Multiple-Choice Questions:

1. SRP is short for:

    a. Software Requirements Process

    b. Sequential Response Protocol

    c. Server Receive Packet

    d. Single Responsibility Principle

    e. None of the above

2. Which of the following best exemplifies SRP?

    a. Writing test code before you write the code under test

    b. Creating mock objects to stand in for other objects in the system

    c. Dividing each user story to be built next iteration into tasks and assigning each task to a developer on the team

    d. For objects of a class *C*, creating a DAO (which knows how to read/write *C* objects to a database) instead of putting all that database accessing logic in the *C* class

    e. Collecting feedback from the customer at the end of each iteration, instead of waiting until the system is finally delivered

Solution:

1. d
2. d

Short-Answer and Multiple-Choice Questions:

1. What does MVC stand for?

Given these options:

   a. Responsible for user interface

   b. Responsible for security of the system

   c. Responsible for "business logic" and domain objects

   d. Responsible for translating between user interface actions/events and operations on the domain objects

   e. None of the above

2. What are the M components in MVC responsible for?

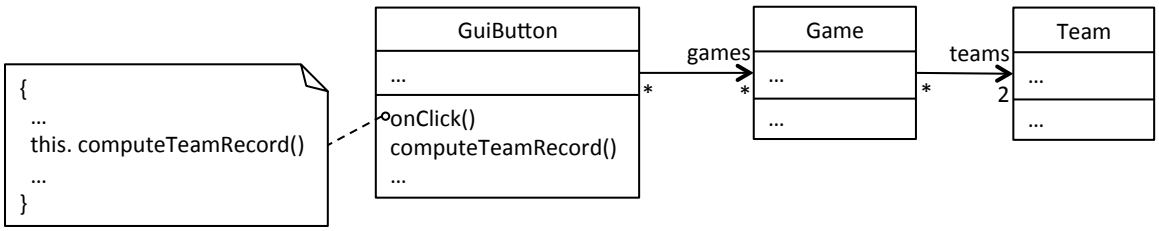
3. What are the V components in MVC responsible for?

4. What are the C components in MVC responsible for?

Solutions:

1. Model-View-Controller
2. c
3. a
4. d

Problem: Does the following design obey the Model-View Separation Principle? Explain your answer.



Solution:

No. The View class, GuiButton, has a method computeTeam-Record(). Computing a teams win/loss record is domain concern. Thus, a view class contains model code, which violates the Model-View Separation Principle.

Here are some figures to consider while answering the following questions.

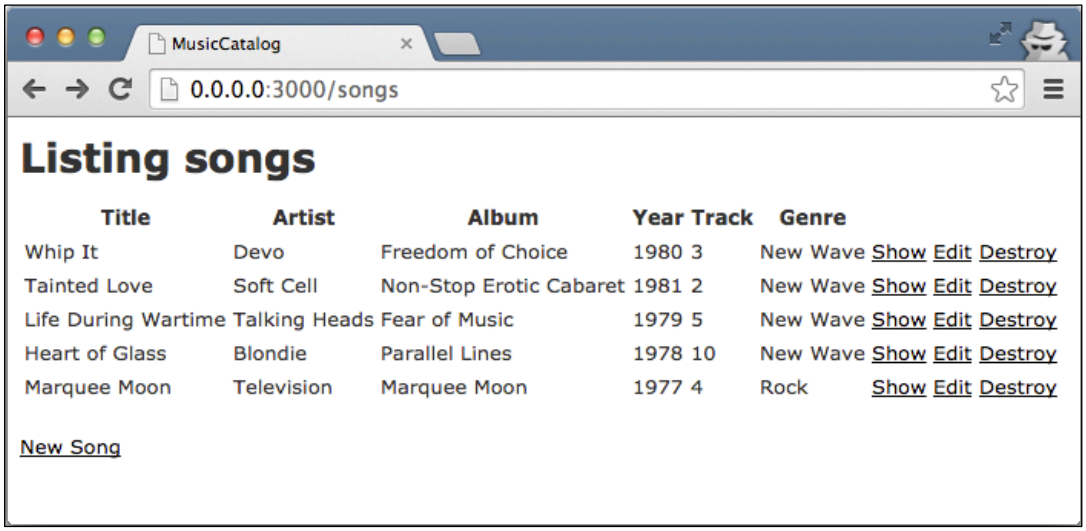


Figure 1. Example page from Music Catalog web app.

```
1  MusicCatalog::Application.routes.draw do
2    resources :songs
3  end
```

