Object Oriented Software Design

Introduction

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

September 12, 2011

Outline

Organization of the course

The course is divided into 2 modules, each one entitles to 6 credits

- **OOSD 1** is open to students of:
  - GPIST
  - GPICT
  - IMCNE
  - MAPNET
- **OOSD 2** is open to students of:
  - GPIST
  - Allievi of Scuola Sant'Anna

OOSD 1

First semester

- **Objective:** to acquire basics knowledge of object oriented design and programming
- Abstract Data Types
- Programming in Java
  - Inheritance, vs. composition, when and when not?
  - Exception safety
  - Design for re-use
  - Testing while programming
  - Generics

Exam:

- assignments: Some small assignment during the course
- A final project to be developed during the course and handed over at the end
- An oral examination

OOSD 2

- the C++ programming language
- Design Patterns in C++
  - How to reuse existing knowledge
- Software Engineering
  - Requirement Analysis
  - Classical Software life cycles
  - Agile and Xtreme programming
- Tools
  - The Boost library
  - SVN
  - Tools for testing
  - Documentation
- Concurrent Programming
  - The Boost thread library
  - Distributed Middleware
  - Design Patterns for Distributed Concurrent Programming

Exam:

- A project and an oral examination

Books and material

Java Programming

- Bruce Eckel's "Thinking in Java"
- Sun's documentation and Tutorials

C++ Programming

- Stroustup's "C++ Language" (Language reference)
- Bruce Eckel's "Thinking in C++", Volume 1 (Learning C++)
- Herb Sutter's "Exceptional C++" (Advanced Tips and Tricks)
- Alexandrescu's "Modern C++ programming" (Meta-programming with Templates)

And of course, these slides, and various material to be found over the Internet

Something to think about

"Computer science is no more about computers than astronomy is about telescopes"

– Edsger Dijkstra

What else might it be about?

Algorithms, programs, processes

- Algorithm:
  - It is a logical procedure thought to solve a certain problem
  - It is informally specified as a sequence of elementary steps that an *execution machine* must follow to solve the problem
  - it is not necessarily expressed in a formal programming language!
- Program:
  - It is the implementation of an algorithm in a programming language, that can be executed by an autonomous machine (calculator)
  - It can be executed several times, every time with different inputs
- Process:
  - An instance of a program that, given a set of input values, produces a set of outputs

Algorithm

- Given a *computational* problem, it is necessary to find a *procedure*, consisting of a *finite set of simple steps* that will produce the solution of the problem.
- Such a procedure is called "Algorithm" in honor of arab mathematician Mohammed ibn-Musa al-Khuwarizmi (VIII century AC)

Figure: al-Khuwarizmi

Examples:

- How to prepare a coffe
- How to buy a coffe from the vending machine
- How to calculate the square root of a number

Calculators

- An algorithm needs a machine to execute it
- Machine here is intended in the abstract sense
  - It can even be a human being, or group of people
  - However, it is important that the algorithm it is *described* so that the machine can execute it without further instructions, or wrong interpretation of what to do
  - Therefore, the steps must be simple, and precisely described

Coffe time!

- Example: in the description of the algorithm to prepare a coffe:
  - we must specify how much coffe to put, so that the *machine* cannot be wrong in preparing it
  - If the *machine* is a calculator (a *stupid* machine!), then we must tell it *exactly* how much coffe to put
  - If the machine is *smart*, we can be less precise, for example, put “coffee” until the machine is full
  - (actually, many human beings not not smart enough to prepare a coffe!)

Programs

- In this course, we are interested in describing an algorithm so that a *computer* can understand and execute it
- How to communicate with a computer?

- We need to use a language that the computer can understand
  - A programming language is not so much different than any human language
  - The main difference is that the interpretation of a *sentence* expressed in a programming language must be unambiguous
    - Human languages, instead, allow plenty of ambiguities!
  - Therefore, a programming language is grounded on a solid mathematical basis

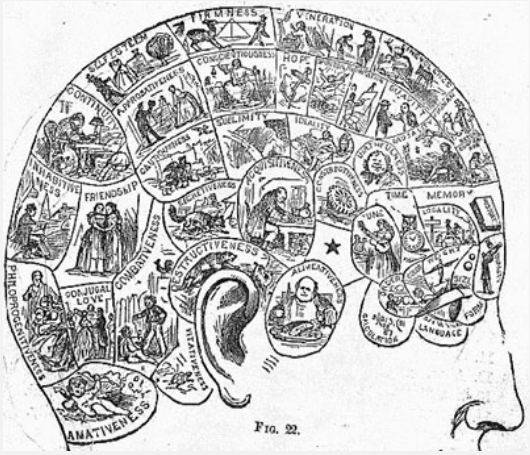
Human beings . . . are very much at the mercy of the particular language which has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality essentially without the use of language and that language is merely an incidental means of solving specific problems of communication and reflection. The fact of the matter is that the "real world" is to a large extent unconsciously built up on the language habits of the group.

