Software Design Review

May 4, 2011

Alex Morrow

This is an updated document from my review sessions last week. Please send feedback to alex.morrow @ olin.edu.

If you do nothing else to prepare for the exam, at least do or re-do the last few diagnostics and labs. And go over the vocabulary, which is included at the end of this document.

Introduction
Our goal for Software Design was to learn how program collaboratively. You and your team have now demonstrated you could do this. The final exam will test your mastery of the material.

Mastery here means the ability to take problems and translate them to solutions. If you Google the “Zen of Python,” you’ll find this koan: “If the implementation is easy to understand, it may be a good idea.” Here’s the Zen of Software Design: “Think Python when you listen but don’t speak it. You'll get the idea."

Concrete to Abstract
This semester you climbed a hill from the concrete concepts of digital computers to the abstraction of object-oriented programming. If you’re a freshman you’ll climb another next year: User-Oriented Collaborative Design (OOCD). The paths of these two courses wind up at the same place. From it, you can get a nice view of interconnected abstractions.

UOCD teaches listening. You learn what it is to be someone other than yourself. You also learn that it's possible to be right about complicated things and wrong about simple things.

After a while a, you hear a faint voice describe a simple problem that, if solved, would help someone. You then learn to describe the problem you think you know how to solve in simple terms, using the vocabulary you've learned by listening. By Thinking Python without speaking Python, you'll find you can structure the simple problem so that it has a simple solution. You become ... The Fool on the Hill. We show said hill in Figure 1. You can look down the western slope and see the problem and down the eastern slope and see the solution.

From the standpoint of studying for the final, the critical thing to realize is that you spent a semester learning how to turn concrete computer concepts into object-oriented solution abstractions. Your time in the future will be spent listening to concrete client concepts and turning them into problem
abstractions. The object-oriented solution abstractions you have learned give you a pretty good way to analyze the hidden structure of problems. If you can structure client problems as object-oriented problem abstractions you'll find it easier to create collaborative teams and explain the solution to them. Most product managers, project managers, interface designers, hardware engineers, software engineers and test and release engineers understand object-oriented concepts. So do entrepreneigneers. The ones with the money.

In studying for the final, therefore, go backwards through the course. If you do nothing else, at least do or redo the last diagnostics and labs. The exercises below go backwards through Think Python, from the most abstract concepts to the most concrete.

An Example
Suppose a stock trading company wants a better system to let its customers manage their portfolios. They describe a simple portfolio for stocks as having problem-space elements Stocks, Investors, and Holdings. A software engineering team enlightened by Think Python may suddenly realize that, in class diagram terms, a Holding represents a HAS-A relationship between an Investor and some Stocks. Without using Python terminology, the team checks with knowledgeable representatives from the client to find out if they are correct. They find out that to account for stock splits, the HAS-A relationship must name a Stock with a specific Issue Date. The team continues this line of questioning until they fully understand each problem object. This understanding includes relationships with other classes, attribute types, and methods, all without bringing up the Think Python terminology.

Figure 1: The Hill of Abstraction
Abstract Interfaces
At the top (most abstract point) of Figure 1 you’ll see the term “Abstract Interface.” An interface is considered abstract if it does not have much of an implementation. Its goal is to relate elements of a problem space one another and the objects in its solution.

There are several ways of representing abstract interface in Python. The simplest is a class whose methods have no function bodies, like this:

```python
class AbstractHand(object):
    """Fundamental operations for Hand of Cards."""
    def __init__(self): pass  # either pass
    def shuffle(self):
        " Shuffle the Cards in the Hand"
        raise NotImplemented  # or raise
    def deal(self,other):
        " Move a card from this Hand to another Hand."
        raise NotImplemented  # NotImplemented
```

These Abstract Interfaces have classes and methods with names, parameters and docstrings, but no implementations other than a pass or raise NotImplemented statement. The idea is that the interfaces given are contracts for elements of the solution. These elements correspond (one hopes, in obvious ways) to elements of the problem space.

