Inheritance of Automata Classes Using Dynamic Programming Languages (using Ruby as an Example)

Kirill Timofeev
SPbSU IFMO
email: ktimofeev@dataart.com

Artyom Astafurov
SPbSU IFMO
email: astaff@dataart.com

Anatoly Shalyto
SPbSU IFMO
email: shalyto@mail.ifmo.ru

Abstract

This paper analyzed two libraries for implementation of automata in dynamic languages. The comparison has been made based on modifying the functionality of the Restful-authentication plugin for Ruby on Rails framework.

These libraries allow transferring a graphical model into a source code. Moreover library developed by the authors of the paper provide a method for creation of state groups which reduces the number of transitions required to implement the automaton in the code. Also this library preserved the hierarchy of the parent automata after inheritance. Using developed library it is possible to perform a reverse engineering and restore the original graphical model from the source code.

1. Introduction

Year over year the dynamic programming languages are used more often in software development. For example, in 2008 the proportion of dynamic languages to the languages with static type checking was 40% [1]. Dynamic languages allow for runtime program extension by dynamic creation of new methods and usage of macro scripts [2]. *Ruby* is a dynamic programming language and is a part of top 10 most popular languages in 2008 [1]. Also this language has been used to develop a popular open source web-application framework *Ruby on Rails*. One of the features of *Ruby on Rails* is a flexible functionality extension mechanism that uses plugins that can be added into *Ruby on Rails* application. *Restlful-authentication* [4] is one of such plugins that allows for web application to support user registration functionality. This plugin is used in 96% [5] of applications developed on *Ruby on Rails* and is implemented using *Automata* approach.

Restful-authentication plugin uses *Acts as State Machine* [6] library that doesn't allow preserve the hierarchy of the parent automata when the automaton is inherited and doesn't have nested groups of states concept. This leads to duplicated code, makes debugging more complicated and also doesn't allow an isomorphic generation of the model based on code when necessary. In paper [2] a method for inheritance of automata classes has been proposed, also a *State Machine on Steroids* library has been developed to support the concept. This library uses the traditional object-oriented paradigm applied to automata and extended with some features of dynamic libraries, which means that automata classes are created in runtime. Using the method proposed by authors of paper [2] will help to eliminate the drawbacks of the *Restful-authentication* plugin implementation.

The goal of this research paper is to compare the approaches to inheritance of automata classes described above using *Restful-authentication* plugin implementation as an example.

2. Graphical Notation Being Used

As a graphical model it is proposed to use state diagram from *UML 2* [7] extended with graphical notation for inheritance of automata classes. This notation has been proposed in [8], as in *UML 2* it is impossible to present inheritance of automata classes. The sample use of an extended graphical notation is shown on fig.1.

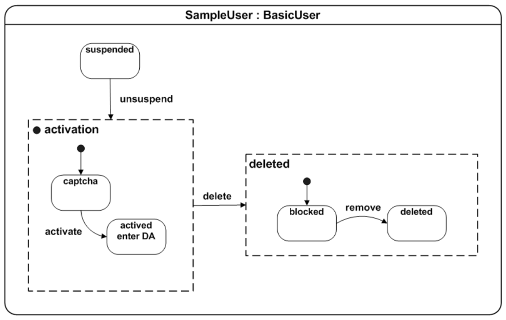

Fig. 1. Extended Graphical Notation

On this figure the automaton class `SampleUser` is presented. This class is inherited from automaton class

`BasicUser` and contains the following changes to its base class:

- added a new group `deleted` which is made of two states: `blocked` and `deleted`. The initial state is set to "`blocked`";
- the group activation has been overridden. It now contains `captcha` state which is marked as initial. A transition from this group to the new group `deleted` as been added;
- new state `suspended` has been added. There is a transition from `suspended` state to `activation` group.

3. Problem Statement

Let's take *Restful-authentication* plugin and review it in more details. Assume, for example, that in order to register successfully the user has to take three steps:

1. Fill in the registration form on the website;
2. Receive an e-mail with activation code;
3. Confirm the registration by entering the activation code into a special form on the website.

In case the user hasn't registered in a given period of time or entered the activation code incorrectly several times, he will be blocked and won't be able to register on the site any longer. The site administrator can block the users as well as delete from the system.

On fig. 2 using the extended graphical notation the automaton for basic user registration (`BasicUser`) is presented.

