BABEŞ-BOLYAI UNIVERSITY
Faculty of Mathematics and Computer Science

Fundamentals of Programming

Lecture 7 – Testing, GRASP patterns

Camelia Chira

Course content

- Introduction in the software development process
- Procedural programming
- Modular programming
- Abstract data types
- Software development principles
- **Testing and debugging**
- Recursion
- Complexity of algorithms
- Search and sorting algorithms
- Backtracking
- Recap

Last time

- Classes
  - Data abstraction
  - Instance attributes vs Class attributes
- UML

Today

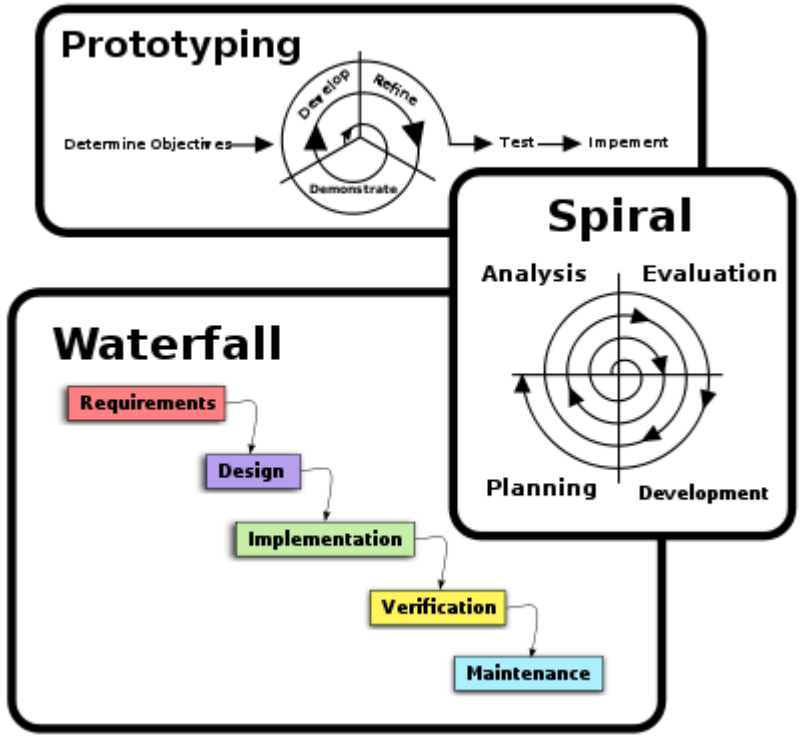
- Testing
  - Concept
  - Testing data
  - Examples
- GRASP

Recap: Testing and debugging

- Separate the code in modules – test and debug them separately
- Document modules and functions
- Debugging the code
  - Identify why a program is not working as expected
  - Study the events that generate an error
  - Use print!
- Testing the code
  - No syntax errors
  - No semantic errors
  - Use assertions
  - Unit testing: validate each unit, test each function separately

Stages in the life of a program (software) Software

- **Requirements definition**
- **Analysis** (decompose the problem in subproblems)
- **Design** (specification of ADT, structure / layers of the application)
- **Implementation**
- **Testing**



Testing

- Testing
  - Observing the behavior of an application for different test data
  - Program verification
  - Continuous activity

- Determine the data for testing (how to choose test data)
  - Exhaustive testing – incomplete testing
  - Black box testing
  - White box testing (or glass box testing)

Why testing

- **"UNTESTED == BROKEN“,** Schlomo Shapiro, EuroPython 2014
- **"... we have as many testers as we have developers. And testers spend all their time testing, and developers spend half their time testing. We're more of a testing, a quality software organization than we're a software organization.“,** Bill Gates (Information Week, May 2002)
- **"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?“,** Brian Kernighan, "The Elements of Programming Style", 2nd edition, chapter 2
- **"Pay attention to zeros. If there is a zero, someone will divide by it.“,** Cem Kaner

https://github.com/krother/python_testing_tutorial

Testing methods: Exhaustive testing

- Exhaustive testing
  - Verify a program for all possible testing data
  - Impossible to apply it in the real world – need to choose a finite number of testing cases
- Incomplete testing
  - Verify a program for some testing data
  - Can be applied in the real world
  - Can use some intuition about natural boundaries of the problem

```
def bigger(a, b):
    """ Given two integer numbers a and b
    Returns True if a > b, False otherwise """
    if a > b:
        return True
    return False
```

  - Random testing
    - more tests increase probability that code is correct

Testing methods: Black box testing

- Choose the testing data based on the specification of algorithms (data, results) without analyzing the code
- Test if the application does what is supposed to do
  - Normal values
  - Boundary conditions on the values e.g empty list, large numbers, etc
  - Error conditions
- Verifies if the application respects the specification
- Designed without looking at the code – can avoid implementation biases

