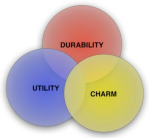
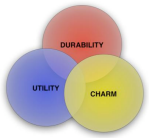


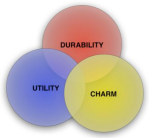
Class 4 SSW565

Gregg Vesonder
Stevens Institute of Technology



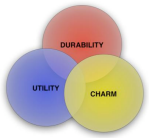
Roadmap

- Logbook
  - me
- Architecture Styles (study aids) - one more time!
- Architecture Description Languages
- Architecture Views
- Sha "Using simplicity to control complexity."
- Usability
- Readings next week - Starr paper on Architecture reviews



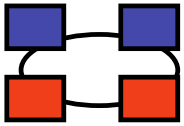
Key Dates

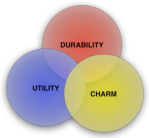
- Thursday class June 18th - MidTerm
- Thursday class June 25th
- June 29th logbooks due
- Final July 20th



Logbook

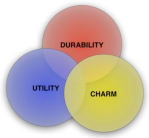
- The Chasm
- The Civil War





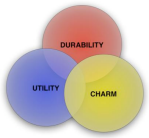
Software Connectors-Redux

- Software connectors specify interaction among components in architectures and designs
- Manifestations of software connectors are buffers, procedure calls, networking protocols, pipes, shared variable access ...
- Connectors are key factors for system properties such as performance, resource use, scalability, reliability security, evolvability
- Architectures separate computation and storage (components) from interaction, transfer of control and data (connectors)
- Data and/or control are transfered along a duct, interaction channel with no associated behavior



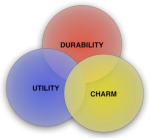
Service Categories of Connectors

- communication- support transmission of data, e.g., stream, pipe
- coordination- support transfer of control, e.g., function calls and method invocations
- conversion -convert the interaction provided by one component to that required by another, e.g., adaptors
- facilitation- mediate and streamline component interaction



Major Connector Types

- procedure call - model flow of control and transfer data, "assembly language of software interconnection"
- event - flow of control caused by an event, once it learns of the event all 'interested' parties are notified and control is yielded to components designated for processing these messages
- data access - permit components to access data maintained by a data store component
- linkage - enable duct formation and therefore tie system components and hold them in a state during operation -required to grow, monitor and repair existing systems



Connector Types - II

- streams - transfer large amounts of data between autonomous processes, often represent connectors with complex protocols of usage (UNIX pipes, TCP/UDP sockets)
- arbitrator - streamline system operation and resolve conflicts, assure system trustworthiness and dependability, negotiate service levels
- adaptor - support interaction between components that were not designed to interoperate
- distributor - identify interaction paths and routing of communication and coordination information, always work in conjunction with other connectors, e.g., DNS. Affects scalability and survivability

From Mehta, Medivdovic & Phadke, 2000

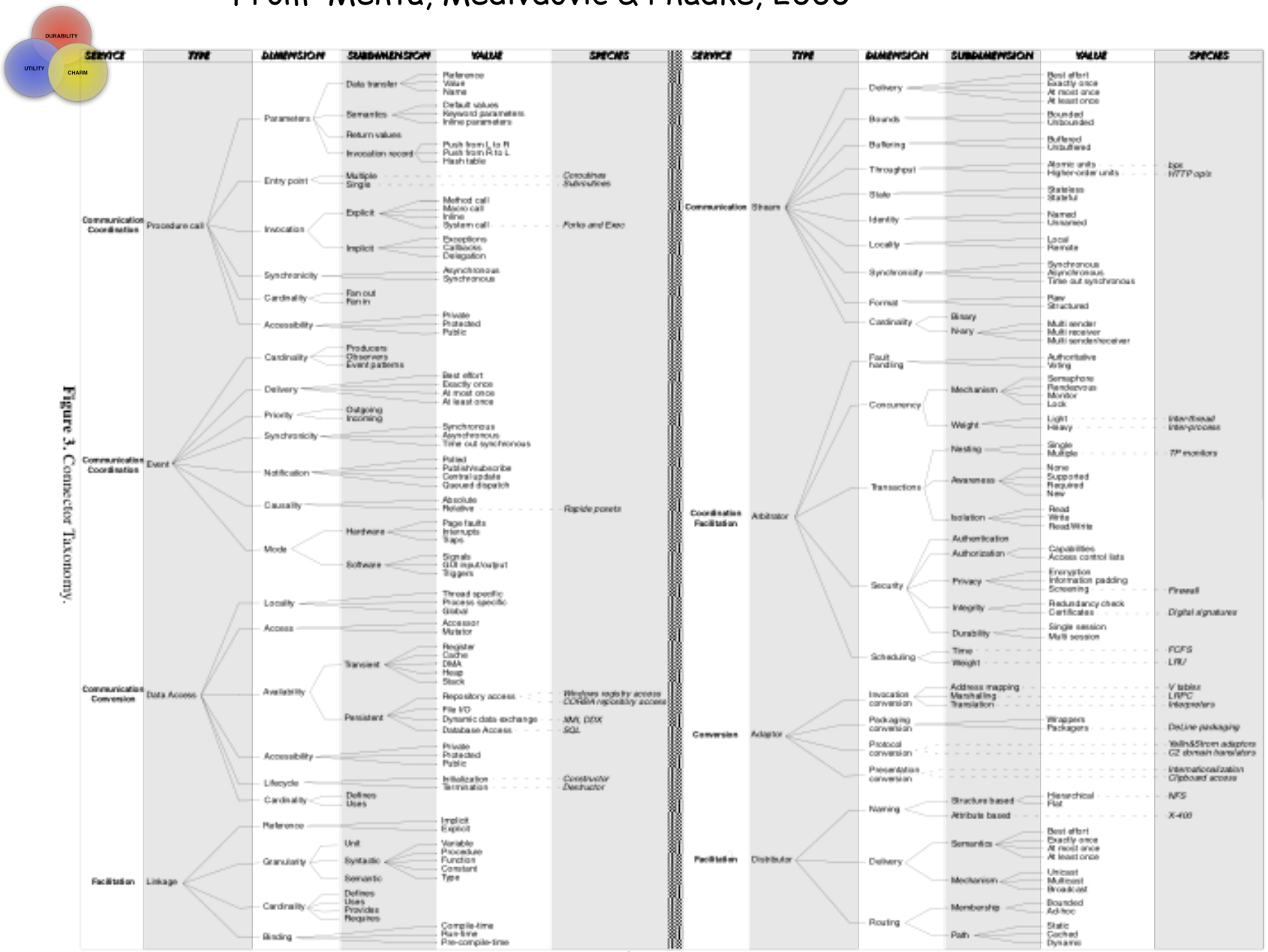
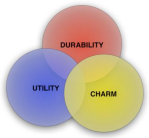
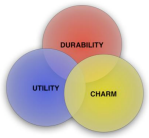


Figure 3. Connector Taxonomy.

