CSC108H Lecture 9

Dan Zingaro
OISE/UT

July 9, 2009

Clickers: Aspects of Software

What is the most important property of software?

- A. Correctness
- B. Efficiency
- C. Reusability
- D. Looks really cool
- E. Has tons and tons of features

Modularity

- Two key aspects of software construction
  - Extendibility: ease of adapting software products to changes of specification (e.g. millennium problem)
  - Reusability: developing components for one software development that can then be used in others (e.g. game engines)
- We use the term **modularity** to refer to the combination of extendibility and reusability
- A software construction method is modular if it helps designers produce software systems made of autonomous elements connected by a simple structure

Criteria for Modularity

A design method worthy of being called modular must satisfy five criteria. We will use Python modules to frame our discussion.

1. Decomposability: does it help decompose a software problem into a small number of less complex subproblems that can be worked on separately?
2. Composability: can the elements be combined in new ways, outside of those for which they were originally designed?
3. Understandability: can a human reader understand each module without having to know the others?
4. Continuity: does a small change in problem specification affect only one or at most a small number of modules? (e.g. constants)
5. Modular protection: does an abnormal runtime condition in one module remain confined to that module, or at worst propagate only to a few neighboring modules?

Rules for Modularity

These criteria imply a number of rules about the use of modules

- Every module should communicate with as few others as possible (continuity, composability, decomposability)
- If two modules communicate, they should exchange as little information as possible (continuity)
- Information hiding (continuity)
  - Only those properties that are part of its specification, not its implementation, should be used by clients of the module
  - When changes are made to the "secret elements", clients are not affected

Open-Closed Principle

- Modules should be both open and closed. . . . huh?
- Open
  - Still available for extension (i.e. so we can add features)
  - Almost impossible to foresee everything that the module should offer
- Closed
  - Available for other modules (i.e. it has a stable interface, and is available for use in other programs)
  - If we never closed a module, every developer would always be waiting for the completion of someone else's job
- Traditional modular structures don't help resolve this dilemma

Open-Closed Principle...

- Assume module A is used by clients B, C and D. Everything's working smoothly ...
- Then, modules E, F and G come along and require an extended version of A. Let's call this extended version A2
- What are our choices?
- If we adapt A, we may break the execution of B, C and D
- If we copy the text of A and paste it into a new module A2 (and keep doing stuff like that), we'll generate a lot of near-identical modules
- We'll see that inheritance lets us create A2 by listing **only the differences** with A

Single Choice

Assume we are managing a collection of publications; we can use a Python list, stored in a module.

```
pub_list = []

def add_book (author, title, year, publisher):
  pub_list.append (
    ('book', author, title, year, publisher))

def add_journal (author, title, year, volume, issue):
  pub_list.append (
    ('journal', author, title, year, volume, issue))

def add_proceedings (author, title, year, editor, place):
  pub_list.append (
    ('proceedings', author, title, year, editor, place))
```

Single Choice...

- Most clients of this module will have to discriminate among types of publications (e.g. a module that creates a bibliography)
- But, what happens if we later add new types of publications?
- Every client that made decisions based on publication type must be modified!
- Single choice: whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list
- Traditional methods do not provide a solution; we will see that polymorphism and dynamic binding help here

Why Reusability?

- Timeliness: by relying on existing components, we have less software to develop
- Decreased maintenance effort
- Reliability: expectation that the components have been tested and validated
- Efficiency: in a large application, you can hardly expect to have an expert for every field touched on in the development
- Consistency: we are influenced by well-written libraries
- Investment: preserve know-how and inventions of best developers

What to Reuse

- Personnel: most common source of reusability is the developers themselves
- Reuse of designs and specifications
- But these aren't the components that can be readily included in a new software product
- We can reuse in terms of an abstract interface description of our modules
- This is not the same as source code, although the modules may be delivered that way
- i.e. we don't have to look at our media module's code to use it

Top-down Design

- Top-down design is a textbook method for software construction
- We begin with a description of the "top function" of the system we want to produce
- e.g. "create a program for generating exams"
- Then, we continue with **refinement steps**, designed to decompose each operation into one or more simpler operations
- e.g. we can refine the above example into the steps: "open question bank", "read questions", "randomly generate questions", etc. and continue decomposing each of these steps ...
- Can be useful for developing individual algorithms, but not for entire systems

Top-down Design...

- Top-down design favors decomposability, but often produces modules that are not easy to combine with modules coming from other sources (e.g. context-specific)
- In the evolution of a system, what may originally have been perceived as the systems main function may become less important over time (e.g. exam generator morphing into a class management system)
- Sometimes, there really is no "top": what is the top of an operating system?
- It places premature emphasis on temporal constraints
- In object-oriented design, we specify each applicable operation, but defer for as long as possible specifying the operations order of execution
- Objects, rather than functions, are a more stable characterization of software's properties; e.g. a payroll system will always deal with employees, hours worked, pay cheques

Classes

- In Python, and lots of other OO languages, a **class** serves as the mechanism for generating new types of objects
- At the same time, classes serve as modular structures, helping us organize our code
- We'll begin with a class of two-dimensional points
- These points will "know how" to translate themselves on the x- and y-axes, and compute their distance to another point
- These features become methods of the class

Two-Dimensional Points (point.py)

```
from math import sqrt

class Point(object):
  '''Two-dimensional points'''

  def __init__ (self):
    self.x = 0
    self.y = 0

  def translate (self, a, b):
    '''Move by a horizontally, b vertically.'''
    self.x += a
    self.y += b

  def distance (self, other):
    '''Return distance between this point and other.'''
    return sqrt ((other.x - self.x) ** 2 \
                 + (self.y - other.y) ** 2)
```

Class Instances...

