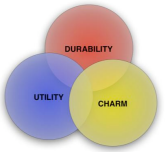
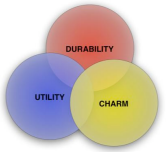


Class 3 CIS 573

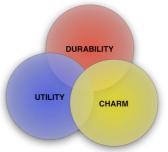
Gregg Vesonder
University of Pennsylvania
Penn Engineering - Computer & Information Science

Penn CIS



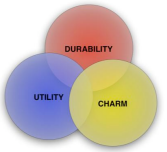
Roadmap

- Continue on 2
- Software Architecture
- Software Reviews
- Readings this class: : S Chapter 11, Brooks chapters 5 & 6 & 9 (3&4);
- Readings next class S Chapter 14; Andersson chapters 1-2



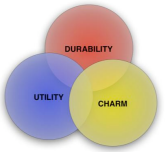
Critical Dates

- Every class project review
- July 23rd Mid Term
- August 6th log books due
- August 11th project presentations
- August 13th Final



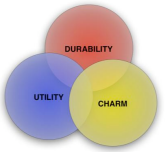
Teams

- Team 1 – Klein Keane, Beck, Buchman, Richardson, Nunez
- Team 2- Wilmarth, Caputo, Xiang, Francis, Nanda?
- Team 3- Noronha, Fang, Treatman, Huang
- Team 4-Whitehead, Liu, Ratnakar



Log Book

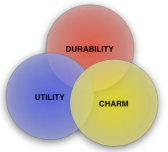
- The Dog & Bone
- Volunteers



Achieving agreement and consensus. Facts often set you free - the press to collect history and push Quantitative Software Engineering.

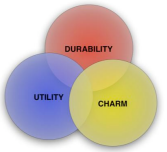
Penn CIS

Drilling Down!



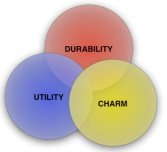
Requirements Elicitation

- Implicit conceptual model of users becomes explicit
- Universe of Discourse (UoD) part of reality we are interested
  - Domain Driven Design - Evans, Ubiquitous Language
- Requires us to become quick learners but
- Much of knowledge is
  - Knowledge taken for granted
  - Tacit-knowledge skillfully applied in the background, not verbalized
  - Involves habits, customs, inconsistencies
  - Influenced by frequency and recency
  - What's needed may be different from what exists



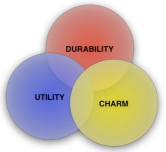
Requirements Elicitation Techniques

- Asking: interview, questionnaire, structured interview, Delphi (group based)
- Task analysis: hierarchical decomposition
- Scenario based analysis: instances of tasks, use-case (not only for OO)
- Ethnography: studying folks in natural setting



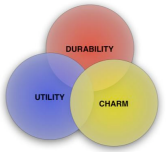
Ethnography

- Discover the way folks actually work
- Discover aspects of cooperation and awareness - information ecology
- Use side by sides, prototyping
- Very important



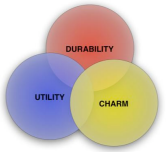
Situated Action and Distributed Cognition

- A simple experiment may not always be diagnostic because:
  - Complex interactions between people, electronic devices and paper resources
  - Physical and social resources are intertwined with use of computer and information technologies
  - Design cannot be separated from patterns of use
  - Users change plans in response to circumstances
- Distributed cognition - knowledge not only in the minds but also distributed in the environment
- Therefore users have to be participants in the design process not just experimental subjects (rigid definition): ethnography, longitudinal studies



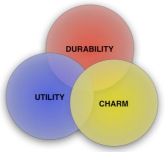
Requirements Elicitation Techniques (cont'd)

- Form analysis: existing forms (may carry over in DTFs)
- Natural language descriptions: training, manuals, …
- Derivation from existing system
- Domain analysis: study existing systems w/ in domain, reusable components



Requirements Elicitation Techniques (cont'd)

- Business Process Redesign - radically redesign the processes, information processing systems should enable
  - At the very least rethink the existing process
- Prototyping
- Usually it is a mixture of these
- PMO/FMO present vs. future method of operation



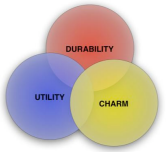
The most important outcome of product definition … a team of stakeholders with enough trust and shared vision …"

Can you build and Succeed w/o great relationships?

Team relations with customers twice as productive as contractual relations (Bernstein & Yuhas, 2005)

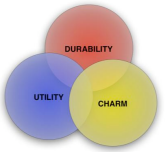
NOT EASY!

COMMUNICATION is the key



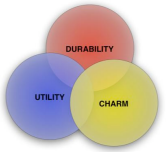
More on Prototyping

- Boehm: "Prototyping (significantly) reduces requirements and design errors, especially for user interfaces." (p 19, Endres & Rombach)
- Types of prototypes:
  - Demonstration prototype - clarifies user requirements, impresses user (GUI)
  - Decision prototype - helps select design alternatives, answers particular questions (design/construction phase)
  - Educational prototype - used to understand technology and its characteristics (performance). Differs from Pilot projects that test a new technology that may or may not be included in the product



Still more on prototyping

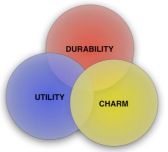
- All three are throwaways - recall from last week that after proto design can begin (and you may use decision protos to help)
- In ENGINEERING most prototypes use different materials that clearly separate it from final proto, e.g. wind tunnel models … only in software are folks tempted



Expert System Process

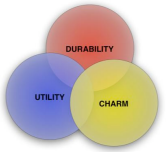
- AKA Knowledge Engineering
  - Knowledge Engineer becomes familiar with domain, architecture and operation
  - KE meets with experts to understand operations and issues
  - Team uses knowledge to create first (and subsequent) passes at rules
  - Experts critique results, provide new knowledge and iterate on previous step until a satisfactory (or best possible) conclusion is achieved

Key to many new Requirements & Design techniques



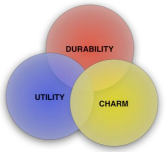
Use Cases - Ivar Jacobson

- Capturing requirements from the user's point of view (and then we adapt) in manageable chunks (OO motivated)
- Actors are entities that reside outside the system and should not be other use cases
- Use case example - "withdraw a book" following normal path and alternate paths
- Need a comprehensive set



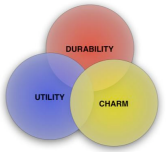
Human Actors (from Wirfs-Brock)

- Who uses the system?
- Who installs the system?
- Who starts up the system?
- Who maintains the system?
- Who shuts down the system?
- Who gets information from the system?
- Who provides information to the system?
- Systems too are actors- use actual system names