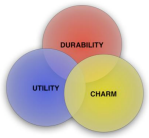


What is the mapping between connector types and architectural styles?



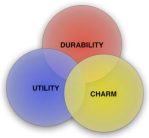
Case Studies from Shaw and Garlan

- Instrumentation software -oscilloscope
  - Issues: reusability and performance
- Mobile Robotics
  - Issues: architecture must accommodate deliberate and reactive behavior, allow for uncertainty, account for dangers(fault tolerance, safety) and flexibility



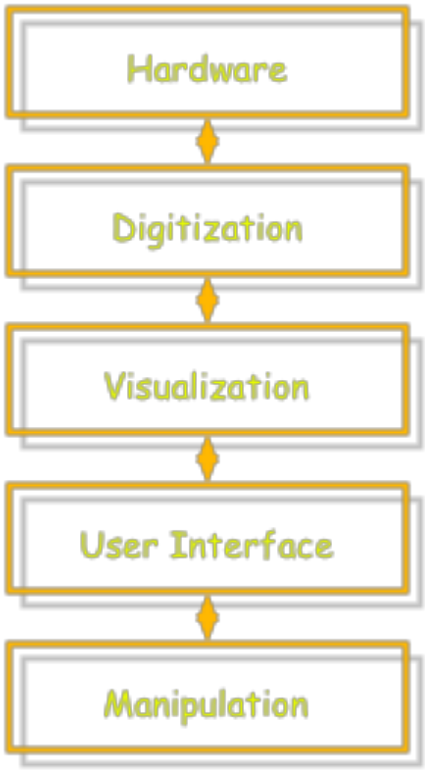
Oscilloscope - OO

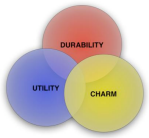
- Clarified data types used in oscilloscope
- Many types identified but could not find an overall model of fit



Oscilloscope - Layered

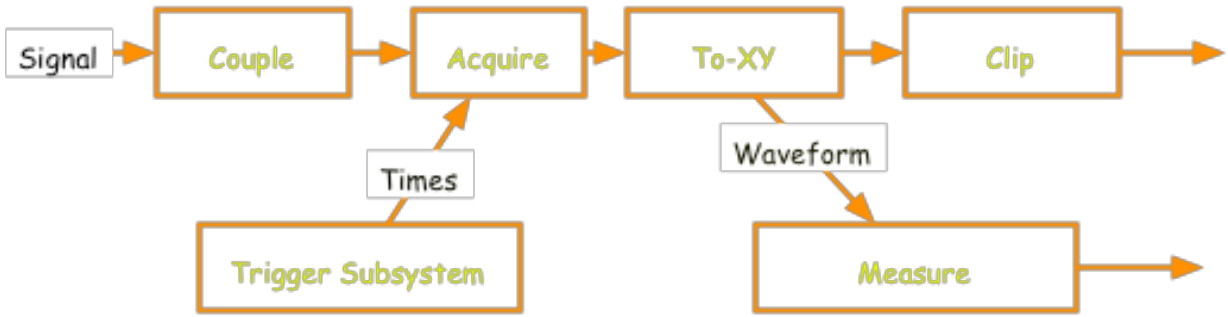
- Partition functions into groups
- Boundaries of abstraction of layers conflicted with needs
- UI needed to touch all layers

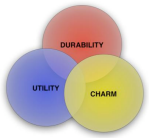




Oscilloscope-Pipe and Filter

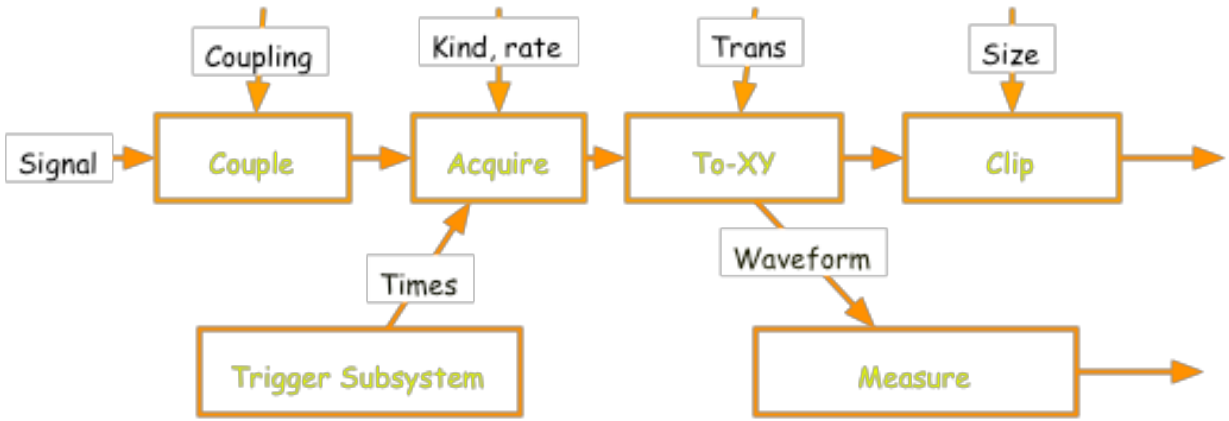
- Oscilloscope functions viewed as incremental transformers of data -did not isolate functions
- Matched with engineers view of signal processing as a dataflow problem
- Difficulty with how UI was handled since user would be at end of pipe and worse isolation than layered