The use of abstract interfaces parallels the language of other written specifications. Blueprints for house remodeling often identify an object abstract class (a light fixture, a bush) with minimal specification (ceiling, perennial) and leave the actual selection to the client. The IS-A relationship is considered common sense (don’t buy a wall sconce if the blueprint calls for a ceiling fixture). Computers have no common sense, so the relationship between objects must be made explicit. In the case of Python IS-A relationships, the relationship is shown by the methods any object satisfying the IS-A contract must have. The IS-A relationship is formalized as class inheritance.

The next lower level of abstraction is the HAS-A relationship. The HAS-A relationship between classes can be represented in a class hierarchy by a minimal implementation of the IS-A relationship, one that can in turn be inherited by all implementation classes. In this minimal implementation only special methods such as __init__ are defined. The HAS-A implementation creates only those instance variables that represent HAS-A relationships. Any other implementation is put in a regular method that the special method calls. This regular method is a stub, to be overridden by an implementation class. For example,
This form of interface building makes the IS-A and HAS-A relationships difficult to change. This is a good thing, and should be encouraged by making them read-only to the implementers. In this way, the IS-A and HAS-A relationships become a contract that make it more difficult for implementers to fail to meet the client's expectations about the elements of the problem space.

This technology example represents the enlightenment about the use of abstraction we hope you obtained from the course. What we are trying to impress on you by the UOCD to Think Python diagram in Figure 1 is simply this: almost any amount of time spent to understand a problem in the client’s terms will be repaid many times over in the ways that matter to both the client and your team. First of all, the client will be able to understand the structure of the program at the highest level, because you and your team will present it in problem space terms. Secondly, your team wants to build a reputation for coming in early and under budget. It's much faster and cheaper to fix design errors during the design phase. In the case of software development, this mean before you’re written any code.

Abstract interfaces are particularly important to get right and to lock in before your team writes any code. They not only define the contracts between implementation team and client. During implementation, however, they also define the contracts between implementation sub-teams and, often, sub-teams and their individual software engineers. Any changes to an interface during coding will be disruptive. The entire project team may have to drop their other work for days or weeks to tear out working code and re-implement it if an abstract interfaces changes. If the need for an abstract interface is discovered late in the implementation cycle, it will be too late. The software engineering team and the entreprengineers will team up to prevent adding it.

**The Zen of Python Programming**

Our focus in this review is to raise awareness in the class about a vital advocacy role played by enlightened software engineers on the UOCD team. They must advocate for a formulation of the problem that lends itself to a straightforward OOCD implementation. The enlightened part of their role during design is that they must make their points by contributing to clarification of the problem model in
ways that continue to engage the client, rather than bringing implementation issues into the design discussion.

An unenlightened programmer might complain that these two top levels of abstraction are worthless, since they contain no programming. Graduates of Software Design, however, having received enlightenment in the Way of the Program, recognize the value of truthful simplicity. They see these abstract classes as the culmination of understanding of the problem and therefore the beginning of the solution. Right?
Review Exercises:
These exercises provide additional testing of how well you understand concepts. Note that they are in order from highest level of abstraction down. Do the ones that will help you. The answer key is on the Software Design website.

Review TK Inter.
Be sure you understand how callables are used to map user mouse and keyboard actions to callbacks. What are the levels of abstraction in TKInter?
Q1: Inheritance:
Imagine that you are working with a client who wants to create a Hoyle.com website in which the rules of every known card game are represented as computer programs. After UOCD sessions with the client, you realize that many card games are small variants of others. You discuss this matter with the client and agree to make it easy to represent variants and variants of variants. You decide to use Python's inheritance mechanisms to represent games.

(q1.1) Why?

to make it easy to have an number of variants of variants, you decide to use super(class,self).__init__(self,*args,**kwargs) as the first line in every __init__ method.

