Language and Library API Design for Usability of Ruby

Usability over Simplicity

Akira Tanaka

National Institute of Advanced Industrial Science and Technology (AIST)

akr@fsij.org

Abstract

The Ruby programming language is designed for easy use. The usability is an important feature since the productivity of programmers depends on it. This paper describes that the design method obtained through the experiences of developing Ruby. The design method can be used to make other languages and libraries easy to use.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Object-oriented languages

General Terms Programming Language, Library, Usability

Keywords Syntax, Library, API, Usability, Simplicity,

1. Introduction

There is no formal way to establish easy-to-use programming languages and library APIs. But the easiness is an important factor for programmers' productivity.

Ruby is the object oriented programming language created by Yukihiro Matsumoto. It is intended to be easy to use. I, Akira Tanaka, feels it has good usability. However it is not simple. It has complex behavior. It is difficult to explain why it is easy.

Basically, the design of Ruby puts importance on good properties of programming languages: succinctness, good naming convention, etc[2]. However, some of Ruby's behaviors violate such good properties to improve the usability. The concrete design method for such behaviors is not well explained.

For example, these two expressions has the different meaning.

- `obj.meth + var`
- `obj.meth +var`

The difference of this example is a space between `+` and `var`. The former has a space, the latter not. In the former, `meth` is a method call without an argument and `+` is a binary operator: `obj.meth() + var`. In the later, `meth` is a method call with an argument and `+` is a unary operator: `obj.meth(+var)`. The space resolves such an ambiguity caused by the fact that the parentheses of a method call is not mandatory.

Here, the semantics depends on white spaces. Such a design is curious form a view point of computer scientists. It is not simple but complex, against tradition, and hard to understand.

But such a curious behavior does realize the usability. By understanding the reason why Ruby adopts it, we can get an insight into the usability of programming languages.

This paper constructed as follows to explain a design method for programming languages and library APIs to be good usability. It is obtained from experiences of Ruby developer (committer).

- Although Ruby is designed to be easy to use, the design method is not explained concretely (Section 2). The explanation will have following benefits.
  - We can make libraries and other languages easy to use as Ruby.
  - We can avoid usability regression when modifying Ruby.
- The design of Ruby focus usability (Section 3). We don't mind complexity of the design if it realize usability. In general, simplicity is good property but it is not the primary goal of Ruby.
- We design Ruby incrementally to improve usability (Section 4). We find flaws of Ruby and fix them. Several issues should be considered for usability:
  - How many frequency of the flaw occur?
  - What the appropriate design according to the frequency?

    - Does the fix prevents other fixes in future?
    - If the fix has an incompatibility, how it should be dealt with?
- We describes future work ((Section 5). We will explain various techniques for usability used in Ruby as design patterns (pattern language). Since the techniques are empirical and sometimes conflict, design patterns should be good format for the explanation. They will accelerate Ruby design process. We also describes possible technology to support incremental design.

2. Usability of Ruby

Ruby is designed to be easy to use programming language. The design policy, such as succinctness, is described briefly in [2, 3] for language itself and [4] for libraries. But the description is not enough to put it into practice. Especially the practice is difficult when usual good properties conflicts usability.

The difficulty causes several problems:

- When we want to realize the usability similar to Ruby in other languages and libraries, it is difficult to determine what behavior should be imitate. We want to ignore the behavior which doesn't contribute to the usability. But it is clear to determine.
- When we want to modify Ruby, it is difficult to consider the modification will degrade the usability or not.

These problems can be solved by understanding how the usability of Ruby is implemented.

The examples follows are explained in following sections.

- optional parenthesis for succinctness and DSL
- blocks for common usages of higher order functions
- shorter names for frequently used methods for succinctness

3. Unusual Design

In this section, we describe the design of Ruby which intend to be easy to use and violates usual language design.

In usual language design, there are several good property: consistency, simplicity, orthogonality, flexibility, succinctness, intuitiveness, DRY (Don't Repeat Yourself), good name, generalness, naturalness, meets programmers' common sense, etc. In the design policy of Ruby, they are also good properties.