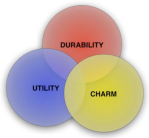




Oscilloscope
Modified Pipe & Filter

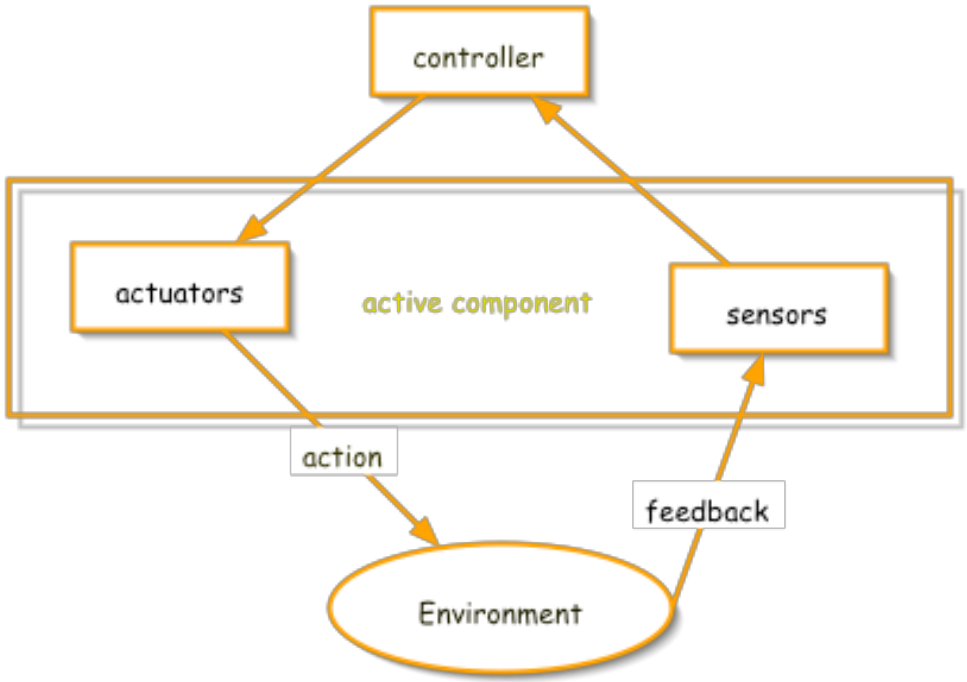
- Associate with each filter a control interface that permitted external entity to control input
- Made further adjustments to improve performance

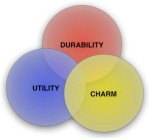




Robot - Control Loop

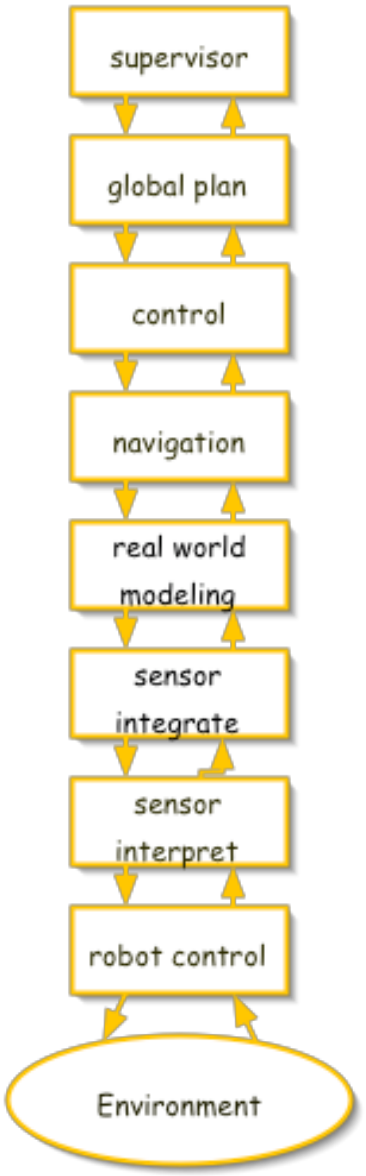
- Simple architecture, but not easy to change modes from adjusting arm to moving base, can't sense, plan & act too well
- Deals with uncertainty through iteration
- Fault tolerance and safety are dealt with through redundancy
- Flexibility - major components dealt with separately
- Good for simple robots, small # of events and simple nondecomposable tasks

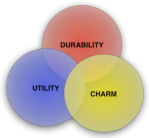




Robots Layered

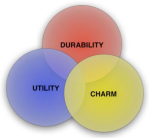
- Abstraction levels to simplify task delegation (navigation, control). Not good for data requiring fast reaction - has to go up and down the stack
- Abstraction deals a bit with uncertainty through knowledge
- Safety and fault tolerance - layered provides checks and balances
- Not so flexible because of interlayer dependencies



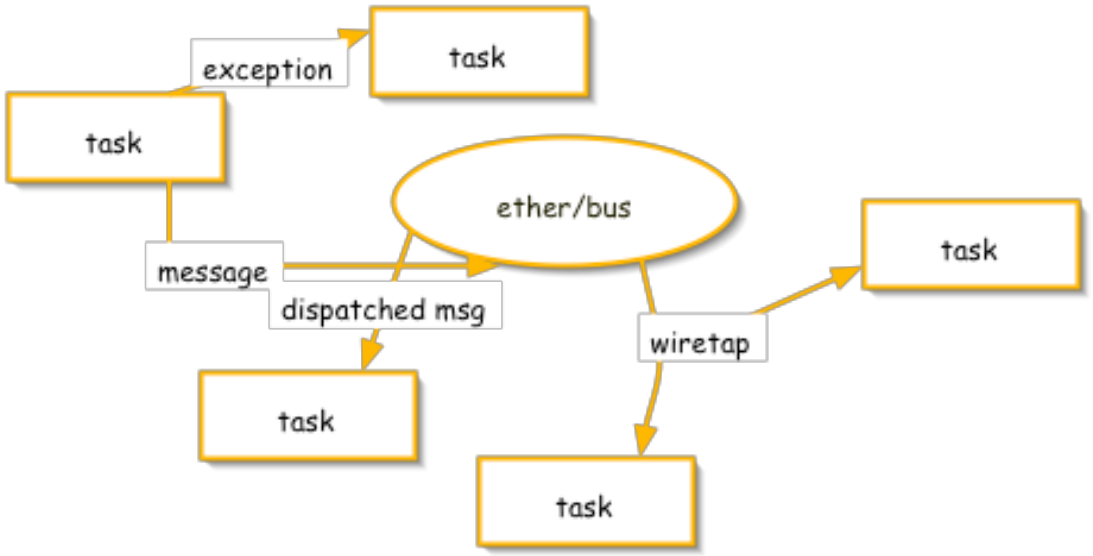


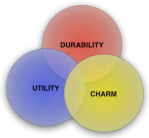
Robot - Implicit Invocation

- Complicated - uses message broadcast and task trees
- Supports 3 features:
  - Exceptions-deals with danger
  - Wiretapping - msgs can be intercepted for safety check
  - Monitors - e.g., check for battery levels
- Evaluation
  - Separates action and reaction, concurrency
  - Uncertainty is a bit more difficult but can modify exception handlers ONCE KNOWN
  - Great for safety - exception, wiretapping, monitoring, multiple handlers for redundancy
  - Implicit invocation allows flexibility - new handlers in some systems need only be registered with central server