Figure 2. config/routes.rb

```
$ rake routes
   Prefix Verb   URI Pattern                  Controller#Action
    songs GET    /songs(.:format)             songs#index
          POST   /songs(.:format)             songs#create
 new_song GET    /songs/new(.:format)         songs#new
edit_song GET    /songs/:id/edit(.:format)    songs#edit
     song GET    /songs/:id(.:format)         songs#show
          PATCH  /songs/:id(.:format)         songs#update
          PUT    /songs/:id(.:format)         songs#update
          DELETE /songs/:id(.:format)         songs#destroy
```

Figure 3. Output of `rake routes` command.

```ruby
# == Schema Information
#
# Table name: songs
#
#  id         :integer          not null, primary key
#  title      :string(255)
#  artist     :string(255)
#  album      :string(255)
#  year       :string(255)
#  track      :integer
#  genre      :string(255)
#  created_at :datetime
#  updated_at :datetime
#

class Song < ActiveRecord::Base
end
```

Figure 4. app/models/song.rb

```ruby
class CreateSongs < ActiveRecord::Migration
  def change
    create_table :songs do |t|
      t.string :title
      t.string :artist
      t.string :album
      t.string :year
      t.integer :track
      t.string :genre

      t.timestamps
    end
  end
end
```

Figure 5. db/migrate/20140930033607_create_songs.rb

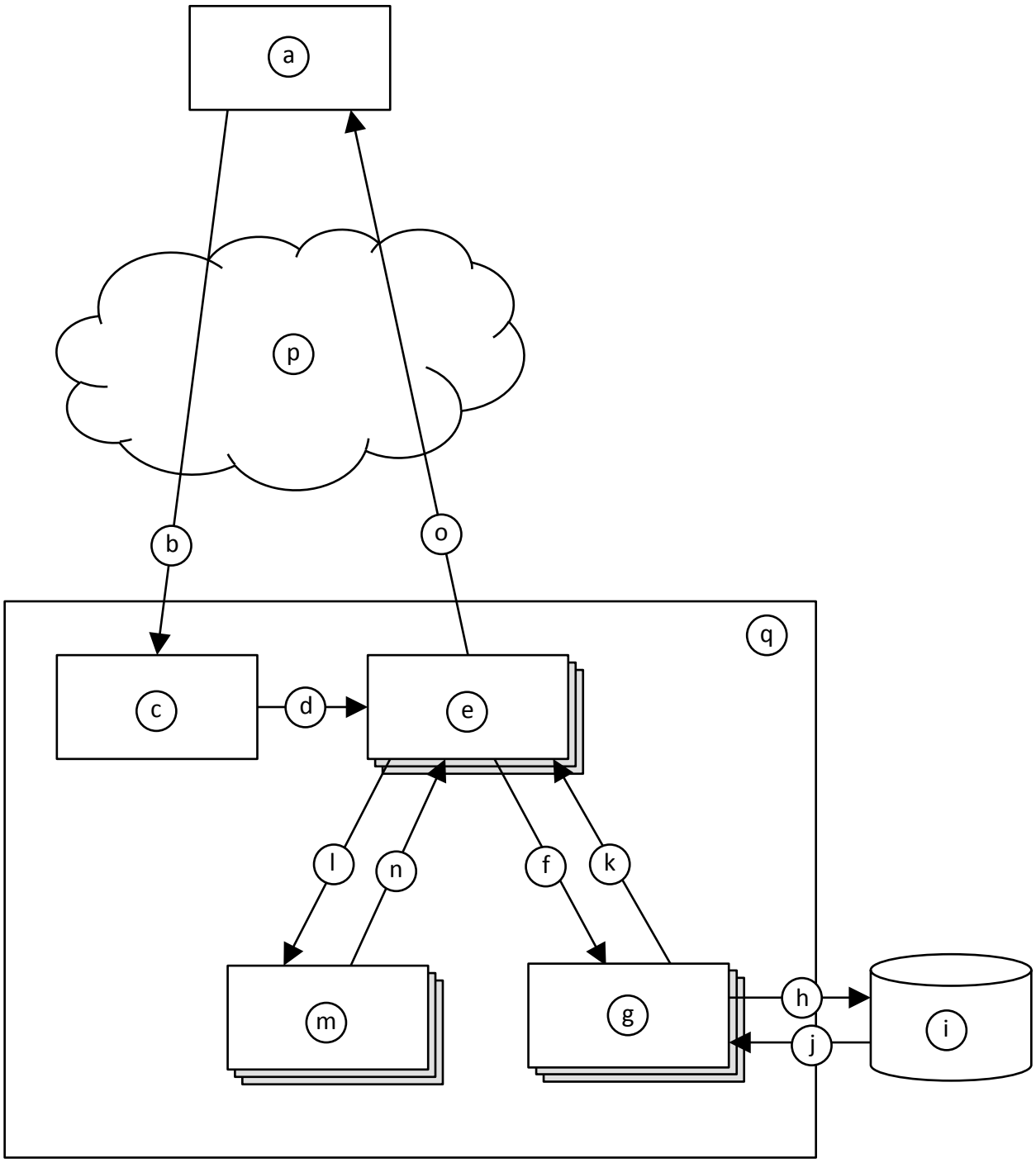
```ruby
class SongsController < ApplicationController
  def index
    @songs = Song.all
  end

  def show
    @song = Song.find(params[:id])
  end

  def new
    @song = Song.new
  end

  def edit
    @song = Song.find(params[:id])
  end

  def create
    @song = Song.new(song_params)
    respond_to do |format|
      if @song.save
        format.html { redirect_to @song, notice: 'Song was successfully created.' }
        format.json { render action: 'show', status: :created, location: @song }
      else
        format.html { render action: 'new' }
        format.json { render json: @song.errors, status: :unprocessable_entity }
      end
    end
  end

  def update
    @song = Song.find(params[:id])
    respond_to do |format|
      if @song.update(song_params)
        format.html { redirect_to @song, notice: 'Song was successfully updated.' }
        format.json { head :no_content }
      else
        format.html { render action: 'edit' }
        format.json { render json: @song.errors, status: :unprocessable_entity }
      end
    end
  end

  def destroy
    @song = Song.find(params[:id])
    @song.destroy
    respond_to do |format|
      format.html { redirect_to songs_url }
      format.json { head :no_content }
    end
  end

  private
    # Never trust parameters from the scary internet, only allow the white list through.
    def song_params
      params.require(:song).permit(:title, :artist, :album, :year, :track, :genre)
    end
end
```

