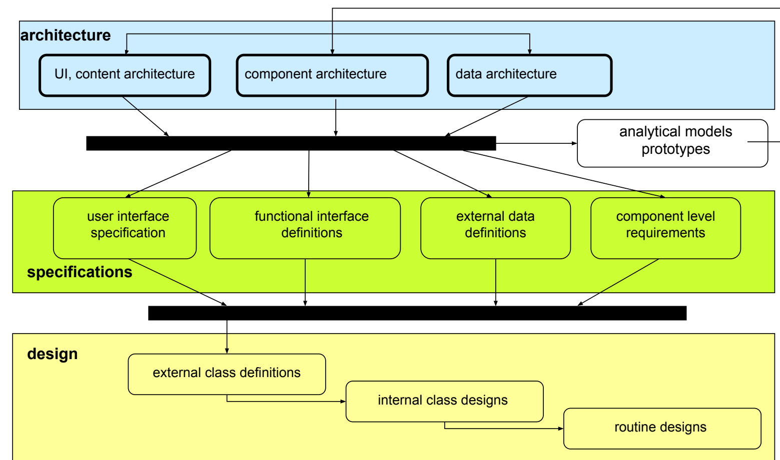
System Architecture (how to)

- who needs how much architecture
- the process of evolving an architecture
  - the nature of the process
  - getting started
  - evaluating and evolving an architecture
  - approaches to addressing hard problems
- describing an architecture
  - challenges and approaches

Model Hierarchy/Succession



Who needs architecture?

- Do you need modular sub-components
  - to decompose a hard problem?
  - to enable independent development?
  - to enable component reuse?
  - to enable heterogeneous interoperability?
  - to enable future enhancements
  - to enable independent delivery/replacement?
- If so, you need an architecture
  - to describe the components & interaction
- If not, move on to specification and design

How much architecture?

- enough to enable assessment
  - will system be able to meet its requirements?
  - how much work will take to build it?
  - what are the likely difficulties and problems?
  - how will it perform?
  - how will it handle errors?
  - how will it accommodate expected growth?
- enough to enable construction
  - each sub-group knows what they have to do
  - pieces are likely to work when combined

A “Wicked” Problem

- some problems come from requirements
  - complex specifications, tight constraints
  - these are usually clear from the start
- some problems come from solutions
  - new requirements created by our approach
  - complexities inherent in our approach
  - costs and weaknesses of our approach
  - these emerge as we elaborate our design
- starting with a solution may not work
  - start by understanding the problems

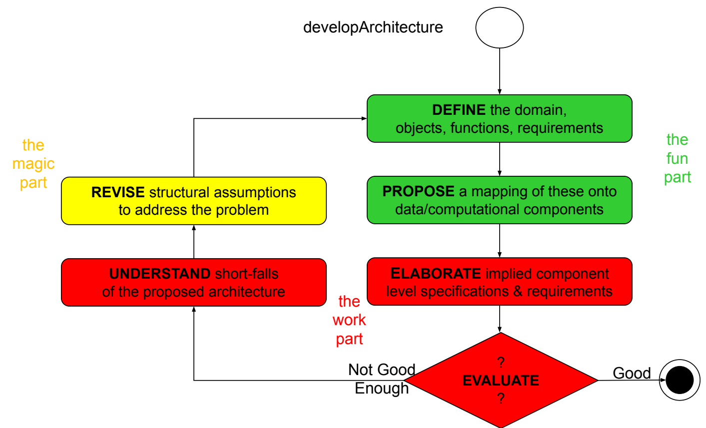
The Architectural Process

- it is seldom one-pass or purely top-down
- it is iterative, “constructive” decomposition
  - our understanding evolves with each iteration
- it usually involves compromise
  - accepting limitations to simplify problems
  - finding a middle-path between conflicting goals
  - to meet time, resource, risk constraints
- the best solutions are often non-obvious
- architecture is <u>research</u>, not <u>development</u>

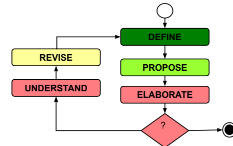
Top-Down vs Bottom-Up

- Top-Down Design (divide and conquer)
  - propose key high level components
  - recurse until implementations are obvious
- Bottom-Up Design (crawl, walk, run)
  - define key building blocks (solve basic probs)
  - use those to design higher level services
  - continue until we have a complete solution
- similarities:
  - both start with the same overall reqts
  - both result in a multi-level architecture
  - neither is done until it all works

