GoF patterns in Ruby

Matthieu Tanguay-Carel - 2007

Creational Patterns

- Abstract Factory
- Builder
- Factory Method # can be deduced from Abstract Factory
- Prototype
- Singleton # available in standard lib (doc in singleton.rb)

Structural Patterns

- Adapter
- Bridge # can be deduced from Abstract Factory
- Composite
- Decorator
- Facade # boring and trivial
- Flyweight
- Proxy

Behavioral Patterns

- Chain of Responsibility
- Command
- Interpreter # skipped
- Iterator # built-in (module Enumerable)
- Mediator # skipped
- Memento
- Observer # built-in (doc in observer.rb)
- State # nice implementation by maurice codik
- Strategy
- Template Method # the simplest is the block yielded to
- Visitor

```ruby
#
# The GoF Abstract Factory pattern
# written by Matthieu Tanguay-Carel
#
# Factories behave in effect like singletons.
# Extra functionality can be tested for with "Object#respond_to? :extra"
# if needed (See GTKFactory).
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

module MyAbstractFactory
    def create_button
        raise NotImplementedError, "You should implement this method"
    end
end

class Win95Factory
    include MyAbstractFactory
    def create_button
        puts "I'm Win95"
        "win95button"
    end
end

class MotifFactory
    include MyAbstractFactory
    def create_button
        puts "I'm Motif"
        "motifbutton"
    end
end

class GTKFactory
    include MyAbstractFactory
    def create_button
        puts "I'm GTK"
        "gtkbutton"
    end

    def extra
        puts "I'm enhanced"
    end
end

class LookAndFeelManager

    @@types2classes = {
        :motif => [MotifFactory,nil],
        :gtk   => [GTKFactory,nil],
        :win95 => [Win95Factory,nil]
    }

    def self.create type
        if !@@types2classes.include? type
            raise NotImplementedError, "I know nothing about type: #{type}"
        end
```

```
        factory_and_instance = @@types2classes[type]

        if factory_and_instance[1].nil?
            puts 'instantiating new factory'
            factory_and_instance[1] = factory_and_instance[0].new #mutex this
        else
            puts 'returning already instantiated factory'
            factory_and_instance[1]
        end
    end
end

if __FILE__ == $0
    factory = LookAndFeelManager.create :gtk
    puts factory.create_button
    factory.extra if factory.respond_to? :extra
end
```

Output

```
instantiating new factory
I'm GTK
gtkbutton
I'm enhanced
```

```ruby
#
# The GoF Builder pattern
# written by Matthieu Tanguay-Carel
#
# The Director class declares the creation process.
# The Builder classes are the concrete builders.
# The builders are free to implement a method or not, and can be
# customised at will by the client.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class Director
    def initialize
        @process = [:create_header, :create_body, :create_footer]
    end
    def build builder
        @process.inject("") {|acc, method|
            acc += builder.send method if builder.respond_to? method
            acc
        }
    end
end

class HTMLBuilder
    def initialize title
        @title = title
    end

    def create_header
        "<html><title>#{@title}</title>"
    end

    def create_body
        "<body>fig leave</body>"
    end

    def create_footer
        "</html>"
    end
end

class XMLBuilder
    def create_header
        "<?xml version='1.0' charset='utf-8'?>"
    end

    def create_body
        "<root>welcome</root>"
    end
end

if __FILE__ == $0
    director = Director.new
    html_builder = HTMLBuilder.new 'xml sucks'
    puts director.build(html_builder)
```

```
    xml_builder = XMLBuilder.new
    puts director.build(xml_builder)
end
```

