MVC:
Model View Controller

Lecture 16
Motivation

- Basic parts of any application:
  - Data being manipulated
  - A user-interface through which this manipulation occurs

- The data is logically independent from how it is displayed to the user
  - Display should be decoupled from content
  - Single-point-of-control over change

- Example: grade distribution in class
  - Could be displayed as a pie chart, or a bar chart, or a cumulative fraction plot, or...
Architecture: Desktop App

- User Interface
  - Graphical events (mouse moves, button pushed)

- Application
  - Processing, Calculating

- Data
  - Persistence, Transactions, Triggers
Model-View-Controller Pattern

- **Model**
  - The data (i.e., state)
  - Methods for accessing and modifying state

- **View**
  - Renders contents of model for user
  - When model changes, view must be updated

- **Controller**
  - Translates user actions (i.e., interactions with view) into operations on the model
  - Example user actions: button clicks, menu selections
Basic Interactions in MVC

Model

Controller

View

Input

“user action”

“change display”

“new state”

“change data”

Output
Implementing Basic MVC in Swing

- **Mapping of classes to MVC parts**
  - View is a Swing widget (e.g., JFrame, JButton, etc.)
  - Controller is an event listener (e.g., ActionListener)
  - Model is an ordinary Java class (or database)

- **Alternative mapping**
  - View is a Swing widget and includes (inner) event listener(s) as event handlers
  - Controller is an ordinary Java class with “business logic”, invoked by event handlers in view
  - Model is an ordinary Java class (or database)

- **Difference: Where is the event listener?**
  - Regardless, model and view are completely decoupled (linked only by controller)
Wiring Parts Together

```java
void actionPerformed(ActionEvent e) {
...
}
```

```java
void multiplyBy(String arg) {
...
}
```
public class CalcView extends JFrame {
    private JButton multiplyBtn = new JButton("X");

    public void register(ActionListener x) {
        multiplyBtn.addActionListener(x);
    }
}

public class CalcController {
    ...
    view.register(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            ...
        }
    });
}

Basic MVC in Objective-C
Implementing MVC in XCode
Implementing MVC in XCode
Basic Web App Skeleton: 3-Tier

User Interface

http

Application

HTML, CSS, Javascript

SQL

Data

http

HTML, CSS, Javascript

SQL

Data
MVC in a Web Application

- **Model**
  - Database (table with rows)
  - Classes that wrap database operations (class with instances)

- **View**
  - HTML (+ CSS, JavaScript) files rendered by client's browser
  - Skeleton files used by server to generate these HTML files

- **Controller**
  - Receives HTTP requests via web server
  - Orchestrates activity (model and view)
MVC with Rails
MVC with Rails
Directory Structure of Rails

depot/
 .... /app
 ........ /controllers
 ........ /helpers
 ........ /models
 ........ /views
 ............. /layouts
 .... /components
 .... /config
 .... /db
 .... /doc
 .... /lib
 .... /log
 .... /public
 .... /script
 .... /test
 .... /tmp
 .... /vendor
 .... README
 .... Rakefile
"Convention Over Configuration"

- Use naming & location conventions to wire components together *implicitly*
- Explicit routing too, based on *names* and pattern matching
- Contrast with:
  - Configuration files (eg XML)
  - Configuration code (eg Swing register listener)
  - Configuration tools (eg IDEs to connect GUI widgets to code snippets)
Wiring Parts Together in Rails

- **Example: Event → Controller wiring**
  - HTTP GET request for URL `/say/hello` gets routed to controller:
    - Class called `SayController`
    - File `say_controller.rb` in `app/controllers`
    - Method `hello`

- **Example: Controller → View wiring**
  - HTTP response formed from:
    - File `app/views/say/hello.html.erb`

- **Example: Model → Database wiring**
  - Class `Order` maps to database table "orders"
  - Attributes of `Order` map to columns of table
  - Instances of `Order` map to a rows of table
Summary

- **Programming Patterns**
  - Common idioms for solving categories of problems
  - Example: Observer pattern, MVC

- **Separation of concerns**
  - Decouple state from business logic
  - Decouple business logic from display

- **Rails: Convention over configuration**
  - Parts are wired together based on naming and structuring conventions
  - Defaults can always be overridden (but better not to fight!)
Rails:
Models
Recall: Rails Architecture

Ruby on Rails
Web Applications

HTTP, RSS, ATOM or SOAP

Apache, WeBrick or Lighttpd
Invokes FastCGI, mod_ruby or CGI processor

XML response
XHTML, CSS, JS & images, XML

Web Server
Forwards
Dispatcher
Loads

CRUDs
Redirects

Controller
Delegates
Renders

Action View
Displays
Responds

Action WebServices
Delivers

Action Mailer

Active Record
Queries
Data or Errors

Database
MySQL, PostgreSQL or Oracle

Handles some validations

Browser or client
Requests
Recall: Rails Architecture
Mapping Tables to Objects

- General strategy for OO languages
  - Table in database -- a class
  - Table columns -- attributes of the class
  - Table rows -- instances of class (objects)

- Application works with database using ordinary language syntax
  - Class methods for finding row(s) in table