(q1.2) Why is this required?

you remember the Think Python days and look again at card.py. You realize that you will have to change card.py in order to use "super" as the basis for the __init__ method.

(q1.3) Why do you have to do this?

(q1.4) What classes will you have to change?

Q2: Classes and Methods
1. An unordered list is an collection type that resembles a set in that it cannot be indexed, but resembles a list in that it can have repeated elements.
2. Create a class called "UList."
3. The docstring should read "Unordered List"
4. The __init__ method should create a HAS-A relationship to a list.
5. The __str__ method should print a representation the user can type it to recreate the object. For example, print Ulist(range(3)) should print "Ulist([1,2,3])"
6. The methods "add" and "pop" should behave like those of set objects.
7. Implement the __or__ operator as the catenation of two Ulists.
8. Q2.1 Here is a rule of thumb that Ulists illustrate. Do you agree with it? "IS-A relationships often add methods. HAS-A relationships often remove methods."
Q3. Classes and Functions

Look over the following function.

Q3 Could you change sqrt(x) to extract the cube root of its argument?

def sqrt(arg):
    def sqrt_nrm(val,tol):
        "Square Root Algorithm -- Newton-Raphson Method"
        def f(v,x): return x ** 2 - v
        def fprime(x): return 2 * x

        vf = float(val)
        guess = vf / 2
        while f(vf,guess) >= tol:
            guess = guess - f(vf,guess) / fprime(guess)
            # draw a stack diagram of the state here for sqrt(0)-
        return guess

    return sqrt_nrm(arg,tol=1e-11)

def f(x): return 2 ** x

print sqrt( f(10) )
Q4: Files
Read over the methods in the io module.
Q4.1 Which method most resembles a class constructor?
Q4.2 Which method is a class instance destructor?
Q4.3 Which methods are class instance iterators?
Q4.4 How can the class instance iterator be restarted?

Q5: Data Structure Selection
Q5.1 What are the four major built-in data structures?
Q5.2 Which data structures are mappings?
Q5.3 Which data structures are invariant?
Q5.4 What is Markov analysis?

Q6: Tuples:
Q6.1 You do remember how to make a one-element tuple, right?
Q6.2 If you use the for loop: for k,v in d: print k,v you see the following error message: ‘need more than 1 value to unpack’ Why?

Q7: Dictionaries:
Q7.1 Create an function to convert a dictionary into a reverse lookup dictionary.
Q7.2 Write a square-root routine that uses a dictionary to avoid recalculating square roots it has already extracted. What is this use of a dictionary called?

Q8: Lists:
Slices: Review slices by reading the Python documentation.
Q8.1 How do you create a copy of a list without using the copy module?
Q8.2 What is a shorter way to write:

```python
[x for x in reversed([3,2,1])]
```

Q9: Strings:
Q9.1: Write a class that provides a mutable string.

Q10: Iteration:
Review the while and for loops and the break and continue statements.
Q10.1: Write a function that tests if a one dictionary is contained in another. issubdictof({2:{3:4}},{1 : {2: {3: 4}}}) is True

Q11: Fruitful Functions:
Q11.1 Can a function return a result without using the return statement?

Q12: Conditionals and recursion:
Q12.1 Review Boolean values; how do the expression (1&2) and the expression (1 and 2) differ?

Q13: TurtleWorld:
Create a class diagram for TurtleWorld.
Why is there a HAS-A relationship from World to Turtle?
Why is there a HAS-A relationship from Turtle to World?

Q14: Functions:
Write a non-recursive definition of the factorial function that does not use a for or while loop. (hint: reduce and lambda)

Q15: Variables, expressions and statements:
Q15.1 What is the syntactic difference between a statement and an expression? Hint: which can you pass to a function?
Q15.2 How are a = [1,0] and c = Card(1,0) similar? (Hint: what are type(a) and type(c)? How would you represent them in an object diagram?

