Procedural Abstraction

Comp-303 : Programming Techniques

Lecture 5

Alexandre Denault
Computer Science
McGill University

Winter 2004

Announcements . . .

- The classroom for Thursday's tutorial has still not been determined.
- *Pizza and Beverages* will be held next Thursday at 17:30, first floor Trottier.
- Assignment 1 will be handed out . . . now!

Last lecture . . .

- What is UML and why learn it?
- References and Tools for UML
- Class diagrams
  - Elements of a class
  - Relations (Hierarchy, Contains, Uses)
  - Cardinality
  - Notes
  - Putting it all together
- State diagrams
- Sequence diagrams
- Requirements & Specifications

Procedural abstraction

- Abstracting a single action using parameterization and specification.
- *Parameterization* abstracts from the identity of the data being used, allowing the procedure to be used in more situations.
  - relevant: presence, number, types of parameters
  - irrelevant: identity of parameters
- *Specification* abstracts from the realization of the action, allowing multiple implementations.
  - relevant: what is done
  - irrelevant: how it is done

Benefits

- Benefits of abstraction by specification
  - Locality
    - The implementation of an abstraction can be read or written without needing to examine the implementations of any other abstractions.
    - The abstraction can be used without examining its implementation.
  - Modifiability
    - An abstraction can be re-implemented without requiring changes to any abstractions that use it.

Specifications

- Specifications should be precise.
- Specifications can be written in a formal language or in an informal language.
  - *Formal* specification languages have a precise meaning and can be checked automatically or even used to generate code (Comp 304).
  - *Informal* specifications are easier to read or write but harder to give precise meaning.
    We will use informal English in comments to define specifications.

Specifications of procedural abstractions

- A function header is formal form of specification :
  - name
  - order and type of parameters
  - return type
  - visibility
- We can improve on this by adding some informal specification :
  - *REQUIRES* specifies use constraints
  - *MODIFIES* lists modified inputs/side-effects
  - *EFFECTS* describes behavior for inputs complying to requirements, outputs and modifications

An example: SquareRoot

```
public float squareRoot (float x) {
   // REQUIRES: x should be positive
   // MODIFIES: nothing
   // EFFECTS: returns an approximation of
         square root of x
}
```

Another example: Arrays

```
public class Arrays {
   // OVERVIEW: stand-alone procedures for manipulating arrays of
   integers

   public static int search (int [ ] a, int x)
      // EFFECTS: if x is in a, returns the index where x is stored
      //   otherwise returns -1

   public static int searchSorted (int [ ] a, int x)
      // REQUIRES: a is sorted in ascending order
      // EFFECTS: if x is in a, returns the index where x is stored
      //   otherwise returns -1

   public static void sort (int [ ] a)
      // MODIFIES: a
      // EFFECTS: Rearranges the elements of a in ascending order
      //   e.g. if a = {3,1,6,1} ,  a_post={1,1,3,6}
}
```

Additional explanation: Arrays

- *MODIFIES* is omitted if no modifications happen.
  - In other words, the properties of parameter objects or global objects are unchanged.
  - If modifications does occur, the state before the procedure is invoked should be related to the state after the procedure returns (the change should be explained).
- *REQUIRES* is omitted if the procedure if nothing is expected of the total.
  - In other words, the procedure functions correctly with any input.
- *EFFECTS* should always be specified. A function should always *do* something.
  - Examples can be used to clarify specifications.

New Objects

- In general, with Java, we tend to favor creating new objects over modifying old ones.
- Since Java is Garbage Collected, old unused objects are eventually automatically disposed.
- This allows us to turn some mutable objects into immutable objects.
- The following is an example of favoring object creation over modifying old ones:

```
public static int [ ] boundArray (int [ ], int n)
   // EFFECTS: Returns a new array containing the
   // elements of a in the order they appear in a except
   // that any elements of a > n are replaced by n
```

Implicit Inputs

- Sometime, the parameters of a function are not found in the function header.
- These parameters are called implicit inputs.
- When using implicit inputs, they should be properly documented in the specification.
- Here is an example of a function using implicit inputs.

```
public static void copyLine()
   // REQUIRES: System.in contains a line of text
   // MODIFIES: System.in and System.out
   // EFFECTS: Reads a line of text from System.in, advances cursor
   //  in System.in to end of line, writes the line on System.out
```

Implementing procedures

- Specifications are written first.
- Procedure bodies are added later.
  - Implementations should modify only those inputs specified in MODIFIES clause.
  - Implementations should produce results according to EFFECTS clause if REQUIRES clause holds.

SearchSorted

```
public static int searchSorted (int [ ] a, int x) {
   // REQUIRES: a is sorted in ascending order
   // EFFECTS: if x is in a, returns the index where x is stored
   //   otherwise returns -1
   // uses linear search
   ...
}
```

- If a is unsorted, we will get the wrong.
- If a is null, we return -1 (or we could throw an exception)
- Includes comment on algorithm used.

Designing Procedural Abstractions

- Procedures are introduced to shorten the calling code, clarify structure, and reduce code duplication.
- Sometimes, a piece of code in a procedure can be further subdivided into another function.
- These subfunctions have a well-defined purpose and allows to separate details of the subfunction from the main function.
- However, we must be careful. Further decomposition could be counter-productive.
- Rule of thumb #1: if you have trouble naming the subprocedure, then it probably shouldn't be separated.
- Rule of thumb #2: if you see almost identical code repeated twice, then it is probably useful to define a procedure.

