Object Oriented Design & Analysis in Designing PLs

By: **Behrooz Nobakht, JellyJ Design Group**

Analysis

1.Domains in Programming Language Development:

Application Domain: Applications of the programming language
Problem Domain: Application of the to-be-developed software

2.Requirement Analysis:

- It should be possible in principle for the program development process to arrive at a program which can be used to solve the given problem.
- The user should be able to use the delivered program to solve most aspects of the problem even if the program is not a complete solution (because 100% solutions are improbable).
- Development progress towards a solution should be predictable.
- Program development should not require excessively many or scarce resources (team training, team size, special-purpose experts, development time, development environment).
- Program use (execution) should not require excessively many or scarce resources (user/operator training, operators, execution time, execution environment).
- Development processes (including maintenance) and delivered programs (including upgrades) improve with repeated use of the language in more and more projects.
- etc ...

3.Program Levels:

Program Application:

- General-Purpose: (PL/I, Algol68)
- Problem-Oriented: (
  System Programming: C, Modula-2, Ada
  Data Processing: Cobol, 4GLs
  Scientific Computing: Fortran
  Simulation: Simula, SmallTalk
  Symbolic Processing: Lisp, Snobol, ML
  …)
- Domain Specific or 4GLs

Application Interface:

- Interactive Processing:
  - Text interaction: Basic, Prolog, Lisp, Logo
  - Graphical Interaction: Visual Basic, SmallTalk, Java
- Real-Time Processing
- Concurrent Processing
  - Ada, SmallTalk, Modula-2, **Java (?)**
- Batch Processing

Program Processor:

- Interpreted programs: Lisp, Prolog, SmallTalk, ML
- Compiled programs: Ada, C, Cobol, Eiffel, Pascal, Fortran
- Virtual Machine:
  - ➔ **Intermediate Languages:** Java
- Machine Oriented

4.Program Representation

- Format-free textual PLs: Ada, C, C++, SmallTalk, Java, …
- Formatted textual PLs: Fortran
- Tabular
- Visual PLs: Icon-based, form-based, diagram-based

5.Architecture

- **Architectural Style:** What is the organization of the SW?
  - Interacting Components (Objects, Agents) ➔ Communication-based Model of Computation (Client/Server, Multi-Agent)
  - Layered (**Stratified Design**):
    - Opaque: Access only adjacent layers
    - Transparent: Access any layer
  - Pipes-and-filters ➔ Data Flow Model & **Sequence Paradigm**
  - Abstract Data Type
  - API-Users
  - Optimizing Alg.
  - Task Spawning
  - Repository
  - Event-Driven
- **Architectural Perspective:**
  - Conceptual Architecture
  - Module Interaction Architecture
  - Code Architecture
  - Execution Architecture
  - **Note:** ***"Many of the reported successes of the surveyed systems resulted from allowing the different***

architectures to develop independently while maintaining the relationship between them. Similarly, some of the problems they reported were a direct result of merging or intermingling these architectures."

6.Programming Environment (Architecture)

- Edit-Compile-Test Programming (C, Pascal)
- Literate Programming
- Interactive Programming (Prolog, ML, SmallTalk (?))
- CASE
- Visual Programming (GUI Builders)
- Generators (4GLs, GUI Builders)
- The management of the SW Artifacts:
  - Each Component in a file
  - Components in libraries
  - Components in a repository
  - With version control
- Programming Reusable Components (Libraries, Frameworks) **vs.** Programming a Ready-to-Be-Used Application

Requirements & Principles

1. Irreducible Requirements

- The Practicality Requirement
  - The Adequacy Requirement:
    - ➔ Problem Domain
    - Ability to express the solutions to all problems to be solved in the program.
  - The Translatability Requirement:
    - There exists a **processor** to translate the programs.
- The Learnability Requirement: There is no point in a language that requires too much learning before understanding enough to achieve anything.
  - **The Programming Languages are for People.**
  - **Brevity**
  - **Simplicity** including **Familiarity** & **Standards**
  - **Understatement Independence**
- The Attractiveness Requirement
  - Good Quality
  - Application-based Attractiveness
  - **Don't talk down to Programmers:**
    - Design for Yourself and Your Friends
    - Give the Programmer as much control as possible

- A language attractiveness also depends on what is "**sexy**" in the targeted programmer community:
  - **Object-Oriented** vs. **Functional**
  - Allowing or excluding **bit manipulation**
  - Being Strongly Typed (Statically, Dynamically, Mixed)
  - Having **verbose** syntax with syntactic **sugar**
  - Having **sugar-free** syntax uniform syntax

- The Productivity Requirement: **Programming Languages are for People**
  - The Correctness Requirement
    - The Error Prevention Requirement
      - **Principle of Locality**
      - **Principle of Lexical Coherence**
      - **Principle of Too Much Flexibility**
    - The Error Detection Requirement
  - The Reusability Requirement
    - *Features:* Abstraction, Naming, Generics
    - *Features:* Inheritance, Delegation mechanisms
    - Portability
    - Factorization
    - Internationalization
    - Module Library
- The Code Management Requirement:
  - The Separability Requirement:
    - Development/Reuse/Maintenance of the program can be split into a team of programmers.

2. Standard Requirements

General Advices:

- **A Program should Be Good for Throwaway Programs.**
- **Programming Languages are for *People***
- **The Human Thought Property**
- **Design for Yourself and Your Friends**
- **Give the Programmer as Much Control as Possible**
- **Aim for Brevity**
- **Admit what Hacking Is**
- **Principle of Frequency**
- **Principle of Locality**
- **Principle of Lexical Coherence**
- **Principle of Distinct Representation**
- **Principle of Too Much Flexibility**
- **Principle of Semantic Power**
- **Simplicity**
  - **KISS**: Keep It Simple Silly
  - The fewer concepts in a language to understand the better.

- Two smaller concepts might be simpler then one more powerful but complicated one.
- **Uniformity:** Rules are **few** and **simple**.
- **Generality:** A small number of general functions provide as special cases of more specialized functions.
- **Familiarity:** Familiar symbols and usages adopted whenever possible.
- **Brevity:** Economy of Expression is sought.
- **Regularity:** The fewer exceptions to the **rule** the better.

- **Consistency:** Similar constructs should look similar.
- **Redundancy:** All error detection is based on redundancy.

3. Orthogonality

Defined for a **construct** in a programming language where has two parts:

- **Semantic Independence**:
  - The semantics of original constructs in the language remain unchanged by the addition of new ones.
- **Compositional Semantics:**
  - If the new construction uses components phrases from the original language, then the semantics of the construction is uniformly defined with respect to the component phrases _ there are no "special cases".
- **Cleanly Integrated Features**
- **Composability of the Features:**
  - Create new Solutions for the problems that would need extra feature.
- **Avoid Special-Purpose Features**
- **Performance Independence**
- **Understatement Independence**

4. Expressiveness (Expressive Power)

5. Evaluating a Programming Language (How To)

Conceptual Modeling Methods Evaluation:

- **Expressibility**
- **Clarity**
- **Semantic Stability**
- **Semantic Relevance**
- **Validation Mechanisms**
- **Abstraction Mechanism**
- **Formal Foundation**

6. Developing Methods for Programming Languages

- **Re-use**
- **Experimental Prototyping**
- **Formalizing**
- **Iterative**
- **Evolving a Programming Language**