Software Design

Profesora Mikaela Grace

Background: Profesora Mikaela

Background: Profesora Mikaela

Stanford University

UNDERGRAD
Mathematical and Computational Sciences
GRADUATE
Theoretical Computer Science
THESIS
Detection of emotion in music

Background: Profesora Mikaela

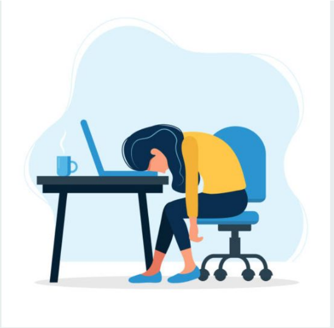
UNDERGRAD
Mathematical and Computational Sciences
GRADUATE
Theoretical Computer Science
THESIS
Detection of emotion in music

SPEECH
Acoustic Modeling
Internationalization
Research

Background: Profesora Mikaela

Stanford University

UNDERGRAD
Mathematical and Computational Sciences
GRADUATE
Theoretical Computer Science
THESIS
Detection of emotion in music

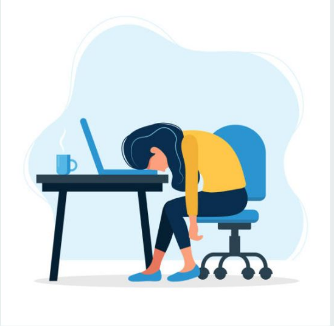


STARTUPS
River: Machine Learning Lead
Altis: NLP Engineer, Advisor
Miai: Interim CTO

SPEECH
Acoustic Modeling
Internationalization
Research

Background: Profesora Mikaela

UNDERGRAD
Mathematical and Computational Sciences
GRADUATE
Theoretical Computer Science
THESIS
Detection of emotion in music



STARTUPS
River: Machine Learning Lead
Altis: NLP Engineer, Advisor
Miai: Interim CTO

SPEECH
Acoustic Modeling
Internationalization
Research

TEC
I’m new here!
(Kind) feedback plz

Software Architecture: Course Intro

- Comprehensive introduction to software architecture and design
- We will cover:
  - Object Oriented Programming (OOP)
  - Design Patterns for code
  - Architectures for common problems

Software Architecture: Course Intro

- Expectations:
  - Quietly listen when I’m lecturing
  - Ask questions!
  - Best-effort and engagement – be here to learn :)

Course Tools

- Excalidraw
  - Or lucidchart
- Python
  - Download pycharm (preferred), VSCode, or Thony
- Learn the basics of vim
- Github
  - Optional, but helpful
- AWS

Course Structure

- Lectures with in-class exercises
- Spanglish
- Grade based upon:
    - Attendance and participation (in-class exercises)
    - Final Exam
    - Project Work
        - I will review your projects and give feedback
        - You incorporate that feedback and re-submit
        - More soon

Defining the Course: Functionality vs Architecture

- Functionality is different than architecture! Though architecture can affect functionality.

  - **Functionality**: what needs the system fulfills. What it “does”.
  - **Architecture**: how the system is structured in order to fulfill those needs

- When the architecture is poorly matched to the functionality, developers will struggle against it.

- Real-world example: a building’s functionality vs its design

Software Architecture: Changing your Viewpoint

- Basketball analogy:
  - Player view
- On the court
  - From his/her/their view, makes decisions about where to pass, whether to shoot, etc

Software Architecture: Changing your Viewpoint

- Basketball analogy:
  - Coach view
- Coach has a set of mental abstractions that allow her to:
  - Convert her perceptions of raw phenomena
    - such as a ball being passed
  - into a integrated understanding
    - such as the success of an offensive strategy
  - She thinks in high-level components
    - NOT low-level individual decisions

Challenges of Architecture

- You just joined a new company, and the codebase is 900,000 lines of code.
  - You haven't read every one of the 900,000 lines
- You need to add a new feature.
  - How can you build on top of what's already there?
  - How do you understand how to do this?
- I'm implementing a new feature. That will add a dependence on another part of the system.
  - Is that ok? What are the consequences of introducing this dependency here?