Requirements Advice

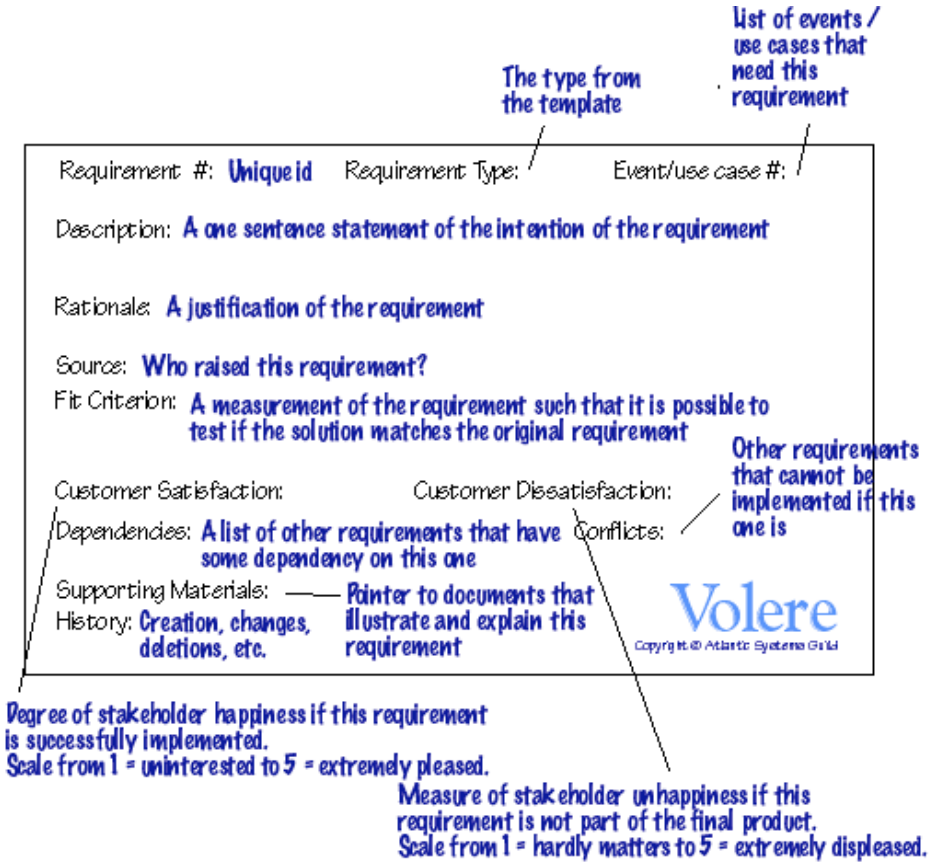
- Invent or borrow a standard format (Volere)
- Use language consistently - glossary
- Highlight key parts of requirements



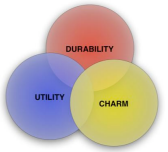
The Requirements Specification

- Key output of the process
- Must be baselined -- it will change and this must be measured
- Requirements spec should be readable, understandable, correct (validated against other docs), complete, internally consistent, ranked for importance or stability, verifiable, modifiable and traceable
- Used even in prototyping to describe outcome
- Serves at least two groups, the user and the designer
- More later since it must be managed

Recording Requirements

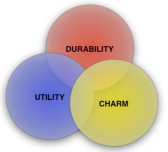


http://www.systemsguild.com/GuildSite/Robs/Template.html



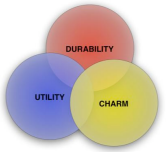
Volere Requirements Specification Contents

- One method, a useful checklist
- Card approach will be seen in several areas
- Focuses on:
  - Product Constraints- restrictions and limitations that apply to project and product
  - Functional Requirements- the functionality of the product
  - Non-Functional Requirements - the product's qualities
  - Product Issues- apply to the project that builds the product



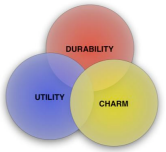
Volere Product Constraints

- Purpose of the product
  - User problem
  - Goals of product
- Client, customer and other stakeholders
  - Client = pays and owns
  - Customer buys
  - Other stakeholders, e.g., management, business SMEs
- Users of the product
  - Users
  - Priorities assigned to users (key, secondary, unimportant) see also what Volere calls stakeholders



Product Constraints (cont'd)

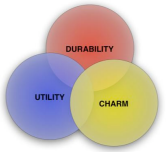
- Requirements constraints
  - Solution constraints –Vista vs. Windows 7
  - Implementation environment
  - Partner applications
  - COTS-commercial off the shelf software (early)
  - Anticipated workplace environment
  - Known deadlines -how long
  - budget
- Naming conventions and definitions
- Relevant facts
- Assumptions



Volere Functional Requirements

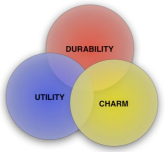
- The scope of the product
  - -context of work, domain knowledge
  - Work partitioning, event list with inputs and outputs
  - Product boundary(users: active, autonomous, cooperative)
- Functional and data requirements
  - Functional - fit criterion (as many place as possible - did I do what I said I'd do)
  - Data (model)

(simple, eh?)



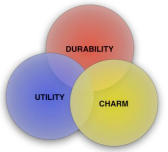
Volere Non-Functional Requirements -"ilities"

- Look and Feel Requirements
- Usability Requirements
  - Ease of use
  - Ease of learning
- Performance Requirements
  - Speed
  - Safety critical (robotics)
  - Precision
  - Reliability and availability
  - Capacity throughput, volume, cpu/memory load



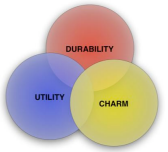
Non Functional (cont'd)

- Operational Requirements
  - Expected physical environment
  - Expected technological environment
  - Partner applications
- Maintainability and Portability Requirements
  - How easy? {how much effort or budget}
  - Special conditions for maintenance
  - portability
- Security Requirements
  - Confidentiality - restricted access
  - File integrity/ system integrity (sync'd)



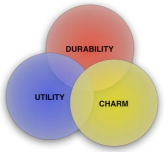
Non Functional (cont'd)

- Cultural and Political Requirements
- Legal Requirements
  - Law pertaining to product
  - Standard compliance



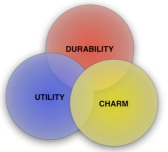
Volere Project Issues

- Open issues
- Off-the-shelf Solutions (early)
  - Existing product
  - Ready made components
  - Copy or model
- New Problems
  - Problems the new system can cause
  - Affect on current systems
  - Users adversely affected by system
  - Limitations in the anticipated environment that inhibit new system
  - Other problems?



