: in *effects* clause, not *modifies.*
- **mutators**: Modifies state of the object on which it was called.
  - Each method has a *side effect* on the callee.
- **producers**: Creates another object(s) of the same type.
  - Common in immutable types, e.g. String substring; prototypes.
  - Must have no side effects.