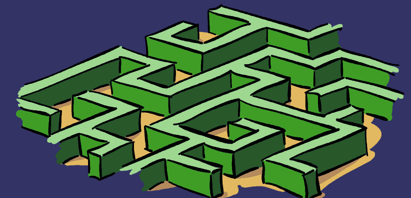
使いやすいライブラリ
API デザイン

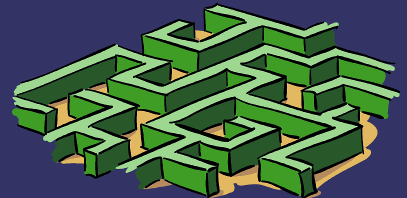
産業技術総合研究所
情報技術研究部門

田中 哲



目的

ユーザの望みを なんとなく かなえてしまう APIを設計する



人間

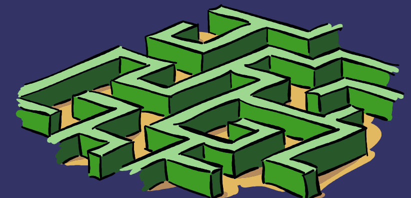
人間は怠惰である

- 人間は短い記述で済ますのが好き
- 人間はものおぼえが悪い

がんばればできるから問題ないという考え方で
デザインされた API は使いにくい
人間の根本的性質に反する

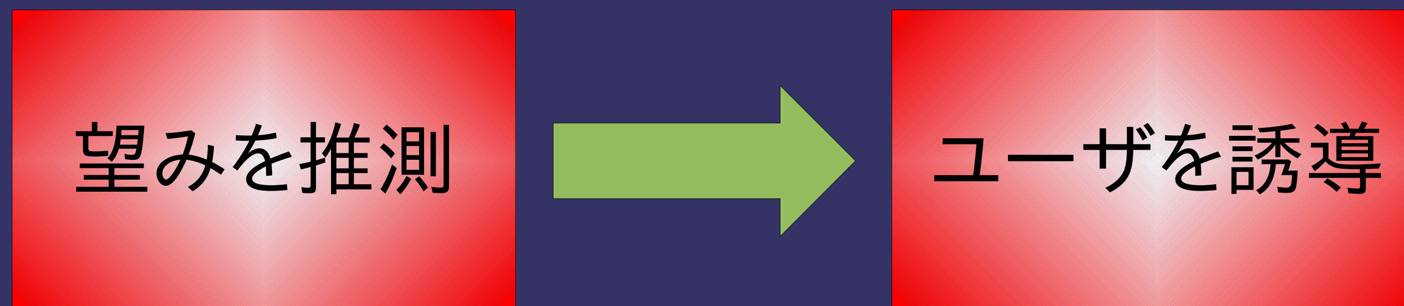
むしろ、怠惰であることを活用してデザインする

怠惰指向設計

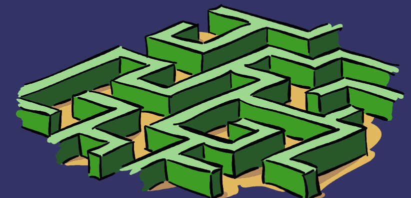


手段

1. ユーザの典型的な望みを推測する
2. (その望みを実現可能な機能を実装する)
3. 望みを実現する機能にユーザを誘導する



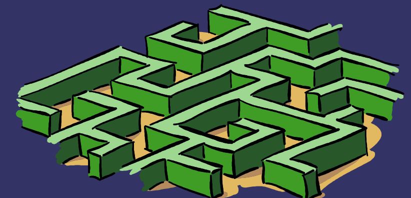
怠惰であることを仮定して推測･誘導



open-uri

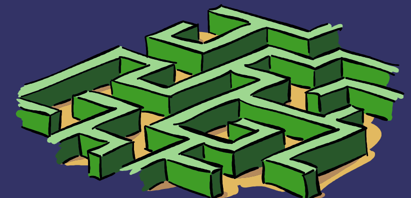
- URI を open できるようにするライブラリ
- 対象は URI 一般だが主に http で使われる
- net/http より簡単にユーザの望みをかなえる

```
require 'open-uri'
open("http://www.ruby-lang.org") {|f|
  print f.read
}
```



どうやって望みをかなえているのか?