Project Issues (cont'd)

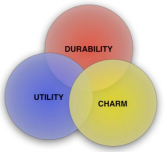
- Tasks
  - Steps to deliver product
  - Development phases {Development Plan}
- Cutover
  - Special requirements to get data and procedures
  - Data modification/translation necessary
- Risks
  - What are they
  - How do you address them



Project Issues (cont'd)

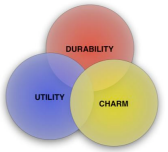
- Costs
- User Documentation
- Waiting Room
- {Worry Beads} - daily, weekly, monthly
- And then there is the inevitable negotiation

Penn CIS



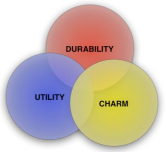
Interface Specification

- Procedural interfaces: APIs, signatures
- Data structures: passed from one subsystem to the other
- Representations of data - e.g., bit packing - the "hidden" information that may change
- Extraordinarily important! Sometimes it waits until (early!) design



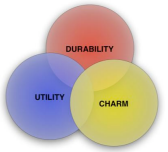
Thought Problems

- You are building a customer care application for a personal computer company with 400 customer care positions in 6 states. The customers can call from around the globe -- what techniques should be used for requirements elicitation and why?
- You are a manager responsible for 2 projects and you have been handed requirements documents for each project, one document is 10 pages long and the other is 200 pages long, what is your approach for V&V?



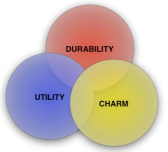
Meyer's Approach

- Natural language has problems: noise, silence, over-specification, contradictions, ambiguity, forward references and wishful thinking
- Sommerville: lack of clarity, requirements confusion (not properly categorized), requirements amalgamation
- Use formal techniques then translate to natural language
- Some formal techniques: entity-relationship modeling, Finite state machines, Structured Analysis and Design Technique (SADT)
- Less formal: Structured natural language, graphical notations



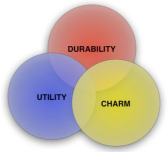
Formal Methods

- Formality = degree of mathematical rigor during analysis and design
  - Mathematically based techniques for describing system properties
  - Strive for consistency, completeness and lack of ambiguity - the usual suspects
  - Issues with less formal approaches: contradictions - statements at variance with each other, usually separated by substantial text in the requirements, ambiguities, vagueness, incompleteness, mixed levels of abstraction
- Mathematics provides a high level of validation



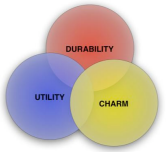
Formal Method Concepts

- Data invariant is a set of conditions that are true throughout the execution of the system that contains a collection of data
  - Example: constraint on table size in a symbol table = maxids and all items are unique names
- State - stored data that the system accesses and alters
- Operator - an action that reads or writes data to a State, e.g. adding and removing names to a symbol table. Operator is associated with two conditions:
  - Precondition - defines circumstances in which a particular operation is valid
  - Postcondition - defines what happens when an operator has completed its action defined by its affect on state
- Brainstorming techniques are useful to develop a data invariant for a complex function, e.g., bounds, restrictions and limitations



Set Preliminaries

- Set - collection of objects and elements, all elements are unique and order is immaterial
- Cardinality - number of items in a set
- Sets are defined by enumerating elements or using a constructive set specification
  - $\{n{:}\mathbb{N} \mid n < 3 \cdot n\}$
    - $n{:}\mathbb{N}$ is signature, specifies range of values considered
    - $n < 3$ is predicate defines how set is constructed
    - n is term provides general form of the item of the set, when n is obvious it can be omitted
  - {0, 1, 2}



More Set Prelims

$\in$ $x \in X$ denotes membership in a set

$12 \in \{6, 1, 2, 12, 22\}$

$x \notin X$ x is not a member of X

$\emptyset$ is empty set

$\emptyset \cup A = A$ and $\emptyset \cap A = \emptyset$

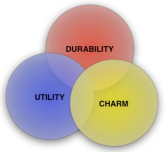
union, cup

intersection, cap

contained in relationships

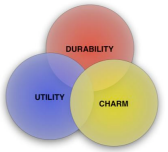
Union combines both sets with duplicates eliminated

Intersection provides list of common elements



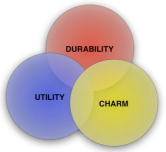
Yet More Set Prelims

- \ is set difference operation removes elements of 2nd from 1st
- X is cross product, each of the elements of the 1st combined with each of the elements of the 2nd
- Powerset is collection of subsets of set
- Logical operators
- Universal quantification
- Sequence, elements are ordered, 1st element is domain, 2nd element is range
- Sequence operators: concatenation, head, tail, front, last



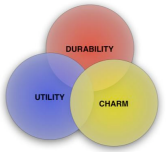
Example: Block Handling

- Blocks of storage held on a file storage device
- State is collection of free blocks, collection of used blocks and queue of returned blocks
- Data Invariants:
  - No block will be marked as used and unused
  - All sets of blocks held in the queue will be subsets of the collection of currently used blocks
  - No elements of the queue will contain the same block numbers
  - Collection of used and unused blocks is the collection of blocks that make up files
  - The collection of {used, unused} blocks have no duplicate block number
- Operations: adds a collection of blocks to the end of queue, checks whether queue of block is empty



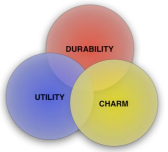
Formal Specification Language

- Three components:
  - Syntax defining specific notation of specification
  - Semantics to help define a universe of objects used to describe the system
  - Set of relations that define the rules indicating which objects properly satisfy the specification
- Syntax is usually derived from formal set theory
- Semantics abstraction usually are things such as states, state transitions, …



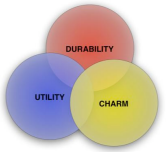
10 Commandments of Formal Methods

- Choose the appropriate notation - that is good match for application
- Formalize but not overformalize - not necessary for every (most) aspects of the system
- Estimate costs - formal methods have high startup, training costs
- Know a formal guru - for consultation
- Keep your standard development methods
- Document sufficiently including natural language commentary



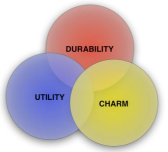
10 Commandments (cont'd)

- Maintain quality standards - Formal Methods do not supplant quality efforts
- Do not be dogmatic - you will still get bugs
- Test, test and test again - does not replace testing
- reuse



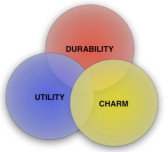
Modeling Approaches

- None are truly comprehensive, usually done to model aspects of the problem.
- Very time intensive
- Some require buying into a full methodology
- Difficult to reflect change and therefore keep synchronized
- Entity-Relationship (Chen), FSMs and SADT (Ross), Data Flow modeling
- Surveying blackboards and white boards usually the pictures are much less formal
- Davis: "The value of a model depends on the view taken, but none is best for all purposes" (E&R)



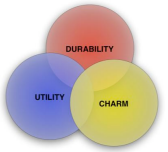
Requirements Validation

- At this stage everything is (usually) informal and incomplete (Boehm's 40%)
- Prototyping/iteration
- Requirements reviews: Sentence by sentence review of document helps as does baselining
- Test case generation -> leads to test plan