Figure 6. app/controllers/songs_controller.rb

```erb
<h1>Listing songs</h1>

<table>
  <thead>
    <tr>
      <th>Title</th>
      <th>Artist</th>
      <th>Album</th>
      <th>Year</th>
      <th>Track</th>
      <th>Genre</th>
      <th></th>
      <th></th>
      <th></th>
    </tr>
  </thead>

  <tbody>
    <% @songs.each do |song| %>
      <tr>
        <td><%= song.title %></td>
        <td><%= song.artist %></td>
        <td><%= song.album %></td>
        <td><%= song.year %></td>
        <td><%= song.track %></td>
        <td><%= song.genre %></td>
        <td><%= link_to 'Show', song %></td>
        <td><%= link_to 'Edit', edit_song_path(song) %></td>
        <td><%= link_to 'Destroy', song, method: :delete, data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Song', new_song_path %>
```

Figure 7. app/views/songs/index.html.erb

Problem: First consider this figure depicting the Rails MVC architecture.



Now, given the architectural diagram, think about how the web page in Figure 1 would have come to be displayed. Fill in each lettered item from the figure (blanks at left) the most appropriate label number (at right). Note that you will not use all of the label numbers.

a. _____

b. _____

c. _____

d. _____

e. _____

f. _____

g. _____

h. _____

i. _____

j. _____

k. _____

l. _____

m. _____

n. _____

o. _____

p. _____

q. _____

1) routes.rb (Figure 2)
2) song.rb (Figure 4)
3) 20140930033607_create_songs.rb (Figure 5)
4) songs_controller.rb (Figure 6)
5) index.html.erb (Figure 7)
6) Ye Olde Internet
7) Rails server
8) Web browser
9) Call to SongsController#index
10) Call to SongsController#show
11) Call to Song::all
12) Data returned by Song::all
13) Call to Song::find
14) Data returned by Song::find
15) Call to CreateSongs#change
16) Data returned from CreateSongs#change
17) Call to index.html.erb (whatever that means)
18) Data returned from index.html.erb
19) Invocation of SQL query
20) Data returned form SQL query
21) HTTP GET request
22) HTTP response
23) Database