Developing an Architecture

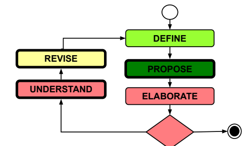


Problem Definition



- identify the types of information involved
  - basic data objects, general contents of each
- identify the primary computations involved
  - with their associated inputs and outputs
- identify the key requirements on each
  - what operations does each have to support
  - performance, manageability, persistence,
  - correctness, reliability, availability

Architectural Proposal



- **PROPOSE** major execution components
  - what types of processing each performs
- **PROPOSE** major data components
  - what types of information each contains
- **PROPOSE** external interfaces
  - type of interface, supported operations

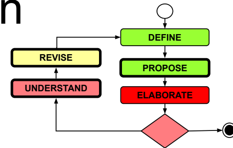
Start with the “Givens”

- We very seldom start w/blank paper
  - requirements may call out standards
    - standard communications protocols
    - existing products with which it must inter-operate
  - organizations may mandate technology
    - standard frameworks & management models
  - obvious Off the Shelf components & kits
    - internal, commercial, open-source
- Start your architecture with these “givens”
  - the missing pieces are what we must define

Filling in the Gaps

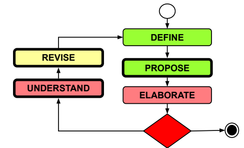
- sometimes missing objects are obvious
  - they come straight from the problem domain
- sometimes the computations are obvious
  - perform transformation X on object Y
  - translate request X into requests x_1,x_2,x_3
- if so, the architecture arises naturally
  - classes do obvious things to obvious objects
  - key components merely translate between user-level and object-level operations
- many problems are, in fact, this simple

Architectural Elaboration



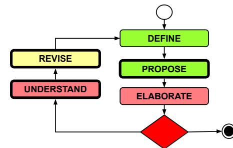
- **ELABORATE** component requirements
  – relationships/interfaces w/other components
- **ELABORATE** component designs
  – how might we implement these requirements
  – what does this imply about other interfaces
- **ELABORATE** interface specifications
  – more detailed characterization

Evaluate the Pieces



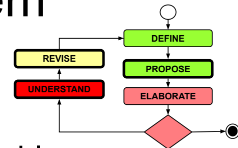
- we can **evaluate** the defined components
  – are their functions clear and limited
  – are responsibilities well compartmentalized
  – do we know how to build (or better, find) each
- we can **evaluate** the defined interfaces
  – how well abstracted do they appear to be?
  – how simple or complex are they?
  – do they embrace applicable standards?
  – do they accomplish good modularity?
  – will they accommodate anticipatable change

Evaluate the System



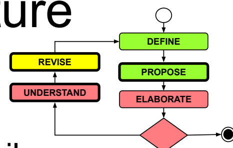
- Will the system meet all requirements?
  – paper-simulate execution of all the major use-cases
  – see what each component will have to do
- Are there any …
  – significant, unanalyzed problems or risks?
  – obvious performance bottlenecks or failure modes?
- How does this architecture …
  – enable reasonable development models
  – provide for testing, integration, support
  – embrace expected anticipatable change

Understand the Problem (this is the real work)



- We must recognize <u>underlying problems</u>
  – must see beyond the most <u>recent symptoms</u>
  – these are often not obvious
    - otherwise you would have designed around them
  – these understandings evolve over time
    - usually after many proposals have all failed
- We must also know available technology
  – what things are easily changed
  – what things are fundamental limitations

Revising the Architecture (the magic part)



- obvious decompositions often fail
  – responsibility can span multiple components
    - ensuring consistency between multiple objects
    - recovering from errors in complex operations
  – correct decisions require collaboration
    - coordinating actions in a distributed system
    - reconciling conflicting views of data or state
  – performance cuts through design hierarchies
  – High Availability requires coupling & isolation
- we may need to "push back" on requirements

Solving Hard Problems

- identify the real underlying problems
  – analysis, study, simulation, hunch, ...
- imagine systems where each doesn't exist
  – **question your assumptions**
    - does this problem really have to be solved?
  – **constrain** or **sub-class** the problem
    - divide and conquer the sub-problems individually
  – **preclude** or **side-step** the key problem
    - ensure problem can't arise or doesn't matter
  – **embrace** the problem
    - accept it as a fundamental fact of life

Precluding a Problem

Problem

Dealing with communications failures in a HA cluster is extremely complex.

Solution

Build apps upon a guaranteed delivery reliable transport layer. The only way a delivery can fail is if the recipient is dead … in which case he is no longer a factor.