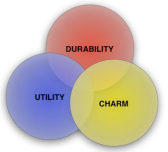
Key Factors for the Review

- Verifiability: testability
- Comprehensibility
- Traceability: source of requirement
- Adaptability: is requirement independent or will changing it affect the rest of system?
- Relationship to other requirements: conflicts, contradicts, contentious, …



Requirements Management

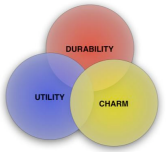
- Change is inevitable
- Therefore:
  - Automated system to manage it
  - Baselined requirements
    - Enduring vs volatile (best guess)
    - Identified and traceable (maybe)
      - Source, requirements and design traceability
  - Link dependent requirements
  - Change management process



Heuristics to Reengineer Requirements

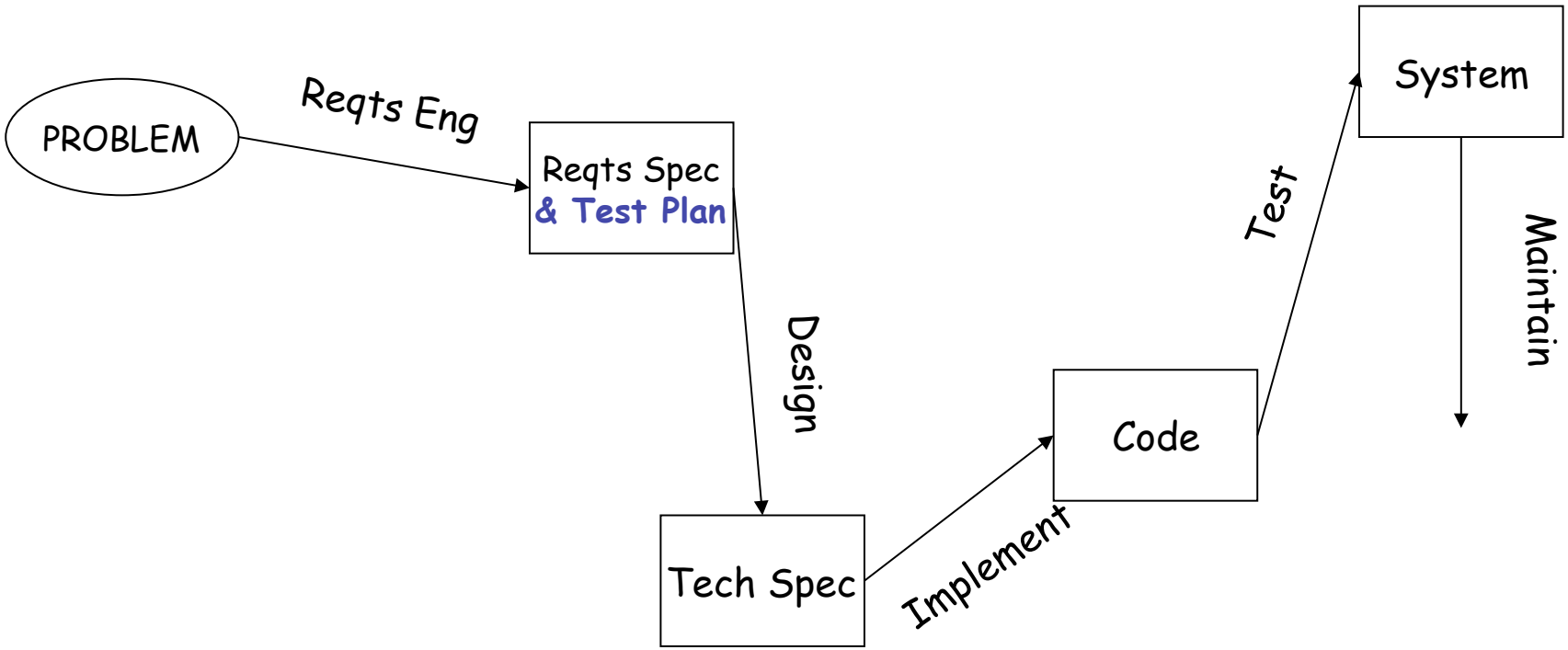
- Move some of the desired functionality into version 2
- Deliver product in stages 0.6, ..
- Cut features
- Refine some features less
- Relax the detailed requirements for some features.

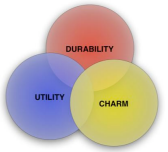
Penn CIS



Your Heuristics for reengineering requirements and preventing feature creep

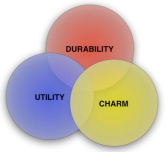
Simplified Model





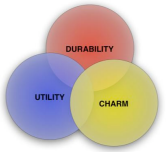
Test Plan

- (more later during testing lecture)
- Get testers involved as early as possible
- Includes scope, approach, resources necessary and schedule
- Lists and describes items and features to be tested
- Testing tasks to be performed
- Roles and Responsibilities - who does what



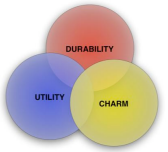
Project Metric Estimates

- Fortunate if requirements are partially done before estimation!
- Used for cost prediction and project monitoring
- Predicting size of system should get easier as project continues - stable specs and fewer tasks remaining
- Two steps:
  - Estimate size
  - Estimate effort and cost from sizing (coding necessary to fulfill requirements). Effort includes resources + calendar time + staff budget constraints
- Cost Prediction usually based on either expected effort or elapsed time



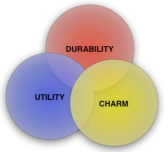
Cost Estimation

- Cost estimation falls into these categories (Kitchenham, 1994):
  - Expert - Delphi Method
  - Analogy
  - Decomposition
  - PERT Models - Delphi Method
  - Mathematical Models - COCOMO, Rayleigh Curve, Function Points
  - Parkinson's Law- work expands to fill time available
  - Pricing to win- cost and effort is whatever customer has to spend
  - Top-Down/Bottom-Up



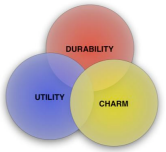
Brooks Chapter 2

- Projects fail usually for lack of calendar time
  - Estimation is optimistic, last bug syndrome
  - Costs vary by staff months, progress does not
  - Sequential nature of some tasks
  - Communication is key difference, farm workers vs. developers
  - Each new worker must be trained by experienced staff
  - Communication rule of thumb n(n-1)/2
- Testing most mis-scheduled part
- Brooks heuristics:
  - 1/3 planning
  - 1/6 coding
  - 1/4 component and early system test
  - 1/4 system test, whole system



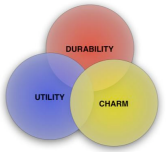
Brooks, Chapter 2 (cont'd)

