C13_Chpa17&Chap18_Interf&Implem-DataEncap-Refactoring_90

May 29, 2020

1 Chapter 17. Classes and Methods

1.1 17.11. Interface and implementation

One of the goals of **object-oriented design** is to make software more *maintainable*, which means that you can keep the program working when other parts of the system change, and modify the program to meet new requirements.

A **design principle** that helps achieve that goal is to **keep interfaces separate from implementations**. For objects, that means that the methods a class provides (implementation) should not depend on how the attributes are represented (interface) and we have to hide the attributes from the implementation.

I we design the interface carefully, **we can change the implementation without changing the interface**, which means that other parts of the program don't have to change.

Code in other parts of the program, meaning outside the class definition, should use *methods* to read and modify the state of the object. They should not access the attributes directly. This principle is called **information hiding**.

1.2 Exercise 17.6

Download the code from this chapter http://thinkpython.com/code/Time2.py Change the attributes of Time to be a single integer representing seconds since midnight. Then **modify the methods and the function** int_to_time to work with the new interface. You should not have to modify the test code in **__main__**. When you are done, the output should be the same as before. Solution: http://thinkpython.com/code/Time2_soln.py

EXAMPLES of using the terms **interface** and **implementation**

- [par.18.3] The built-in function **cmp** has the same *interface* as the method **__cmp__**: it takes two values ...
- [par.18.9] Here's a **program design suggestion**: whenever you override a method, the *interface* of the new method should be the same as the old. It should take the same parameters, return the same type, and obey the same preconditions and postconditions. If you obey / respect this rule, you will find that any function designed to work with an instance of a superclass, like a Deck, will also work with instances of subclasses like a Hand or PokerHand. If you violate this rule, your code will collapse like (sorry) *a house of cards*.
- [par.18.1] One problem with this *implementation* is that it would not be easy to compare cards to see which had a higher rank or suit.

- [par. 17.8] Unfortunately, this *implementation* of addition is not commutative.
- API = Application Programming *Interface*
- GUI = Graphical User *Interface*
- ...

2 Chapter 18. Inheritance

2.1 18.10. Data encapsulation. Refactoring

Chapter 16 demonstrates *a development plan* we might call **object-oriented design**: we identified objects we needed Time, Point or Rectangle and defined classes to represent them. In each case there is an obvious correspondence between the object and some entity in the real world (or at least a mathematical world).

But sometimes it is less obvious what objects you need and how they should interact. In that case you need a different development plan. In the same way that we discovered function interfaces by **encapsulation and generalization**, we can discover class interfaces by **data encapsulation**.

2.1.1 4.8 A development plan (page 37)

A *development plan* is a process for writing programs.

The development plan called **encapsulation and generalization** consists of the next steps for developing functions interfaces: 1. Start by writing a small program with no *function* definitions. 2. Once you get the program working, **encapsulate** it in a *function* and give it a name. 3. **Generalize** the *function* by adding appropriate parameters. 4. Repeat steps 1–3 until you have a set of working *functions*. Copy and paste working code to avoid retyping (and re-debugging). 5. Look for opportunities to improve the program by **refactoring**. For example, if you have similar code in several places, consider *factoring* it into an appropriately general *function*.

A development plan for designing classes, meaning objects and methods, is based on **data encapsulation** and consists of the next steps: 1. Start by writing *functions* that read and write *global variables* (when necessary). 2. Once you get the program working, look for associations between *global variables* and the *functions* that use them. 3. Encapsulate related variables as **attributes** of an object. 4. Transform the associated functions into **methods** of the new **class**.

2.1.2 4.7 Refactoring (page 36)

The process of re-arranging a program to improve function interfaces and facilitate code reuse is called **refactoring**. In the case of section 4.7. (see the examples), we noticed that there was similar code in arc and polygon, so we ***factored it out*** into polyline.

If we had planned ahead, we might have written polyline first and avoided refactoring, but often we don't know enough at the beginning of a project to design all the interfaces.

Once we start coding, we understand the problem better. Sometimes **refactoring** is a sign that you have learned something.

No code is ever right the first (or second) time. **Refactoring** the code once you understand the problem and the design trade-offs more fully helps keep the code maintainable. [https://nsls-ii.github.io/scientific-python-cookiecutter/guiding-design-principles.html]

2.2 Exercise 18.5

Download the code from sectionn 13.8 (http://thinkpython.com/code/markov.py) and follow the steps described above to encapsulate the global variables as attributes of a new class called Markov. Solution: http://thinkpython.com/code/Markov.py

EXAMPLES of using the terms **factoring**, **factorize**, **refactoring**

- to factor = to calculate the prime factors of a number
- te refactor = to (re)arrange the factors / the components into a better arrangement
- code refactoring = the process of *restructuring* existing computer code (changing the factoring) without changing its internal behaviour
- refactoring = rearranging a program to improve function interfaces and facilitate code reuse; if there is similar code in two places, then we **factor it out** into a new code/function
- ...

2.3 SUCCES!!!