Output

```
<html><title>xml sucks</title><body>fig leave</body></html>
<?xml version='1.0' charset='utf-8'?><root>welcome</root>
```

```ruby
#
# The GoF Prototype pattern
# written by Matthieu Tanguay-Carel
#
# The Note and Clef classes are the prototypes.
# The deep copy used here will not work if the instances
# have singleton methods (the ruby meaning of singleton).
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class PrototypeManager
    def initialize
        @prototypes = {}
    end

    def register key, prototype
        raise IndexError, "a prototype is already \
            assigned to this key: #{key}" if @prototypes.include? key
        @prototypes[key] = prototype
    end

    def unregister key
        raise IndexError, "this key is not \
            registered: #{key}" if !@prototypes.include? key
        @prototypes.delete key
    end

    def get key
        @prototypes[key].deep_copy
    end
end

class Object
    def deep_copy
        Marshal.load(Marshal.dump(self))
    end
end

class Note
    attr_accessor :duration
    def initialize duration
        @duration = duration
    end
end

class Clef
    attr_accessor :type
    def initialize type
        @type = type
    end
end

if __FILE__ == $0
    manager = PrototypeManager.new
    manager.register(:half_note, Note.new(2))
    manager.register(:full_note, Note.new(4))
```

```ruby
    manager.register(:treble, Clef.new("high pitch"))
    manager.register(:bass, Clef.new("low pitch"))

    clef = manager.get :bass
    puts "clef's type: #{clef.type}"
    note = manager.get :half_note
    puts "note's duration: #{note.duration}"
    note.duration = 6
    puts "note's duration: #{note.duration}"
    other_note = manager.get :half_note
    puts "note's duration: #{other_note.duration}"
end
```

Output

```
clef's type: low pitch
note's duration: 2
note's duration: 6
note's duration: 2
```

```ruby
#
# The GoF Adapter pattern
# written by Matthieu Tanguay-Carel
#
# The Adapter offers exactly the same interface as the adaptee, but it can
# override any method or add new ones.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class Adaptee
    def talk
        puts "I'm Adaptee"
    end
    def whine
        puts "Stop bullying me!"
    end
end

class Adapter
    def initialize
        @adaptee = Adaptee.new
    end

    def talk #override
        puts "Let me introduce you to Adaptee!"
        @adaptee.talk
        puts "That was my adaptee"
    end

    def do_other_stuff
        puts "I'm versatile"
    end

    def method_missing method
        if @adaptee.respond_to? method
            @adaptee.send method
        else
            raise NotImplementedError, "This method is not " + \
                "available on this interface"
        end
    end
end

if __FILE__ == $0
    adapter = Adapter.new
    adapter.talk
    adapter.whine
    adapter.do_other_stuff
end
```

Output

```
Let me introduce you to Adaptee!
I'm Adaptee
That was my adaptee
Stop bullying me!
I'm versatile
```

```ruby
#
# The GoF Composite pattern
# written by Matthieu Tanguay-Carel
#
# The Component module contains the common behavior between the leaf
# and composite. The component being a module, two classes are free to
# share the same interface without being in the same object hierarchy.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

module Component #file system entity
    attr_accessor :name
    attr_accessor :owner

    def initialize name
        @name = name
    end

    def children
        @children ||= []
    end

    def rename new_name
        @name = new_name
    end

    def to_s
        @name
    end

    def add_child *new_children
        new_children.each {|child|
            children.push child
            puts "adding #{self} as owner of #{child}"
            child.owner = self
        }
    end

    def remove_child child
        children.delete child
    end
end

class MyFile
    include Component
    attr_accessor :file_type

    def initialize name, type
        @file_type = type
        super name #we need to call super whatever happens
                   #see ruby cookbook's recipe 9.8
    end
end

class MyDir
    include Component
```

```ruby
    attr_accessor :icon

    def is_dir; true; end

    def initialize name, icon
        @icon = icon
        super name
    end
end

if __FILE__ == $0
    #setup
    root  = MyDir.new 'root', :ginger
    puts "created directory root with icon in the form of a #{root.icon}"
    music = MyDir.new 'music', :clef
    jewel = MyDir.new 'jewel', :guitar
    notes = MyFile.new 'notes', :text
    puts "created file notes whose file type is #{notes.file_type}"
    movie = MyFile.new 'ratatouille', :mpeg
    todos = MyFile.new 'todos', :text
    song  = MyFile.new 'iloveyou', :mp3

    root.add_child notes, movie, todos
    root.add_child music
    music.add_child song
    music.add_child jewel

    #use case 1
    puts 'prefixing all components as if they were the same type'
    def recursive_prefix prefix, component
        component.rename(prefix + component.name)
        component.children.each {|child|
            recursive_prefix prefix, child
        }
    end
    recursive_prefix 'prefixed_', root

    #use case 2
    puts "extracting all directories"
    def all_directories root
        root.children.inject([]){|acc,component|
            if component.respond_to? :is_dir
                acc << component
                acc.push *all_directories(component)
            end
            acc
        }
    end
    all_directories(root).each {|d| puts d}

    #use case 3
    puts "going up the hierarchy"
    def get_master component
        component = component.owner while !component.owner.nil?
        component
    end
```

```
    puts get_master(song)
    puts get_master(jewel)
end
```