- "Gutless estimating … urging of patron"
- Solutions:
  - Industry wide need to publish data
  - "Take no small slips"
  - Trim the task
- **Brook's Law: "Adding manpower to a late software project makes it later"**
- Caveats:
  - #of months depends on sequential constraints
  - #of staff depends on # of independent subtasks (and ability to match talent to task)



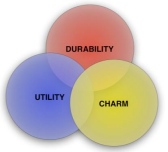
Brooks, Chapter 8, Calling the shot

- Do not estimate the whole task by estimating coding and multiplying by 6!
- Small programs : large programs :: 100 yard dash : 1 mile
  - Effort increases as a power of size w/o factoring communication
- Unrealistic assumptions as to how much of a Developer's time is allotted to development - studies show only 50% of the time
- Productivity is also related to complexity of the task, more complex, less lines/year - high level languages help (still relevant today)



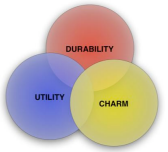
Software Estimation

- Begins by designating a unit of estimation:
  - Initially KLOC and then divide by LOC/month figures (3KLOC/yr, average programmer productivity rate -- Futrell, et.al.)
  - In addition to LOC, function points, feature points, object (application) points, number of bubbles on Data Flow Diagram, objects, classes, number of pages of user documentation, …
- Some data was collected by IBM - regression analyses
- Evolved to an estimate through regression analyses adjusted by unique characteristics of the project
  - assessed by 10 to n factors that weight the final estimate



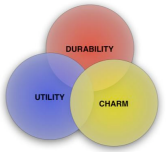
Easy?

- 15% of all software projects fail to meet their goals, overruns of 100-200% are common.
- "When performance does not meet the estimate, there are two possible causes: poor performance or poor estimates. In the software world, we have ample evidence that our estimates stink, but virtually no evidence that people in general don't work hard enough or intelligently enough." -- Tom DeMarco
- Boehm - no project can finish in less than 75% of theoretical time



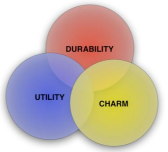
Capers Jones Table

Language	Ratio-Source:Executable
Assembler	1:1
Macro-Assembler	1:1.5
C	1:2.5
ALGOL	1:3
COBOL	1:3
FORTRAN	1:3
Pascal	1:3.5
RPG	1:4
PL1	1:4
MODULA-2	1:4.5
Ada	1:5
PROLOG	1:5
LISP	1:5
FORTH	1:5
BASIC	1:5
LOGO	1:6
4th-GLs	1:8
APL	1:9
OBJECTIVE-C	1:12
SMALLTALK	1:15
Query-Languages	1:20
Spreadsheets	1:50



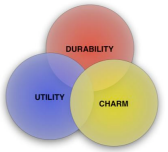
SLOC!

- Source Lines Of Code (basis for KLOC):
  - Make sure single statement, not two separated by semicolon
    - Syntax issue
  - All delivered, executable statements (OA&M, test harness)
  - Count data definitions only once
  - No Comments?
  - Count all instances of calls, subroutines, …
  - Class project once - an extra credit adventure
- No industry standards
- This can be toyed with if you base compensation on it!
- Of course LOC is a "results measure"



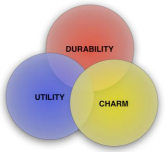
COCOMO

- COnstructive COst MOdel
- Initially based on Boehm's analysis of a database of 63 projects - models based on regression analysis of these systems
- Linked to classic waterfall model
- Effort is number of Source Lines of Code (SLOC) expressed in thousands of delivered source instructions (KDSI) - excludes comments and unmodified utility software
- Model has 3 versions and considers 3 types of systems:
  - Organic - simple business systems
  - Embedded - avionics
  - Semi-detached - management inventory systems



COCOMO System Types

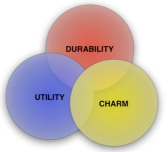
	SIZE	INNOVATION	DEADLINE	CONSTRAINTS
Organic	Small	Little	Not tight	Stable
Semi-Detached	Medium	Medium	Medium	Medium
Embedded	Large	Greater	Tight	Complex hdw/ customer interfaces



COCOMO Formula

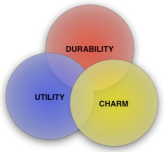
Effort in staff months $= b * KDLOC^c$

	b	c
organic	2.4	1.05
semi-detached	3.0	1.12
embedded	3.6	1.20



A Retrospective on the Regression Models

- They came to similar conclusions:
  - Time:
    - Watson-Felix $T = 2.5E^{0.35}$
    - COCOMO(organic) $T = 2.5E^{0.38}$
    - Putnam $T = 2.4E^{0.33}$
  - Effort:
    - Halstead $E = 0.7\ KLOC^{1.50}$
    - Boehm $E = 2.4\ KLOC^{1.05}$
    - Watson-Felix $E = 5.2\ KLOC^{0.91}$

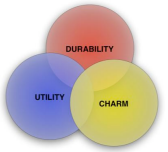


Intermediate COCOMO

- Adds 15 attributes of the product that has to be rated on a six point scale from Very Low to Extra High
- There are 4 categories of attributes: product, computer, personnel and project.
- The ratings are reflected in P of the equation
  - P < 1, less effort; P > 1 more effort

Effort in staff months $=(b*KDLOC^{C})$ *P

Effort = A * $Size^{B}$ * M (Sommerville)

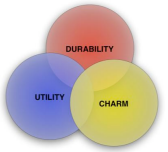


Intermediate COCOMO Attributes

- PRODUCT:
  - RELY - required reliability
  - DATA- data bytes per DSI (smaller db)
  - CPLX - code complexity (VH= real time)
- COMPUTER:
  - TIME - execution time, % used
  - STOR - storage requirements, % used
  - VIRT - changes made to hdw and OS
  - TURN- Dev turnaround time, batch vs interactive
- PERSONNEL
  - ACAP - analyst capability, skills
  - PCAP - programmer capability
  - AEXP- applications experience
  - LEXP - language experience
  - VEXP- virtual machine experience
- PROJECT
  - MODP - Modern Development Practices
  - TOOL - use of sfw tools
  - SCED - amount of schedule compression - recall Boehm quote, 75% limit