However, sometimes Ruby overrides the properties by usability. I.e. the design of Ruby prefer usability over the properties when they conflict. Ruby don't need consistency including rare usage. Ruby don't need succinctness including rare usage. Ruby don't need orthogonality including rare usage. Ruby don't need simplicity including rare usage.

For example, continuation (call/cc) on dynamic typed languages endorse consistency between arguments and return value because it pass former to later. This mismatch, multiple values of arguments v.s. single value of return value, can be solved by that function call can have multiple return values as Scheme. However continuation is rarely used in Ruby, the consistency is not important.

The design of Ruby is not intended to simplify the behavior. Actually the whole behavior including rare usage is complex. Some of the complexity is intentional for usability.

In general, simplicity is a good property. It derives many things from few principles. So programmers don't need to memorize many things except the principles. Another benefit is that simplicity ease programming language research. But Ruby prefer direct usability over such benefits.

In this section, we describe several examples of Ruby's complex design for usability.

3.1 Succinctness over Simplicity

In this section, we explain the example shown in the section 1. The example shows us Ruby depends on a space in a method call. If we want to choose a simple behavior, the following can be considered.

- make the parenthesis of method call mandatory.
- even if the parenthesis is optional, define the behavior regardless of the space.

If the parenthesis is mandatory, the ambiguity of + operator doesn't occur. The + of `obj.meth()+var` is a binary operator. The + of `obj.meth(+var)` is a unary operator. Also, the syntax rules can be reduced because we don't need rules for the parenthesis omitted.

Even if the parenthesis is optional, the behavior regardless of the space simplify the information notification between the tokenizer and the parser.

We didn't choose the simple behavior for Ruby. The reason behind it is succinctness.

There are many situations which don't cause ambiguities even without the parenthesis. The method call is not ambiguous if no arguments are given, only one argument is given and it is a variable, etc. If we require the parenthesis, Ruby loses succinctness for such situations.

3.2 Intuitiveness over Simplicity

The example in the section 1 also show Ruby's design policy which prefer intuitiveness over simplicity. The intuitiveness is for average programmers. Although programmers vary, they have many shared knowledge. For example, there are common textbooks and the programmers can understand pseudo code in the textbooks. Programmers who know application domain can understand the notation used by the domain. So programmers have common intuition in a degree. The detailed reason are follows.

- DSL

  DSL, Domain Specific Language, is a language which correspond to a target domain. It can represent logic in the domain intuitively. DSLs are classified as external DSLs and internal DSLs. An external DSL is an independent programming language. An internal DSL is a library in some programming language. The library provides vocabulary for the domain.

  The parenthesis of method call have an impression of function call. The impression hides the impression of the domain. So the syntax with optional parenthesis appropriate for DSL. It expose the impression of the domain. So programmers easily sense the logic in the domain.

  For example, Ruby has a DSL to manipulate Ruby runtime. The DSL is constructed by methods to load a library, define/remove a constant, define/remove a method, etc. `require` method loads the library `foo` as follows:

  ```
  require 'foo'
  ```

  `require` method is used without parenthesis in general. This reduces the impression of function call and programmers consider this as a declaration. Since the parenthesis is noise in the domain, it increase the cost to read/write/understand the code. Therefore the syntax with optional parenthesis avoid the cost.

- Proximity

  The syntax with optional parenthesis has benefits as above. However it causes the ambiguity. Ruby uses the Gestalt law of proximity to resolve the ambiguity. The law means that near objects are perceived as grouped together. `obj.meth +var` is grouped as `obj.meth` and `+var`. Ruby parses the expression as the perception. So the semantics of the expression is similar to the perception. This reduces the cost to read/write/understand the expression.

- Utility Methods
  The class library of Ruby also prefer usability over simplicity. For example, Array class has `push` and `pop` method. `push` inserts an element at the end of the array. `pop` deletes an element at the end of the array. Since the array size is changed dynamically, programmers can use the array as a stack intuitively.

  Such utility methods tends to be increased because method addition is a major way to introduce a new feature. So the class tends to have more feature and be more complex.

These design decision means that we choose usability over simplicity in Ruby.

3.3 Usage Frequency

The frequency of usage can also be a reason to override simplicity.

