Software Architecture (Just Enough)

1. Analyzing Software seems informal
2. FineBugs, Lint (Looks at 'C' program) - Formal Analysis

We can recognize cars on the road, know whether they're hybrid/not-hybrid. Correlate this with software where we should be able to recognize on whether this is a 2-tier system / 3 - tier system.

Views - use the right view to analyze the metric that you are looking to improve/scale.

Quality Attributes (Non-functional requirements)

Performance, Scalability, Modularity, Usability, Security
Optimizing across many QA's, cannot maximize all
Design Space - Choose a point that enhances: Fast/Cheap/Secure

Eg: Denormalize DB's - Why? Redundant data, makes retrieval faster but maintenance becomes costly.

2FA - Tradeoff between security & (speed/convenience)

Architecture Driver

Stimulus - Agent / Situation that triggers
Response - Reaction to stimulus

QA Scenario Grading

Importance to stakeholder - low, medium, high
Implementation difficulty - low, medium, high

A problem could thus be rated as (H,H), or (L,H) or (H,L) and based on these decisions need to be made on which QA should be looked at.

Figuring out single points of failures

Related to programming language choice ?? Most likely, no!
Related to dependency between modules ?? Can be!
Number of copies running, distributed system ??

What kind of diagram would be helpful for figuring out single point of failures?

If we were looking at security. Choose the view that helps this. How can the bad guy get into the system?
Constraints on given projects:

Must run on 32 MHz CPU, must use particular DB schema, stick to a particular programming language. ---> Annoying! But required.

3-tier system

User Interface → Business Logic → Data Logic
Cache things in RAM in the middle tier to make system faster.
UI goes directly to DB will return stale data. It needs to be consistent.

UI never talks directly to DB.

Idempotency: Divide a lot of work across a lot of machines. If the machine dies in the middle of doing it. Eg: resizing photos, push out photos to machines, resize and they hand it back.

Each one of the instructions is idempotent. So the same job could be repeated and the expected result would be the same. This happens if we perceive a system to have failed.

Examples of Architectures (QA's)

- Plugins must use cross-platform APIs to read files → Portability (XBOX media center plugins)
- EJBeans must not start own threads → Manageability
- EJBeans must not write to local files → Distribution

MapReduce

Example of how it works
Spider crawling web-pages. Look at all the words in that web-page. Return a list of words with frequencies. Partial indexes get combined and make a combined index.

What is it actually?
Bunch of work → Chunks of work → Taking all partial indices and making a master index

Architectural Style Types

Client - Server Model (Obvious Constraint: Server cannot request anything off the client)
N - tier System
MapReduce

Conceptual Model

Canonical Model Structure

1. Domain Model
2. Boundary Model
3. Internals Model
4. Code Model

Just enough software architecture - How much do we actually need to know!

Consider failures → Analyzing Options → Designing a solutions

Website building

Problems
Broken Links
Cross-browser Compatibility
Hard-drive crashes - DB failures
Click Latency (Bad Queries)

Choices
When design happens / What analyses happen / Precision / Depth

Existing Architecture Techniques

Protocol Analysis (Client- Server communication)
Component and Connector Modelling (MVC)
Queuing Theory
Schedulability Analysis

Map from Risks → Techniques

- Identify and prioritize risks
- Apply relevant architecture activities
- Re-evaluate

Rackspace Email Client Server Problems of nailing down errors from the logs

1. Architecture was distributed system, so to get the errors all logs needed to be scanned.
2. V2: Adding a MYSQL DB that used to store all data in indexed/pre-processed format at every interval (Con: Latency between error and time between next storage)
3. V3: Streamed data into distributed file system, Map Reduce job which indexed the files every certain interval.