Q16: The Zen of Programming:
Q16.1 Master, what is a computer program?
Glossary for Review

Chapter 1:

problem solving: The process of formulating a problem, finding a solution, and expressing the solution.
high-level language: A programming language like Python that is designed to be easy for humans to read and write.
low-level language: A programming language that is designed to be easy for a computer to execute; also called “machine language” or “assembly language.”
portability: A property of a program that can run on more than one kind of computer.
interpret: To execute a program in a high-level language by translating it one line at a time.
compile: To translate a program written in a high-level language into a low-level language all at once, in preparation for later execution.
source code: A program in a high-level language before being compiled.
object code: The output of the compiler after it translates the program.
executable: Another name for object code that is ready to be executed.
prompt: Characters displayed by the interpreter to indicate that it is ready to take input from the user.
script: A program stored in a file (usually one that will be interpreted).
interactive mode: A way of using the Python interpreter by typing commands and expressions at the prompt.
script mode: A way of using the Python interpreter to read and execute statements in a script.
program: A set of instructions that specifies a computation.
algorithm: A general process for solving a category of problems.
bug: An error in a program.
debugging: The process of finding and removing any of the three kinds of programming errors.
syntax: The structure of a program.
syntax error: An error in a program that makes it impossible to parse (and therefore impossible to interpret).
exception: An error that is detected while the program is running.

Chapter 2:

value: One of the basic units of data, like a number or string, that a program manipulates.
type: A category of values. The types we have seen so far are integers (type int), floating-point numbers (type float), and strings (type str).
integer: A type that represents whole numbers.
float: A type that represents numbers with fractional parts.
string: A type that represents sequences of characters.
variable: A name that refers to a value.
statement: A section of code that represents a command or action. So far, the statements we have seen are assignments and print statements.
assignment: A statement that assigns a value to a variable.
state diagram: A graphical representation of a set of variables and the values they refer to.
keyword: A reserved word that is used by the compiler to parse a program; you cannot use keywords like if, def, and while as variable names.

Chapter 3:
function: A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.
function definition: A statement that creates a new function, specifying its name, parameters, and the statements it executes.
function object: A value created by a function definition. The name of the function is a variable that refers to a function object.
header: The first line of a function definition.
body: The sequence of statements inside a function definition.
parameter: A name used inside a function to refer to the value passed as an argument.

Chapter 4
instance: A member of a set. The TurtleWorld in this chapter is a member of the set of TurtleWorlds.
initialization: An assignment that gives an initial value to a variable that will be updated.
increment: An update that increases the value of a variable (often by one).
loop: A part of a program that can execute repeatedly.
encapsulation: The process of transforming a sequence of statements into a function definition.
generalization: The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).
keyword argument: An argument that includes the name of the parameter as a “keyword.”
interface: A description of how to use a function, including the name and descriptions of the arguments and return value.
refactoring: The process of modifying a working program to improve function interfaces and other qualities of the code.
development plan: A process for writing programs.
docstring: A string that appears in a function definition to document the function’s interface.
precondition: A requirement that should be satisfied by the caller before a function starts.
postcondition: A requirement that should be satisfied by the function before it ends.

Chapter 5
modulus operator: An operator, denoted with a percent sign (%), that works on integers and yields the remainder when one number is divided by another.
**boolean expression:** An expression whose value is either True or False.

**relational operator:** One of the operators that compares its operands: ==, !=, >, <, >=, and <=.

**logical operator:** One of the operators that combines boolean expressions: and, or, and not.

**conditional statement:** A statement that controls the flow of execution depending on some condition.

**condition:** The boolean expression in a conditional statement that determines which branch is executed.

**compound statement:** A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

**branch:** One of the alternative sequences of statements in a conditional statement.

**chained conditional:** A conditional statement with a series of alternative branches.

**nested conditional:** A conditional statement that appears in one of the branches of another conditional statement.

**recursion:** The process of calling the function that is currently executing.

**base case:** A conditional branch in a recursive function that does not make a recursive call.

**infinite recursion:** A recursion that doesn’t have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

---

**Chapter 6**

**temporary variable:** A variable used to store an intermediate value in a complex calculation.

**dead code:** Part of a program that can never be executed, often because it appears after a return statement.

None: A special value returned by functions that have no return statement or a return statement without an argument.

