COMP 121
Software Engineering

Software Architecture

Spring 2021

(Inspiration from Ben Liblit)
Introduction

- **Software architecture** is the high-level structure and organization of a software system
  - The “big picture” or the “blueprint”: What are the components of the system, and how they fit together

- Useful reading on software architecture

- Let’s learn about one example architecture
  - Model-view-controller, a way of separating user interface from app’s domain logic

- Then we’ll talk more about architecture in general, and examine several more architectural styles
Model-View-Controller (MVC) Arch.

Model (database)
- reads
- reads/updates

View (e.g., html)
- renders

Controller (business logic)
- sends request
- renders response

User (e.g., browser)
- request response

User sends request to Controller, which processes the request and sends a response. The Model reads from or updates the database, and the View renders the response for the User to see.
Example: Ruby on Rails

- Rails is a web app framework written in Ruby
  - Developed by David Heinemeier Hansson as part of Basecamp; released separately as Rails in 2004
- MVC framework
  - Model = database (sqlite, mysql, postgres, etc)
  - View = `.html.erb` files, i.e., html with embedded Ruby
  - Controller = methods that handle web requests
- Side note: real-world apps include many languages
  - Ruby, HTML, CSS, JavaScript, SQL, …
- Very quick tour of Rails next, with some code
  - Learn more at https://guides.rubyonrails.org
Sending a Web Request

• Browser sends a request for a web page

```
$ nc -c www.cs.tufts.edu 80
GET / HTTP/1.0
```

HTTP/1.1 200 OK
Date: Mon, 18 Feb 2019 20:57:47 GMT
Server: Apache/2.2.15 (Red Hat)
X-Powered-By: PHP/5.3.3
Content-Length: 848
Content-Type: text/html; charset=UTF-8
Connection: close

<html>
<head>
<title>Tufts University ECE and CS Departments</title>
</head>
</html>
```
Rails Server Internal Sequence

• Server receives a request
• It first routes the request to a controller method
• That method accesses the db using models
• When the controller is done, it renders a view
• The view file is sent back to the web browser

• Note: HTTP is stateless
  ▪ Each request connects, gets result, drops connection
  ▪ Web server stores state in db and in browser cookies
  ▪ Servers and OSes play a lot of tricks to avoid making so many connections
Rails Models

• Examples from talks, a web site for displaying a list of talks in a CS department

```ruby
# db/schema.rb
create_table "talks" do |t|
  t.text "title"
  t.text "abstract"
  t.text "speaker"
  t.integer "owner_id"  # talk creator
end
```

```ruby
# app/models/talk.rb
class Talk < ActiveRecord::Base
  validates_presence_of :owner  # an invariant!
end
```
Rails Routing: URLs to Methods

```ruby
# config/routes.rb
Talks::Application.routes.routes.draw do
  resources :talks
end
```

```
$ rake routes
  talks  GET  /talks(.:format)  talks#index
  POST   /talks(.:format)  talks#create
  new_talk  GET  /talks/new(.:format)  talks#new
  edit_talk  GET  /talks/:id/edit(.:format)  talks#edit
  talk  GET  /talks/:id(.:format)  talks#show
  PATCH  /talks/:id(.:format)  talks#update
  PUT    /talks/:id(.:format)  talks#update
  DELETE /talks/:id(.:format)  talks#destroy
```

- A route maps an HTTP verb and URL to a method
  - Example: GET /talks/12 calls TalksController#show
Rails Controllers

• Receive a request

```ruby
# app/controllers/talks_controller.rb
class TalksController < ApplicationController
  def show
    # params maps :id to the id in the URL
    # Talk.find does a db query
    # @f for any f is a field (instance variable)
    @talk = Talk.find(params[:id])
    # notice the method just returns nothing!
  end
end
```
Rails Views

```
# app/views/talks/show.html.erb
<div class="center-header">
  <div class="talk">
    <div class="title"><%= @talk.title %></div>
    <div class="speaker"><%= @talk.speaker %></div>
  </div>
<div class="abstract">
  <% if @talk.abstract == "" %>
    <span class="title">No abstract</span>
  <% else %>
    <span class="title">Abstract</span>
    <div class="abstract-body"><%= @talk.abstract %></div>
  <% end %>
</div>
...
```

- Didn’t need to def Talk#title and Talk#speaker
  - Implemented using reflection and knowledge of db!
MVC Pros and Cons

- Separation between data and interface is key
  - Views can be replaced, changed, customized, expanded
  - As db is read and written, changes reflected in all views
    - Centralized store of “truth” for the system state
  - Scalable deployment easier, e.g., multiple controller/view instances communicate with one db
    - System is quiescent when controller method returns

- Several potential drawbacks
  - Many kinds of changes to model require changing view and controller
    - E.g., removing a db column, changing the name of a table
  - Views and controllers are closely coupled
  - Added complexity
    - E.g., even simple Rails app has many files in many locations
What is Software Architecture?

- Tends to be more *abstract* than any coding feature
  - E.g., it may talk about things that are represented by sets of classes rather than a single class
- Is *hard to change* after building a system!
- Helps guide *division of work* by different developers
- Includes *decisions, principles, and vision* that led to the design
  - Informs later decisions as the system evolves
  - (Design patterns are smaller scale than architectures)
Pipe and Filter Architecture

- Each component has inputs and outputs
  - Component reads input stream, produces output stream

- Components are *filters*, connections are the *pipes*
- A *stream* is a sequence of data of unknown length

- Key design properties/questions
  - Filters should not share internal state
    - Filters don’t know how they’re connected, only the pipes do
  - At any joins, data from pipes needs to sync up
  - Filters and pipes have to agree on input/output data types
Pipe and Filter: Unix Commands

• Ex: Count `httpd` instances running (off by one)