Solution:

a. 8

b. 21

c. 1

d. 9

e. 4

f. 11

g. 2

h. 19

i. 23

j. 20

k. 12

l. 17

m. 5

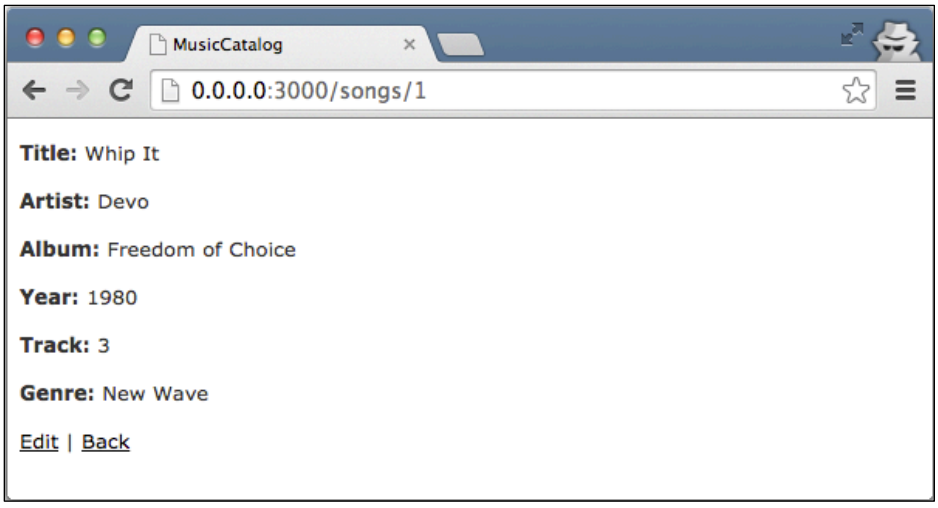
n. 18

o. 22

p. 6

q. 7

Problem: In Figure 1, if you were to click the "Show" link for "Whip It", this page would display.



Write the ERB file for this page. Assume that a layout, application.html.erb, already exists, so your ERB need only include the main content being displayed. Your ERB must include the following types of HTML elements: **p** and **strong**.

Solution:

It's OK to omit line 1.

```
1  <p id="notice"><%= notice %></p>
2
3  <p>
4    <strong>Title:</strong>
5    <%= @song.title %>
6  </p>
7
8  <p>
9    <strong>Artist:</strong>
10   <%= @song.artist %>
11 </p>
12
13 <p>
14   <strong>Album:</strong>
15   <%= @song.album %>
16 </p>
17
18 <p>
19   <strong>Year:</strong>
20   <%= @song.year %>
21 </p>
22
23 <p>
24   <strong>Track:</strong>
25   <%= @song.track %>
26 </p>
27
28 <p>
29   <strong>Genre:</strong>
30   <%= @song.genre %>
31 </p>
32
33 <%= link_to 'Edit', edit_song_path(@song) %> |
34 <%= link_to 'Back', songs_path %>
```

Problem: Modify the web app such that the page from Figure 1 includes only songs from 1980 or later. Here are a few hints:

- To create a new array:
  - `my_array = Array.new`
- To add an item to the end of an array:
  - `my_array.push(my_item)`
- To convert a string to an integer:
  - `my_int = my_string.to_i`

Solution:

Here's one straightforward way to solve the problem by changing SongsController#index (in songs_controller.rb):

```
1   class SongsController < ApplicationController
2     def index
3       # BEFORE:
4       #@songs = Song.all
5       #
6       # AFTER:
7       @songs = Array.new
8       Song.all.each do |song|
9         if song.year.to_i >= 1980 then
10          @songs.push(song)
11        end
12      end
13    end
```

(The rest of the file remains unchanged.)

Problem: Imagine that you wanted to change the web app such that it now stores the name of the song-writer with each song. Answer the following in plain English.

a. How would you go about updating the web app's "M" (as in MVC) component?

b. How would you change the "V" files in the above figures?

c. How would you change the "C" files in the above figures?

Solution:

a. To update the model ("M") component, you would need to create a new migration (similar to Figure 5). A common way to do this would be with this Rails command:

```
$ rails generate migration AddSongwriterToSongs songwriter:string
```

This command generates an appropriate migration file. Note that the class name after `migration` must be of the form `Add`*`Xxx`*`To`*`Yyy`*.

b. The view ("V") files above (i.e., the ERBs) would need to also display the songwriter values by adding appropriate HTML and calls to `song.songwriter`.

c. In the controller ("C") file above (song_controller.rb), the `song_params` method would need to be updated to account for the `:songwriter` parameter.