System Architecture (how to)

- how much architecture do we need
- designing for the future
  - how do we identify likely types of change
  - how do we accommodate future change
- the process of evolving an architecture
  - the nature of the process
  - starting with the obvious
  - evaluating an architecture
  - hard problems and non-obvious architectures

Who needs architecture?

- Do you need modular sub-components
  - to decompose a hard problem?
  - for independent development?
  - for component reuse?
  - for heterogenous inter-operatbility?
  - for future enhancements
  - for independent delivery or replacement?
- If so, you need an architecture
  - to describe the components & interaction
- If not, you can move straight on to 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

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

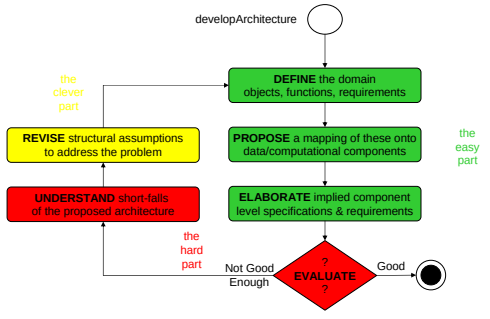
A "Wicked" Problem

- some problems come from requirements
  - complex specifications, tight constraints
  - these are usually clear from the start
- some problems come from solutions
  - constraints implied by our approach
  - complexities inherent in our approach
  - performance of our approach
  - these emerge as we elaborate our design
- problems & solutions are entangled
  - they evolve with/are affected by one-another

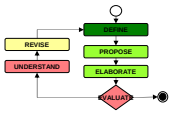
The Architectural Process

- is not a "one-pass", "top-down" elaboration
- architecture usually involves compromise
  - accepting limitations to simplify problems
  - to best satisfy conflicting goals
  - to address time, resource, risk considerations
- architecture must anticipate design issues
  - each component must be build-able
  - it needs information, potency, and slack
- seldom one obvious or right architecture
  - but some are much better than others

Developing an Architecture

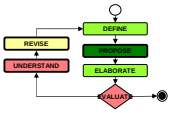


Problem Definition



- 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

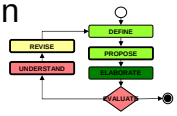
Start with the "Givens"

- 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

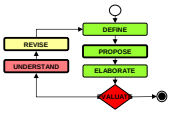
- sometimes missing objects are obvious
  - they come straight from the problem domain
- sometimes the computations are obvious
  - perform transformation X on object Y
  - translate request X into requests x_1, x_2, x_3
- if so, the architecture arises naturally
  - classes do obvious things to obvious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

Architectural Elaboration



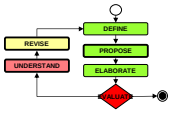
- **ELABORATE** component requirements
  - relationships/interfaces w/other components
- **ELABORATE** component designs
  - how might we implement these requirements
  - what does this imply about other interfaces
- **ELABORATE** interface specifications
  - more detailed characterization

Evaluate the Pieces



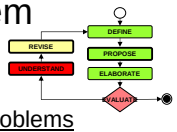
- we can **evaluate** the defined components
  - are their functions clear and limited
  - are responsibilities well compartmentalized
  - do we know how to build (or better, find) each
- we can **evaluate** the defined interfaces
  - how well abstracted do they appear to be?
  - how simple or complex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

Evaluate the System



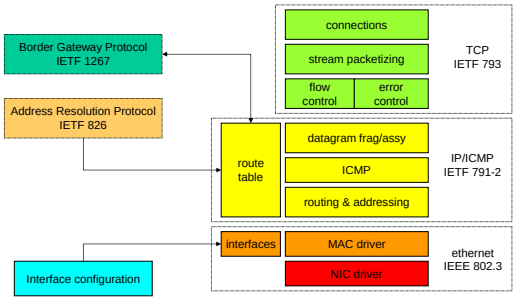
- Will the system meet all requirements?
- Are there any …
  - significant, unanalyzed problems or risks?
  - obvious performance bottlenecks?
  - obvious points of failure?
- How does this architecture …
  - enable reasonable development models
  - provide for testing, integration, support
  - embrace expected technology evolution

Understand the Problem



- We must recognize underlying problems
  - must see beyond the most recent symptoms
  - 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

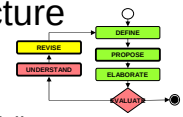
an “Architectural” Problem*



(an “Architectural” Problem)

- Protocol Layering
  - a classic hierarchically layered architecture
  - TCP provides sessions, error, flow control
  - IP provides datagram addressing and routing
- I need fast and reliable transport
  - deliver this message in a few ms, BAMN
  - we need prompt fail-over to alternate links
    - “no-response” detection happens inside of TCP
    - link assignment happens inside of IP
    - there is no direct communication between them

Revising the Architecture



- 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
    - many levels and channels are very expensive
  - High Availability requires coupling & isolation

Solving Hard Problems

- identify the crux problems
  - analysis, study, common wisdom, hunch, ...
- imagine systems where each doesn’t exist
  - **preclude** or **side-step** the key problem
    - seek a design where this problem can’t arise
  - **constrain** or **sub-class** the problem
    - divide and conquer the sub-problems individually
  - **embrace** the problem
    - accept it as a fundamental fact of life
- build your architecture on this foundation

For the next lecture

- Wikipedia: Introduction to UML
  - background on its evolution
- Rational: UML Activity Diagrams
  - good overview of UML "flow charting"
- Braun: UML Physical Diagrams
  - representing physical things
- Kampe: Why we Model
  - treatise on goals and techniques
- Ambler: Agile Modeling Principles
  - good overview of XP modeling philosophy

Supplementary Slides

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 his no longer a factor.

Side-Stepping a Problem

Problem:

An automatic network system upgrade can fail in a million 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 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.