- Example: Java POJOs, Rails models
Directory Structure of Rails

depot/
  ....../app
  ............/controllers
  ............/helpers
  ............/models
  ............/views
  ................../layouts
  ....../components
  ....../config
  ....../db
  ....../doc
  ....../lib
  ....../log
  ....../public
  ....../script
  ....../test
  ....../tmp
  ....../vendor
  ....../README
  ....../Rakefile
A Bit of Configuration

- Which database to use?
  - SQLite is the easiest (no setup!)
  - MySQL has better performance
  - PostgreSQL favored for Heroku deployment

- Different environments: development, test, production
  - Default is development (for rake command)

- See config/database.yml

```yaml
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```
Database Tables

- A database is a collection of *tables*
  - Naming convention: table names plural
- Each table has a list of *columns*
- Each column has a *name* and a *type*
- A table has a list of *rows*

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>fname</strong> <em>(string)</em></td>
<td><strong>lname</strong> <em>(string)</em></td>
<td><strong>buckid</strong> <em>(integer)</em></td>
</tr>
<tr>
<td>Marco</td>
<td>Pantani</td>
<td>22352022</td>
</tr>
<tr>
<td>Primo</td>
<td>Carnera</td>
<td>334432</td>
</tr>
<tr>
<td>Cher</td>
<td></td>
<td>34822039</td>
</tr>
</tbody>
</table>

**students**
Table Constraints

- Invariants on table entries beyond type information
  - "Iname is not null"
  - "buckid is unique"

- Often want a unique identifier for each row (a "primary key")
  - Easy: Include another (integer) column
  - Database responsible for assigning this value every time a row is added
  - No way to change this value after creation
## Primary Key With Autoincrement

<table>
<thead>
<tr>
<th>id</th>
<th>fname</th>
<th>lname</th>
<th>buckid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Marco</td>
<td>Pantani</td>
<td>22352022</td>
</tr>
<tr>
<td>3</td>
<td>Primo</td>
<td>Carnera</td>
<td>334432</td>
</tr>
<tr>
<td>4</td>
<td>Cher</td>
<td></td>
<td>34822039</td>
</tr>
</tbody>
</table>
Linking Tables

- Different tables can be related to each other
  - "Each student has exactly 1 major"
  - "Each student can own 1 (or more) vehicles"
- Keys are used to make this connection
  - Include a column in table X containing keys from table Y ("foreign keys")
  - For examples:
    - Student table includes a column identifying a student's major
    - Vehicle table includes a column identifying a (student) owner
- Association is an invariant between tables
### Association: Students & Vehicles

#### Students

<table>
<thead>
<tr>
<th>id (key)</th>
<th>fname (string)</th>
<th>lname (string)</th>
<th>buckid (integer)</th>
<th>major (foreign key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Marco</td>
<td>Pantani</td>
<td>22352022</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Primo</td>
<td>Carnera</td>
<td>334432</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>Cher</td>
<td>34822039</td>
<td>3</td>
</tr>
</tbody>
</table>

#### Vehicles

<table>
<thead>
<tr>
<th>id (key)</th>
<th>owner (foreign key)</th>
<th>license (string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>K3F 443L</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>F8L 220J</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>GOHBUX</td>
</tr>
</tbody>
</table>
# Associations

<table>
<thead>
<tr>
<th>id (key)</th>
<th>owner (for. key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

## Vehicles

<table>
<thead>
<tr>
<th>id (key)</th>
<th>major (for. key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

## Students

<table>
<thead>
<tr>
<th>id (key)</th>
<th>major (for. key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

## Programs

<table>
<thead>
<tr>
<th>id (key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>
Schema

- Definition of table structure
  - Table name
  - Column names and types
  - Constraints

- Usually database manager-specific

- See db/schema.rb for *Ruby-based* schema description
  - Allows independence from particular DB manager
  - Schema is versioned by timestamp (really by "migration"...)
Example schema.rb

ActiveRecord::Schema.define(:version => 20121025193013) do

  create_table "students", :force => true do |t|
    t.string   "name"
    t.integer  "buckid"
    t.datetime "created_at", :null => false
    t.datetime "updated_at", :null => false
  end

end
Migrations

Q. Who writes schema.rb?
A. It is generated! (tool called rake)
Golden rule: Never edit schema.rb directly
Instead, write a migration

A migration is Ruby code (a class) that represents a change in schema
- Create new tables (including column names and column types)
- Modify existing tables (adding/removing columns, or changing associations)
- Delete ("drop") existing tables
Migration Classes

- See db/migrate
- Filename consists of
  - Timestamp (UTC) of creation
  - Class name (descriptive of delta)
  - Example: class CreatePosts in 20121026180030_create_posts.rb
- Consequence: Migrations are run in a consistent order
  - Deltas do not commute, so order is important
- Class extends ActiveRecord::Migration
  - Contains method change
  - This method invoked by rake db:migrate
Example Migration Class

class CreatePosts < ActiveRecord::Migration
  def change
    create_table :posts do |t|
      t.string :name
      t.string :title
      t.text :content
      t.timestamps
    end
  end
end
### Result of raking this Migration

<table>
<thead>
<tr>
<th>:posts</th>
<th>:id</th>
<th>:name (string)</th>
<th>:title (string)</th>
<th>:content (text)</th>
<th>:created_at (time)</th>
<th>:updated_at (time)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>id</strong> (key)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Column Type Mappings

<table>
<thead>
<tr>
<th>Migration</th>
<th>Ruby</th>
<th>SQLite</th>
<th>Postgres</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>:boolean</td>
<td>Boolean</td>
<td>boolean</td>
<td>boolean</td>
<td>tinyint(1)</td>
</tr>
<tr>
<td>:date</td>
<td>Date</td>
<td>date</td>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>:decimal</td>
<td>BigDecimal</td>
<td>decimal</td>
<td>decimal</td>
<td>decimal</td>
</tr>
<tr>
<td>:float</td>
<td>Float</td>
<td>float</td>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>:integer</td>
<td>FixNum</td>
<td>integer</td>
<td>integer</td>
<td>int(11)</td>
</tr>
<tr>
<td>:string</td>
<td>String</td>
<td>varchar(255)</td>
<td>character</td>
<td>varchar(255)</td>
</tr>
<tr>
<td>:text</td>
<td>String</td>
<td>text</td>
<td>clob(32768)</td>
<td>text</td>
</tr>
<tr>
<td>:time</td>
<td>Time</td>
<td>time</td>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>:timestamp</td>
<td>Time</td>
<td>datetime</td>
<td>timestamp</td>
<td>datetime</td>
</tr>
</tbody>
</table>
Schema Deltas In Migrations

- In addition to creating tables, the change method can also
  - Modify columns of an existing table
    - `add_column`, `remove_column`, `rename_column`, `change_column`
  - Modify and delete tables
    - `change_table`, `drop_table`

- Example: `xxx_add_author_to_posts.rb`