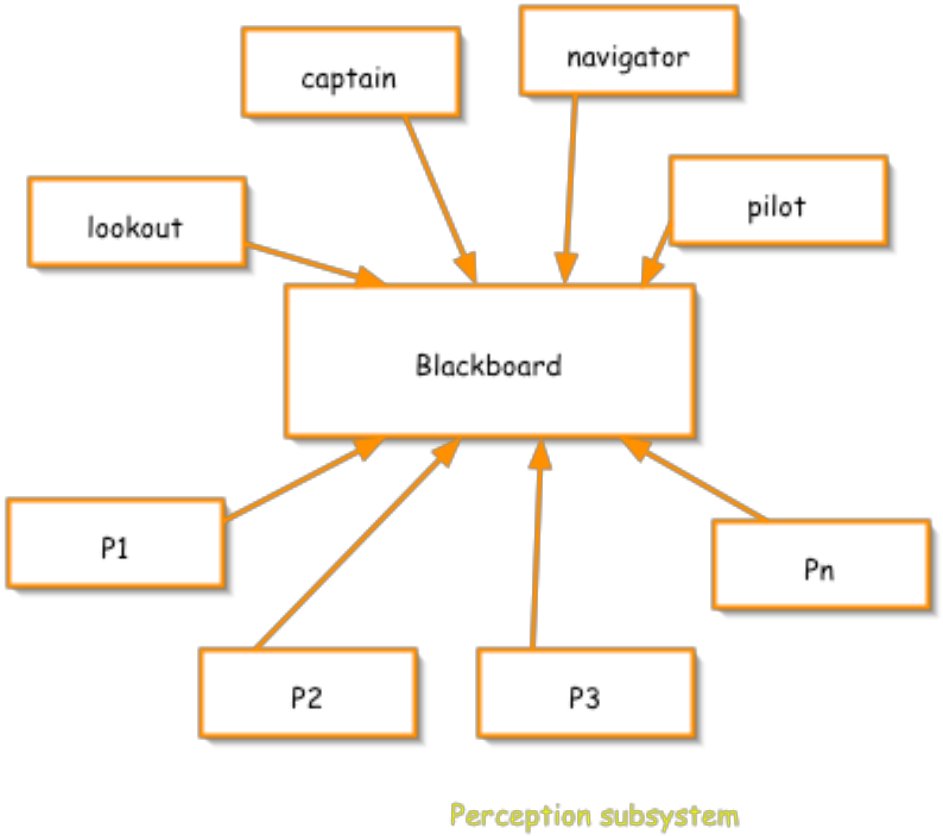


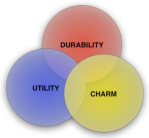
Robot - implicit invocation





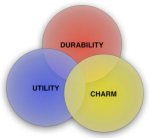
Robot -Blackboard Model





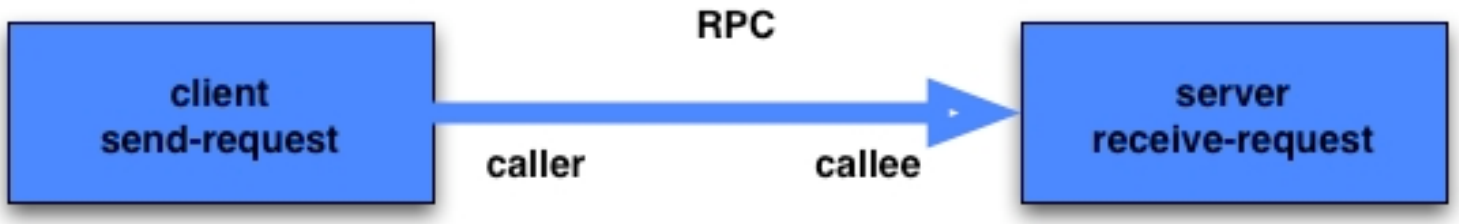
Architecture Description Languages (ADLs)

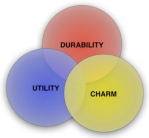
- ACME from CMU.
- 5 types of entities:
  - Components
  - Connectors
  - Systems (combo of 1st two)
  - Ports - interfaces defined by sets of them, simple/complex
  - Roles caller/callee, read/write,…
  - (actually 2 more that are representions)



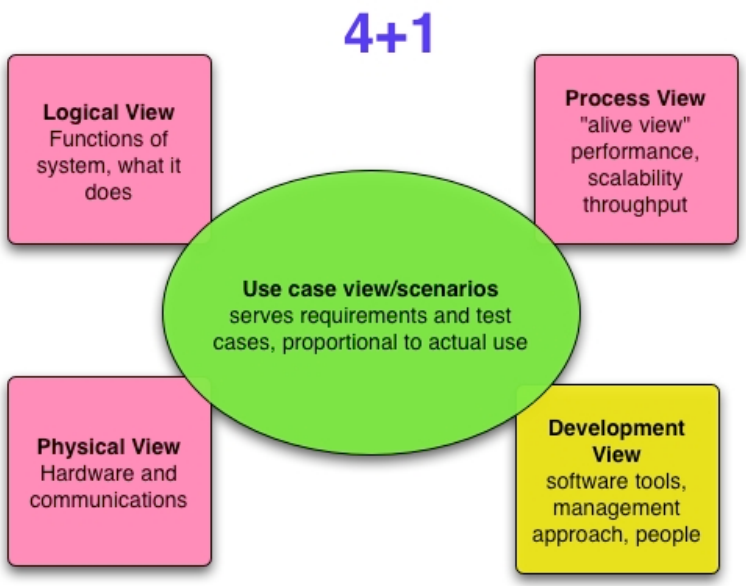
Simple ADL Example

```
System Simple_cs = {
  Component client = {Port send-request}
  Component sever = {Port receive-request}
  Connector rpc = {Roles {caller,callee}}
  Attachments : {client.send-request to
rpc.caller; server.receive-request to
rpc.callee}
}
```

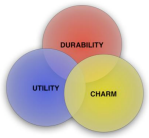




Architecture Approach

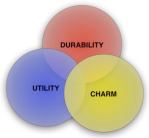


- Ilitie's, non functional requirements are represented in 3 of the 4 quadrants
- Spend some time on the ilities



Simplicity to Control Complexity Reliability and Architecture

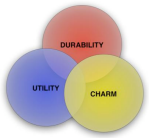
- Sha(2001) - do not worry about the math
- Two software reliability approaches
  - Fault avoidance = formal specification and verification + rigorous software development process
  - Fault tolerance through diversity
- But dividing resources for diversity can produce mixed results depending on architecture
- Thesis: Key to improving reliability is simple and reliable core components
- Not earth shattering but his solution is novel!