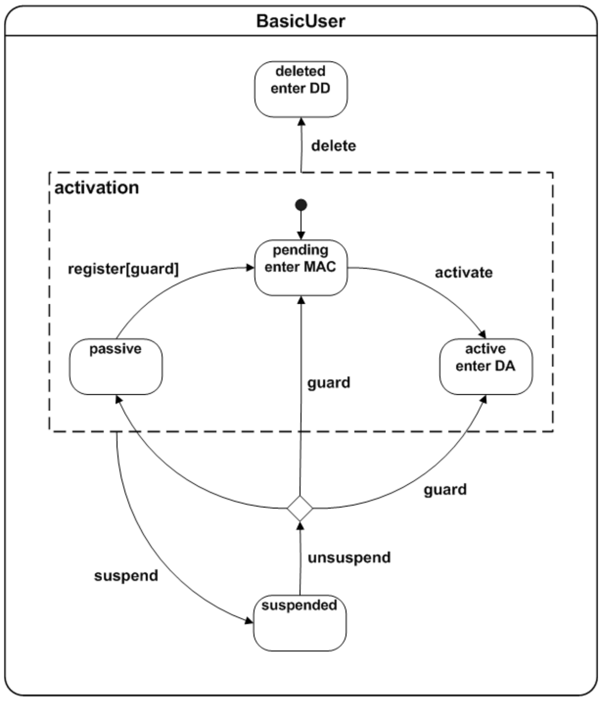

Fig. 2. Basic user registration

It consists of `activation` group and `suspended` and `deleted` states. The nested group `activation` contains three states: `passive`, `active` and `pending`, the latter is marked as initial.

The `AdvancedUser` automaton (fig. 3) is inherited from `BasicUser` automaton and has the following functionality:

1. After entering the activation code the user should perform two additional actions: recognize Captcha (to avoid spam registrations) and submit user data on a profile page. In order to do this, a nested group `active` should be created.
2. The deletion of the user is performed in two stages: on the first stage the user is blocked, but his messages are still being stored in the system, on the second stage both: the user and his data are deleted. In order to do this a nested group `deleted` should be created.

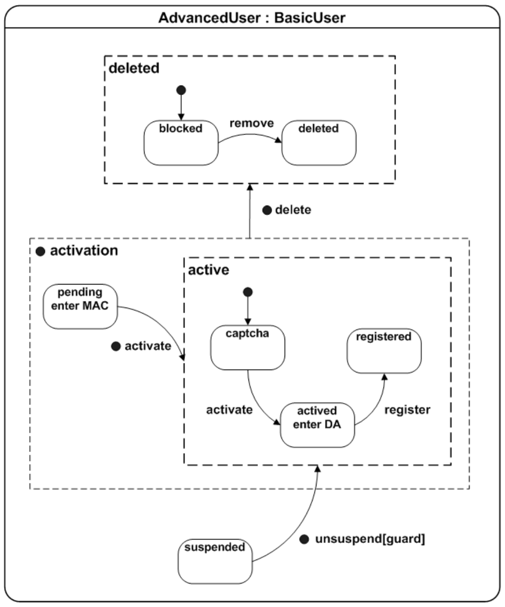

Fig. 3. Extended User Registration Automata

4. The Implementation of Inheritance using Act as State Machine Library

Let's look at the `BasicUser` class that represents a basic user in *Restful-authentication* plugin using *Acts as State Machine* library. It provides four methods:

- `acts_as_state_machine :initial` – sets the initial state of the automata;
- `state` – creates the state. As optional parameters it accepts lambda-functions [2]: the actions to be done when entering or leaving the state;
- `event` – named transition;

- `transitions` – defines from/to states for making a named transition. As an optional parameter the method accepts a lambda-function that can contain a guard condition.

This library doesn't support nested groups. In this case in order to make an isomorphic transfer of a diagram with nested groups into the code it is required to modify the original `BasicUser` diagram as shown on fig. 4 by "flattering" it:

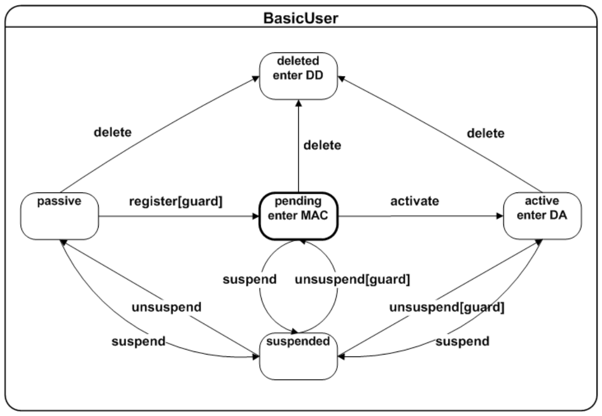

Fig. 4. A Basic User Registration Automata Without Nested Groups

Below is a *Ruby* code snippet that implements the model described above:

```
class BasicUser
  acts_as_state_machine :initial => :pending

  state :passive
  state :pending, :enter =>
:make_activation_code
  state :active,  :enter => :do_activate
  state :suspended
  state :deleted, :enter => :do_delete

  event :register do
    transitions :from => :passive, :to =>
:pending,
                :guard => Proc.new {|u| !
(u.crypted_password.blank? &&
u.password.blank?) }
  end

  event :activate do
    transitions :from => :pending, :to =>
:active
  end

  event :suspend do
    transitions :from => [:passive,
:pending, :active], :to => :suspended
  end

  event :delete do
    transitions :from => [:passive,
:pending, :active, :suspended], :to =>
:deleted
  end

  event :unsuspend do
    transitions :from => :suspended, :to =>
:active,
                :guard => Proc.new {|u| !
u.activated_at.blank? }

    transitions :from => :suspended, :to =>
:pending,
                :guard => Proc.new {|u| !
u.activation_code.blank? }
    transitions :from => :suspended, :to =>
:passive
  end
end
```

Let's create a new class `AdvancedUser` which is represented on the fig. 5. This class will be inherited from `BasicUser` class. The new class will inherit the internal structure of the `BasicUser` automaton created using the methods of *Acts as State Machine* library. Since the library doesn't support nested groups the hierarchy of the parent `BasicUser` automata will be lost in the inherited `AdvancedUser` automata.

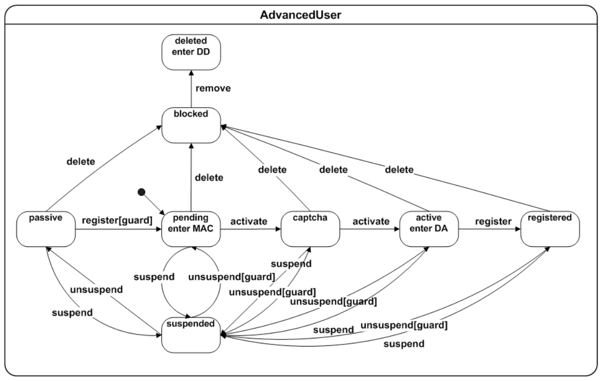

Fig. 5. The Model of the `AdvancedUser` Automata Class After Losing the Hierarchy of the Parent Automata

Below is the code that implements the `AdvancedUser` automaton:

```
class AdvancedUser < BasicUser
  state :captcha
  state :registered
  state :blocked

  event :activate do
    transitions :from => :captcha, :to =>
:actived
    transitions :from => :pending, :to =>
:captcha
  end

  event :register do
    transitions :from => :activated, :to =>
:registered
  end
  event :unsuspend do
    transitions :from => :unsuspend, :to =>
:captcha
  end

  event :suspend do
    transitions :from => [:passive,
:pending, :active, :suspended, :captcha,
:registered], :to => :suspended
  end

  event :delete do
    transitions :from => [:passive,
:pending, :active, :suspended, :captcha,
:registered], :to => :deleted
  end
end
```

The approach described above has the following drawbacks:

- The loss of hierarchy of the parent automata. This results the inability to do the isomorphic transfer of the code into the diagram preserving the structure of the original diagram;
- No support for nested groups of states which results in increase of the number of transitions in the automaton.

5. The Implementation of Inheritance Using State Machine on Steroids Library

In order to eliminate the drawbacks of the described above approach, the author of this paper have implemented a new library called *State Machine on Steroids*. Like *Acts as State Machine* it uses dynamic programming language to implement automata classes. The library provides the following six methods:

- `automaton` – the method accepts the block as an argument. The block has a *DSL*-syntax (*Domain Specific Language*) designed to describe the automaton [2];
- `group` – creates the nested group of states;
- `initial` – sets the initial state for the automaton;
- `state` – creates the state in the automaton in runtime. As an optional parameter it accepts a lambda-function that represents an action on entering/leaving the state;
- `event` – named transition;
- `transitions` – defines the starting and the destination states for the named transition. As an optional parameter the method accepts a lambda-function that serves as a guard condition for the transition.