Testing methods: White box testing

- Choose testing cases based on the code of the algorithms
  - Cover all possible execution flows based on the implementation
  - Path-complete: if every potential path through code is tested at least once

- Recommendations
  - `If`
    - Test all parts of conditionals
  - `While, for`
    - Test all cases to exit the loop
    - Loop not entered
    - Loop executed only once or several times

Examples
Black box testing

```
def isPrime(x):
    '''
    checks if a number is prime or not
    Data: x - a positive integer number
    Results: True, if x is prime,
    False if x is composed
    raise ValueError if x < 0
    '''
   if (x < 0):
        raise ValueError("give a positive
         number to be tested...")
   else:
        if (x < 2):
            return False
        else:
            d = 2
            while (d * d <= x):
                if (x % d == 0):
                    return False
                d = d + 1
            return True
```

```
def testIsPrime():
    #black box testing
    #test a prime number
    assert isPrime(5) == True
    #test a composed number
    assert isPrime(15) == False
    #test 0
    assert isPrime(0) == False
    #test a negative number
    try:
        isPrime(-3)
        assert False
    except ValueError as ex:
        print("some errors: " + str(ex))
        assert True
```

```
def testIsPrime():
    for i in range(-100, 1):
        try:
            isPrime(i)
            assert False
        except ValueError:
            pass

    primes = [2, 3, 5, 7, 11, 13, 17, 19]
    for i in range(2, 20):
        assert isPrime(i) == (i in primes), "this is the
value where it fails: " + str(i)
```

Examples
White box testing

```
def isPrime(x):
    '''
    checks if a number is prime or not
    Data: x - a potivie integer number
    Results: True, if x is prime,
    False if x is composed
    raise ValueError if x < 0
    '''
S1  if (x < 0):
S2      raise ValueError("give a positive
          number to be tested...")
S3  else:
S4      if (x < 2):
S5          return False
S6      else:
S7          d = 2
S8          while (d * d <= x):
S9              if (x % d == 0):
S10                 return False
S11             d = d + 1
S12         return True
```

```
def testIsPrime():
    #white box testing (cover all paths)
    # x < 0 => S1, S2
    try:
        isPrime(-5)
        assert False
    except ValueError as ex:
        print("some errors: " + str(ex))
        assert True

    #0 <= x < 2 => S3, S4, S5
    assert isPrime(0) == False
    assert isPrime(1) == False

    #x = 2 or x = 3 => S3, S6, S7, S12
    assert isPrime(2) == True
    assert isPrime(3) == True

    #x = 11 => S3,S6,S7,S8,S11,S8,S11,S12
    assert isPrime(11) == True

    #x = 15 => S3,S6,S7,S8,S11,S8,S9,S10
    assert isPrime(15) == False

testIsPrime()
```

Examples

```
def sumOfEvenValues(l):
    '''
    computes the sum of even values from a list
    Data: l - a list of integers
    Results: sum of even values of l
    '''
    s = 0
    for i in range(0, len(l)):
        if (l[i] % 2 == 0):
            s = s + l[i]
        return s
```

```
def testSumOfEvenValues():
    #empty list
    assert sumOfEvenValues([]) == 0

    #no even value
    assert sumOfEvenValues([5,1,7,3]) == 0

    #one even value
    assert sumOfEvenValues([5,2,7,3]) == 2

    #more even values
    assert sumOfEvenValues([5,2,7,4,6,3])==12

    #all values are even
    assert sumOfEvenValues([4,8,2,6]) == 20

testSumOfEvenValues()
```

Examples

```
def changeElement(l, pos, el):
    '''
    change the pos-th element of list to el
    Data: a list l, a position, a new element
    Results: list l' with l[pos] == el
    raise IndexError if pos < 0 or
    pos >= len(l)
    '''
    if (pos < 0):
        raise IndexError("pos must be positive")
    else:
        if (pos >= len(l)):
            raise IndexError("position must be
                  smaller to the length of list")
        else: #a valid position
            l[pos] = el
            return l
```

```
def testChangeElement():
    #black box testing and white box testing

    #negative position
    try:
        changeElement([1,3,9,2], -2, 5)
        assert False
    except IndexError as ex:
        print("some errors: " + str(ex))

    #position > len(l)
    try:
        changeElement([1,3,9,2], 6, 5)
        assert False
    except IndexError as ex:
        print("some errors: " + str(ex))

    #valid data
    assert changeElement([1,3,9,2],2,5)==[1,3,5,2]

testChangeElement()
```