- 典型的な望みに API をチューンしてある
  - 簡単なコードで望まれる動作をする
  - 覚えることが少ない



open-uri vs. net/http
API リスト

open-uri

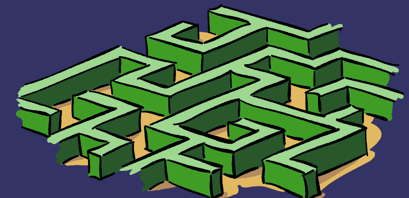
- open(uri) {|f| }
- uri.open {|f| }
- uri.read

機能はだいたい同じ

net/http

- Net::HTTP.get_print
- Net::HTTP.get
- Net::HTTP.start {|h| h.get }
- Net::HTTP.start {|h|
  h.request_get {|r| }}
- h = Net::HTTP.new
  h.start {|h| h.request_get {|r| }}
- h = Net::HTTP.new; h.start {|h|
  q = HTTP::Get.new
  h.request(q) {|r| }}

機能もそれなりに違う
なんでこんなにあるの？

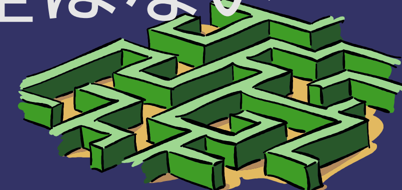


open-uri vs. net/http
http でとってきて表示

- open("http://host") {|f|
  print f.read
  }
- print URI("http://host").read

- Net::HTTP.get_print(
  URI("http://host")
- print Net::HTTP.get(
  URI("http://host")

- open の使いかたはみんな知ってる
- Net::HTTP.get_print を覚えてなんか嬉しい？
- まぁ、このくらいならどっちもたいした差はない



open-uri vs. net/http レスポンスヘッダ取得

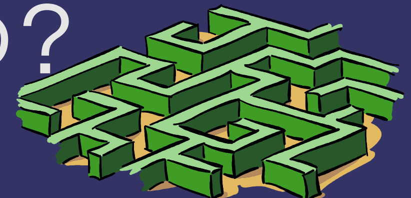
```
open("http://host"){|f|
  p f.content_type
  print f.read
}
```

```
r = Net::HTTP.get_response(
  URI("http://host"))
p r["Content-Type"]
print r.body
```

- content_type とか ["Content-Type"] という指定方法を覚えるのはしかたない
- でも、なんで新しいメソッドを覚えないといけないの？

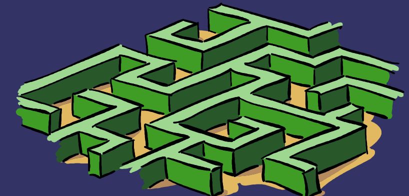
open-uri vs. net/http
リクエストヘッダ指定

- ```
  open("http://host",
    "User-Agent"=>"moz"){|f|
    print f.read
  }
  ```
- ```
  Net::HTTP.start("host") {|h|
    r = h.get("/",
        "User-Agent"=>"moz")
    print r.body
  }
  ```
- "User-Agent"=>"moz" という指定方法を覚えるのはしかたない
- でも、なんで新しいメソッドを覚えないといけないの？
- あと、なんで URI が使えなくなっちゃったの？



open-uri vs. net/http
SSL

- ```
  open("https://host") {|f|
    print f.read
  }
  ```
- ```
  require 'net/https'
  h = Net::HTTP.new("host", 443)
  h.use_ssl = true
  h.ca_file = "/etc/ssl/certs/ca-certificates.crt"
  h.verify_mode = OpenSSL::SSL::VERIFY_PEER
  h.start {
    r = h.get("/")
    print r.body
  }
  ```
- ブラウザなら https ってするだけでアクセスできるのに、なんでそれだけじゃできないの?
- なんでサーバを検証しろってわざわざいわないといけないの? 443って覚えないといけないの？
- またメソッド増えてるし
- ライブラリの名前も違うし

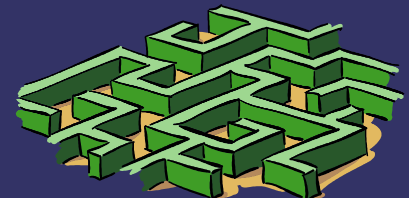


open-uri vs. net/http
プロキシ

- http_proxy 環境変数
- http_proxy=http://proxy:8080/
  export http_proxy

- klass = Net::HTTP::Proxy(
    "proxy", 8080)
  print klass.get(
    URI("http://host"))