Output

```
created directory root with icon in the form of a ginger
created file notes whose file type is text
adding root as owner of notes
adding root as owner of ratatouille
adding root as owner of todos
adding root as owner of music
adding music as owner of iloveyou
adding music as owner of jewel
prefixing all components as if they were the same type
extracting all directories
prefixed_music
prefixed_jewel
going up the hierarchy
prefixed_root
prefixed_root
```

```ruby
#
# The GoF Decorator pattern
# written by Matthieu Tanguay-Carel
#
# This pattern is made trivial by Ruby's meta methods.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

module Bordered
    attr_accessor :color
    attr_accessor :width
end

module Scrollable
    def position
        @position ||= 0
    end
    def scroll offset
        @position = position + offset
    end
end

class Widget
    attr_accessor :content
    def initialize content
        @content = content
    end
end

if __FILE__ == $0
    widget = Widget.new "flagada jones"
    widget.extend(Bordered)
    widget.color = :blue

    widget.extend(Scrollable)
    widget.scroll 3

    puts widget.kind_of?(Scrollable)
    puts widget.kind_of?(Bordered)
end
```

Output

```
true
true
```

```ruby
#
# The GoF Flyweight pattern
# written by Matthieu Tanguay-Carel
#
# The Glyph instances are the flyweights.
# Each glyph knows how to draw itself, given the context.
# You can supply a block to Glyp#draw to draw something else than
# the glyph itself.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class Glyph
    attr_accessor :char
    def initialize char
        puts "initializing with #{char}"
        @char = char
    end

    def draw context #hash expecting :color and :size and :x and :y as keys
        inner_html = block_given?? yield(@char) : @char
        "<span style='color:#{context[:color]}; font-size:#{context[:size]};\
        position:absolute; top: #{context[:y]}px; " + \
        " left: #{context[:x]}px'>#{inner_html}</span>"
    end
end

class FlyweightFactory
    def initialize
        @flyweights = {}
    end

    def get charsym
        return @flyweights[charsym] if @flyweights.include? charsym
        @flyweights[charsym] = Glyph.new charsym
    end
end

if __FILE__ == $0
    #a few tests
    factory = FlyweightFactory.new
    a = factory.get :a
    a2 = factory.get :a
    puts "Flyweights are the same object: #{a.eql?(a2)}"
    b = factory.get :b
    b2 = factory.get :b
    puts "Flyweights are the same object: #{b.eql?(b2)}"

    #draw a rectangle containing letters in random contexts
    File.open('test.html','w') {|file|
        file.write "<div style='width:800px; height:600px; " + \
            "border:1px #ccc solid; background-color:#efefff;'"
        colors = ['red', 'blue', 'grey']
        sizes = ['24pt', '8pt', '14pt']
        context = {}
        syms = [:a, :b, :b, :b, :c, :d, :e, :e, :f, :d, :e, :e, :f]
        syms.each {|s|
```

```
            index = rand 3
            index2 = rand 3
            x = rand 800
            y = rand 600
            context[:color] = colors[index]
            context[:size] = sizes[index2]
            context[:x] = x
            context[:y] = y
            file.write factory.get(s).draw(context) {|char|
                "#{char}?!"
            }
        }
    }
end
```