Let's implement the automaton presented on fig. 2:

```
class BasicUser
  include StateMachineOnSteroids

  automaton :user, :initial => :activation do
    state :deleted

    group :activation, initial => :pending do
      state :pending do
        transition :activate, :to => :actived
      end

      state :passive do
          transition :register, :to =>
:pending,
                  :guard => Proc.new {|u| !
(u.crypted_password.blank? &&
u.password.blank?) }
      end

      state :activated do
      transition :suspend, :to => :suspended
      transition :delete, :to => :deleted
    end

    state :suspended do
      event :unsuspend do
        transition :to => :active,
                  :guard => Proc.new {|u| !
u.activated_at.blank? }
        transition :to => :pending,
                  :guard => Proc.new {|u| !
u.activation_code.blank? }

        transition :to => :passive
      end
    end
  end
end
```

In code above every call to the method of the library creates a separate class. This approach preserves the structure of the parent automata `BasicUser` after inheriting it.

Let's implement the inherited class `AdvancedUser` (fig. 3):

```
class AdvancedUser < BasicUser
  automaton :user do
    group :deleted, initial => :blocked do
      state :blocked do
        transition :remove, :to => :deleted
      end

      state :”user::activation::deleted” do
    end
    group :activation do
      group :active, initial => :captcha do
        state :captcha do
          transition :activate, :to =>
:activated
        end
        state :”user::activated” do
          transition :register, :to =>
:registered
        end
        state :registered
      end
    end
    transition :”user::delete”, :to =>
:deleted
    transition :”user::activate”, :to =>
:actived
    state :suspended do
      event :”user::unsuspend”, :to => :active
    end
  end
end
```

Using the described above *State Machine on Steroids* library we were able to eliminate the drawbacks of *Acts of State Machine* library, discussed in Part 3 of this paper:

- a method for creation of state groups has been introduced which reduces the number of transitions required to implement the automaton in the code;
- the hierarchy of the parent automata is preserved after inheritance. This allows to reference to the previously created classes, for example we can reference `:"user::deleted"`. Also this helps to solve the problem of automatic isomorphic transfer of the source code back to the graphical model.

6. Side-by-Side Comparison of Acts as State Machine and State Machine on Steroids Libraries

State Machine on Steroids library retains all the advantages of the dynamic and object-oriented features of *Ruby* programming language, such as:

- the ability to create the code using domain specific language. As a result, the domain experts will be able to easier understand, verify and modify the source code;
- the support for the development of self-documenting code;
- increase of the quality, reliability and maintainability of the programs;
- preservation of the hierarchy during the inheritance of the automata.

On a special note, one of the advantages of the *State Machine on Steroids* library is the ability to perform an isomorphic transfer of the graphical notation into the source code and vice versa. This also allows to avoid duplicate code that is produced as a result of the implementation of group transitions in *Acts as State Machine* library.

The number of transitions and states required to implement the automaton displayed on the fig. 3 is shown in the table below for the libraries described above. Please note that *State Machines on Steroids* library simplifies the code making it less redundant.

Table. The Comparison of Libraries

Library	Transitions	States	Groups
Acts as State Machine	20	8	-
State Machine on Steroids	9	8	3

7. Conclusions

In this paper we have analyzed two libraries for implementation of automata in dynamic languages. The comparison has been made based on modifying the functionality of the *Restful-authentication* plugin.

These libraries allow for transferring a graphical model into a source code. At the same time a well-known *Acts as State Machine* library doesn't preserve the hierarchy of the parent automata after inheritance and doesn't support isomorphic implementation of nested groups of states.

The *State Machine on Steroids* library developed by the authors of the paper eliminates the drawbacks listed above. Also using this library it is possible to perform a reverse engineering and restore the original graphical model from the source code.

8. References

[1] TIOBE. "TIOBE Software: Tiobe Index". http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[2] Timofeev K. I., Astafurov A. A., and Shalyto A. A. "Automata classes inheritance with dynamical language Ruby" / Software Engineering Conference (Russia) 2008. http://www.secr.ru/?pageid=4548&submissionid=5270

[3] David H. H. "Ruby on Rails". http://www.rubyonrails.com/

[4] Grant G. "Restful Authentication Generator". http://github.com/technoweenie/restful-authentication/tree/master

[5] Szinek P. "Rails Rumble Observations, part II – trends in gem/plugin usage". http://www.rubyrailways.com/rails-rumble-observations-part-ii-trends-in-gemplugin-usage/

[6] Scott B., "Acts as State Machine". http://agilewebdevelopment.com/plugins/acts_as_state_machine.

[7] "Object Management Group. Official UML Specification". http://www.uml.org/#UML2.0

[8] Shopyrin D. G., and Shalyto A. A., "Graphical notation for automata classes inheritance" // Programming. 2007. #5, pp. 62–74. http://is.ifmo.ru/works/_12_12_2007_shopyrin.pdf