For example, a method name should match the following regular expression:

```
[A-Za-z_][0-9A-Za-z_]*[!?]?
```

I.e. it start with an letter or an underscore, followed by zero or more digits, letters and underscores, optionally followed by `!` or `?`.

This syntax is not simple because the last `!` or `?`. If we choose simple syntax, we can consider a syntax without the last character like C or a syntax with various character in any position like Scheme.

This complex syntax is chosen to use the naming practice of Scheme in Ruby. Scheme uses function names which ends with `?` for predicates and `!` for destructive functions. It is just a convention in Scheme because the syntax is not special for the usage. On the other hand, Ruby's syntax is specialized for the usage. This complexity realize the usage in non-S-expression language and prevent too cryptic method names.

`!` is mainly used for destructive methods as Scheme. However Ruby uses `!` only for some of destructive methods. It is not consistent. This is also because usage frequency. Since most Ruby programs are imperative style, there are too many destructive method calls to pay attention. So Ruby uses `!` only for methods valuable to pay attention, such as there are both destructive and non-destructive method and programmers carefully choose them.

The big feature of Ruby, block, is also uses usage frequency. Ruby's block is similar to higher order function in functional languages. For example, `map` can be used as follows in Ruby, Scheme and Haskell.

```
Ruby: [1, 2, 3].map {|x| x * 2 }
Scheme: (map (lambda (x) (* x 2)) '(1 2 3))
Haskell: map (\x -> x * 2) [1, 2, 3]
```

Ruby's `map` is a method of Array class which takes a block. In above example, `{|x| x * 2 }` is a block.

Ruby's block is not an expression. The syntax of block is defined with the syntax of method call. So, a block can be described only with a method call. The block is passed to the method as a hidden argument which is separated from usual arguments. This differs from lambda expression in functional languages. Scheme and Haskell can describe lambda expression as an individual expression. It is passed to `map` function as a usual argument.

This causes following pros and cons.

pro succinct description because it don't need keywords such as lambda.

pro one can terminate the method by break statement in the block.

con a method can take only one block.

Ruby's blocks are limited from higher order functions because only one block can be given for a method. But this is not a big problem because usage frequency. Since it is rare that we need to specify two or more functions, the block's benefits surpass its problem by the limitation.

The library design also utilize the usage frequency. For example, Ruby defines `p` method which is usable anywhere. It prints arguments for debugging purpose which is easy to understand for programmers. The method name, p, is inconsistent with other methods because it is too short in the sense of Ruby naming convention. It is intentional because debug printings are very common. In general, too short names are incomprehensible and tends to conflict. But `p` has no such problem because almost all Ruby programmers knows it.

This kind of naming convention, assigning short names for features frequently used, are called Huffman coding which term is borrowed from data compression.[1]

Huffman coding is applied for writing and reading programs. For writing, shorter and too short names reduces number of types. However too short names, such as p, is can be problematic for reading. So too short names should be used only if it is sure that most programmers have no problem with reading. `p` is an example of such name as explained above. In most case, names can be shorter until single word which can be understand the meaning by programmers.

Ruby uses the frequency of usage for usability. This means Ruby focus major usage and don't focus rare usage. This "focus" is implemented in various levels of Ruby: syntax, semantics and library API.

4. Incremental Design

Ruby is designed to realize the usability using various techniques usability described in section 3. However, we cannot define the complex behavior at once.

Therefore we need incremental design for usability. The design should be refined by feedback. Since we cannot find the best design at beginning, this process is unavoidable. We must find flaws and fix them.

The "flaw" means a bad usability. The process to improve the usability is follows.

- Find flaw of usability
- Design the fix the flaw
- Deal with the incompatibilities

4.1 Find flaw of usability

At first, we must find flaw to refine the design. There are several starting point to find it.

- No feature
- Not enough feature
- Feature is available but not easy to use
- Feature is available but difficult to find it

But we don't provide all features requested in the programming language and the standard library. If the flaw causes a trouble frequently, it is an important problem. If the flaw is difficult to avoid in an application but easy to fix in the programming language and the standard library, it is appropriate to fix by them.

