Chapter 3 - Language Design Principles

Thought question

- What characteristics should be present in your “perfect language”?

- What have you liked/disliked about Ruby?
The language design problem

- Language design is difficult, and success is hard to predict:
  - Pascal a success, Modula-2 (successor to Pascal – fixed syntactic ambiguity and added modules) a failure
  - Algol60 a success, Algol68 a failure
  - FORTRAN a success, PL/I a failure

- Conflicting advice

Efficiency

- The “first” goal (FORTRAN): execution efficiency.
- Still an important goal in some settings (C++, C).
- Many other criteria can be interpreted from the point of view of a type of efficiency:
  - programming efficiency: writability or expressiveness (ability to express complex processes and structures)
  - maintenance efficiency: readability.
(saw this as a goal for first time in Cobol)
Other kinds of efficiency

- efficiency of execution (optimizable)
- efficiency of translation. Are there features which are extremely difficult to check at compile time (or even run time)? e.g. Algol68 – solved dangling else problem (case/esac if/fi, do/od closing)
- Implementability (cost of writing translator)

What language features do you think effect cost of execution?
Language features that aid efficiency of execution

- Static data types allow efficient allocation and access.
- Manual memory management avoids overhead of "garbage collection".
- Simple semantics allow for simple structure of running programs (class variables, run time inheritance type checking - expensive).
- Expensive operations – array size increase

What language features encourage the production of reliable code?

Can an IDE compensate?

Analogy: word with spell/grammar checking. Make far fewer mistakes.
Reliability and Programming Language?

**Hurts**
- bad syntax source of bugs
  (if $a=b$ when mean if $a==b$)
- complex syntax distracts the programmer (errors are more likely)
- Name hiding - confusion
- machine dependencies (length of Integer could cause different results when ported)

**Helps**
- rigorous type checking
- eliminate undefined variables (always init)
- runtime checks (array bounds, types)
- simple syntax
- less implicit computation (force explicit casts)
- compiler helps (check for declared but not used)
- correct return value types

Internal consistency of a language design: **Regularity**

- Regularity is a measure of how well a language integrates its features, so that there are no unusual restrictions, interactions, or behavior. Easy to remember.
- Regularity issues can often be placed in subcategories:
  - **Generality**: combining closely related constructs into a single more general one. Ex. Ruby: Arrays, stacks, queues – all handled the same.
  - **Orthogonality** (independence): features combined in any combination. Ex. Ruby collections/ranges/arrays/files all have ‘each’
  - **Uniformity**: Do similar things look the same, and do different things look different?
Regularity - examples of deficiencies

- In pascal, procedures can be passed as parameters, but no procedure variable to store them (not a true object).
- Pascal has no way of specifying an array of unknown length (as a parameter).

Orthogonality: independence

- features combined
  - any way
  - all combinations make sense
  - meaning is consistent – regardless of what other features are being used.

Example: I can ask for ‘each’ element of an array, hash, file, range, etc. It always makes sense and does the expected thing.

At seats, from Ruby give examples of orthogonality.
At seats, from C++ give examples of non-orthogonality.
Orthogonality Example

- a car has orthogonal components and controls (e.g. accelerating the vehicle does not influence anything else but the components involved exclusively with the acceleration function). On the other hand, a non-orthogonal design might have its steering influence its braking (e.g. electronic stability control), or its speed tweak its suspension.

Language example

- An instruction set is said to be orthogonal if any instruction can use any register in any addressing mode. This terminology results from considering an instruction as a vector whose components are the instruction fields. One field identifies the registers to be operated upon, and another specifies the addressing mode. An orthogonal instruction set uniquely encodes all combinations of registers and addressing modes.
For examples of **non-orthogonality** consider C++:

- We can convert from integer to float by simply assigning a float to an integer, but not vice versa. (not a question of ability to do – generality, but of the way it is done)
- Arrays are pass by “what appears to be” reference while integers are pass by value.
- A switch statement works with integers, characters, or enumerated types, but not doubles or Strings.

### Orthogonal – independent decisions

<table>
<thead>
<tr>
<th></th>
<th>Sweater</th>
<th>Coat</th>
<th>Shirt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td>S,M,L,XL</td>
<td>S,M,L,XL</td>
<td>S,M,L,XL</td>
</tr>
<tr>
<td><strong>Sleeve Length</strong></td>
<td>Short/Long</td>
<td>Short/Long</td>
<td>Short/Long</td>
</tr>
<tr>
<td><strong>Color</strong></td>
<td>RGB</td>
<td>RGB</td>
<td>RGB</td>
</tr>
</tbody>
</table>
Non-Orthogonal – dependent decisions (if sweater, no choice on sleeve length)

<table>
<thead>
<tr>
<th></th>
<th>Sweater</th>
<th>Coat</th>
<th>Shirt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>S,M,L,XL</td>
<td>M,L,XL (no S)</td>
<td>S,M,L,XL,XXL</td>
</tr>
<tr>
<td>Sleeve Length</td>
<td>Long (no Short)</td>
<td>Long (no Short)</td>
<td>Sleeveless/Short/Long</td>
</tr>
<tr>
<td>Color</td>
<td>RGB Yellow, Purple</td>
<td>RGB</td>
<td>RGB</td>
</tr>
</tbody>
</table>

Orthogonality

a relatively small set of primitive constructs can be combined in a relatively small number of ways. Every possible combination is legal.