Challenges of Architecture

- How can you find abstractions that let you focus on the right level of detail and reduce the amount of work?
- How can design solutions be described, generalized, and shared?
- How do we describe and refer to common architectural styles?

Software Architecture: This lecture

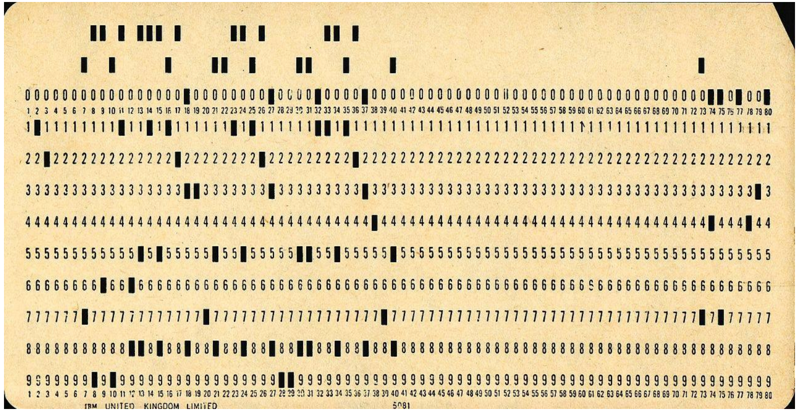
- Brief history of architecture
- Example of differing architectural choices
- Overview of design choices

Software Architecture: History

- Decade after decade, software systems have seen orders-of-magnitude increases in their size and complexity.
- Increases in software size and complexity have been matched by advances in software engineering.
    - Assembly language programming -> higher-level languages and structured programming.
    - Procedures -> objects.
    - Software reuse: subroutines -> extensive libraries and frameworks.

Punch Cards (1970s)

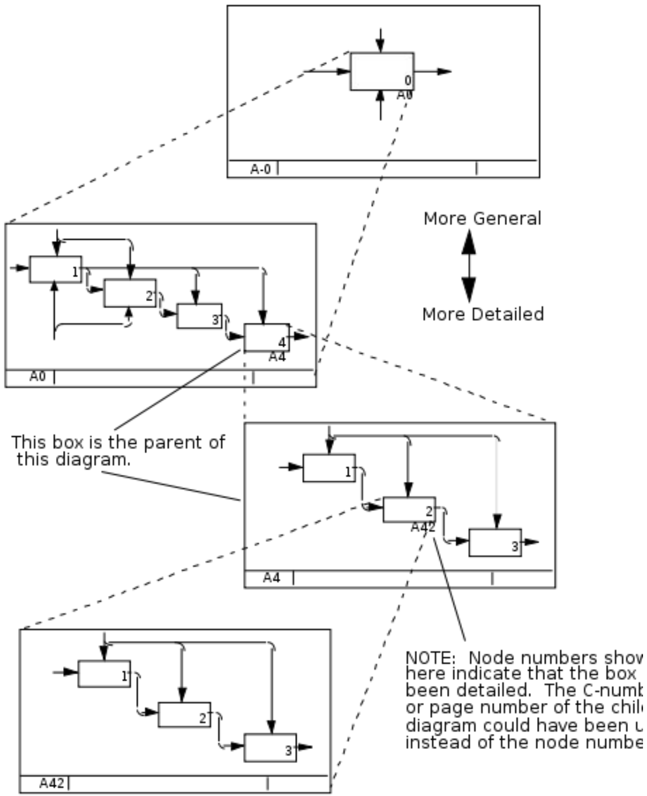
- Code written on punch cards with holes in them
- Changing the code means printing a new punch card.
- Organization of code very primitive.