Two types of complexity

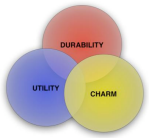
- Computational complexity - # of steps to complete computation
- Logical complexity - # of steps to verify correctness (cases or states or testing process)

	Computational complexity	Logical Complexity
quicksort	low	high
bubblesort	high	low



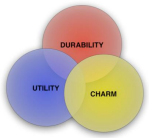
Quicksort and Bubblesort

- Bubble sort is simple -- start at 1 end and compare first two elements, switch if in wrong order and then proceed until largest "bubble" to the top, e.g., 8,3,5,4->3,8,5,4->3,5,8,4 ->…, then go iterate after getting to the top
- Quicksort
  - Divide- Partition data into two partitions around a value
  - Sort by recursively applying quicksort
  - Combine - they are sorted
  - Choosing partition is tricky: random, median of three, …



Reliability and Complexity

- The higher the complexity, the more difficult to specify, design, develop and verify
- Postulates
  - P1-complexity breeds bugs
  - P2-all bugs are not equal, easy bugs spotted early, decreasing rate of improvement with development effort
  - P3-all budgets are finite - diversity is not free it divides the effort
- Simple reliability model



Simple Reliability Model

$R(t) = e^{-\lambda t}$

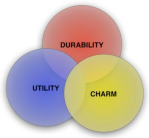
λ - failure rate proportional to software complexity C and inversely proportional to development effort E.

$$R(t) = e^{-kCt/E}$$

if we concentrate on relationship between complexity and development effort :

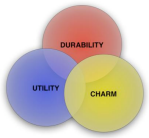
$$R(E, C) = e^{-C/E}$$

Null



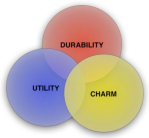
Two Diversity Methods
N-version

- N version programming, divide Development effort into n independent teams design and implement different program versions for same specification, hoping faults independent. At runtime results are "voted" and majority of outputs is selected (median used if output floating point)
- Running through equations with assumptions - reliability of single version development is better than N version over wide range of effort.
- FAA DO 178B discourages use of N-version programming



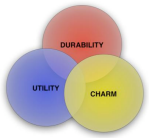
Two Diversity Methods
Recovery Block

- Construct different alternatives and submit to acceptance test. When input data arrives system checkpoints its state and executes primary alternative. If result passes acceptance test, use it, else roll back to checkpoint and iterate through alternatives until one passes or exception is generated
- When development effort is low single version is better but as effort rises blocking is better
- Basically recovery block scores better than N version because only one version needs to be correct
- But getting perfect acceptance tests is difficult



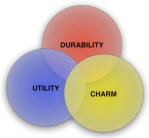
So how do we use simplicity to control complexity?

- Separate critical requirements from desirable properties
  - Assumption- useful but unessential features cause complexity
- Returning to sorting case - what is critical is sorting the list correctly what is desirable is sorting it fast
- Solution: sort the same list using quicksort and have the result sorted by bubblesort
  - If quicksort works correctly then bubblesort should sort the result from quicksort in a single pass! Recall from the table that quicksort is more logically complex.
- This same methodology can be used in a host of optimization tasks



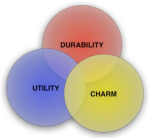
Case study: Boeing 777 flight control software

- Feedback control system - loop continuously corrects errors -- difference between actual and checkpoint.
- Boeing 777 uses 2 flight control systems:
  - New 777 system, fine tuned optimized control
  - Old reliable, simpler, tested 747 system
- If 777 software deviates from control envelope of 747 software for 777, 747 software takes over.



More General Case: Simplex Architecture

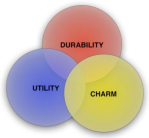
- Two components, High assurance control subsystem and High Performance control subsystem.
- Normally HPC controls but if operation falls outside HAC envelope, HAC controls
- Paper has a way of computing such an envelope but beyond scope of course
- Also can manage upgrades this way
- The software that implements decision rule is simple and easy to verify



HAC vs. HPC

- High Acceptance Controller
- Application Level - well understood classical, tradeoff performance for stability/ simplicity
- System Software Level - high assurance, no frills OS
- Hardware level - well established, simple fault tolerant
- System Development and Maintenance - high assurance process
- Requirements Management- limits requirements to critical functions and essential services, stable and changes very slowly

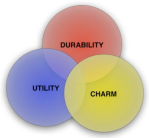
- High Performance Controller
- Application Level - advanced technologies
- System Software Level- real time, dynamic and sophisticated development features
- Hardware Level - standard industrial hardware
- System Development and Maintenance - standard practices
- Requirements Management - emphasizes features and performance, requirements change relatively quickly



On UI Architecture

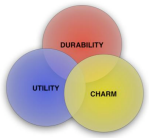
If the designers of X-Windows built cars, there would be no fewer than five steering wheels hidden about the cockpit, none of which followed the same principles -- but you'd be able to shift gears with your car stereo. Useful feature, that.