The Status Of Linguistics As A Science, 1929, Edward Sapir

Learning a programming language

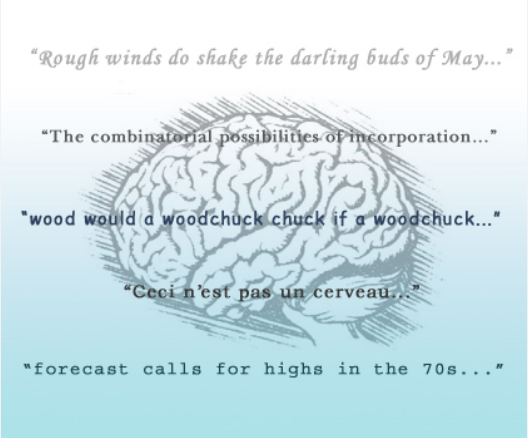
- Learning a programming language is not so different from learning a natural language

- First of all, it is important to point out that the structure of the language influences the way we think at problems
  - It is well known that people from different countries tend to think in different ways
  - Natural language is the primary mental knowledge representation system [a]
  - Therefore, reasoning is heavily influenced by language



[a] See http://en.wikipedia.org/wiki/Cognitive_psychology. Image taken from "Myth, Ritual, and Symbolism" http://bit.ly/cLLmgc

Natural language

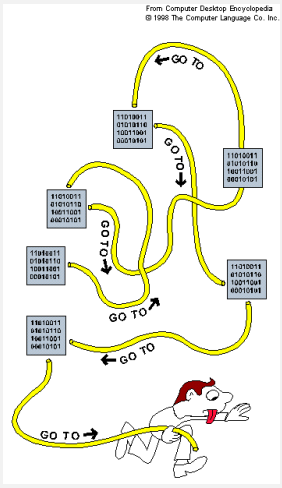


- To be fluent in another language (e.g. English), it is important to *think* sentences in that language
  - Thinking a sentence in the mother-toungue (e.g. Italian) and then mentally translate to the new language (e.g. English) often produces unnatural sentences
  - When speaking in a new language, we should also "switch" to a new way of thinking[1]

[1] Image taken from "Natural Language and the Computer Representation of Knowledge" `http://bit.ly/ccdzLh`

Language structure

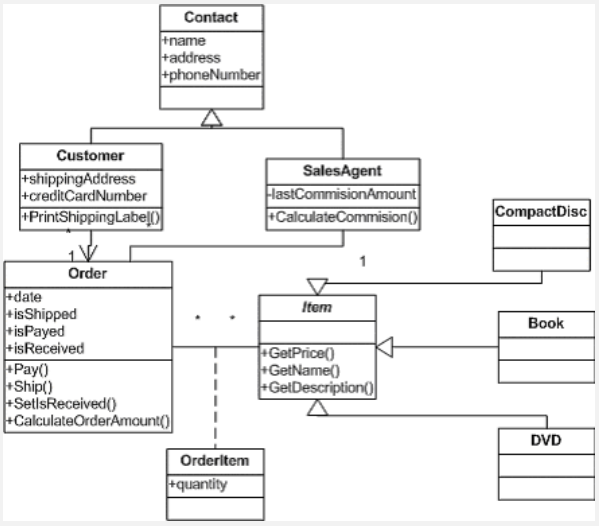
Back to programming languages



- A low level language (MS Visual Basic, C, etc.) lacks strong structure
  - Emphasis on programs as collection of functions acting on global and local data
  - Correspondingly, low-level programs made by beginners tend to lack in structure
  - When they have, it is difficult to find and understand it

Object Oriented Language Structure

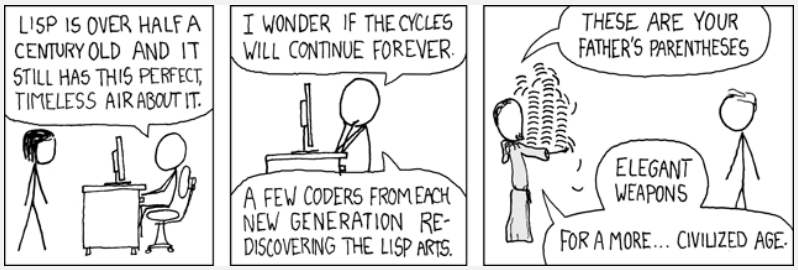
- Object-oriented programming languages favor a structure
  - Emphasis on programs as set of objects communicating through well-defined interfaces
  - Everything is an object!
- The programmers has to model the world as a set of object



Functional programming

Functional programming[2]

- Functional languages impose yet another structure
  - Emphasis on functions as first-class concepts
  - Progams manipulate functions as well as data



[2] Comic strip from `http://xkcd.com`

Learning the basics

Where to start from?