http://www.columbia.edu/cu/computinghistory/fisk.pdf

Structured Design (1960s -70S)

- Can use representations beyond software to describe structure of code or program
- First "architectural diagrams"
- Make intentional choices about structure in order to make software easier to modify and maintain



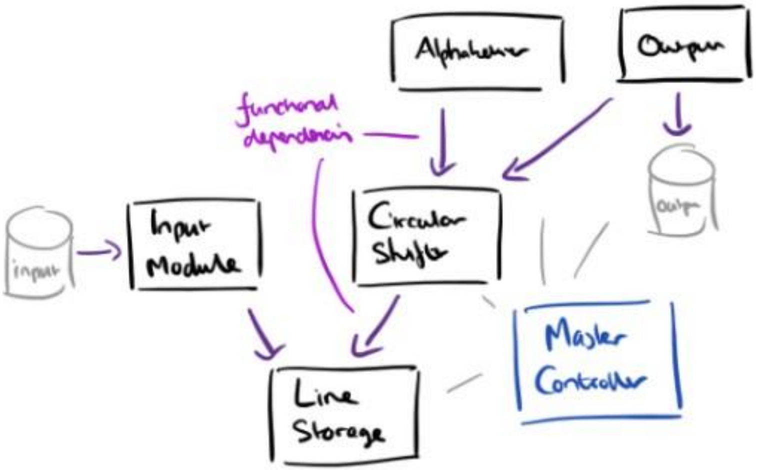
Structured Design (1960s -70S)

- In 1968 **Edsger Dijkstra** wrote a now famous letter titled “GOTO Considered Harmful”
  - He argued that GOTO statements complicate the reasoning about runtime execution
    - They make it much harder to read the code and understand what it’s doing.
    - Best to avoid GOTO statements
  - Looking back at this debate today, it is hard to imagine disagreeing.But at the time, it was controversial
    - Developers were accustomed to working within the old set of abstractions.
    - They focused on the **constraints** of the new abstractions rather than the **benefits**

```
#include <iostream>
using namespace std;

int main () {
   // Local variable declaration:
   int a = 5;

   // do loop execution
   LOOP:do {
      if( a == 15) {
         // skip the iteration.
         a = a + 1;
         goto LOOP;
      }
      cout << "value of a: " << a << endl;
      a = a + 1;
   }
   while( a < 10);

   return 0;
}
```

Information Hiding (1970S)

- Software contains design decisions which may change
- Code made more maintainable by **hiding** design decisions in module
- Can change some decisions, without that change rippling outward and causing changes to dependencies
- Encapsulation and interfaces



Architectural Styles (1990S)

- Structural constraints on elements and element relationships can be codified as architectural styles

- Any system following an architectural style has specific properties inherent to the architectural style
  - Allows for re-use and easier communication