Intermediate COCOMO Attributes

http://www.cs.unc.edu/~stotts/COMP145/cocomo6.gif



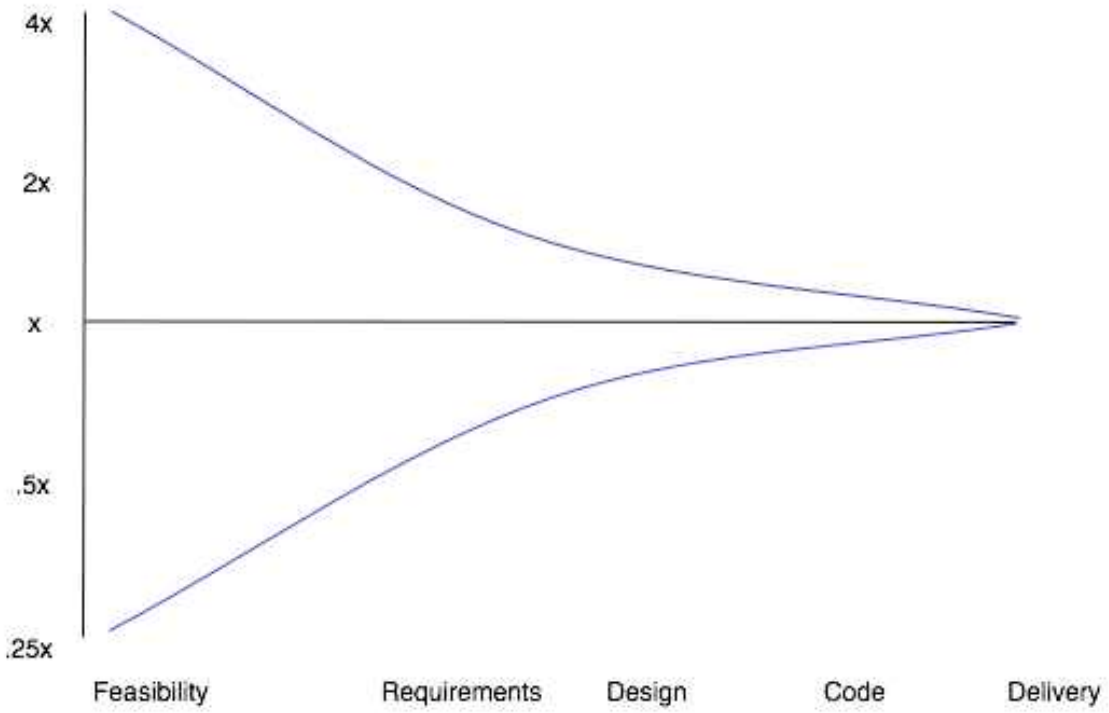
Project Characteristics Table

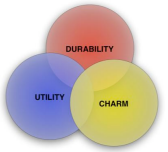
Cost adjustments for computing the EAF (Effort Adjustment Factor)

	v.low	low	nominal	high	v.high	ex.high
product attributes						
required software reliability	0.75	0.88	1.00	1.15	1.40	
database size		0.94	1.00	1.08	1.16	
product complexity	0.70	0.85	1.00	1.15	1.30	1.65
computer attributes						
execution time constraints			1.00	1.11	1.30	1.66
main storage constraints			1.00	1.06	1.21	1.56
virtual machine volitility	0.87	1.00	1.15	1.30		
computer turnaround time		0.87	1.00	1.07	1.15	
personnel attributes						
analyst capability	1.46	1.19	1.00	0.86	0.71	
applications experience	1.29	1.13	1.00	0.91	0.82	
programmer capability	1.42	1.17	1.00	0.86	0.70	
virtual machine experience	1.21	1.10	1.00	0.90		
programming language experience	1.14	1.07	1.00	0.95		
project attributes						
use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
use of software tools	1.24	1.10	1.00	0.91	0.83	
required development schedule	1.23	1.08	1.00	1.04	1.10	

Estimate Uncertainty

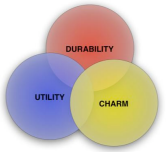
(by stage in life cycle)





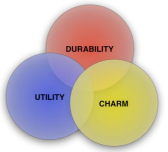
COCOMO II

- Mapped to life cycle, three phases:
  - Application Composition Model -based on counting Object points where objects are screens, reports and 3GL (C, FORTRAN,BASIC) modules weighted by difficulty. Object points are easier to determine at an earlier time.
  - Early Design Model uses unadjusted function points which are converted to SLOC based on language. Rather than Function Point cost structures, uses its own cost drivers which are related to third model, the Post Architecture Model. Cost drivers are: product reliability and complexity, required reuse, platform difficulty, personnel experience, personnel capability, facilities and schedule. Rated on 7 point scale.



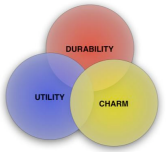
COCOMO II-2

- Post Architecture Model is most detailed model. Differs from original *COCOMO* in set of cost drivers, and range of values to parameters. New cost drivers are:
  - Documentation needs
  - Personnel continuity
  - Required reusability
  - Multi-site development
  - (-) computer turnaround time
  - (-) use of modern programming practices



Application (Object) Points

- Count of separate screens displayed, range from 1 to 3 points as function of complexity
- Count of reports produced, range from 2 to 8 points as function of difficulty to do
- Count of modules in programming language, each 10 points



Project Duration

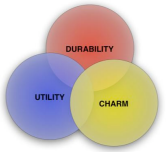
- *COCOMO II* =

$$TDEV = 3 * (PM)^{(0.33+0.2*(B-1.01))}$$

PM is effort computation

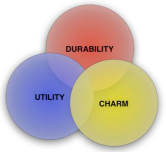
B reflects size of project

TDEV is duration of development in months



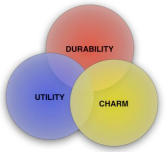
The COCOMO Family

- COCOMO II - estimates cost, effort and schedule of a perspective project
- COCOTS - estimates cost, effort and schedule associated with using commercial off-the-shelf components (COTS) in a software development project. (research project)
- COQUALMO - explores relationship between cost, schedule and quality
- CORADMO - estimates cost of developing software using rapid application development techniques.
- COPROMO - predicts the most cost effective allocation of investment resources in new technologies intended to improve productivity
- COPSEMO - estimates cost of developing software as distributed over development activity stage
- COSYSMO - estimates the system engineering tasks in software intensive projects
- http://sunset.usc.edu/research/cocomosuite/suite_main.html



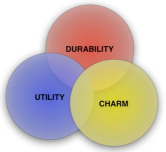
Function Points

- A break from KLOC - important because estimates can be based on design data vs. results based measures
- Albrecht and Gaffney, '70s, IBM
- Based on counting data structures
- Great for business apps or any that are data dependent, not as good for apps that have a heavy algorithmic component, e.g., compilers
- Five factors in the Function point Analysis Model:
  - I # of input types that are user inputs changing data structures.
  - O # of output types
  - E # of inquiry types, input controlling execution. E.g., menu selection
  - L # of logical internal files, internal data used by system such as index files
  - F # of interfaces data output or shared with another app



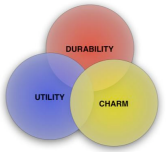
Function Point Calculations

- Unadjusted Function Points
  - UFP= 4I + 5O + 4E + 10L + 7F
- The constants can be altered if we know more about the data (next slide)
  - Each input type has a number of data element types (attributes) and refers to other file types. As these increase, the complexity level increases, resulting in changes to the constants
- Also "backfiring" - Capers Jones relating FP to LOC