- Similarly to learning natural languages, you should start from learning to read and understand programs written by others
- **To understand**: you should try to *mentally* execute the code, just as a computer would do
  - To be able to do this, you have to understand first how the “abstract machine” executes each part of the program
  - If you learn how to do this, you will not need sophisticated debuggers
- Then, you can learn to write code, keeping in mind how it is executed by the machine
  - That’s the most difficult thing to do, because at the beginning you haven’t the “right structure” in your mind yet

Human interaction

Computer programming has important social aspects

- At the beginning you learn to write small programs all by yourself
  - It’s you and the PC
    - the teacher gives you an exercise, you have to transform the *specification* into a program
    - Or, you want to write a program for yourself, so you also give the specification
  - When you code in this way, you have complete control of all the aspects

Large programs

- However, when you become a professional programmer, it's a completely different story
  - Professional programs get larger and larger (tens of thousands of lines of code), and it takes much longer to write them
  - Even if you are the only programmer, you often lose control of what you have done the week before (*what was the meaning of that* `k` *variable?*)
  - Most often, however, the program code is written by a team
  - Depending on the project, the team can consist also of 5-6 programmers, and there can be many teams working on different parts of the same project
  - It is important to learn how to **work in a group**
- Therefore, it is important to be able to communicate between partners without misunderstandings
- The **team leadership** is a key factor for the success of the project

Specification

How to communicate with customers?

- A professional team writes programs for someone else (*the customer*)
- So, the customers **must tell** then team **exactly** what they want
- That's the **specification**
  - Still done in natural language
  - A very difficult aspect
- Most of the failures in completing/delivering a corrects software project are caused by
  - Wrong or ambiguous initial specification
  - Wrong interpretation of the specification
  - Continuously changing specification after the design

Software engineering

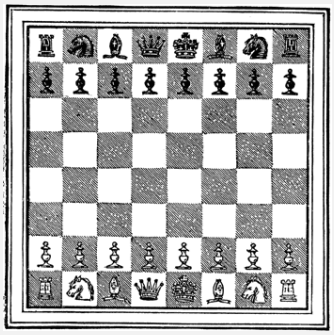
- Software engineering is *also* a social science
  - Team management
  - Project planning
  - Risk management
  - Dealing with the customer
  - Continuosly evolving specification
- All these aspects are as important as learning how to program

How to become a good software designer

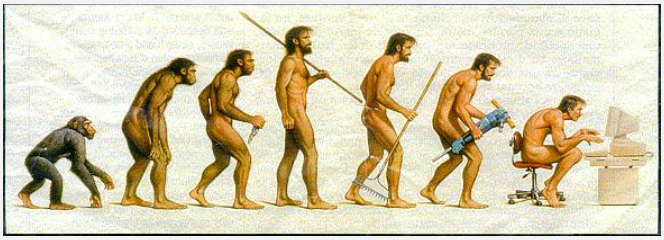
- How to become a software design master?
  - Engineering approach
  - Lot of experience
- Learning to develop good software is similar to learning to play good chess

How to become a chess master

- First, learn the rules
  - e.g., names of pieces, legal movements, captures, board geometry, etc.
- Second, learn the principles
  - e.g., relative value of certain pieces, power of a threat, etc.
  - But principles are abstract. How to apply them in practice?
- Third, learn the patterns by studying games of other masters
  - These games have certain patterns that must be understood, memorized, and applied repeatedly until they become second nature.



To become a software design master



- First, learn the rules
  - e.g., programming languages, data structures, etc.
- Second, learn the principles
  - e.g., software engineering principles such as separation of concerns, etc.
  - But principles are abstract. How to apply them in practice?
- Third, learn the patterns by studying designs of other masters
  - These designs have certain patterns that must be understood, memorized, and applied repeatedly until they become second nature.

This course

This course is a tentative to form “Software Design Masters” in Object Oriented software technology

- A bottom-up approach
  - Object Oriented Languages
  - Programming Techniques
  - Design Techniques
  - An introduction to Software Engineering principles and tools
- This course has been designed to be interactive
  - I need your help to carry on until the end!