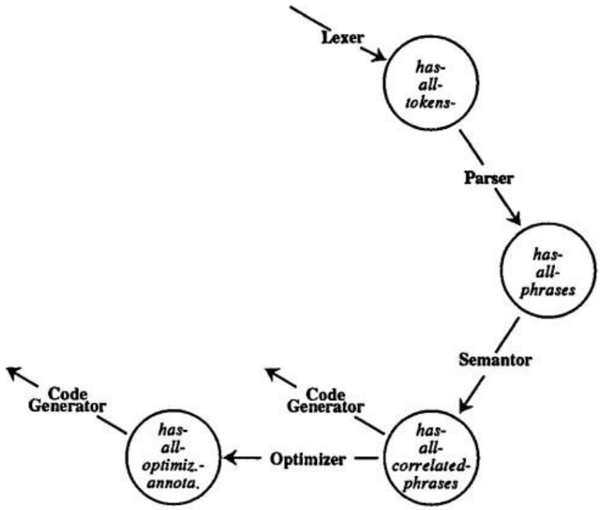
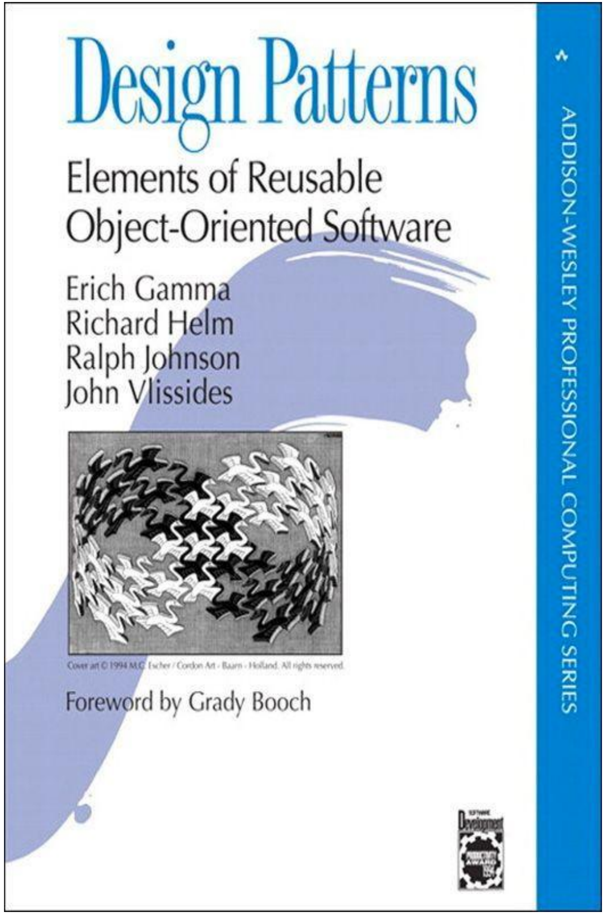


Figure 3: Data View of Sequential Compiler Architecture.

Perry and Wolf. (1992). Foundations for the study of software architecture. FSE.

Design Patterns (1990S)

- Reusable solution to a problem in a context
- Rather than solving problems from scratch, experts borrow existing solutions to common design problems.
- Giving them names allows them to be recognized and taught
- We will cover many modern design patterns in this class



Agile Software Development (2000S)

- Architecture built upfront can be mismatched to goals, particularly in ever-changing tech startup environment
- Software should be flexible enough to accommodate those needs

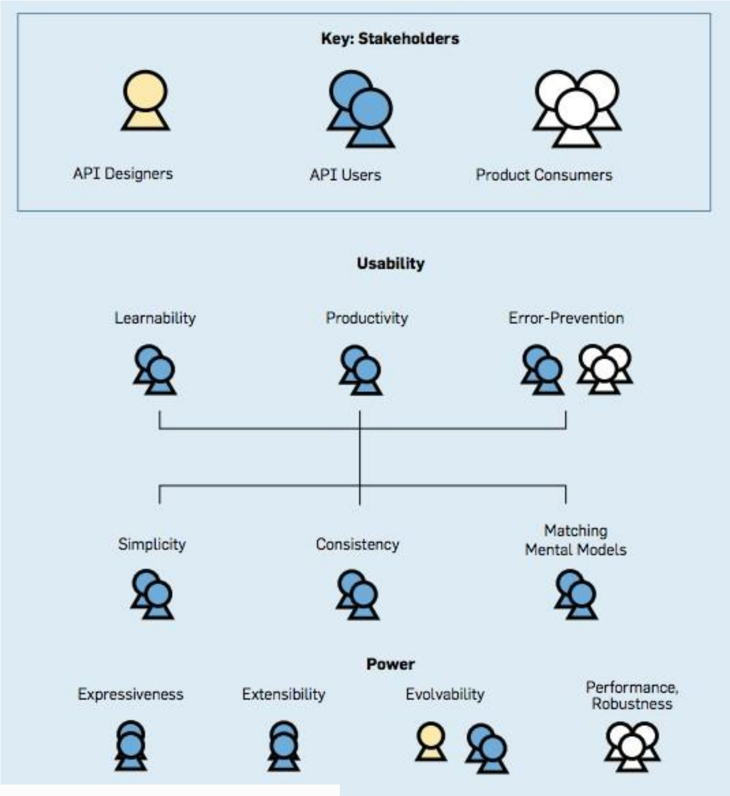
http://agilemanifesto.org/

APIS (2000S)

- Web increased availability and number of libraries and frameworks, often as free open source projects

- APIs became the way that services interacted with each other.

