## System Architecture (how to)

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



## 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

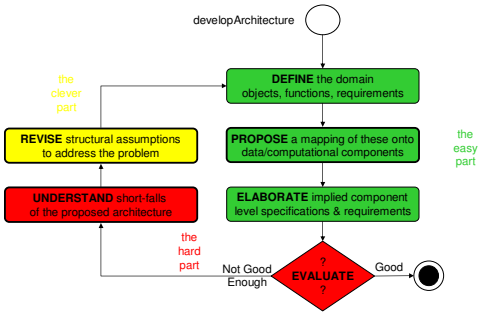
- It is good to think of more general abstractions
  - improves productivity through code reuse
    - if you properly understand the more general abstraction
    - if you actually do reuse the code
- The most general solution is not always the best
  - general solutions often involve more code
  - more 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

## 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, power, and slack
- seldom one obvious or right architecture
  - but some are much better than others



## Developing an Architecture






## 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
- these must be mapped to components



## Architecture – Development (the easy part)

- **PROPOSE** major execution components
  - what types of processing each performs
- **PROPOSE** major data components
  - what types of information each contains
- **ELABORATE** requirements on each
  - relationships/interfaces w/other components
  - where each component will physically reside
  - share of system-level requirements



## 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

## Evaluating the Pieces

- we can **evaluate** the defined components
  - are their functions clear and limited
  - are responsibilities well comparementalized
  - 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



## Evaluating 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



## When it isn't good enough ...

- obvious architectures don't always work
  - 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
  - may be critical non-technical requirements



## Know your enemy, Know yourself

*So it is said that if you know others and know yourself, you will not be imperiled in a hundred battles;*

*If you do not know others but know yourself, you win one and lose one;*

*If you do not know others and do not know yourself, you will be imperiled in every single battle*

Sun Tzu



## Understanding 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



## 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

## Design Patterns

- Much is written about design patterns
  - architectural, class structure, algorithmic
  - good solutions to recurring problems
  - I encourage you to study about them
- view them as Chess gambits or Go joseki
  - using them will improve your game
  - studying them will improve your understanding
  - but in the end, they are merely tactical aids
    - they can't analyze the board for you
    - they aren't a substitute for strategy



## Further Reading

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns: Elements of reusable object-oriented software, Addison-Wesley, 1995
- Mary Shaw, David Garlan, Software Architectures: Perspectives on an emerging discipline, Prentice-Hall, 1996



## For Next Lecture

- McConnell 6 – Class Design
  - excellent discussion of the considerations in defining and designing classes
- McConnell 7 – Routine Design
  - excellent discussion of what and how much to put into a single routine
- McConnell 9 – Pseudo Code Programming
  - advice on reasonable use of pseudo-code



## 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.