The UNIVERSITY of VERMONT

Benjamin Fenster
CS-100 (Damon)
25 January 2003

Notes – Abstraction

I. Concept
   a. Ignore “unnecessary” details
   b. The key to being a good developer is to be able to abstract.
   c. Types of abstraction
      i. Procedural
      ii. Data abstractions
      iii. Collections (“Iteration” in the text)
      iv. Type Hierarchies. Generalize groups together.

II. Procedural
   a. Simplest form of abstraction
   b. Define a new high-level abstraction that ignores sub-level details.
   c. “Walk over there” is really “pick up foot, move forward…”
   d. Abstraction by Specification
      i. Don’t say *how* it works. Say *what* it does.
      ii. If you don’t specify something, it’s allowed to vary
      iii. Which details are important will vary from procedure to procedure.
      iv. Means many implementations can fit into the same specification.
   e. Advantages
      i. Locality
         1. Don’t need to understand the entire program. Just understand your piece in full and abstract the rest.
         2. Enables multi-programmer projects since nobody needs to understand the details of the whole thing.
      ii. Modularity. Can replace the inner workings of a class, function, whatever, without affecting users.
      iii. Why study procedural abstraction in an O.O. class?
         1. Methods ARE procedures.
         2. Abstraction by specification is why inheritance works.
   f. Specifications
      i. Need to describe the behavior precisely and accurately
      ii. Precisely means it’s a very small, well-defined point.
      iii. Accurately means the point is in the right place.
      iv. What does the function do?
      v. Formal Definitions
         1. Use a mathematically precise language (one of many).
         2. Results in necessarily precise specifications, so you only need to be concerned about accuracy
      vi. Informal Specifications
         1. Plain English
         2. Easier, clearer communication, if done right.
         3. It’s easy to be ambiguous with plain language, but there’s higher accuracy because you won’t accidentally say the wrong thing.
      vii. Use some structure to help eliminate the ambiguity
         1. REQUIRES. What’s needed for the call? What must be true?
         2. MODIFIES. What inputs/etc have been modified
         3. EFFECTS. What will the modification look like?
      viii. Total Functions operate on all possible inputs. There’s no REQUIRED specification.
      ix. Partial Functions have REQUIRES fields – some possible inputs aren’t allowed.
      x. MODIFIES can be an input, a system state (global) or something reachable through an input.
      xi. Specs should limit the implementation only as much as necessary.

        1. What's necessary for the user? Specify that!
        2. What can they ignore? Don't specify that!

g. Generalization
    i. As much as possible, broaden the set of legal inputs.
    ii. Be careful though, being too general *guarantees* underspecification.

h. Specificity
    i. Procedures should only do one thing!
    ii. Be able to say what the thing is, or you won't be able to implement it.
    iii. You may have some difficulty with the specification details, but be sure to understand the general idea

III. Data
  a. Procedural Abstraction depends on consistent use of data types
  b. What types should you use for complex data?
      i. Your code depends heavily on what you pick.
      ii. What operators are available for a given type?
      iii. All functions using the data will be affected by any changes you make.
  c. Data Abstraction
      i. Isolates data and any behavior associated with it.
      ii. Take decisions about the data out of the specification and make the specification more about observable behavior.
      iii. Also includes procedural abstraction, so one must know the latter to use the former.
  d. Essentially making a new data type – nothing more than a class.
  e. Called object-based programming
  f. Documentation
      i. Class OVERVIEW: a short description, maybe a half-dozen words.
      ii. Can add longer comments, examples of code use, whatever, but keep them separate from the overview.
  g. Choosing Operations
      i. It's far more important to pick the right operations than it is to get the right internal representation.
          1. It's hard to change the available operations because other code refers to them.
          2. Implementation can be changed at any time.
          3. Start with the simple, "stupid" version of the implementation and then make it better later as needed.
      ii. Consider what people will do with the data.
      iii. Allow all the reasonable operations
      iv. Keep it simple and clean.
      v. Provide good building blocks, not complete specialization.
      vi. Combine tasks only when there's a huge benefit, otherwise let the users write combinations.
      vii. Consider Alternatives
          1. If you absolutely need more than one implementation, make several distinct options.
          2. Consider the performance gains achieved by combining operations (open connection, send, close).
      viii. Enumerate constructors.
          1. The exception to the above rule, in a way.
          2. Take a large range of potential data.
          3. Provide an empty constructor.
  h. Standard Operations
      i. Remember the operations inherited from `Object`.
      ii. `.equals()`

1. Are two objects the same? That could be asking one of two different questions.
2. Object Identity
    a. Two objects are the same if changing one affects the other.
    b. Hard to tell for immutable objects – they can't be changed so how would you know?
    c. Tested using the == operator. Asks "are they the same memory?"
    d. If a == b they will ALWAYS be identical (there's no way to make them be different).
3. Do they have the same values right now?
    a. If two sets have the same elements at this particular moment, they can be called "equal."
    b. Tested with the `.equals()` method.
    c. If `a.equals(b)` it may or may not stay that way.
    d. .equals() should be…
        i. Symmetric. `a.equals(b)` matches `b.equals(a)`
        ii. Reflexive. Equal to self.
        iii. `a.equals(b) && b.equals(c)` matches `a.equals(c)`
        iv. Null preserving. An object cannot equal `NULL` unless it actually *is* `NULL`.
        v. Consistent. Remains the same over time if neither object is modified.
4. The book describes `.equals()` as object identity.
    a. That's incorrect!
    b. The *default* implementation is indeed object identity.
    c. Many, many classes reimplement `.equals()` to test values.
5. Possible to test primary values and ignore secondary information.
    a. A Set might have storage size information in addition to its actual elements.
    b. Testing just the elements could be a valid use of `.equals()`

iii. `.hashCode()`
1. Returns an integer that can be used to identify this object.
2. May or may not be unique among other objects' hash codes.
3. If `a.equals(b)` then `a.hashCode()` MUST be equal to `b.hashCode()`

iv. .clone()
1. Makes a copy of `this`
2. `a != a.clone()` because it's a *copy*
3. `a.equals(a.clone())` should be true!
4. By default, all fields are copied. That may or may not be okay!
    a. If copying all fields means keeping a "pointer" to some object in both a and `a.clone()` then the two objects will have access to the same data!
    b. If necessary, `.clone()` will need to call `.clone()` on the embedded object.
    c. `newR = super.clone();  newR.students = students.clone();`

v. toString()
1. Be able to convert your data to a readable string.
2. Simply returns data in the most basic string format conceivable.