We can estimate the frequency by investigating the similar requests in the past. Also, existing programs can be investigated for a code to avoid the flaw. For example, when we guess a code snippet is an idiom, single method which replace the idiom will improve the usability.

Since Ruby is developed in the bazaar model, any Ruby programmer can find flaws of Ruby. Such flaws are discussed in the mailing lists. Sometimes flaws are found in discussion, so open discussion is useful.

The archive of the mailing lists is useful to investigate the requests in the past. The source code search engines, such as Google Code, is useful to investigate existing programs. We can search idioms and other candidates to improve usability in many programs.

4.2 Design the fix the flaw

In general, there are two or more ways to fix flaw. So we need to design the fix for better usability. Since incompatibilities should be avoided, method addition is a good fix in general. Section 4.3 details about dealing with incompatibilities.

When we add a method, we must define its name and behavior.

The good method name is a name which is easy to understand the behavior. However Huffman coding is applied for methods which is frequently used. So we estimate the frequently of the method.

If the method is frequently used, it should have a short name or define as an operator. Since most programmers knows operators in the language already, operators are easier to adopt. This happens even if programmers doesn't sure precious behavior of the method. They have some expectation on operators and common method names such as `A << B` appends A to B, `A[B]` extract something by B in A, etc.

However the frequency is just an estimate. It can be failure. For example, we tends to assign operators to primitives but primitiveness doesn't mean it is used frequently. If we used a too short name or an operator for a feature, we may have trouble in future. When we find another feature which should be used more frequently, it is difficult to find a name shorter than that. If an operator is used, it is very difficult to find a name easier than the operator. We will need incompatible renaming to preserve Huffman coding.

Therefore short names and operators should be used only if we are certain that the feature is used frequently. If we are not certain, a longer name should be used. It doesn't causes problems in future. We can alias it with a shorter name when we are certain. It doesn't cause incompatibility because longer names are still usable.

The method should be implemented experimentally to examine the behavior.

This examination is easy in Ruby because Ruby's classes are open. It means we can define new methods in the existing classes. For example, we can define `to_proc` method in the builtin class `Symbol` as follows:.

```
class Symbol
  def to_proc
    lambda {|obj, *args|
      obj.send(self, *args)
    }
  end
end
```

The `to_proc` method is an example which is already taken by Ruby. The method is experimented by a third party at first. It is re-implemented in Ruby later. Recent Ruby has the method by default.

The classes can be bigger because we prefer method addition. The big classes are useful to try various methods. If we add a class for new feature, we must create the instance of the class to try the feature.

The method may have two or more names because shorter names are defined later. Although this violates minimalism, Ruby doesn't intend to be minimum. Perl has a slogan TMTOWTDI (There's More Than One Way To Do It). Ruby also has similar nature.

4.3 Deal with the incompatibilities

Improving usability may break compatibility. So, we should consider language and library design without incompatibility in future improvement.

If we change a programming language and a library, it can cause incompatibilities. The incompatibilities break application programs. So they should be avoided if possible.

Various changes can be classified as follows.

- compatible changes
  - new syntax
  - new class
  - new method
  - relax method arguments
  - define undefined behavior
- incompatible changes
  - remove class
  - remove method
  - restrict method arguments
  - change return values
  - change side effects

Strictly speaking, the new methods can also conflicts because applications can add the method by open class. However they are not big problem in practice because we don't use open class extensively. We assume new methods doesn't cause incompatibility here.

Since incompatibility should be avoided, we should choose compatible changes such as method addition.

However several techniques to avoid future incompatibilities in method addition.

- Arguments should be checked strictly. We can add new features by relax the arguments in future.
- Short names and operators should be used only if we are certain to they are used frequently. This reduces a possibility that we cannot find a shorter method name for methods more frequently used in future.
- Describe undefined behavior explicitly in the manual. We can add new features by changing and defining the behavior.

If we really cannot avoid incompatibilities, we can use following practices to reduce pain for application programmers.

- Incompatibilities should be introduced when the major version number is incremented. The programmers can update the application at a time for each major version.
- Warnings should be generated before incompatibilities introduced. The warnings notify that the application doesn't work well in the next major version.