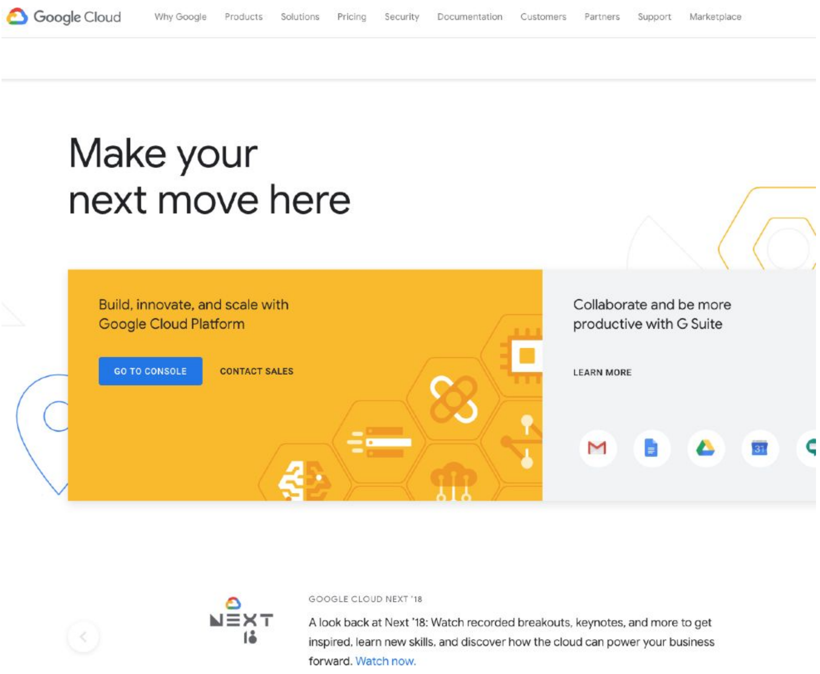
- What happens when we need to change an interface that others depend on?



Myers & Stylos, [Improving API Usability], CACM 59 (6), 2016

Software Ecosystems (2010S)

- Businesses expose web services
- Market for for software and services
- APIs create value for organizations
- Systems of systems, where no single owner controls the design of the system from end to end
- Work more distributed, through crowdsourcing, hackathons, bug bounties



Software Architecture: Example

- **What follows is a description of three systems with the same functionality, yet different architectures.**
    - Rackspace is a real company that manages hosted email servers.
    - Customers call up for help when they experience problems.
    - To help a customer, Rackspace must search the log files that record what has happened during the customer's email processing.
    - Because the volume of emails they handle kept increasing, Rackspace built three generations of systems to handle the customer queries

Software Architecture: Example

- **Version 1: Local log files.**
  - Each service on each email server writes to a separate log file.
  - To answer customer inquiry, execute grep query on that server.
- Challenges
  - As system gained users, overhead of running searches on email servers increased.
  - Required engineer, rather than support tech, to perform search

Software Architecture: Example

Version 2: Central database.