  ```sh
  ps -ef | grep httpd | wc -l
  ```

• Commands take ASCII chars as input
  ■ What about unicode?

• Every command has `standard in` and `standard out`
  ■ But there’s also `standard error`; where does that go?
  ■ Normally to stdout, but you can redirect it

    ```sh
    # send both stdout and stderr to file.log
    ./script.sh > file.log 2>&1
    ```

• One command can launch another
  ■ See `pipe`, `fork`, and `exec`* C library functions
  ■ Most languages have a library to make these easier to use
Using Pipe and Filter in Java

• Standard input, output, and error are standard

```java
class System {
    static PrintStream out; // stdout
    static InputStream in; // stdin
    static PrintStream err; // stderr
}
```

• To launch subprocesses, use `ProcessBuilder`

```java
ProcessBuilder pb = new ProcessBuilder("ls");
Process p = pb.start();
p.waitFor();
InputStream is = p.getInputStream(); // output of p!
BufferedReader br =
    new BufferedReader(new InputStreamReader(is));
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```
Pipe and Filter? A Compiler

- A compiler is a sequence of transformations
  - Source text converted to *tokens* and then parsed to yield AST
  - The AST becomes a control-flow graph (CFG), which is successively simplified
  - Ultimately the CFG is simplified so much it can be output as an assembly or machine code file
- Except, all stages share state (e.g., symbol table)
Pipe and Filter Advantages

• Overall behavior is a composition of filter behaviors
  ▪ Component connections are significant and obvious
• Potentially good reuse by creating different compositions of filters
• Can test each filter in isolation
• Filters can be replaced individually
Pipe and Filter Disadvantages

• Not good for interactive use
  ▪ Focused on converting inputs to outputs, not supporting user interactions

• Pipes are narrow; hard to pass complex data
  ▪ E.g., compilers not really pipe-and-filter

• Overhead for parsing/unparsing data when read from/sent to a pipe
  ▪ Though, components that are conceptually pipes could run as part of the same process
  ▪ In which case they could pass data structures through “pipes”
Layered Architecture

• Organized as a hierarchy
  ▪ Layer provides services to layer above it
  ▪ Layer is a client of the layer below it

• Example: running a Java program

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Operating system (kernel)</th>
<th>Java virtual machine</th>
<th>Java program</th>
</tr>
</thead>
</table>

19
Open Systems Interconnection Model

• “Network stack” or “protocol stack”

<table>
<thead>
<tr>
<th>Layer 7: Application (HTTP)</th>
<th>(the actual application)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer 6: Presentation</td>
<td>(encoding, compression, crypto)</td>
</tr>
<tr>
<td>Layer 5: Session</td>
<td>(sequences of communication)</td>
</tr>
<tr>
<td>Layer 4: Transport (TCP)</td>
<td>(segmentation, acks, multiplex)</td>
</tr>
<tr>
<td>Layer 3: Network (IP)</td>
<td>(addressing, routing, etc)</td>
</tr>
<tr>
<td>Layer 2: Data link (Ethernet)</td>
<td>(sending data frames between nodes)</td>
</tr>
<tr>
<td>Layer 1: Physical (IEEE 802.3u)</td>
<td>(sending raw data over wires/radio/etc)</td>
</tr>
</tbody>
</table>
LAMP Stack

<table>
<thead>
<tr>
<th>PHP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
</tr>
<tr>
<td>Apache</td>
</tr>
<tr>
<td>Linux</td>
</tr>
</tbody>
</table>

- Slightly old-school web server structure
Layered Architecture Tradeoffs

• Advantages
  ▪ Good fit for system with increasing levels of abstraction
  ▪ Changing one layer affects at most two others
  ▪ Can interchange implementations of same layer interface

• Disadvantages
  ▪ May not be able to identify clean layers
  ▪ Might need to jump layers for performance or functionality
Client-Server Architecture

- Clients communicate with server, typically over network
- Tradeoffs
  - Server is central point of failure
    - Replication can help, but then consistency is
    - CAP theorem: Pick two of consistency, availability, and partition tolerance
  - Any client-to-client communication must go through server
Peer-to-Peer Architecture

- Machines communicate with each other
  - Popularized by Napster (!), 1999

- Several challenges/tradeoffs
  - Trust between nodes
  - More equal upload/download volume compared to client server
  - Location of data on network not centralized
Software Architecture Activities

• Initial design/development/evaluation

• Maintain architecture over time
  ▪ Architectural drift — implementation decisions that aren’t encompassed by the architecture, but don’t conflict with it
  ▪ Architectural erosion — implementation decisions that actually violate the architecture
  ▪ What to do?
    - Change the architecture or change the code!

• Evolve architecture as requirements change

• Key challenge: Architecture is not code, hence it will inevitably drift from the code