- The methods above all took a first parameter called `self`
- Since we can generate multiple two-dimensional points from the class, `self` indicates which of these **instances** is the current instance
- This way, methods called on an instance know whose variables to change!
- Python passes the current instance to each method when we make a method call; we do not explicitly include `self` as a parameter
- To use the class, we become a **client** of the class, by creating an object of its type

```
p1 = Point()
P1.x, p1.y
p1.translate (3, 4)
p1.x, p1.y
```

Object Creation

- To create an object of class `c`, we use `c()`, optionally providing parameters in the parentheses
- This statement executes the `__init__` method (called a constructor) of the class that we are instantiating
- The `__init__` method of class `Point` simply sets the `x` and `y` coordinates to zero
- For increased flexibility, we can create a new version of the class whose `__init__` method accepts two parameters, defaulting to `0` if not provided

Two-Dimensional Points, New Constructor (point2.py)

```
from math import sqrt

class Point(object):
  '''Two-dimensional points'''

  def __init__ (self, x = 0, y = 0):
    self.x = x
    self.y = y

  def translate (self, a, b):
    '''Move by a horizontally, b vertically.'''
    self.x += a
    self.y += b

  def distance (self, other):
    '''Return distance between this point and other.'''
    return sqrt ((other.x - self.x) ** 2 \
                 + (self.y - other.y) ** 2)
```

Class Attributes

- In a function-oriented approach, we might translate a point with `translate (p1, 3, 4)`
- In pure OO, there is no such thing as a function separate from an object
- Each object is an instance of some class, and the class determines the attributes of the object
- Class attributes may be functions or variables of any type
- In particular, this lets us reference other objects from our objects, as if they were values of standard Python types

Line Segments (segment.py)

```
from point2 import Point

class Segment(object):
  '''Line segments'''

  def __init__ (self, x1, y1, x2, y2):
    self.p1 = Point(x1, y1)
    self.p2 = Point(x2, y2)

  def translate (self, a, b):
    '''Move by a horizontally, b vertically'''

    self.p1.translate (a, b)
    self.p2.translate (a, b)

  def length (self):
    '''Length of the line segment'''

    return self.p1.distance (self.p2)
```

Object Identity

- Every object we create has a unique identity
- Two objects with different identities may have identical fields
- Changing fields of an object does not change its identity
- In Python, we use `is` to test object identity, and == to test whether two (possibly different) objects have the same field values

```
a = [1, 2, 3]
b = a
a == b # True
a is b # True
b = [1, 2, 3]
a == b # True
a is b # False!
```

Clickers: Object Identity

What is the output of the following program?

```
from point import Point

p1 = Point()
p2 = Point()
p1.translate (2, 2)
p2.translate (2, 2)
print p1 is p2
```

- A. `True`
- B. `False`

Classes and Instances

- When we run a `class` statement, we get a class object (much as `def` gives us a function object)
- Just like with modules, top-level assignments within a class statement (not nested in a `def`) generate attributes in a class object
- These class attributes are shared by all instances of the class: `def` statements become methods, whereas variable assignments become shared data across instances
- Each time a class is called with `()` function syntax, it creates and returns a new instance object, which has links to class attributes
- Unlike in other OO languages, you can assign a new object to any class or instance attribute at any time (but this is usually not what you want to do!)

Class Variable Example (counter.py)

This class keeps track of how many times it has been instantiated; `instances` is available through `Counter.instances`, or through any of the class' instances, but it is only **one** variable!

```
class Counter(object):

  instances = 0

  def __init__(self):
    Counter.instances += 1
```

Objects as Strings (point3.py)

- Trying to use `str (p)` or `print p` on a `Point` object `p` gives us a memory address, not a useful string representation
- We can provide a `__str__` method that tells Python what to display in these contexts

```
class Point(object):
  ...

  def __str__(self):
    return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Comparing Objects

- When we ask Python to do a comparison like `a < b` or `a == b`, Python calls `cmp`
- `cmp(a, b)` returns -1 when `a` is less than `b`, 0 if they are equal, or 1 when `a > b`
- Unfortunately, `cmp` doesn't know what to do with our point objects yet, so it defaults to comparing them by memory address
- We can provide our class with a `__cmp__` method that works like `cmp`
- Let's make points with smaller `x` values be "less than" points with bigger `x` values; if they're tied, we'll further sort on `y`

Comparing Objects (point4.py)

```
class Point(object):
  ...

  def __cmp__ (self, other):
    if self.x < other.x:
      return -1
    elif self.x > other.x:
      return 1
    else:
      if self.y < other.y:
        return -1
      elif self.y == other.y:
        return 0
      else:
        return 1
```

References

- Bertrand Meyer. Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997.
- Gerald M. Weinberg. The Psychology of Computer Programming, Dorset House Publishing, 1998.