- Every few minutes, log data sent to central server and indexed in relational database
- Support techs could query log data through web-based interface
- Challenges
  - Hundreds of servers constantly generating log data --> took long to run queries, load data
  - Searches became slow; could only keep 3 days of logs
  - Wildcard searches prohibited because of extra load on server
  - Server experienced random failures, was not redundant

Software Architecture: Example

- **Version 3: Indexing cluster.**
  - Save log data into distributed file system (Hadoop)
  - Indexing and storage distributed across 10 commodity machines, parallelized
  - Index results about 15m stale
  - All data redundantly stored
  - Indexed 140 GB of log data / day
  - Web-based search engine for support techs to get query results in seconds
  - Engineers could write new types of queries
    - exposed to support techs through API

Software Architecture: Example

Comparing the three systems

- They all have roughly the same **functionality** (querying email logs to diagnose problems) yet they have different **architectures**.
- Their architecture was a separate choice from their functionality.
- This means that when you build a system, you can choose an architecture that best suits your needs, then build the functionality on that architectural skeleton.
- What else can these systems reveal about software architecture?

Software Architecture: Example

Despite having the same functionality, the three systems differ in three **Quality attributes: modifiability, scalability, and latency.**

- Ease of modifiability
  - V1 and V2 supported ad hoc queries in seconds by writing a new grep expression or changing SQL query
  - V3 required a new program to be written to build a new query type
- Scalability
  - V3 more scalable
- Liveness/Freshness of results
  - V1 always got latest results, V3 short delay

Software Architecture: Example

- Trade-offs. There was no free lunch: promoting one quality inhibited another.
  - V3 had scalability, but that cost it in terms of modifiability and latency.
  - V1 was flexible and low-latency, but not scalable.
  - **Scalability, latency, modifiability, etc., usually trade off against each other.**
    - Maximizing one quality attribute means settling for less of the others.

Software Architecture: Example

- **Abstractions and constraints.**
- In software, bigger things are usually built out of smaller things.
- You can always reason about the smaller things in a system (like individual lines of code) but usually you will find it more efficient to reason about the larger things (like clients and servers).
- For example, the third system in the Rackspace example scheduled jobs and stored data in a distributed file system.
  - Easier to reason about “jobs” than “individual lines of code”

Examples of Quality Attributes

- **Latency**: how fast is the system
- **Scalability**: how well does adding more computing resources translate to better performance
- **Maintainability**: how hard is system to change
- **Reliability**: how likely is the system to be available
- **Extensibility**: in what ways can new components be added without changing existing components
- **Portability**: in what environments can the system be used
- **Testability**: how easy is it to write tests of the system's behavior

Architecture: Defining your requirements

- Lists of requirements and features systems should include
  - Defining your functionality
  - Also: what does it NOT include?
  - Implicit requirements:
    - Shipping to a customer in a marketplace – what requirements feed into that?
- List of quality attributes by which to compare alternative designs
  - both of which offer the same functionality

Risk-Driven Architecture

- My father has a degree in mechanical engineering and does various projects at home and at work
    - Mailbox
    - Gas tank
- Your effort and architectural complexity should be commensurate with the risks that failure brings

Risk-Driven Architecture: The risk of Rewriting

- Team spent 1 year building v1, decided to throw it away and build v2.
  - How can this be avoided?
- What risks cause software to need to be rewritten to meet its requirements?
  - Changes in latency requirements, amount of traffic

Extensibility

- **Change** is the only constant thing in a programmer's life.
    - You released a video game for Windows, but now people ask for a macOS version.
    - You created a GUI framework with square buttons, but several months later round buttons become a trend.
    - You designed a brilliant e-commerce website architecture, but just a month later customers ask for a feature that would let them accept phone orders.

- There's a bright side: if someone asks you to change something in your app, that means someone still cares about it.
- That's why all seasoned developers try to provide for possible future changes when designing an application's architecture

Excalidraw and Architecture Diagrams

- UML
  - Important to have standardized ways of representing information
    - Class Diagrams: structure of classes
    - Interaction Diagrams: one use case
  - Book
  - UML quick guide
- Excalidraw.com
- Lucidchart

Python Style

- **https://peps.python.org/pep-0008/**
- https://google.github.io/styleguide/pyguide.html
-