The incompatible change and its warning can be implemented at a time in Ruby. Ruby has two develop branch: stable and development. The warning is implemented to the stable branch. The incompatible change is applied to the development branch. The inconsistency between the warning and the change can be avoided in this style of development. Also, application programmers can try the development version to study the incompatibility.

In Ruby, application can use open class to implement a new method in an older Ruby which don't have the method. For example, `to_proc` method in Symbol class can be implemented for the older Ruby by the compatibility definition as follows. Note that `:foo.respond_to? :to_proc` returns true if the symbol, `:foo`, has `to_proc` method.

```
if !(:foo.respond_to? :to_proc)
  class Symbol
    def to_proc
      lambda {|obj, *args|
        obj.send(self, *args)
      }
    end
  end
end
```

So application can use new methods even in the older Ruby by defining the methods.

The compatibility definitions can be removed when the older Ruby is fade out and the application discontinue support for it. No other code need to be modified at the time.

5. Summary and Future Work

This paper explains Ruby language and library is designed for usability utilize the usage frequency. The incremental design process for the usability is also explained.

However the design principle is not popular even in Ruby community. So, sometimes third party libraries are not easy to use as Ruby.

The incremental design process is not supported well by the implementation. There are ideas for mechanism to support the process.

5.1 Usability of Ruby in Future

It is important to explain the design principle of Ruby to preserve the usability of Ruby.

There are change requests for Ruby which the main reason is simplicity and doesn't focus usability. It is possible to spoil the usability if the request is accepted.

So, it is important to popularize the usability principle. If the principle is popular, the requests which degrade the usability will be decreased.

Currently we work on "language patterns" which are design patterns for designing easy to use languages and libraries. It describes DSL, structure by white spaces, etc.

The format of design patterns is appropriate for this kind of knowledge. It's because the techniques are rules of thumb. Sometimes the techniques conflicts each other. For example, the `p` method is bad name but the name is supported by Huffman coding rule. This knowledge is not possible to formulate as axioms and theorems.

The explanation by the design patterns provides vocabulary to discuss usability of programming languages and libraries.

5.2 Incremental Design in Future

If we can reduce problems by incompatibilities, we can accelerate improvement of the usability of Ruby.

There are several possible mechanisms to reduce the problems.

Since Ruby is dynamic language, most warnings are generated at runtime. Some of the warnings inform the application will be broken with future Ruby. They are only useful when the application is updated, useless otherwise. Since many useless warnings hides real warnings, we can't produce many warnings for incompatibilities. So, it is useful that a mechanism which selects warnings to generate. If the warnings for incompatibilities are not generated in useless cases, we can add many warnings.

The module mechanism can also be improved for treating incompatibilities. Since Ruby has open class, method addition can cause incompatibilities. The incompatibilities can be reduced by name spaces for method names. We are considering the module systems for method names such as selector namespace, difference-based modules[5], classboxes[6], etc. They eases library usability improvement because an old method and new method can coexist even if they have same method name.

Acknowledgments

The author make grateful acknowledgment for the valuable comments from Yukihiro Matsumoto (matz), Kenichi Handa, Mikiko Nnishikimi, Satoru Tomura, Yutaka Niibe (gniibe) and anonymous referees.

References

[1] Larry Wall. Programming is Hard, Let's Go Scripting... December 2007, `http://www.perl.com/lpt/a/997`

[2] Yukihiro Matsumoto. The Power and Philosophy of Ruby. O'Reilly Open Source Software Convention (OSCON), July 2003, `http://www.rubyist.net/~matz/slides/oscon2003/`

[3] Yukihiro Matsumoto. The World of Code (in Japanese). Nikkei BP, May 2009, ISBN 978-4-8222-3431-7

[4] Akira Tanaka. open-uri, Easy-to-Use and Extensible Virtual File System. International Ruby Conference, October 2005, `http://www.a-k-r.org/pub/rubyconf2005-presen.pdf`

[5] Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP), pages 62–88, LNCS 2374, 2002.

[6] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. In Computer Languages, Systems and Structures, Volume 31, Number 3-4, pp. 107-126, May 2005.