Black box testing	White box testing
Advantages	
+ Acces to source code is not required	+ Knowing about the code: makes testing it easier
+ Testing can be **reused** if implementation changes	+ Can help find hidden defects
+ Efficient for large code sources	+ Can help optimize code
+ Separates implementer - tester	+ Easier to obtain high test coverage
Drawbacks	
- Test coverage might be low	- Problems with code that is completely missing
- Testing might be inefficient	- Requires access to the code and good knowledge of the code

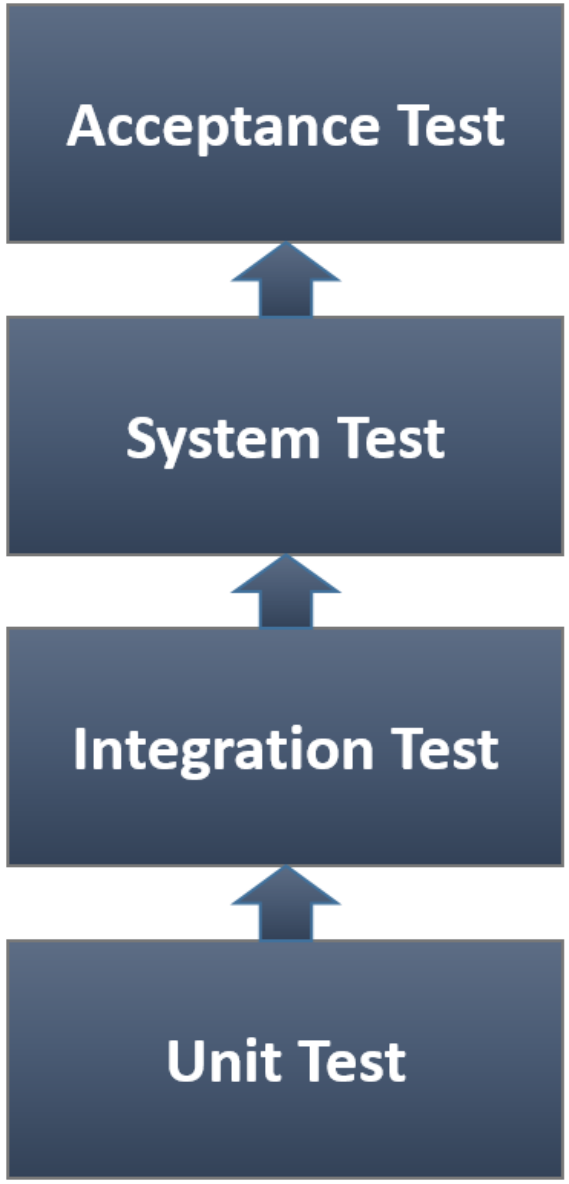
Testing Levels

- Tests are frequently grouped
  - By where they are added in the software development process
  - By the level of specificity of the test

- Testing
  - Manual testing
  - Automated testing
    - Writing software to do testing that would otherwise need to be done manually
    - PyUnit - Python unit testing framework

Testing Levels

- Unit Test
  - Verify the functionality of a specific section of code e.g. a function
  - Test small parts of the program independently
- Integration Test
  - Test different parts of the system in combination
  - Bottom-up approach: based on the results of unit testing
- System Test
  - Tests how the program works as a whole after all modules have been tested
- Acceptance Test
  - Check that the system complies with user requirements and is ready for use



Testing in Python

- Automated Testing
  - The process of writing programs to do testing that would otherwise need to be done manually
  - Use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions
- **unittest**
  - Python framework for writing Unit Tests, Integration Tests and Acceptance Test
  - provides a class **TestCase** and a **main()** method

```
from unittest import TestCase, main OR import unittest
```

Testing in Python: unittest

- Test classes in Python
  - Test classes should extend TestCase and contain at least one method starting with test_
  - Test methods contain assertions (assertEqual, assertTrue, etc)

```
from unittest import TestCase, main

class AdditionTests(TestCase):
    def test_add(self):
        self.assertEqual(add(3, 4), 7)
```

- Running the tests
  - **unittest.main** method looks for all classes derived from TestCase
  - Runs all the tests and reports

```
if __name__ == "__main__":
    main()
```

- Methods setUp() and tearDown() can be used to prepare testing and clean up afterwards