Output

```
initializing with a
Flyweights are the same object: true
initializing with b
Flyweights are the same object: true
initializing with c
initializing with d
initializing with e
initializing with f
```

```ruby
#
# The GoF Proxy pattern
# written by Matthieu Tanguay-Carel
#
# The Image class is the proxy. It should override the operations
# the clients need before costly processing has to take place.
# The attr_proxy method allows the Proxy module to automatically
# remove the overridden methods once the real subject is created.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

module Proxy
    def self.included cls
        puts "creating the attr_proxy method"
        cls.instance_eval {
            def proxy_methods
                @proxy_methods ||= []
            end
            def attr_proxy name
                proxy_methods << name
            end
        }
    end

    #call this to set the proxy object
    def proxy real_cls, constructor_args
        @real_cls = real_cls
        @constructor_args = constructor_args
    end

    def real_subject
        @real_subject or nil
    end

    def method_missing method, *args
        if real_subject.nil?
            @real_subject = @real_cls.new *@constructor_args
            puts "instantiating real subject"
            self.class.proxy_methods.each {|proxy_meth|
                puts "removing #{proxy_meth} from proxy"
                self.class.instance_eval {
                    remove_method proxy_meth
                }
            }
        end
        if real_subject.respond_to? method
            real_subject.send method, *args
        else
            raise NotImplementedError, "This method (#{method}) is " + \
                "not available on this interface"
        end
    end
end

class Image
    include Proxy
```

```ruby
    attr_accessor :mtime
    attr_proxy :mtime
    attr_proxy :mtime=
    attr_proxy :to_s

    def to_s
        "proxy_image"
    end
end

if __FILE__ == $0
    #create the proxy
    img = Image.new
    img.proxy(File, ["img.jpg", 'w'])
    img.mtime = "a few hours ago"
    puts "proxy methods:"
    img.class.proxy_methods.each {|m| puts m}
    puts ''

    #use the proxy
    puts "image's last modified time is #{img.mtime}"
    puts "image's string representation: #{img}"
    puts ''

    #force creation of the real subject
    img.write "im stuck in an image !\n"
    puts "image's last modified time is #{img.mtime}"
    puts "image's string representation: #{img}"
    puts "file written to!"
end
```

Output

```
creating the attr_proxy method
proxy methods:
mtime
mtime=
to_s

image's last modified time is a few hours ago
image's string representation: proxy_image

instantiating real subject
removing mtime from proxy
removing mtime= from proxy
removing to_s from proxy
image's last modified time is Sun Oct 14 17:25:17 +1000 2007
image's string representation: #<Image:0xb7bfcbbc>
file written to!
```

```ruby
#
# The GoF Chain of Responsibility pattern
# written by Matthieu Tanguay-Carel
#
# Each handler needs to be added to a chain and needs to be given
# an operation.
# The handler's operation is a block that should return false if the request
# should be sent forward.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class Array
    def element_after item
        inject(false){|acc, elem|
            return elem if acc
            elem == item ? true : false
        }
        nil
    end
end

class Chain
    def initialize
        @chain = []
    end

    def add_handler *handlers
        handlers.reverse.each {|h|
            @chain << h
            h.chain = self
        }
    end

    def forward caller, request
        next_soul = @chain.element_after caller
        raise Exception.new("End of chain: caller has no forward " + \
            "neighbor available in this chain") if next_soul.nil?
        next_soul.handle request
    end
end

module Handler
    attr_accessor :chain
    def handle request
        raise Exception.new("Handler without a chain") if @chain.nil?
        raise Exception.new("Handler without an operation") if @operation.nil?
        chain.forward self, request if !@operation.call request
    end

    def operation &block
        @operation = block
    end
end

def protect
    begin
```

```
        yield
    rescue Exception
        puts $!
    end
end

if __FILE__ == $0
    @chain = Chain.new

    #create some handlers and add them to chain
    default_handler = Object.new
    default_handler.extend Handler
    default_handler.operation {|request|
        puts "Default handler: the chain of responsibility could not handle" + \
            " the request: #{request}"
    }

    coward = Object.new
    coward.extend Handler
    coward.operation {|request|
        puts "I'm not getting my hands dirty. Let's forward."
        false
    }

    hard_worker = Object.new
    hard_worker.extend Handler
    hard_worker.operation {|request|
        if request.respond_to? :include? and request.include? "work"
            puts "Request handled!"
            true
        else
            puts "Could not handle request... forwarding."
            false
        end
    }

    @chain.add_handler default_handler, hard_worker, coward

    #tests
    protect {
        puts "\nSending first test request"
        coward.handle "test"
    }
    protect {
        puts "\nSending work request"
        coward.handle "work"
    }
    puts "\nMaking it fail"
    foreigner = Object.new
    foreigner.extend Handler
    protect { foreigner.handle "me" }
    foreigner.operation {|request| puts "Guten Tag"}
    protect { @chain.forward foreigner, "hehe" }
end
```