-- Marus J. Ranum, Digital Equipment Corporation



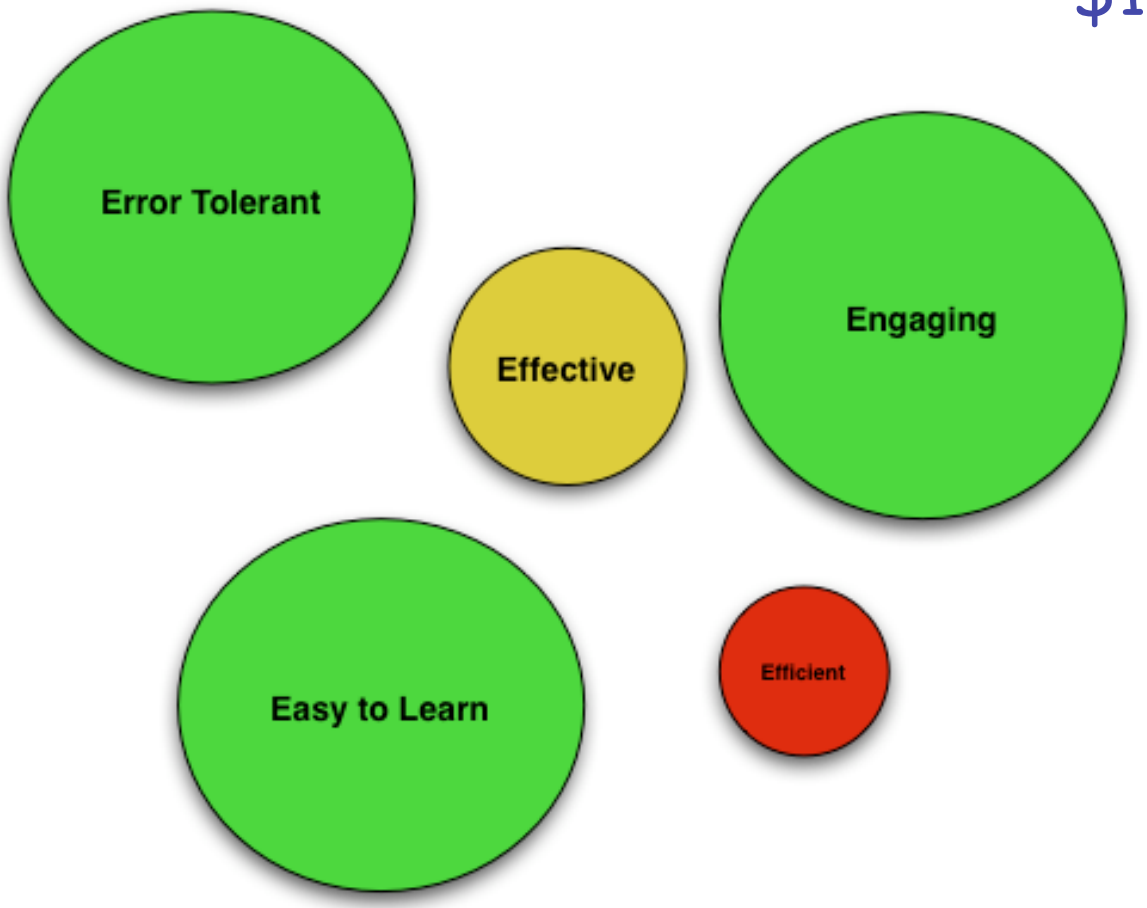
Evaluate User Experience, 5 E's

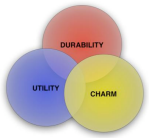
DIMENSION	KEY NEEDS	DESIGN TACTICS
Effective	Accuracy	Focus on places in the interface for potential error and protect against them. Look for opportunities to provide feedback and confirmations
Efficient	Operational Speed	Present only most important information. Work on smooth, direct navigation. Interaction style should minimize actions required
Engaging	Attract users	Consider what aspects of the product are most attractive and incorporate into design
Easy to learn	Just-in-time instruction	Step by step interfaces that help users navigate through complex tasks. Provide training in small chunks if possible
Error tolerant	Validation	Look for places where selection and calculators can replace data entry. Error messages provide opportunities to correct problems



Success Criteria, 5E's
(rational weighting)

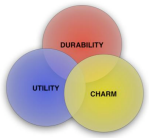
$100





Wide Range of Choices

- Real life vs. simulation
- Actual users …….. User reps/experts
- Actual tasks ……. Task descriptions
- Real environment .. Controlled environment
- Users with domain knowledge ….. Users w/o domain knowledge
- Usability checkers: W3C to commercial
- … the profile



Usability Evaluation Techniques

User Observations

Performance Measurement

Remote Evaluation

Usability Inspections

Heuristic Inspection

Participatory Heuristic Evaluation

Guideline Reviews

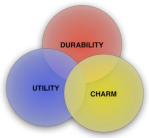
Standards Inspection

Cognitive Walkthrough

Peer Reviews

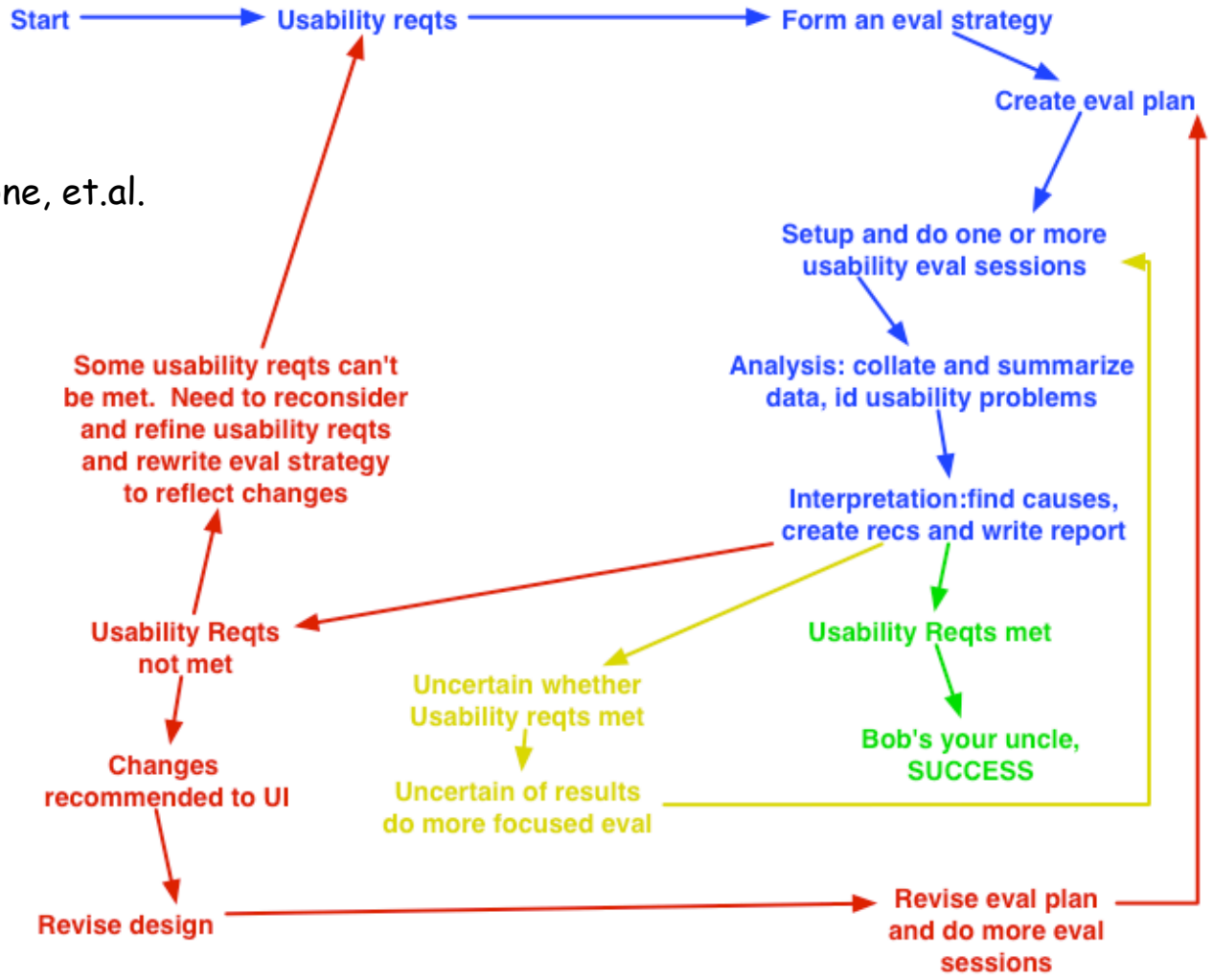
Psychological Experimentation

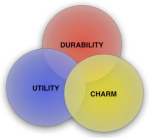
Adapted from Stone, et.al.