Example

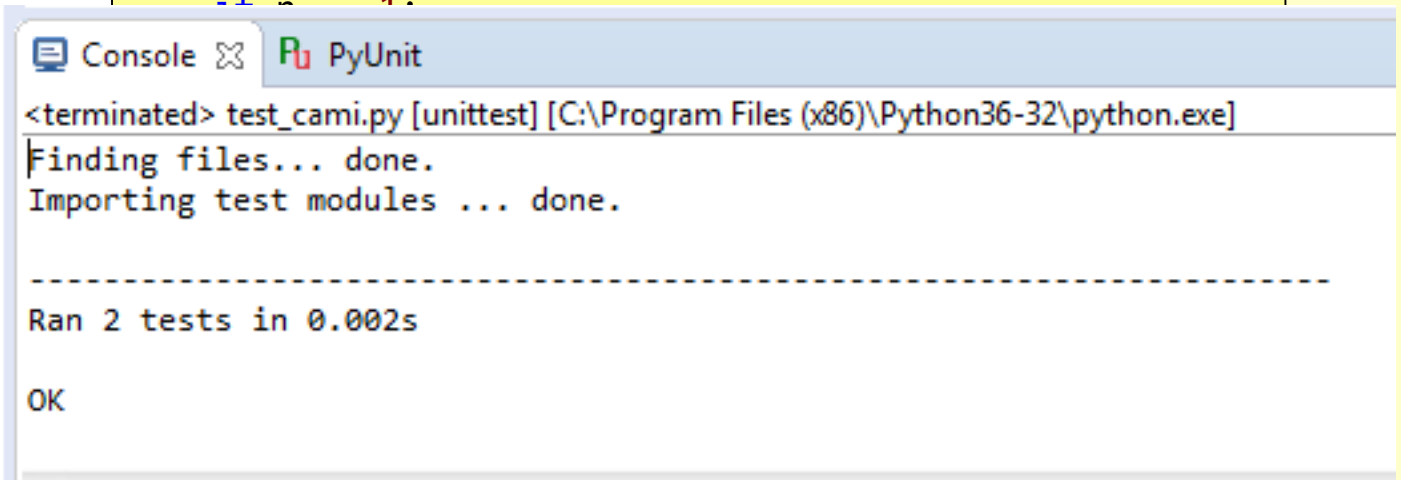
```
def isPrime(n):
    """
    Verify if a number is prime
    Return True if n is prime, False otherwise
    Raise ValueError if n <= 0
    """
    if n <= 0:
        raise ValueError("The number needs to be
    if n == 1:
        return False
    if n <= 3:
        return True
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

```
import unittest
from utils import isPrime

class IsPrimeBlackBoxTest(unittest.TestCase):

    def setUp(self):
        unittest.TestCase.setUp(self)

    def tearDown(self):
        unittest.TestCase.tearDown(self)

    def test_IsPrimeBlackBox(self):
        for i in range(-100, 1):
            try:
                isPrime(i)
                assert False
            except ValueError:
                assert True

        primes = [2, 3, 5, 7, 11, 13, 17, 19]
        for i in range(2, 20):
            self.assertTrue(isPrime(i) == (i in primes),
"The value where it fails: " + str(i))

if __name__ == "__main__":
    unittest.main()
```

Example

```
def isPrime(n):
    """
    Verify if a number is prime
    Return True if n is prime, False otherwise
    Raise ValueError if n <= 0
    """
    if n <= 0:
        raise ValueError("The number needs to be positiv
    if n == 1:
```

```
import unittest
from utils import isPrime

class IsPrimeWhiteBoxTest(unittest.TestCase):
    def setUp(self):
        unittest.TestCase.setUp(self)

    def tearDown(self):
        unittest.TestCase.tearDown(self)

    def test_IsPrimeWhiteBox(self):
        try:
            isPrime(-5)
            assert False
        except ValueError:
            assert True

        self.assertFalse(isPrime(1))
        self.assertTrue(isPrime(2))
        self.assertTrue(isPrime(3), 3)
        self.assertFalse(isPrime(6), 6)
        self.assertTrue(isPrime(7), 7)
        self.assertFalse(isPrime(8), 8)

if __name__ == "__main__":
    unittest.main()
```



Assert methods in unittest.TestCase

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b

Method	Checks that
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b

https://docs.python.org/3/library/unittest.html

Example

domain.Person

```
class Person(object):

    def __init__(self, name, age):
        self.__name = name
        self.__age = age

    def __str__(self):
        return self.__name + " , " + str(sel

    def getName(self):
        return self.__name

    def setName(self, name):
        self.__name = name

    def getAge(self):
        return self.__age

    def setAge(self, age):
        self.__age = age

    def incrementAge(self):
        self.__age += 1
```

test.PersonTest

```
import unittest
from domain.Person import Person

class PersonTest(unittest.TestCase):

    def setUp(self):
        self.person = Person("Simpson", 8)

    def test_IncrementAge(self):
        self.person.setAge(9)
        self.person.incrementAge()
        self.assertEqual(self.person.getAge(), 10)