Properties of procedures

- *Minimally constraining*: a specification should constrain details of the procedure only to the extent necessary. Only details that matter to the user are constrained to leave the implementor the greatest freedom for efficient implementations.
- *Undetermined*: a procedure is considered undetermined if the acceptable output is a set (as opposed to a unique output).
- *Deterministic Implementation*: these procedures will produce exactly the same output when give the same input. Undetermined procedures can be non-deterministic, but it usually takes extra program effort.

Properties of procedures

- *Generality*: a specification is more general if it can handle a larger class of inputs.
  For example: working with any size of array instead of fixed size
- *Simplicity*: procedures should have a well-defined purpose, easy to explain, independent of context of use. If it is hard to name the procedure there may be a problem.

Partial vs. total procedures

- A procedure is *total* if its behavior is specified for all legal inputs. Otherwise it is *partial*.
- *Partial* procedures are always specified with a REQUIRES clause. They are less safe than *total* procedures, and should only be used:
  - when the context of use is limited (private helper procedures).
  - when they enable a substantial benefit such as better performance.
- Library procedures intended for general use should always be *total*.

Summary

- A procedure is a mapping from inputs to outputs, with possible modification of inputs.
- Its specification describes its behavior, providing a contract between users and implementors.
- The specification does not change when the implementation changes. This provides locality and modifiability.
- Specifications should be minimal and can be undetermined / non-deterministic.
- Desirable properties include simplicity and generality.
- Implementations should be total when possible, and may be partial when the context of use is limited and controlled, such as for private helper procedures.

Assignment 1

In this assignment, you must implement the BigInt data abstraction. You may not use floating point numbers or any of the Big* number classes provided by Java.

BigInt can store integer numbers from -2^120 to 2^120 no loss of precision. These numbers must be stored in 4 integers (part1, part2, part3, part4).

a) fill in the bodies of the methods of class BigInt
b) add methods "boolean smallerThan(BigInt other)" and "boolean largerThan(BigInt other)" with specifications and test code.

Bonus: add a "BigInt divide(BigInt other)" method with specifications and test code.

Make sure you respect specifications. This means that a method without REQUIRES specifications should accept all inputs (Total implementation), and that a method with REQUIRES specs should preserve those specs.

BigInt class

```
public class BigInt {

// OVERVIEW: this class implements BigInt numbers
// BigInt numbers consist of a large number store in 4 intergers. You are
// free to choose how the number will be store. The only condition is that
// all four integers (and no other) must be used to store the numerial part
// of the number.
//
// (Can’t start adding float, integers or arrays to store number.)
//
// A BigInt is printed with an apostrophy every three characters.
//   for example 123’456’789’123’465’000
// If the number is larger than 2^120 or smaller than -(2^120), the BigInt
// number is "Not a Number" (NaN). Any operation involving a NaN has a NaN
// as a result. For predicates (methods returning a boolean), NaN is
// considered to be larger than any other number. BigInt numbers are
// immutable.
```

Methods to implement

```
// -- constructors --
public BigInt (int n) {
// -- inherited from Object --
public String toString () {
public boolean equals (Object o) {
// -- predicates --
public boolean isPositive () {
public boolean isNegative () {
// -- arithmetic --
public BigInt add (BigInt other) {
public BigInt subtract (BigInt other) {
public BigInt multiply (BigInt other) {
public BigInt negation () {
public BigInt absoluteValue () {
public BigInt square () {
public BigInt cube () {
public BigInt power (int degree) {
```

Testing

```
BigInt a = new BigInt(123456789);
BigInt b = new BigInt(2);
BigInt c = new BigInt(-444444444);
BigInt d = new BigInt(999999999);
BigInt f = new BigInt(0);

System.out.println(a);
System.out.println(b);
System.out.println(a.add(b));
System.out.println(a.subtract(b));
System.out.println(a.multiply(b));
System.out.println(a.square());
System.out.println(a.cube());
System.out.println(a.cube().add(a.cube()));
System.out.println(a.subtract(c));
System.out.println(c.power(125));
System.out.println(c.absoluteValue().negation());
System.out.println(c.power(125).multiply(f));
System.out.println(a.equals(a));
```

Policies on Cooperation

- You are free to discuss this assignment with your friends.
- You may *NOT* share code with your friends.
- If you do work with someone, mention it in your header.
- I *cannot* give a failing grade to students I find cheating …

  …I *can* report him to the Dean of Science.

Hints

- Think before you code.
  - How am I going to store the numbers?
  - What base?
- Obvious trick to make the assignment simpler.
- Create *many* more test cases.
- Start early. It’s a lot trickier than it looks.
- The assignment is due Tuesday, February 3th at 11:55 pm

Tool of the day: JCreator

- JCreator is a powerful IDE for Java.
- It provides the user with a wide range of functionality:
  - Project management
  - Project templates
  - Code-completion
  - Debugger interface
  - Editor with syntax highlighting
  - and so on ...
- It's *Free*!
- You can find more information at

  `http://www.jcreator.com/`