Designing Classes
Part 1

Basis

- Look at slides on Abstract Data Types
  - » **They give much of the underlying basis**
    - **Types of features**
    - **Properties of features**
    - **Documentation**
- Here we give some additional guidelines

Designing Classes

- Experience shows it is critical to properly design class interfaces, especially in multi-person projects
- Want a set of design principles that can lead to quality and durable classes

  There are no rules

- We are interested in how a class will appear to its clients

  Not the internals

- Make an interface

  *** simple**
  *** easy to remember**
  *** easy to learn**
  *** able to withstand change**

Side Effects – Referential Transparency

- Functions should not have side effects
  - » **Do not return a value and change state**
  - » **A contentious issue**

    Efficiency being the prime motivation for functions with side effects
- Supporting rationale

  An expression is referentially transparent if any sub-expression may be replaced with its value without changing the expression

Side Effects – Example Problem

- Functions are used in expressions, consider the following

  INTEGER get_integer is ... end

  Read integer from an input and return the result

- Use it in an expression (as functions are intended to be used)

  result get_integer + get_integer

  Reads two integers from the input

- Referential transparency says we can do

  result 2 * get_integer

  Reads one integer from the input

Side Effects – Problem Solution

- For input the design should be as follows

  get_integer is a procedure that saves value in an attribute

  last_integer : integer
  get_integer is ... last_integer ▯ the_value end

  Reference attribute when you want the value

  result ▯ 2 * last_integer
  or result ▯ last_integer + last_integer

  Both expressions use one integer from the input

  Use get_integer twice to read two values

  - **Program is clear with no surprises**
  - **Can reason more easily about the program**

Side Effects – Counter Rationale

- Consider the case of removing an item from a data structure

  remove (KEY : key) : DATA is ... end

  - **Need to search for the object**
  - **Useful to return data associated with the key**
  - **Have function with side effects**

- Consider alternative

  data ← search (key)
  remove (key)

  Two searches – inefficient

Side Effects – Rebuttal on Efficiency

- Use the same design as get_integer

```
last_data  : DATA
remove (KEY key)
                    is ...  last_data ?=  the_value end
```

  - **Remove saves the data in an attribute**
  - **User accesses the data if they want it**
  - **Clearer as to what is happening**

- Keeping the last value, or current position (cursor) is a useful design strategy
  - **Reduce number of functions with side effects**
  - **Can have operations relative to current position**

Side Effects – Rebuttal on Efficiency – 2

- Random number generation

  value ⇐ random

  Changes the "seed" on each call

- Poor abstraction ⇒ poor design
- Good abstraction ⇒ good design
  - **The underlying notion is of a sequence of random numbers**
  - **This abstraction is data based – not operation based**

random.forth
value ⇐ random.item

Active Data Structures

- Fits with functions with no side effects
  - » **Maintain**

    current object
    current position
    etc.

  - » **Provide methods that are relative to current**

    after, next, forth
    before, previous, back
    replace (data) – work on current

  - » **For singly linked lists**

    Automatically save pointer to previous node for the client

Side Effects – Other Problems

- Even when the programmer knows about the side effects problems can occur

  Suppose you program the following where f_b is a function with side effects

  r = f_a (f_b , f_b)

  An optimizing compiler, may see f_b as a function and replace one of the calls with the result of the other call

  r = f_a (f_b , f_b)

  Which call is done first? Compiler dependent. Order of parameter evaluation is rarely part of a language definition

How Many Arguments for a Feature

- Arguments come in two types
  - » **operand**
    - **Value needed to do work**
    - **Must appear as an argument**
  - » **option**
    - **Value used to make a choice as to how to do the work – output in blue in 20 point Helvetica**
    - **Should not appear as an argument**
- For a good design

  Options are set with independent procedures

  object.set_font(...) object.set_font_size(...)

Class Size

- Should not be an issue
  - » **Include what must be included**
    - **Design a complete, orthogonal set of methods**
    - **User has a simple, complete control of objects**
    - **No side effects among functions**
  - » **Include additional methods that can be justified**
    - **Increase the efficiency of combinations of operations**
    - **Simplify user manipulation of objects**
    - **Provide aliases**
      - **Easier use**
      - **Keep uniform names across classes for equivalent semantics**