if __name__ == "__main__":
    unittest.main()
```

Debugging

- When testing indicates the presence of errors -> debugging

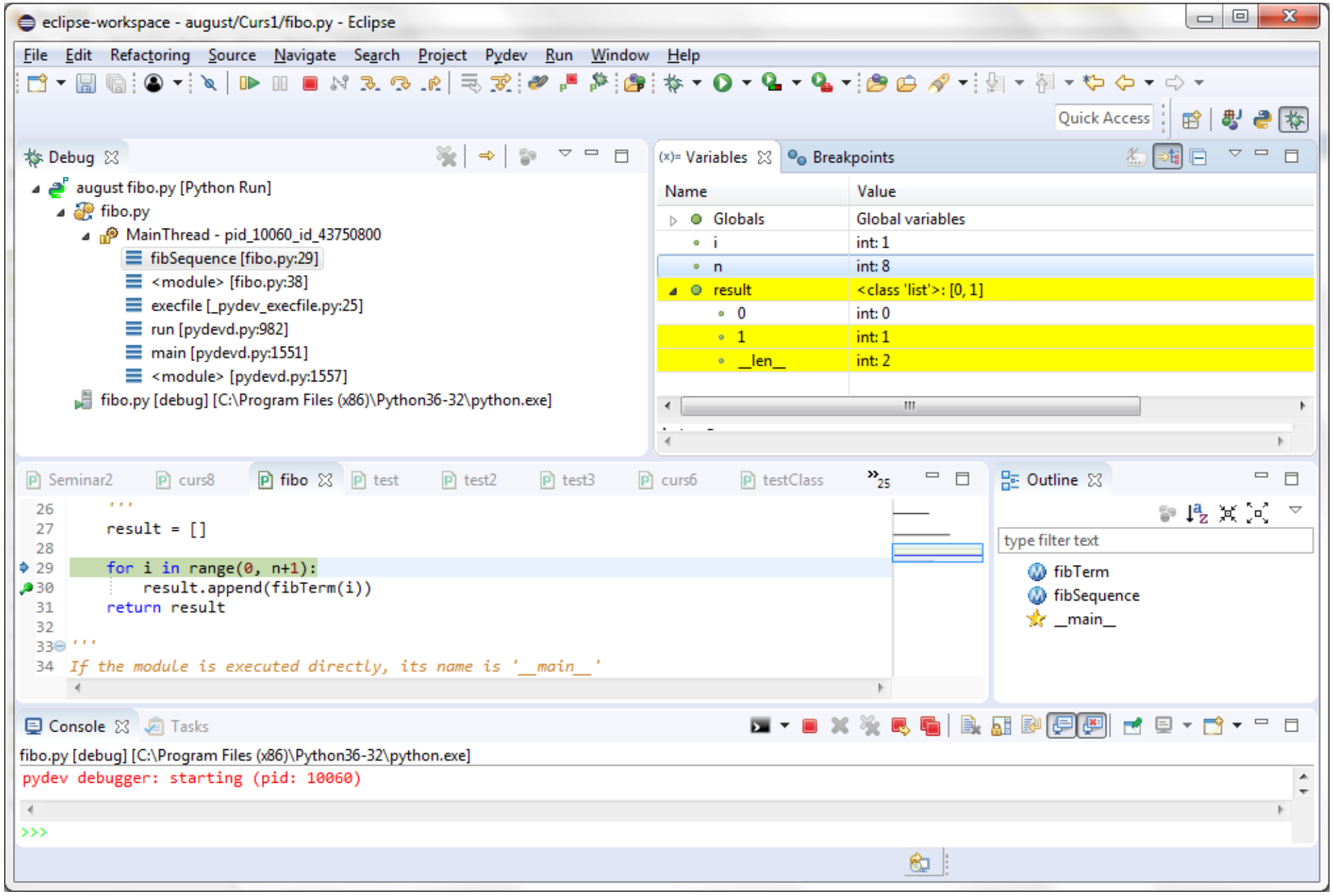
- Debugging the code
  - Identify why a program is not working as expected
  - Study the events that generate an error
  - Use print!

- Rewrite the program with the purpose of eliminating the errors

Debugging in Eclipse

- Debug view
  - View the current execution trace (stack trace)
  - Execute step by step, resume/pause execution

- Variables view
  - View variable values

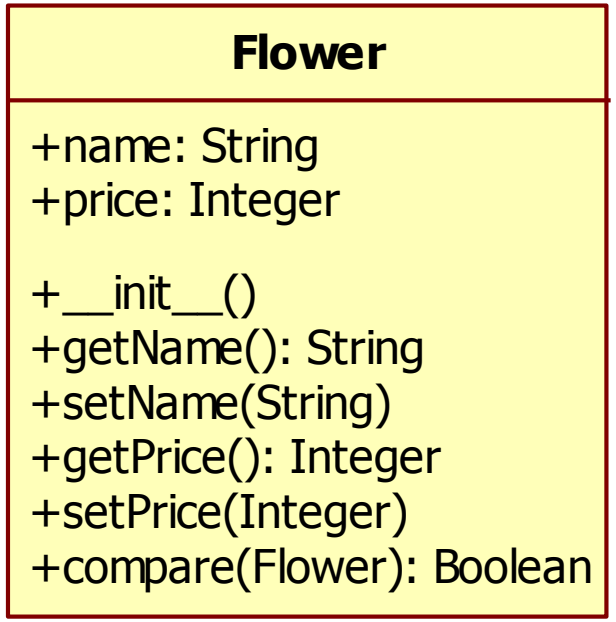

eclipse-workspace - august/Curs1/fibo.py - Eclipse
File Edit Refactoring Source Navigate Search Project Pydev Run Window Help
Quick Access
Debug
august fibo.py [Python Run]
fibo.py
MainThread - pid_10060_id_43750800
fibSequence [fibo.py:29]
<module> [fibo.py:38]
execfile [_pydev_execfile.py:25]
run [pydevd.py:982]
main [pydevd.py:1551]
<module> [pydevd.py:1557]
fibo.py [debug] [C:\Program Files (x86)\Python36-32\python.exe]
(x)= Variables
Breakpoints
Name
Value
Globals
Global variables
i
int: 1
n
int: 8
result
<class 'list'>: [0, 1]
0
int: 0
1
int: 1
__len__
int: 2
Seminar2
curs8
fibo
test
test2
test3
curs6
testClass
26 '''
27 result = []
28
29 for i in range(0, n+1):
30 result.append(fibTerm(i))
31 return result
32
33 '''
34 If the module is executed directly, its name is '__main__'
Outline
type filter text
fibTerm
fibSequence
__main__
Console
Tasks
fibo.py [debug] [C:\Program Files (x86)\Python36-32\python.exe]
pydev debugger: starting (pid: 10060)
>>>

Coding style

- Readability
  - The main attribute of style
  - A good programmer writes code that humans can understand
- Coding style
  - Comments
  - Text formatting (indentation, white spaces)
  - Specification
  - Good names for entities (classes, functions, variables) of the program
  - Meaningful names
  - Use naming conventions

Recap: UML (see Lecture 6)

- Specification of a class