Usability Eval Process

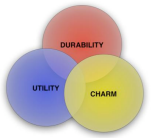
Adapted
from Stone, et.al.





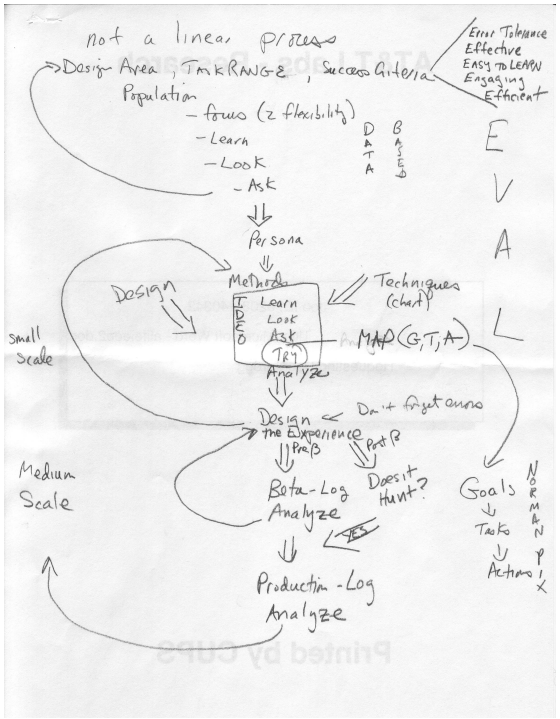
8 Golden Rules

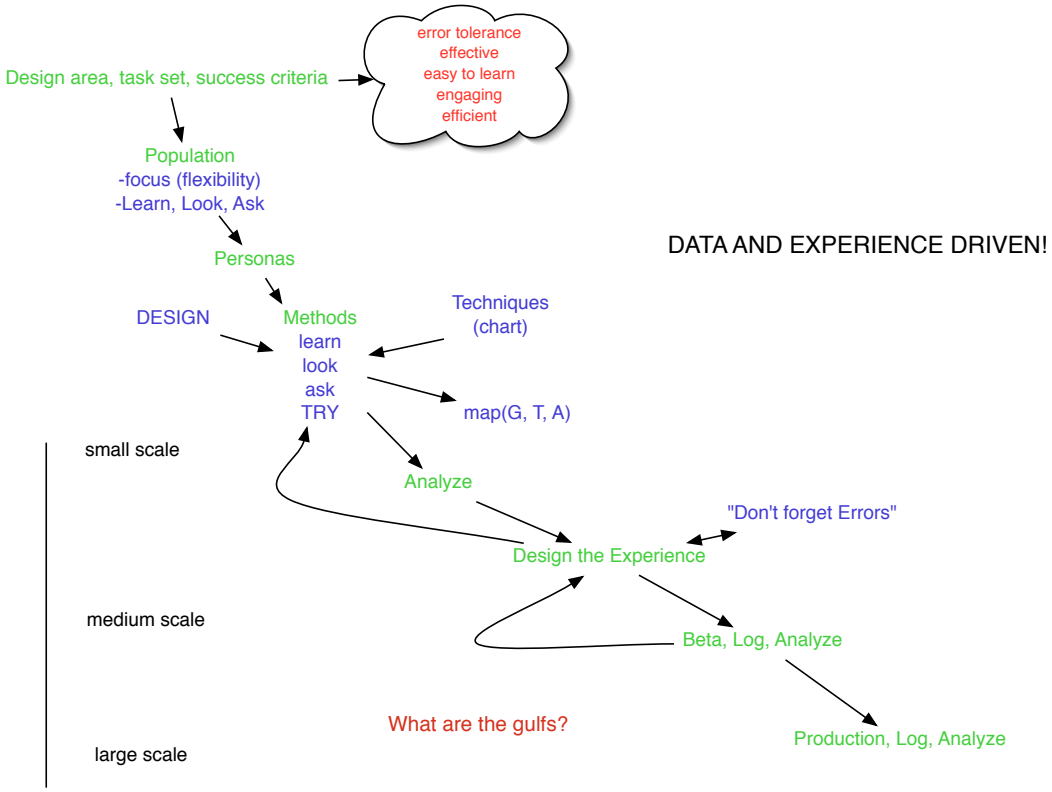
- Strive for consistency
- Cater to universal usability and design for change
- Offer informative feedback
- Design dialogue to yield closure (beginning, middle and end)
- Prevent errors
- Permit easy reversal of actions
- Support internal locus of control - user is in charge
- Reduce short term memory load

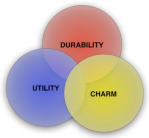


In Summary

- User Experience Design is neither linear nor rigid!

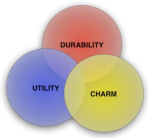






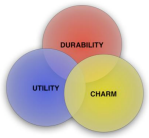
Why spend effort on the UI?

- Increased efficiency
- Improved productivity
- Reduced errors
- Reduced training - strive for game like training
- Improved acceptance



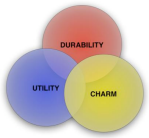
Simplicity

- http://lawsofsimplicity.com
- Reduce
  - How simple can you make it <-> How complex does it have to be
- Organize
- Savings in time feel like simplicity
- Knowledge makes everything simpler
- Simplicity and complexity need each other



Simplicity - 2

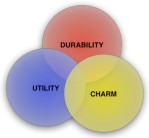
- What lies in the periphery of simplicity is definitely not peripheral - context
- More emotions are better than less
- In simplicity we trust
- Some things can never be made simple!
- The One
  - *Simplicity is about subtracting the obvious and adding the meaningful*



LoS

- REDUCE
- ORGANIZE
- TIME
- LEARN
- DIFFERENCE
- CONTEXT

- Emotion
- Trust
- Failure
- The one
  - Away
  - Open
  - Power

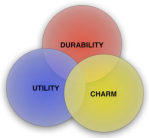


How simple can you make it

REDUCE!

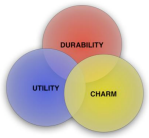
How complex does it have to be

- Easiest way to simplify is to reduce
- SHE
  - Shrink - small is good and comforting
  - Hide - most used controls, interface code - hide complexity - perceived sense of control
  - Embody - what shows is of quality



Organize

- Organization makes a system of many appear fewer
- SLIP
  - Sort - the items
  - Label - the categories
  - Integrate - combine similar groups
  - Priority - combine highest priority into single set: 80/20 rule
- "Humans are organization animals."