Is orthogonal if it can be used without consideration as to how its use will affect something else (no side effects)

Language features are orthogonal when they can be understood in isolation and freely combined with no surprises. Orthogonality thus allows a simple definition to cover lots of possibilities.

Orthogonality makes it easier to understand what happens when things combine.

Is closely related to simplicity - the more orthogonal, the fewer rules to remember.

Can make implementation more complicated as implement things that might not really need.
Orthogonal: other facets

- Performance independence: A feature should be such that its implementation does not impose significant overheads on programs that do not require it. Ex. array element deletion

- Understatement independence: A user need only know about the subset of the language explicitly used to write the program. Ex. reserve word
  
Understatement is a form of speech which contains an expression of less strength than what would be expected.

Thought question

Where does non-orthogonality come from? If it is “bad design”, why not eliminate it?
Other design principles

**Simplicity**: make things as simple as possible, but not simpler (Einstein). (Pascal, C)
- We can make things so simple that it doesn’t work well – no string handling, no reasonable I/O (some languages left out of documentation), no regular expressions
- Can be cumbersome to use or inefficient.

**Expressiveness**: make it possible to express conceptual abstractions directly and simply. (Scheme)
- Helps you to think about the problem. A lot like having a usable notation for a problem.
- Ruby, for example, allows you to return multiple arguments:
  
  ```
  a,b = swap(a,b)
  ```
Other design principles

- **Extensibility**: allow the programmer to extend the language in various ways. Types, operators, create libraries, extend classes. In Ruby, we were able to extend even built-in types.

Other design principles

- **Security**: programs cannot do unexpected damage. (Java)
  - secure communication
  - authentication
  - access control
  - **sandbox** – in which run safely without risk to systems or users
Other design principles

- **Preciseness**: having a language definition that can answer programmers’ and implementers’ questions. If it isn’t clear, there will be differences in implementations.
  
  Example: Declaration in local scope (for loop) unknown/known after exit
  
  Example: implementation of switch statement
  
  Example: constants – expressions or not?
  
  Example: how much accuracy of float?
  
  Example: order of operations

Other design principles (cont.)

- **Machine-independence**: should run the same on any machine. (Java- big effort)

- **Consistent with accepted notations** – easy to learn and understand for experienced programmers (Most languages today, but not Smalltalk & Ruby)
  
  - 42 factorial (sends 42 to function factorial – Smalltalk)
  
  - Symbols, yield, parens optional, use of $1
Wikipedia moment:

- **Syntactic sugar** is a term coined by Peter J. Landin for additions to the syntax of a computer language that do not affect its expressiveness but make it "sweeter" for humans to use.

- Syntactic sugar gives the programmer (designer, in the case of specification computer languages) an alternative way of coding (specifying) that is more practical, either by being more succinct or more like some familiar notation.
  - `operator+(5,7)` or `5+7`

C++ case study

- Thanks to Bjarne Stroustrup, C++ (1983) is not only a great success story, but also the best-documented language development effort in history:
Major C++ design goals

- OO features: class, inheritance
- Strong type checking for better compile-time debugging
- Efficient execution
- Portable
- Easy to implement
- Good interfaces with other tools

Dictionary Moment

polymorphism - providing a single interface to entities of different types.

- virtual functions provide dynamic (run-time) polymorphism through an interface provided by a base class.
- Overloaded functions and templates provide static (compile-time) polymorphism
Supplemental C++ design goals

- C compatibility (but not an absolute goal: no gratuitous incompatibilities)
- Incremental development based on experience.
- No runtime penalty for unused features.
- Multiparadigm- OO and generic programming (templates to get parametric polymorphism or pass in function to another algorithm)
  - dynamic or compile time resolution
- Stronger type checking than C
- Learnable in stages
- Compatibility with other languages and systems

C++ design errors

- Too big?
  - C++ programs can be hard to understand and debug
  - Not easy to implement
  - Defended by Stroustrup: multiparadigm features are worthwhile
- No standard library until late (and even then lacking major features)
  - Stroustrup agrees this has been a major problem
### Review: Language Design

#### Features

- **Efficiency**
  - execution
  - translation
  - implementation
- **Reliability**
- **Readability**
- **Writability**
- **Expressiveness**

- **Regularity**
  - generality,
  - orthogonality
  - uniformity
- **Simplicity**
- **Extensibility**
- **Security**
- **Preciseness**