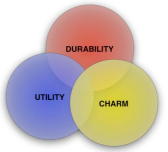
Complexity Table

TYPE:	SIMPLE	AVERAGE	COMPLEX
INPUT (I)	3	4	6
OUTPUT(O)	4	5	7
INQUIRY(E)	3	4	6
LOG INT (L)	7	10	15
INTERFACES (F)	5	7	10



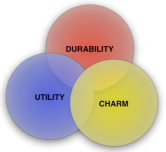
Adjusted Function Points

- Application characteristics are considered
- Considers 14 characteristics (environmental factors)on a 6 point scale (0-5)
- Total Degree of Influence (DI) is sum of scores.
- DI is converted to a technical complexity factor (TCF)
  - TCF = 0.65 + 0.01DI
- Adjusted Function Point is computed by
  - FP = UFP * TCF
- For a given language there is a direct mapping from Function Points to LOC
- COUNTING FUNCTION POINTS IS NOT EASY!
- Feature points - extension of function points to deal with expanded set of applications, such as embedded and real time systems.
- And Object points- at a higher level than function points, one object point to each unique class or object



Application Characteristics for Function Point Analysis

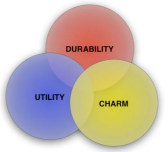
- Data Communications
- Distributed Functions
- Performance
- Heavily used configuration
- Transaction rate
- Online Data Entry
- End-user efficiency
- Online update
- Complex processing
- Reusability
- Installation ease
- Operational ease
- Multiple sites
- Facilitate change



Initial Conversion

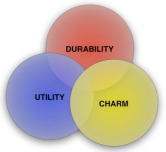
http://www.qsm.com/FPGearing.html

Language	Median SLOC/function point
C	104
C++	53
HTML	42
JAVA	59
Perl	60**
J2EE	50
Visual Basic	42



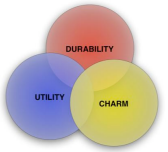
Delphi Method

- A group of experts can give a better estimate
- The Delphi Method (aka Wideband Delphi):
  - Coordinator provides each expert with spec
  - Experts discuss estimates in initial group meeting
  - Each expert gives estimate in interval format: most likely value and an upper and lower bound
  - Coordinator prepares summary report indicating group and individual estimates
  - Group iterates until consensus



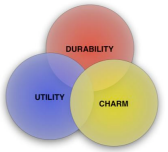
Other Estimation Models/Products

- KnowledgePlan - Capers Jones
- SLIM - Putnam
- SEER - Jensen (worked with Putnam)
- Your favorite here
  - Earned Value Analysis/Management-track how you are doing (COCOMO-II has elements of that)
  - In 2002 ~50 software estimation tools available
- Other: interface screen counts, classes and methods (nouns and verbs), …
- History of success/feedback, inter and intra product is essential!



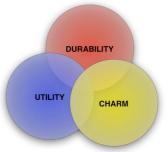
Requirements Process & Estimates

1. ICED-T analysis: Intuitive, Consistent, Efficient, Durable, Thoughtful
2. Simplified Quality Functional Deployment
3. Compute effort
4. Estimate staff and development time
5. Revise requirements to meet above (if exceeds cost, time)
6. Redo 1-4
7. Replan with GANTT chart (optional)
8. Review MOV (Measurable Operational Value) to see if it is worth it



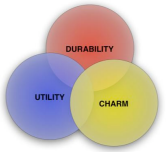
Heuristics to do Better Estimates

- Decompose Work Breakdown Structure to lowest possible level
- Review assumptions with all stakeholders
- Do your homework - past organizational experience
  - HISTORY: organization -> corporation
- Retain contact with Developers
- Update estimates and track new projections (and warn)
- Use multiple methods
- Reuse makes it easier (and more difficult)

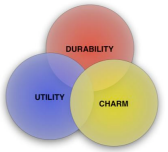


Heuristics to Cope with Estimates

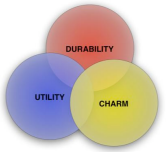
- (from S. McConnell, "How to defend an unpopular schedule, IEEE Software, 13(3), 1996.)
- More developers, if it is early enough
- Higher output developers (order of magnitude difference)
- More admin support
- Increase degree of developer support (faster machines, …)
- Eliminate red tape (50% issue)
- Devote full time end user to project
- Increase level of exec sponsorship to break new ground (new tools, techniques, training)
- Set a schedule goal date but agree to reassess after detailed design
- Use broad estimation ranges rather than single point estimates



Your Heuristics to do and Cope with Estimates?

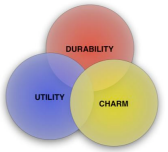


How many of you do risk management, to what extent and is it throughout the life cycle?



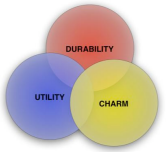
Some Risks

- Staff turnover
- Management change
- Hardware unavailability
- Requirements change
- Specification delays
- Size underestimate
- Failing/ Lack of support software
- Technology change
- Product competition
- Environmental factors



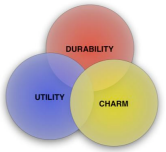
Risk Management

- General signals: new domain, new developers, new management, dictated schedules …
- Risk Management process
  - Identify risk factors
  - Determine risk exposure
  - Develop strategies to handle risks (avoidance, transfer, acceptance)
  - Handle it



Top 10 Risk Factors (Boehm)

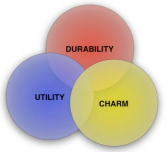
- Personnel shortfall
- Unrealistic schedule or budget
- Wrong functionality
- Wrong user interface
- Gold plating (fun and games)
- Requirements volatility
- Bad external components
- Bad external tasks (subcontracts)
- Real time shortfalls
- Capability shortfall (bleeding edge)



Software Risk Management

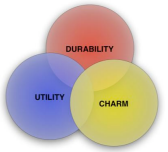
- (much of this adapted from http://www.eas.asu.edu/~riskmgmt/intro.html and the SEI)
- Risk is defined as exposure to harm or loss, not only probability but effect as well.
- NOT RISK AVOIDANCE
- The SEI (and others) phases of risk analysis are:
  - Identify
  - Analyze
  - Plan
  - Track
  - Control