Output

```
Sending first test request
I'm not getting my hands dirty. Let's forward.
Could not handle request... forwarding.
Default handler: the chain of responsibility could not handle the request: test
End of chain: caller has no forward neighbor available in this chain

Sending work request
I'm not getting my hands dirty. Let's forward.
Request handled!

Making it fail
Handler without a chain
End of chain: caller has no forward neighbor available in this chain
```

```ruby
#
# The GoF Command pattern
# written by Matthieu Tanguay-Carel
#
# The Command instance is initialized with its receiver.
# Commands can be grouped by registering children to a macro command.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class Command
    attr_accessor :receiver
    def initialize receiver
        @receiver = receiver
        @commands = []
    end

    def register_command *command
        @commands.push *command
    end

    def execute
        @commands.each {|cmd| cmd.save }
        @commands.each {|cmd| cmd._execute }
        save
        _execute
    end

    def undo
        @commands.each {|cmd| cmd.undo }
    end

    #implement the following methods in the subclasses
    protected
    def save
    end

    def _execute
    end
end

class TextCommand < Command
    def save
        @last_state ||= Marshal.load(Marshal.dump(@receiver.text))
        super
    end
    def undo
        @receiver.text = @last_state
        @last_state = nil
        super
    end
end

class UppercaseCommand < TextCommand
    def _execute
        @receiver.text.upcase!
        super
```

```
    end
end

class IndentCommand < TextCommand
    def _execute
        @receiver.text = "\t" + @receiver.text
        super
    end
end

module Invoker
    attr_accessor :command
    def click
        @command.execute
    end

    def undo
        @command.undo
    end
end

class Document
    attr_accessor :text
    def initialize text
        @text = text
    end
end

if __FILE__ == $0
    text = "This is a test"
    doc = Document.new text
    upcase_cmd = UppercaseCommand.new doc
    button = Object.new.extend(Invoker)
    button.command = upcase_cmd

    puts "before anything"
    puts doc.text
    button.click
    puts "after click"
    puts doc.text
    button.undo
    puts "after undo"
    puts doc.text

    puts "\nNow a macro command"
    allCmds = Command.new doc
    indent_cmd = IndentCommand.new doc
    allCmds.register_command upcase_cmd, indent_cmd

    big_button = Object.new.extend(Invoker)
    big_button.command = allCmds
    puts "before anything"
    puts doc.text
    big_button.click
    puts "after click"
    puts doc.text
```

```
    big_button.undo
    puts "after undo"
    puts doc.text
end
```

Output

```
before anything
This is a test
after click
THIS IS A TEST
after undo
This is a test

Now a macro command
before anything
This is a test
after click
        THIS IS A TEST
after undo
This is a test
```

```ruby
#
# The GoF Memento pattern
# written by Matthieu Tanguay-Carel
#
# The Originator can save and load itself.
# The Caretaker (the main function in this case) never has to touch
# the memento objects.
#
# This implementation is a bit naive:
#   - saves should be kept in files
#   - Marshal will not always work (singleton methods, bindings, etc..)
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

module Originator
    def saves
        @saves ||= {}
    end

    def save key
        puts "saving key #{key}"
        saves[key] = Marshal.dump self
    end

    def restore key
        puts "restoring key #{key}"
        include_state Marshal.load(saves[key])
    end

    def include_state other
        other.instance_variables.each {|var|
            instance_variable_set(var, other.instance_variable_get(var)) \
                if var != "@saves"
        }
    end
end

class Example
    include Originator
    attr_accessor :name, :color

    def initialize name, color
        @name = name
        @color = color
    end
end

if __FILE__ == $0
    ex = Example.new "Matt", "blue"
    puts "my name is #{ex.name}"
    ex.save :now
    ex.name = "John"
    puts "my name is #{ex.name}"
    ex.save :later

    ex.restore :now
```

```
    puts "my name is #{ex.name}"
    ex.restore :later
    puts "my name is #{ex.name}"
end
```