```ruby
class AddAuthorToPosts < ActiveRecord::Migration
  def change
    add_column :posts, :author, :string
  end
end
```
Migrations as History

- Change defined by migration can be undone
  - Migrations give a *linear* history of deltas
  - Schema is the result of applying them (in order)
- Can move forward/backward in history
  - Create database only (no schema) defined in config/database.yml
    - $ rake db:create
  - Update schema.rb (ie compare its version number to list of migrations)
    - $ rake db:migrate
  - Rollback schema.rb to earlier point in history
    - $ rake db:rollback
  - Load schema defined in db/schema.rb
    - $ rake db:schema:load
Schemas, Migrations, Models

- `schema.rb`
- `migrations`
- `models`
- `database.yml`
- `db:create`
- `db:migrate`
- `db:schema:load`
Migrations vs Schema

- **Golden rule: Never edit schema.rb**
  - It is regenerated every time you do a migration
  - *Every* change in schema means writing a migration

- **Commit schema.rb to version control**
  - Deployment in fresh environment means loading schema, not reliving the full migration history

- **Commit migrations to version control**
  - Once a migration has been shared, to undo it you should create a *new* migration (preserve the linear history)
Models

- Programmatic way for application to interact with database
  - Collection of Ruby classes
  - Extend ActiveRecord::Base
  - Found in app/models

- Each class corresponds to a table
  - Note: Models are *singular* (tables are *plural*)
  - Include attributes (corresponding to columns)
    class Post < ActiveRecord::Base
      attr_accessible :content, :name, :title
    end
Class Methods for Models

- Create a new instance with new
  ```ruby
  p1 = Post.new
  p2 = Post.new author: "Xi", title: "Hola"
  □ Warning: this only creates the model (object) it does not modify the database
  ```

- Create instance and add it to database
  ```ruby
  p3 = Post.create author: "Zippy"
  ```

- Retrieve a particular row from table
  ```ruby
  @post = Post.find 4 # or Post.find_by_id
  @post = Post.find_by_author "Xi"
  @blog = Post.find :all # or Post.all
  @student = Student.find_by_buckid 543333
  @post = Post.first
  @post = Post.last
  ```
Object Methods for Models

- To save a model (object) as a row in the database
  ```ruby
  p = Post.new author: 'Xi'
p.save #commits change to database
  ```
- Read/write attributes like an ordinary Ruby class
  ```ruby
  @post = Post.find_by_author 'Xi'
t = @post.title #=> nil
  @post.title = 'A Successful Project'
  @post.save #don't forget to save
  ```
- To force saving through to database
  ```ruby
  @post.update_attribute author:, "Me"
  ```
  % Deprecated! Bypasses validation (coming)
- To delete a row from the table
  ```ruby
  @post.destroy #no save needed
  ```
Summary

- **Databases: Tables, columns, rows**
  - Structure defined in a schema
  - Rails uses Ruby code to generate schema

- **Migrations**
  - Ruby code describing change to schema
  - Syntax look declarative

- **Models**
  - Ruby classes that mirror database tables
  - Class names from table (singular vs plural)
  - Attributes from columns

- **Code generation**
  - Schema generated by schema.rb
  - Schema.rb generated by rake on migrations
  - Migrations and models generated by rails generate