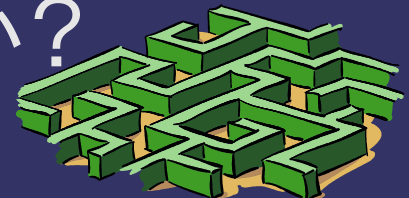
- http_proxy 環境変数ならもう知ってる
- なんで、新しいメソッドを覚えないといけないの？



open-uri vs. net/http
Basic認証

- open("http://host",
  :http_basic_authentication =>
  ["user","pass"]) {|f|
    print f.read
  }
- Net::HTTP.start("host") {|h|
    q = Net::HTTP::Get.new("/")
    q.basic_auth "user", "pass"
    r = h.request(q)
    print r.body
  }

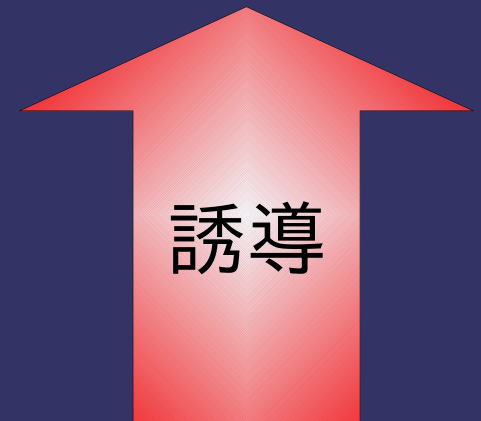
- user, pass の指定方法を覚えないといけないのはしかたない
- でも、なんで新しいクラスを覚えないといけないの？
- なんで新しいメソッドを覚えないといけない？



open-uri の狙い

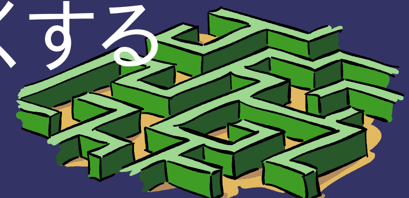
- ユーザの典型的な望み
  - とりあえずとってくる
  - レスポンスヘッダ
  - リクエストヘッダ
  - SSL
  - プロクシ
  - Basic認証

使いやすくする
あまり覚えなくても
使えるようにする



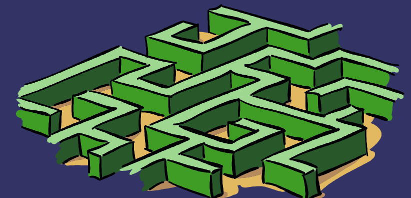
- あまり典型的じゃない望み
  - インクリメンタルな処理
  - パーシステント接続
- だれも望まないこと
  - 落し穴

使いにくくする
もしくは
使えなくする



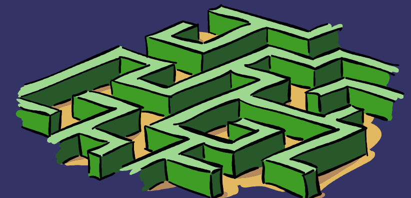
使いやすい *API*

- ユーザの望みをかなえる
  - 直接的に要求されたことだけでなく、
ユーザが意識していない将来的な望みも含めて
- 同じことをやるのに覚えることが少ない
  - とくに普段よく行う使いかたについて



具体的にはどうデザインするか？

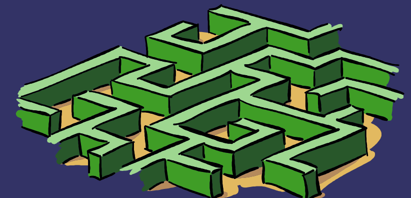
- ユーザの望みを推測する
- ユーザを誘導する
- フィードバックで改善する



ユーザの望みを推測する

- 典型的な望みもあれば、稀な望みもある
- 典型的なものを調べる

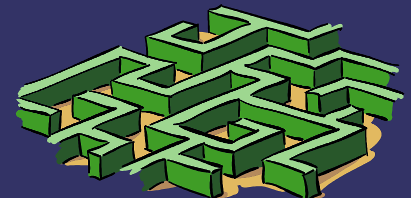
- 対象領域
- メタファ
- 常識
- 一貫性



対象領域

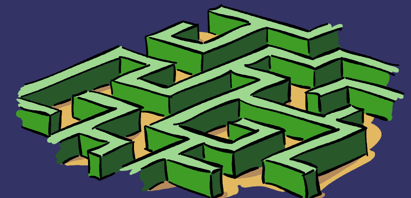
- 対象領域を知ればユーザの望みはある程度わかる
- 自発的にライブラリを作ろうと思い立つ人は自分自身の望みを持っているはず
  初心忘れるべからず
  規格を調べるうちに忘れてしまうことが多い