**incremental development:** A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

**scaffolding:** Code that is used during program development but is not part of the final version.

**guardian:** A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

---

**Chapter 7**

**multiple assignment:** Making more than one assignment to the same variable during the execution of a program.

**update:** An assignment where the new value of the variable depends on the old.

**decrement:** An update that decreases the value of a variable.

**iteration:** Repeated execution of a set of statements using either a recursive function call or a loop.

**infinite loop:** A loop in which the terminating condition is never satisfied.
Chapter 8

object: Something a variable can refer to. For now, you can use “object” and “value” interchangeably.
sequence: An ordered set; that is, a set of values where each value is identified by an integer index.
item: One of the values in a sequence.
index: An integer value used to select an item in a sequence, such as a character in a string.
slice: A part of a string specified by a range of indices.
empty string: A string with no characters and length 0, represented by two quotation marks.
immutable: The property of a sequence whose items cannot be assigned.
traverse: To iterate through the items in a sequence, performing a similar operation on each.
search: A pattern of traversal that stops when it finds what it is looking for.
counter: A variable used to count something, usually initialized to zero and then incremented.
method: A function that is associated with an object and called using dot notation.
invocation: A statement that calls a method.

Chapter 9

file object: A value that represents an open file.
problem recognition: A way of solving a problem by expressing it as an instance of a previously solved problem.
special case: A test case that is atypical or non-obvious (and less likely to be handled correctly).

Chapter 10

list: A sequence of values.
element: One of the values in a list (or other sequence), also called items.
index list: A list that is an element of another list.
nested list: A list that is an element of another list.
list traversal: The sequential accessing of each element in a list.
mapping: A relationship in which each element of one set corresponds to an element of another set. For example, a list is a mapping from indices to elements.
accumulator: A variable used in a loop to add up or accumulate a result.
augmented assignment: A statement that updates the value of a variable using an operator like +=. reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.
map: A processing pattern that traverses a sequence and performs an operation on each element.
filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.
object: Something a variable can refer to. An object has a type and a value.
equivalent: Having the same value.
identical: Being the same object (which implies equivalence).
reference: The association between a variable and its value.
aliasing: A circumstance where two or more variables refer to the same object.
delimiter: A character or string used to indicate where a string should be split.
**Chapter 11**

dictionary: A mapping from a set of keys to their corresponding values.
key-value pair: The representation of the mapping from a key to a value.
item: Another name for a key-value pair.
key: An object that appears in a dictionary as the first part of a key-value pair.
value: An object that appears in a dictionary as the second part of a key-value pair. This is more specific than our previous use of the word “value.”

**Chapter 12**
tuple: An immutable sequence of elements.
tuple assignment: An assignment with a sequence on the right side and a tuple of variables on the left. The right side is evaluated and then its elements are assigned to the variables on the left.
gather: The operation of assembling a variable-length argument tuple.
scatter: The operation of treating a sequence as a list of arguments.
DSU: Abbreviation of “decorate-sort-undecorate,” a pattern that involves building a list of tuples, sorting, and extracting part of the result.
data structure: A collection of related values, often organized in lists, dictionaries, tuples, etc.
shape (of a data structure): A summary of the type, size and composition

**Chapter 13**
deterministic: Pertaining to a program that does the same thing each time it runs, given the same inputs.
pseudorandom: Pertaining to a sequence of numbers that appear to be random, but are generated by a deterministic program.
default value: The value given to an optional parameter if no argument is provided
override: To replace a default value with an argument.
benchmarking: The process of choosing between data structures by implementing alternatives and testing them on a sample of the possible inputs.