```
class Flower:
    def __init__(self):
        self.name = ""
        self.price = ""

    def getName(self):
        return self.name

    def setName(self, n):
        self.name = n

    def getPrice(self):
        return self.price

    def setPrice(self, p):
        self.__price = p

    def compare(self, other):
        if ((self.name == other.name) and
            (self.price == other.price)):
            return True
        else:
            return False
```

Recap:Design principles (see Lecture 4)

- Managing dependency:
  - Single Responsability
  - Separation of Concerns
  - Low Coupling
  - High Cohesion
- Create software:
  - Easy to understand, modify, maintain, test
  - Classes – data abstraction, encapsulate, hide implementation
  - Easy to test, easy to reuse
- Example: create an application to manage students
  - CRUD operations - **C**reate **R**ead **U**pdate **D**elete
  - Features – *Create student, List students, Find a student, Delete a student*

Recap: Layered Architecture (see Lecture 4)

- Layered architecture
  - Low coupling between modules
    - Modules do not need to know details about other modules
  - High cohesion of each module
    - The elements of a module should be highly related
- Layers:
  - User Interface Layer (presentation layer)
  - Application Layer (service layer or controller layer)
  - Infrastructure Layer (data access, other persistence)
  - Domain Layer (business logic layer)
- Coordinator

GRASP

- General Responsibility Assignment Software Patterns (or Principles)
  - Guidelines for assigning responsibility to classes and objects

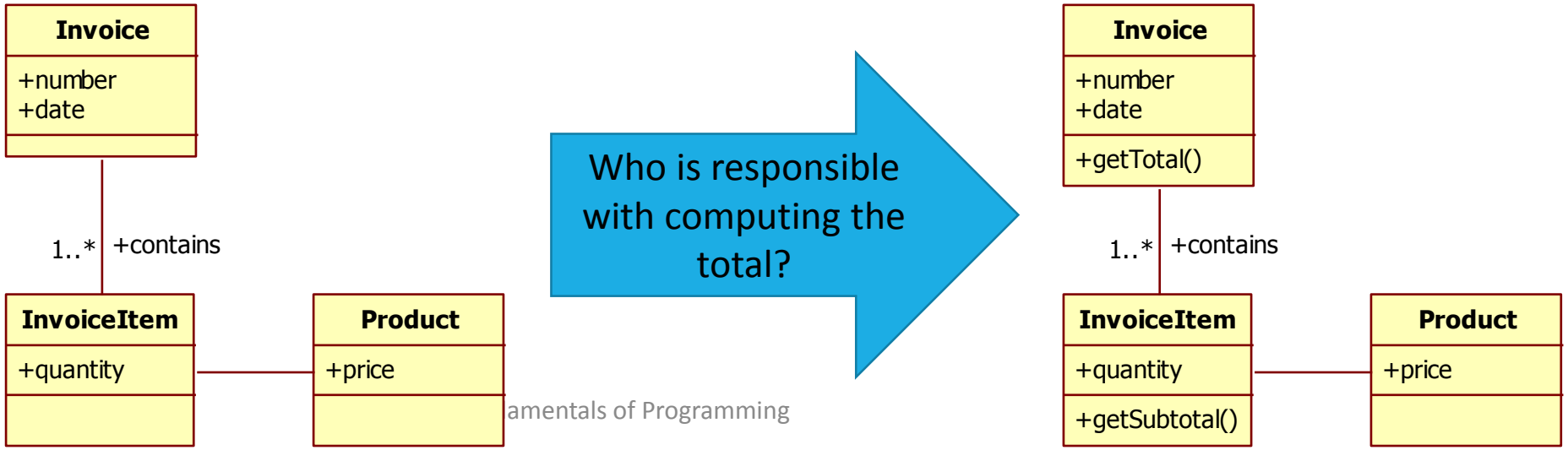
  - High Cohesion
  - Low Coupling
  - Information Expert
  - Creator
  - Pure Fabrication
  - Controller

GRASP: High cohesion, Low couplimg

- High cohesion
  - The responsibilities of a given element are strongly related and highly focused
  - To increase cohesion: break programs into classes and subsystems
  - Low cohesion means that an element has too many unrelated responsibilities => problems: hard to understand, hard to reuse, hard to maintain
- Low coupling
  - Low dependency between classes
  - Low impact in a class of changes in other classes
  - High reuse potential
  - *Examples*
    - Class A has an attribute that is an instance of class B
    - Class A has a method that references an instance of class B in any form

GRASP: Information Expert

- Assign a responsibility to the class that has the information necessary to fulfill the responsibility
- The principle of Information Expert:
  - Look at a given responsibility (e.g. method), determine the information needed to fulfill it, and then determine where that information is stored