- http ならとりあえずデータをとってくる
- SSL ではサーバを検証する



メタファ

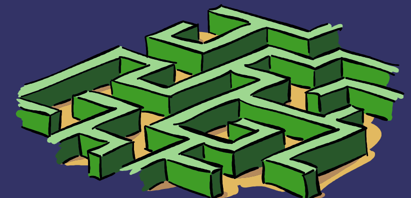
- メタファがあるとユーザの期待をライブラリの作者が予想可能
- メタファがあれば覚えやすく思いだしやすい
- open-uri のメタファ:
  URI でアクセスできるファイルシステム
  - open できる: はい、URI を open できます
  - 読み込める: はい、普通の IO のように読み込めます
  - seek できる: はい、それもできます
  - 書き込める:
    すいません、それはまだ作ってません
  - ディレクトリをたどれる:
    すいません、それもまだです



常識

- ユーザに常識があれば、ユーザの望みをライブラリの作者が予想できる
- もともと知っていることは新しく覚える必要がない
- 環境の知識
  - Ruby の組み込みメソッド (とくに良く使われてるもの)
  - Unix
  - 標準: POSIX, RFC など

- each で繰り返し
- openメソッドの挙動
- http_proxy環境変数

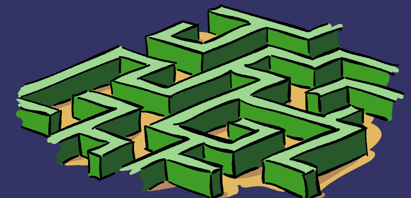


一貫性

- 一貫性があれば類推できる
- ユーザの類推をライブラリの作者が予想可能
- 類推できればうろ覚えでも済む

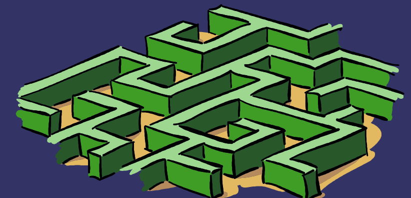
- 一貫性を求めすぎると不幸を生む
  対象領域の都合に比べれば一貫性は些細な話

- open と URI#open はほぼ同等の機能を提供する



ユーザを誘導する

- 怠惰であることを利用する
  - 知っていることで済ます
  - 短いコードを好む
- 普段使うメソッドやクラスを減らす
- 設定無しが良い設定
- ハフマンコード
- DRY: Don't Repeat Yourself

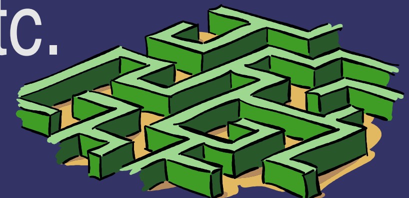


普段使うメソッドやクラスを減らす

- 少なければ最初に適切なものが選ばれる可能性があがる
- 適切なものに誘導する
- 少なければ覚えることも少ない

- ただし、ミニマリズムに陥ってはならない
- 「普段使う」ものを少なくする
- 「稀に使う」ものは多くてもよい

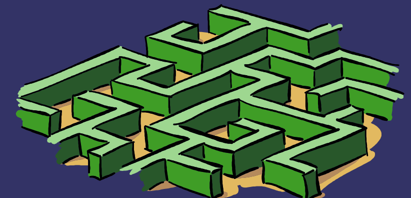
- open vs. Net::HTTP.get, Net::HTTP#get, etc.



設定無しが良い設定

- 書かなくてよいのであれば覚える必要もない
- プログラマが書かないで動かし始め、そのまま最後まで書かなくて済むのが一番幸せ
- 書かないように誘導する

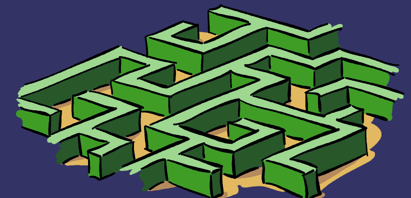
- SSL 認証局証明書



ハフマンコード

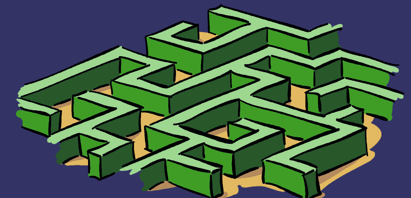
- よく使うものに短い表現を割り当てる
- あまり使わないものに長い表現を割り当てる
- 短いからよく使う
- 長いからあまり使わない
- 短い表現は覚えやすい (暗号的でなければ)
- ユーザの望みどおりに動作する API にユーザを誘導する