Side-Stepping a Problem

Problem:

An automatic network system upgrade can fail in a myriad ways, and we have to recover from each of them.

Solution:

Don't try to recover from such failures. Just fall back to the last known safe state, and start all over again.

Constraining a Problem

Problem:

I want to prove the correctness of some OS code, but interrupts/preemptions can happen at any time and do anything.

Solution:

Don't allow interrupts to happen at any time, and constrain what they can do. Then prove no overlap between my code and the few things interrupts can do.

Sub-Classing a Problem

Problem:

Reconciling conflicting file system updates after a network partition and rejoin.

Solution:

Don't try to solve the general problem. Use type specific modules for different types of files (e.g. directories, mail-boxes, versioned files, etc). Solve those with easy solutions, and get manual assistance with the few that remain.

Embracing a Problem

Problem

I want continuous access to files, but I can't always talk to my file servers.

Solution

Accept that your connection to remote file servers is intermittent, and stop using them as your primary file access path. Maintain all needed files in a local cache and have a proxy server keep the cache up to date.

Describing an Architecture

- others must understand your architecture
  - so they can review it
  - so they can build, test, and support it
  - so they can use it
- complex architecture can't be described
  - it must be understood
  - it must be taught, in progressive lessons
  - recognizing readers' starting points
- these lessons must be planned

Conceptual Understanding

- describe the external system view
  - as seen by clients
- describe the critical data
  - by its content and purpose
- describe the active components
  - by their roles and responsibilities
  - by their relationships and interactions
- describe the system in operation
  - how common operations are performed
  - how common errors are handled

Formal Specification

- overall system requirements
  - functional, performance, availability, …
- describe the data
  - schemas, properties, invariants
- describe active components
  - functional requirements
  - external interfaces
  - perhaps internal structure

Making it Understandable

- use visual representations
  - visual relationships are easier to absorb
  - use consistent visual metaphors
- build on existing knowledge
  - use standard terminology whenever possible
  - draw similes to existing familiar systems
  - define all new concepts and terms up-front
- provide examples (ideally in visual form)
  - of deployment and operation
  - so they can visualize the system in operation

For Next Lecture (design reviews)

- Wiegers: Inspections
  - principles and detailed techniques
- Wiegers: Seven Deadly Sins
  - how reviews most commonly go wrong
- McConnell 21.4
  - walk-throughs and other less formal processes

Supplementary Slides

Design for the Future

- few programs are “write and forget”
  - we will be adding new features to them
  - we will adapt them to new applications
  - these will exceed cost of initial development
- good architecture anticipates such change
  - defers much work to future releases
  - but lays foundation for anticipated changes
- this might seem to require omniscience
  - many types of change are easily predictable

Anticipating Change
Fixing Known Limitations

- limitations, poorly implemented features
  - feature/quality compromises for schedule
  - usually easily recognized in current design
- design interfaces w/improvements in mind
  - consider a few better implementations
  - find abstraction that encompasses them all
  - define interfaces accordingly
    - better abstraction for the clients
    - ensure flexibility for better implementations

Anticipating Change
Obvious Enhancements

- known features left out of current release
  - feature compromises for schedule
  - features we aren't yet ready to implement
  - these too are easily anticipated
- architect and design for them from the start
  - design classes to support these features
  - design APIs with these extensions in mind
  - cleanly stub-out the missing functionality
  - be prepared to add new APIs later

Anticipating Change
Portability

- Change imposed by evolving markets
  - new operating systems, hardware platforms
  - internationalization
  - supporting new protocols
- Consider how these would affect design
  - what components would be replaced
  - what interfaces might have to change
- Compartmentalize these changes
  - isolate affected functions into a few modules
  - abstract interfaces to embrace other implementations

Anticipating Change
Mechanism/Policy Separation

- Identify policy decisions in your design
  - places where correct behavior is ill defined
- Consider the range of possible policies
  - not just the ones you consider most reasonable
- Design your mechanisms for the full range
  - decision rules are user-configurable
  - consider how users might configure them
  - define appropriate configuration mechanisms
- Provide default configuration rules

Anticipating Change
Generality

- It is good to think of more general abstractions
  - improves productivity through code reuse
    - if you properly understand the more general abstraction
    - if you actually do reuse the code
- The most general solution is not always the best
  - general solutions often involve more code
  - more abstracted interfaces may be less intuitive
  - the generality may never actually be reused
- Few back-yard sheds need marble columns
  - don't spend too much time on speculative investments