**Chapter 14**
persistent: Pertaining to a program that runs indefinitely and keeps at least some of its data in permanent storage.
format operator: An operator, %, that takes a format string and a tuple and generates a string that includes the elements of the tuple formatted as specified by the format string.
format string: A string, used with the format operator, that contains format sequences.
format sequence: A sequence of characters in a format string, like %d, that specifies how a value should be formatted.

text file: A sequence of characters stored in permanent storage like a hard drive.

directory: A named collection of files, also called a folder.

path: A string that identifies a file.

relative path: A path that starts from the current directory.

absolute path: A path that starts from the topmost directory in the file system.

catch: To prevent an exception from terminating a program using the try and except statements.

database: A file whose contents are organized like a dictionary with keys that correspond to values.

Chapter 15

class: A user-defined type. A class definition creates a new class object.

class object: An object that contains information about a user-defined type. The class object can be used to create instances of the type.

instance: An object that belongs to a class.

attribute: One of the named values associated with an object.

embedded (object): An object that is stored as an attribute of another object.

shallow copy: To copy the contents of an object, including any references to embedded objects; implemented by the copy function in the copy module.

deep copy: To copy the contents of an object as well as any embedded objects, and any objects embedded in them, and so on; implemented by the deepcopy function in the copy module.

object diagram: A diagram that shows objects, their attributes, and the values of the attributes.

Chapter 16

prototype and patch: A development plan that involves writing a rough draft of a program, testing, and correcting errors as they are found.

planned development: A development plan that involves high-level insight into the problem and more planning than incremental development or prototype development.

pure function: A function that does not modify any of the objects it receives as arguments. Most pure functions are fruitful.

modifier: A function that changes one or more of the objects it receives as arguments. Most modifiers are fruitless.

functional programming style: A style of program design in which the majority of functions are pure.

invariant: A condition that should always be true during the execution of a program.

Chapter 17

object-oriented language: A language that provides features, such as user-defined classes and
method syntax, that facilitate object-oriented programming
object-oriented programming: A style of programming in which data and the operations that manipulate it are organized into classes and methods.
method: A function that is defined inside a class definition and is invoked on instances of that class.
subject: The object a method is invoked on.
operator overloading: Changing the behavior of an operator like + so it works with a user-defined type.
type-based dispatch: A programming pattern that checks the type of an operand and invokes different functions for different types.
polymorphic: Pertaining to a function that can work with more than one type.

Chapter 18

encode: To represent one set of values using another set of values by constructing a mapping between them.
class attribute: An attribute associated with a class object. Class attributes are defined inside a class definition but outside any method.
instance attribute: An attribute associated with an instance of a class.
veen: A method or function that provides a different interface to another function without doing much computation.
inheritance: The ability to define a new class that is a modified version of a previously defined class.
parent class: The class from which a child class inherits.

Chapter 19

GUI: A graphical user interface.
widget: One of the elements that makes up a GUI, including buttons, menus, text entry fields, etc.
child class: A new class created by inheriting from an existing class; also called a “subclass.
option: A value that controls the appearance or function of a widget.
keyword argument: An argument that indicates the parameter name as part of the function call.
callback: A function associated with a widget that is called when the user performs an action.
bound method: A method associated with a particular instance.
event-driven programming: A style of programming in which the flow of execution is determined by user actions.
event: A user action, like a mouse click or key press, that causes a GUI to respond.
event loop: An infinite loop that waits for user actions and responds.
item: A graphical element on a Canvas widget.
bounding box: A rectangle that encloses a set of items, usually specified by two opposing corners.
pack: To arrange and display the elements of a GUI.
geometry manager: A system for packing widgets.

binding: An association between a widget, an event, and an event handler. The event handler is called when the event occurs in the widget.