- p : あまりによく使う。名前としては非常識
- File.open(n) {|f| f.read } より File.read(n)



ハフマンコードの由来

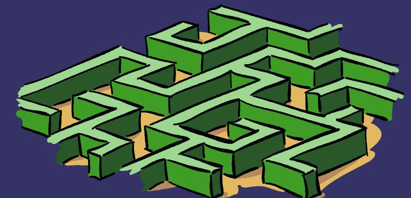
- データ圧縮
  頻出する文字に短いビット列を割り当て、あまり出現しない文字に長いビット列を割り当てる
- perl
  よく使う用法に短い表現をわりあてる



DRY: Don't Repeat Yourself
繰り返して書くな

- 必要なら1回書くのはしかたない
- でも、2回以上書くのは無駄
- どこにどうやって書くか覚えないといけない
- 矛盾した記述が可能という落し穴になる
- DRYにより落し穴を回避

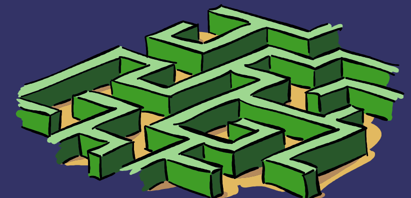
- ConditionVariable#wait(mutex)
- def initialize(foo) @foo = foo end



フィードバックで改善する

- はじめから完璧なデザインを行うことはできない
- 推測を失敗しているかもしれない
- 誘導できていないかもしれない
- フィードバックを受けて改善していく

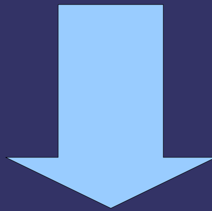
- イディオム
- 繰り返される驚き
- 覚えにくい事項



イディオム

- 典型的な望みを一発で満たすメソッドがない
  ⇒短いコンビニエンスメソッドを追加する

File.open(n) {|f| f.read }

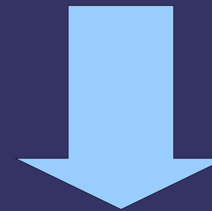


File.read の追加

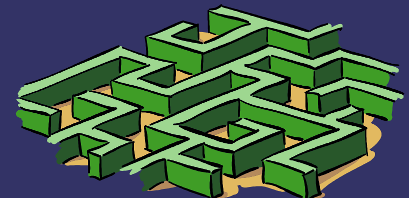
File.read(n)

/foo(?:#{re.source})bar/

Regexp#to_s の追加



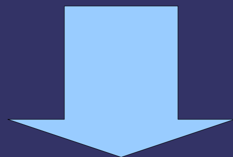
/foo#{re}bar/



繰り返される驚き

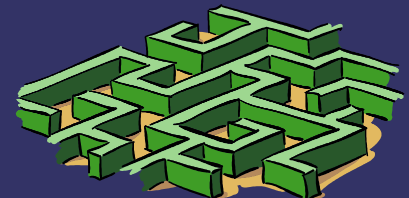
- 典型的な望みとは異なる挙動になっている
  - 望みを満たす新しい短いメソッドを追加
    - Iconv.iconv ってなんで配列を返すの? ⇒Iconv.conv追加
  - 変える (互換性に注意)
    - fork でスレッドが親子両方にできて困る ⇒ 子では殺す
- あるはずの機能が見つからない
  - 見つかる所につける

Iconv.iconv("EUC-JP", "UTF-8", "abc") ⇒ ["abc"]



望ましい挙動の短いメソッドを追加

Iconv.conv("EUC-JP", "UTF-8", "abc") ⇒ "abc"

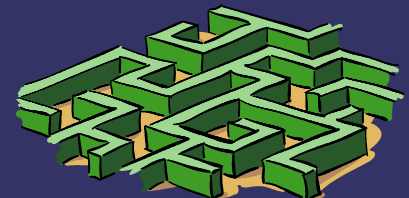


覚えにくい事項

- 典型的な望みの実現法が自然でない
  - より自然で短い方法を考えて方法を導入する (RubyUnit)

```
require 'runit/testcase'
require 'runit/cui/testrunner'
class TestC < RUNIT::TestCase
  def test_foo
    ...
  end
end
RUNIT::CUI::TestRunner.run(TestC.suite)
```

test/unit では不要

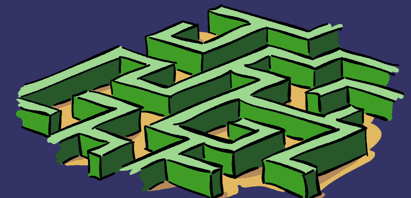


フィードバックのための事前の備え

- 短いメソッドを追加することが必要
- 既存のが短すぎると不可能

- 最初は長いメソッドを使う
- とくにプリミティブは長くしておく
- オペレータは熟慮に熟慮を重ねた上で使う

- 不幸な例: CGI#[]