  - Place the responsibility on the class with the most information required
- Example



GRASP: Creator

- Which class is responsible for creating objects?
- Creator pattern: responsible for creating an object of the class
  - Class B should be responsible for creating instances of class A if:
    - Instances of B contain instances of A
    - Instances of B closely use instances of A
    - Instances of B have the initializing information for instances of A and can use it for creation
- Example
  - Task 1: Create student
  - Task 2: Store student (Repository)
  - Task 3: Add student (Controller)
  - Task 4: Create UI

Task 1: Create student

```
import unittest
from domain.Student import Student

class StudentTest(unittest.TestCase):

    def test_create(self):
        '''
        Testing Student creation
        '''
        st = Student("Zara", 21, 10)
        self.assertEqual(st.getName(), "Zara")
        self.assertEqual(st.getAge(), 21)
        self.assertEqual(st.getGrade(), 10)

if __name__ == "__main__":
    unittest.main()
```

```
class Student(object):

    def __init__(self, name, age, grade):
        self.__name = name
        self.__age = age
        self.__grade = grade

    def __str__(self):
        return self.__name + " " + str(self.__grade)

    def getName(self):
        return self.__name

    def setName(self, name):
        self.__name = name

    def getAge(self):
        return self.__age

    def setAge(self, age):
        self.__age = age

    def getGrade(self):
        return self.__grade

    def setGrade(self, grade):
        self.__grade = grade
```

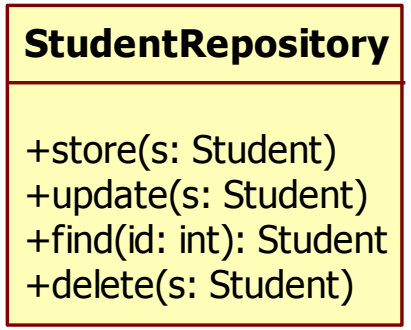
GRASP: Pure Fabrication

- Pure Fabrication - a class added to an application in order to achieve low coupling, high cohesion and reuse
- When an expert violates high cohesion and low coupling => assign a highly cohesive set of responsibilities to an artificial class that does not represent a concept in the problem domain
- Example
  - Task 2: Store student (in memory, file or database)
  - Expert pattern - Student is the "expert" to perform this operation => low cohesion, poor reuse
  - Solution: Pure Fabrication

Task 2: Store student (Create Repository)

- Solution – Pure Fabrication
  - Create a class with the responsibility to store students (StudentRepository)
  - Student class – easy to reuse, high cohesion, low coupling
  - Repository will manage a list of students (persistent storage)

-



Repository

- Represents all objects of a certain type as a conceptual set
- Objects can be added, updated, removed and retrieved from the repository (persistent storage)

Task 2: Store student (Create Repository)