Output

```
my name is Matt
saving key now
my name is John
saving key later
restoring key now
my name is Matt
restoring key later
my name is John
```

```ruby
#
# The GoF State pattern
#
#   Here is Maurice Codik's implementation.
#   I only added an "if __FILE__ == $0", tweaked the layout, and fixed
#   a typo.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
#
# Copyright (C) 2006 Maurice Codik - maurice.codik@gmail.com
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the "Software"),
# to deal in the Software without restriction, including without limitation
# the rights to use, copy, modify, merge, publish, distribute, sublicense,
# and/or sell copies of the Software, and to permit persons to whom the
# Software is furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included
# in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL
# THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
# OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
# ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
# OTHER DEALINGS IN THE SOFTWARE.
#
# Each call to state defines a new subclass of Connection that is stored
# in a hash.  Then, a call to transition_to instantiates one of these
# subclasses and sets it to the be the active state.  Method calls to
# Connection are delegated to the active state object via method_missing.

module StatePattern
  class UnknownStateException < Exception
  end

  def StatePattern.included(mod)
    mod.extend StatePattern::ClassMethods
  end

  module ClassMethods
    attr_reader :state_classes
    def state(state_name, &block)
      @state_classes ||= {}

      new_klass = Class.new(self, &block)
      new_klass.class_eval do
        alias_method :__old_init, :initialize
        def initialize(context, *args, &block)
          @context = context
          __old_init(*args, &block)
        end
      end
```

```
      @state_classes[state_name] = new_klass
    end
  end

  attr_accessor :current_state, :current_state_obj

  def transition_to(state_name, *args, &block)
    new_context = @context || self

    klass = new_context.class.state_classes[state_name]
    if klass
      new_context.current_state = state_name
      new_context.current_state_obj = klass.new(new_context, *args, &block)
    else
      raise UnknownStateException, "tried to transition to " + \
        "unknown state, #{state_name}"
    end
  end

  def method_missing(method, *args, &block)
    unless @current_state_obj
      transition_to :initial
    end
    if @current_state_obj
      @current_state_obj.send(method, *args, &block)
    else
      super
    end
  end

end

class Connection
  include StatePattern
  state :initial do # you always need a state named initial
    def connect
      puts "connected"
      # move to state :connected. all other args to transition_to
      # are passed to the new state's constructor
      transition_to :connected, "hello from initial state"
    end
    def disconnect
      puts "not connected yet"
    end
  end
  state :connected do
    def initialize(msg)
      puts "initialize got msg: #{msg}"
    end
    def connect
      puts "already connected"
    end
    def disconnect
      puts "disconnecting"
      transition_to :initial
    end
```

```ruby
  end
  def reset
    puts "resetting outside a state"
    # you can also change the state from outside of the state objects
    transition_to :initial
  end
end

if __FILE__ == $0
    c = Connection.new
    c.disconnect # not connected yet
    c.connect    # connected
                 # initialize got msg: hello from initial state
    c.connect    # already connected
    c.disconnect # disconnecting
    c.connect    # connected
                 # initialize got msg: hello from initial state
    c.reset      # reseting outside a state
    c.disconnect # not connected yet
end
```

Output

```
not connected yet
connected
initialize got msg: hello from initial state
already connected
disconnecting
connected
initialize got msg: hello from initial state
resetting outside a state
not connected yet
```

```ruby
#
# The GoF Strategy pattern
# written by Matthieu Tanguay-Carel
#
# Sorter is the Context object. It allows to choose between sorting
# implementations.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

class QuickSort
    def sort arr
        return [] if arr.length == 0
        x, *xs = *arr
        smaller, bigger = xs.partition{ |other| other < x }
        sort(smaller) + [x] + sort(bigger)
    end
end

class MergeSort
    def sort array
        if array.length <= 1
            return array
        end
        middle = array.length / 2
        left = array[0...middle]
        right = array[middle...array.length]
        left = sort left
        right = sort right
        return merge(left,right)
    end

    def merge left,right
        result = []
        while left.length > 0 and right.length > 0
            left.first <= right.first ? result << left.shift : result << right.shift
        end
        result.push *left if left.length > 0
        result.push *right if right.length > 0
        return result
    end
end

class Sorter
    @@default_strategy = QuickSort.new
    def self.sort arr, strategy=nil
        strategy ||= @@default_strategy
        strategy.sort(arr)
    end
end

def print_elems arr
    arr.each {|elem| $stdout.write "#{elem} "}
    puts ''
end

def get_random_array size
```

```
    arr = []
    size.times do arr << rand(100) end
    arr
end

require 'benchmark'
if __FILE__ == $0
    arr_length = 1000
    arr1 = get_random_array arr_length
    puts "Sorting first array"
    #print_elems arr1
    puts "Time taken for QuickSort: #{Benchmark.measure {
        arr1 = Sorter.sort(arr1, QuickSort.new)
        print_elems arr1[0...40]
    }}"

    puts "\nSorting second array"
    arr2 = get_random_array arr_length
    #print_elems arr2
    puts "Time taken for MergeSort: #{Benchmark.measure {
        arr2 = Sorter.sort(arr2, MergeSort.new)
        print_elems arr2[0...40]
    }}"
end
```