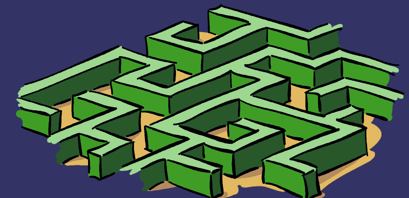


例題: *digest*の問題

- feature名とクラス名の非一貫性
  - digest/md5　Digest::MD5
  - digest/sha1　Digest::SHA1
  - digest/sha2　Digest::SHA256
  - digest/sha2　Digest::SHA512
- ファイルをダイジェストするメソッドがない
  - たまに話題があると次のようなコードが出る
    Digest::SHA256.hexdigest(File.read(name))
  - 大きなファイルだとメモリを喰い尽くして危険

覚えにくい事項
非一貫性

(不幸を呼ぶ)イディオム



digest
*feature*名とクラス名の非一貫性

- 案1: digest/sha256 とか digest/sha512 を導入する

  digest/sha2 のほうが短いのに誰が使うというのだ?

  digest/sha2 から digest/sha256 へは誘導できない

- 案2: 全部 digest だけで済むようにする
  - 問題: 全部読み込むのは無駄
  - autoload で解決可能
  - 提案 [ruby-dev:28689]
  - メンテナの反応がない

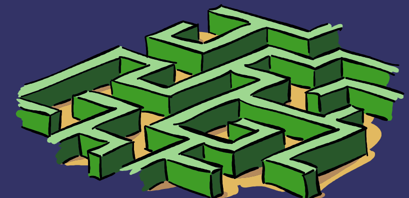
digest/sha2 から
digest へは誘導できる

digest
ファイルをダイジェストするメソッド

- 繰り返し ML で話題になっている
- 危険なコードがよく出る

```
Digest::SHA256.hexdigest(File.read(name))
```

(不幸を呼ぶ)イディオム



digest
ファイルをダイジェストするメソッド

- 不幸を呼ぶイディオム
  - Digest::SHA256.hexdigest(File.read(name))
- 探すといくつか案がある
  - マニュアル:
    Digest::SHA256.open(name).hexdigest

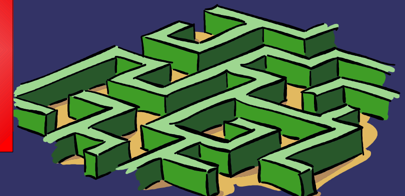
    openしたらcloseするという知識に反する

  - [ruby-talk:116637] mneumann:
    Digest::SHA256.from_io(open(name)).hexdigest

    長すぎて誘導不能

  - [ruby-talk:184826] ara.t.howard:
    File.sha256(name).hexdigest

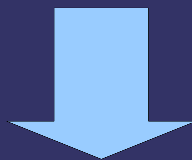
    Fileにあることを覚える必要有



digest
ファイルをダイジェストするメソッド

- Digest::SHA256.file(name).hexdigest はどうか
  - Digest::SHA256.hexdigest(File.read(name)) より短い
  - 既存の知識に反しない
    - close 不要の open を導入しない

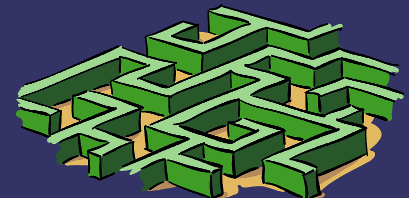
Digest::SHA256.hexdigest(File.read(name))



メソッド追加で誘導

Digest::SHA256.file(name).hexdigest

- 他の案: Digest::SHA256.hexdigest_file(name)



まとめ

- 使いやすいライブラリのデザインのしかた
- ユーザの前提知識と新しい知識を考える
  - ユーザの望みを推測する
    - 対象領域
    - メタファ
    - 常識
    - 一貫性
  - ユーザを誘導する
    - 普段使うメソッドやクラスを減らす
    - 設定無しが良い設定
    - ハフマンコード
    - DRY: Don't Repeat Yourself
  - フィードバックで改善する
    - イディオム
    - 繰り返される驚き
    - 覚えにくい事項