```
def test_storeStudent():
    repo = StudentRepository()
    assert repo.size() == 0

    s1 = Student(1, "Zara", 10)
    repo.store(s1)
    assert repo.size() == 1

    s2 = Student(2, "Erin", 10)
    repo.store(s2)
    assert repo.size() == 2

    s3 = Student(2, "Carla", 10)
    try:
        repo.store(s3)
        assert False
    except:
        assert True
```

```
class StudentRepository(object):

    def __init__(self):
        '''
        Manage a list of Student objects
        '''
        self.__students = {}

    def store(self, student):
        '''
        Stores a student.
        Input: student is of type Student
        Raises an exception if the repository already
        contains a student with the same id.
        '''
        if student.getId() in self.__students:
            raise ValueError("A student with id " +
str(student.getId()) + " already exists.")

        self.__students[student.getId()] = student

    def size(self):
        return len(self.__students)
```

GRASP: Controller

- Controller: the first object beyond the UI layer that receives and coordinates ("controls") a system operation
  - Delegates to other objects the work that needs to be done
  - Coordinates or controls the activity
  - It does not do much work itself
- Controller encapsulate knowledge about the current state of a use case presentation layer decoupled from problem domain
- Example
  - Task 3: Add student
  - Create controller

Task 3: Add student (create controller)

```
def testCreateStudent():
    repo = StudentRepository()

    ctrl = StudentController(repo)

    s = ctrl.createStudent(1, "Zara", 10)
    assert s.getId() == 1
    assert s.getName() == "Zara"

    try:
        s = ctrl.createStudent(1, "Erin", 10)
        assert False
    except:
        assert True
```

```
class StudentController():
    '''
    Controller for CRUD operations on Student list.
    '''

    def __init__(self, repo):
        self.__repo = repo

    def createStudent(self, i, name, grade):
        '''
        Creates a student and stores it in the repository.
        Input: the id of student as int, name of student as
        string, age and grade of student as ints
        Returns a student.
        Raises ValueError if id of student already exists.
        '''
        s = Student(i, name, grade)
        self.__repo.store(s)

    def addStudent(self, s):
        self.__repo.store(s)
```

Task 4: Create UI

```
class StudentUI(object):
    def __init__(self, controller):
         self.__controller = controller

    @staticmethod
    def printMenu():
         ...

    def mainMenu(self):
        while True:
            try:
                StudentUI.printMenu()
                command = input("Enter command: ").strip()
                if command == "0":
                    print("exit...")
                    return
                elif command == "1":
                    s = StudentUI.readStudent()
                    self.__controller.addStudent(s)
                ...
                else:
                    print("Invalid command!")
            except Exception as e:
                print("Error encountered : " + str(e))
```

Coordinator: App start

```
from infrastructure.studentRepo import StudentRepository
from application.studentController import StudentController
from ui.console import StudentUI

from domain.Student import Student

def start():
    #create repository
    repo = StudentRepository()
    repo.store(Student(1, "Erin", 10))
    repo.store(Student(2, "Zara", 10))

    #create controller, provide repository
    controller = StudentController(repo)

    # create UI, provide controller
    ui = StudentUI(controller)
    ui.mainMenu()
```

Recap today

- Testing
  - Concept
  - Testing data
  - Examples
  - Testing in Python: unittest
- Organize a software application
  - Layers
  - GRASP patterns
  - Examples

Next time

- Recursivity

Reading materials and useful links

1. The Python Programming Language - https://www.python.org/
2. The Python Standard Library - https://docs.python.org/3/library/index.html
3. The Python Tutorial - https://docs.python.org/3/tutorial/
4. M. Frentiu, H.F. Pop, Fundamentals of Programming, Cluj University Press, 2006.
5. MIT OpenCourseWare, Introduction to Computer Science and Programming in Python, https://ocw.mit.edu, 2016.
6. K. Beck, Test Driven Development: By Example. Addison-Wesley Longman, 2002. http://en.wikipedia.org/wiki/Test-driven_development
7. M. Fowler, Refactoring. Improving the Design of Existing Code, Addison-Wesley, 1999. http://refactoring.com/catalog/index.html

Bibliography

The content of this course has been prepared using the reading materials from previous slide, different sources from the Internet as well as lectures on Fundamentals of Programming held in previous years by:

- Prof. Dr. Laura Dioşan - www.cs.ubbcluj.ro/~lauras
- Conf. Dr. Istvan Czibula - www.cs.ubbcluj.ro/~istvanc
- Lect. Dr. Andreea Vescan - www.cs.ubbcluj.ro/~avescan
- Lect. Dr. Arthur Molnar - www.cs.ubbcluj.ro/~arthur