Output

```
Sorting first array
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 4
Time taken for QuickSort:   0.030000   0.000000   0.030000 (  0.030721)

Sorting second array
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3
Time taken for MergeSort:   0.030000   0.000000   0.030000 (  0.029816)
```

```ruby
#
# The GoF Template pattern
# written by Matthieu Tanguay-Carel
#
# The module Template implements the boilerplate of the algorithm.
# Some hooks are optional and some mandatory.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

module Template
    #mandatory_methods = ["tagname", "content"]
    #optional_methods = ["font_size", "background_color"]
    def generate
        str = "<#{tagname}"
        styles = ''
        styles += "font-size:#{font_size};" if respond_to? :font_size
        styles += "background-color:#{background_color};" \
            if respond_to? :background_color
        str += " style='#{styles}'" if !styles.empty?
        str += ">#{content}</#{tagname}>"
    end
end

class Body
    def tagname
        "body"
    end
    def content
        "hello"
    end
    def font_size
        "18pt"
    end
    include Template
end

if __FILE__ == $0
    b = Body.new
    puts b.generate
end
```

Output

```
<body style='font-size:18pt;'>hello</body>
```

```ruby
#
# The GoF Visitor pattern
# written by Matthieu Tanguay-Carel
#
# Depends on Rubytree (gem install rubytree).
# The Node module contains the whole logic. A visitor can only implement
# the callbacks it is interested in.
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

require 'rubygems'
require 'tree'

module Node
    def accept visitor
        if self.kind_of? StringNode
            visitor.visit_string self if visitor.respond_to? :visit_string
        elsif self.kind_of? IntegerNode
            visitor.visit_int self if visitor.respond_to? :visit_int
        end
    end
end

class StringNode
    include Node
    attr_accessor :string
    def initialize val
        @string = val
    end
end

class IntegerNode
    include Node
    attr_accessor :int
    def initialize val
        @int = val
    end
end

class PrintingVisitor
    def visit_string node
        puts node.string
    end
    def visit_int node
        puts node.int
    end
end

class RevertingVisitor
    def visit_string node
        puts node.string.reverse!
    end
end

if __FILE__ == $0
    myTreeRoot = Tree::TreeNode.new("ROOT", StringNode.new("this is the root node"))
```

```ruby
    myTreeRoot << Tree::TreeNode.new("child1", StringNode.new("madam im adam")) \
        << Tree::TreeNode.new("grandchild1", IntegerNode.new(3)) \
        << Tree::TreeNode.new("grandchild2", IntegerNode.new(2))
    myTreeRoot << Tree::TreeNode.new("child2", StringNode.new("race car")) \
        << Tree::TreeNode.new("grandchild3", StringNode.new("damn, i agassi " + \
            "miss again. mad"))

    puts "PRINTING visitor..."
    @pvisitor = PrintingVisitor.new
    myTreeRoot.each { |node| node.content.accept @pvisitor }

    puts "\nREVERTING visitor..."
    @rvisitor = RevertingVisitor.new
    myTreeRoot.each { |node| node.content.accept @rvisitor }
end
```

Output

```
PRINTING visitor...
this is the root node
madam im adam
3
2
race car
damn, i agassi miss again. mad

REVERTING visitor...
edon toor eht si siht
mada mi madam
rac ecar
dam .niaga ssim issaga i ,nmad
```