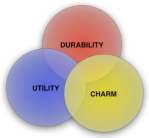


How can you make the wait shorter

Time

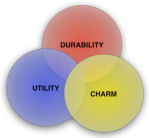
How can you make the wait more tolerable

- Savings in time feel like simplicity
- Best interface may be to automate (no interface)
- Indicate time remaining
- Rarely exclude time!



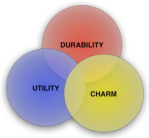
Learn

- "knowledge makes everything simpler"
- Professor or student BRAIN
  - *Basics* are the beginning (tacit): feel confident
  - *Repeat often* simplicity and repetition are related
  - *Avoid creating desperation* - gentle, inspired start to learning: feel safe
  - *Inspire* with examples
  - *Never forget to repeat*: instinctive



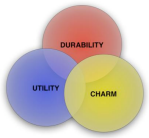
Learn 2

- Good Design
  - Eases process of understanding (form with function)
  - Provides sense of instant familiarity
  - Surprises!
  - Cell phone, digital camera, car and instruction manuals
- "In the beginning of life we strive for independence, at the end of life it is the same"



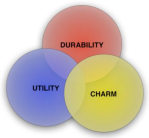
Differences

- Simplicity and complexity need each other
- Complexity provides and even greater appreciation for simplicity

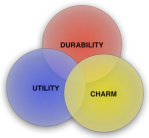


Context

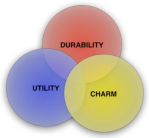
- Laser beam or light bulb?
- "What lies in the periphery of simplicity is definitely not peripheral"
- White space
  - Web pages
  - My home
- "Given an empty space or any extra room technologists would invent something for the expanse"
- Nothing is an important something - focuses on the something



Ambience

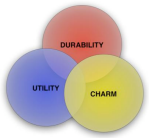


How directed can I stand to feel <-> *How directionless can I afford to be*



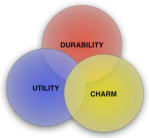
Roadmap

- Architecture
- Case Studies
- Styles
- Connectors
- Reliability
- UI Architecture
- Performance
- Non functional requirement gathering
- Documentation
- Evaluation



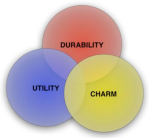
Exercise

- Make additions to each view of the 4+1 architecture of GLOOM – be sure to use connectors and architectural styles



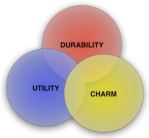
Scenarios "+1"

- Single player: using a practice version solely against bots
- Tournament play: 2 players pitted against each other with brackets hosted on the game server
- Multiplayer environments with up to 12 players

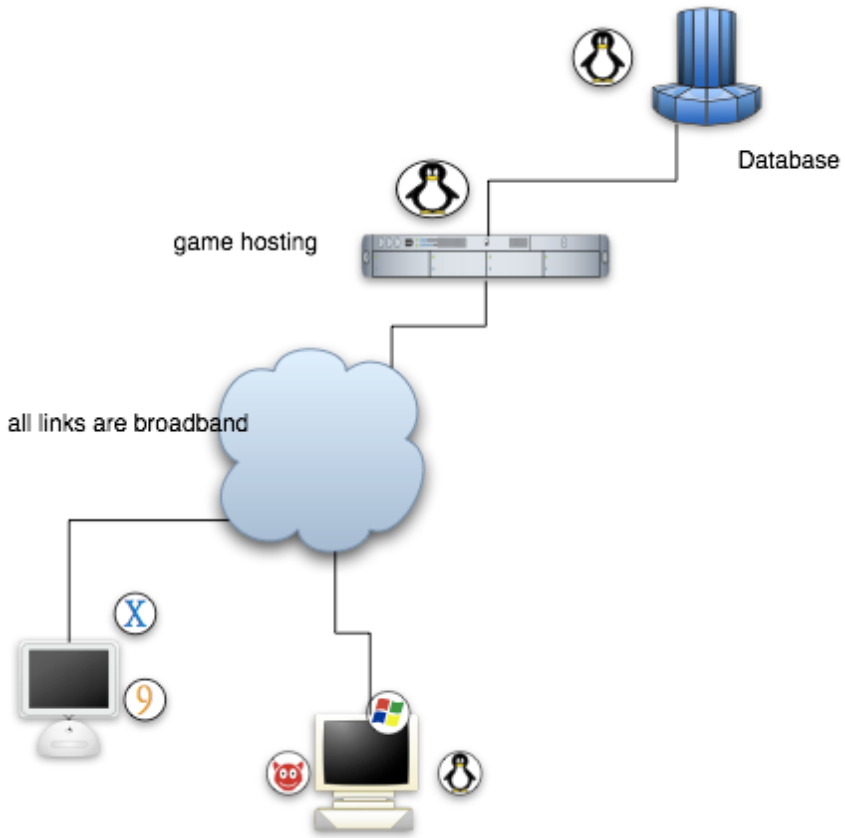


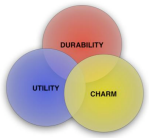
Gloom Logical View

- No diagrams (and this is a bit ahead of things). The game itself is hosted using the Blackboard architectural style with each of the players and bots acting as knowledge sources
- The bots themselves gain their intelligence using a rule base architectural style.
- Different mazes and game results are stored in a repository, a relational data base.



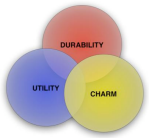
Gloom Physical View





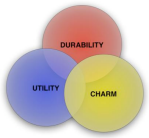
Gloom Process View

- Each player is in its own Java sandbox.
- The bots for a given game session are one java sandbox and they communicate with a blackboard instance, also a separate process.
- Initial environment is uploaded from database and final scores are downloaded to database



Development View

- Java 1.4.2 is the Java version.
- Using an Oracle 9.0 database
- Blackboard software is done in house
- Clips is the rule based language used
- Since most of the coding is in java we will use Sun's coding conventions - http://java.sun.com/docs/codeconv/
- The team will use eXtremeProgramming



Other References

- Shaw, M and Garlan, D., Software Architecture, 1996, Prentice Hall.
- Sha, L. Using simplicity to control complexity, IEEE Software, July/August, 2001, p 20-28
- Foote, S. The Civil War: A narrative, Fort Sumter to Perryville, vol.1, Random House, 1958.
- Smith, C.U. and Williams, L.G. Introduction to Software Performance Engineering, Addison-Wesley, 2001
- Stone, Jarrett, Woodroffe and Minocha User Interface Design and Evaluation, 2005, Morgan-Kaufmann.
- Maeda, John The laws of simplicity, MIT Press, 2006.