C O M M U N I C A T I O N



Identify Risk

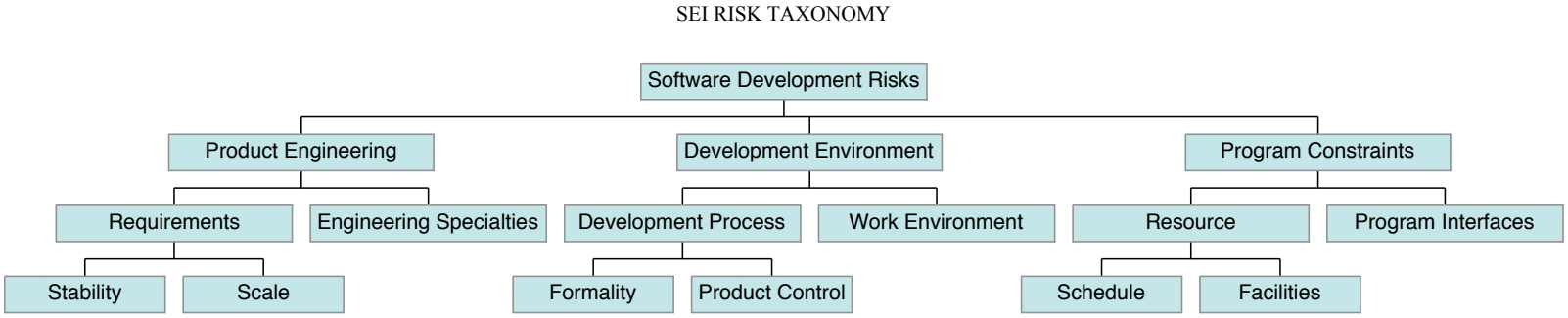
- Risks can be known, unknown and unknowable or known knowns, known unknowns (apply to this project), unknown unknowns :-)!
- SEI method of risk identification (and management) based on following assumptions:
  - Risks are often known by tech staff but poorly communicated
  - A repeatable method is necessary for risk management
  - Must cover all areas
  - Attitude must be non-judgmental and supportive so that controversial views can be heard
  - Success or failure of the project can not be based solely on risk assessment

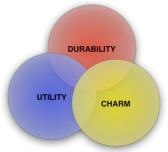


SEI Development Risk Taxonomy

- 3 major categories:
  - Product engineering - what is being developed
  - Development Environment - how it is being developed
  - Program constraints - contractual, organization and operational factors

Taxonomy Expanded
(adapted from CMU/SEI-93-TR-6)



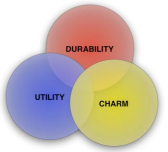


A. Product Engineering
   1. Requirements
      a. Stability
      b. Completeness
      c. Clarity
      d. Validity
      e. Feasibility
      f. Precedent
      g. Scale
   2. Design
      a. Functionality
      b. Difficulty
      c. Interfaces
      d. Performance
      e. Testability
      f. Hardware Constraints
      g. Non-Developmental Software
   3. Code and Unit Test
      a. Feasibility
      b. Testing
      c. Coding/Implement-ation
   4. Integration and Test
      a. Environment
      b. Product
      c. System
   5. Engineering Specialties
      a. Maintainability
      b. Reliability
      c. Safety
      d. Security
      e. Human Factors
      f. Specifications

B. Development Environment
   1. Development Process
      a. Formality
      b. Suitability
      c. Process Control
      d. Familiarity
      e. Product Control
   2. Development System
      a. Capacity
      b. Suitability
      c. Usability
      d. Familiarity
      e. Reliability
      f. System Support
      g. Deliverability
   3. Management Process
      a. Planning
      b. Project Organization
      c. Management Experience
      d. Program Interfaces
   4. Management Methods
      a. Monitoring
      b. Personnel Management
      c. Quality Assurance
      d. Configuration Management
   5. Work Environment
      a. Quality Attitude
      b. Cooperation
      c. Communication
      d. Morale

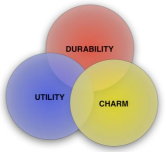
C. Program Constraints
   1. Resources
      a. Schedule
      b. Staff
      c. Budget
      d. Facilities
   2. Contract
      a. Type of Contract
      b. Restrictions
      c. Dependencies
   3. Program Interfaces
      a. Customer
      b. Associate Contractors
      c. Subcontractors
      d. Prime Contractor
      e. Corporate Management
      f. Vendors
      g. Politics

Figure A-1 Taxonomy of Software Development Risks



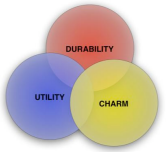
Analyze Risk

- Probability of risk, USAF Handbook categories are very low, low, medium, high and very high
- Impact of risk, USAF Handbook categories are negligible, marginal, critical and catastrophic
- Risks are rarely independent
- A matrix is used to determine overall risk for different categories (e.g., effort, performance, schedule, cost, support)



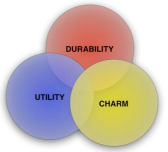
Sample Impact/Probability Matrix
(used to calculate overall risk)

Impact/Probability	V. High	High	Medium	Low	V. Low
Catastrophic	H	H	M	M	L
Critical	H	H	M	L	0
Marginal	M	M	L	0	0
Negligible	M	L	L	0	0



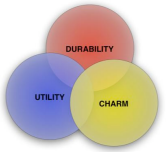
Plan for the Risks

- What can you do:
  - Mitigate impact by developing a contingency plan should risk occur and identify the trigger to initiate the contingency plan
  - Avoid the risk by changing something
  - Accept the risks and the consequences if it occurs
  - Study the risk further so that you can decide on one of the above
- In addition:
  - Specify why risk is important
  - What info is need to track status of risk
  - Who is responsible for Risk Management and what is the cost



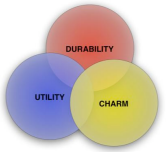
Risk Tracking and Control

- Track like everything else in the project monitoring status of risks and actions taken to address them. Appropriate risk metrics should be in place.
- Control, the risk process should be in place in the beginning, deviations to the plan should be corrected, triggering events should be handled and the process should be assessed for effectiveness.



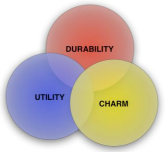
So Far

- Software Process Models
- Software Project Planning (woosh!)
- Requirements
- Estimation
- Risk Analysis
- Next
  - Software Architecture



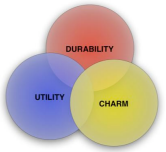
Thought Problems

- What mechanism would you establish to provide better estimates of time and effort for software projects in your group, division and company
- You've just been promoted and now head a new team (with all new hires, none that you know) that has been assigned to do a high risk project - what is your plan of attack for risk management?



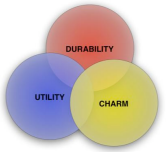
Brooks, Chapter 3

- "small is beautiful, big is necessary" - Bob Factor
- Programmer productivity can vary as much as an order of magnitude and the space and speed of the code can be as much as 5 times better
- Systems should be built by as few people as possible, a small sharp team of < 10
- However OS 360 (1963-'66) required 5,000 staff years. Even with order of magnitude improvement, team of < 10 would take over 50 years!



Chapter 3 (cont'd)

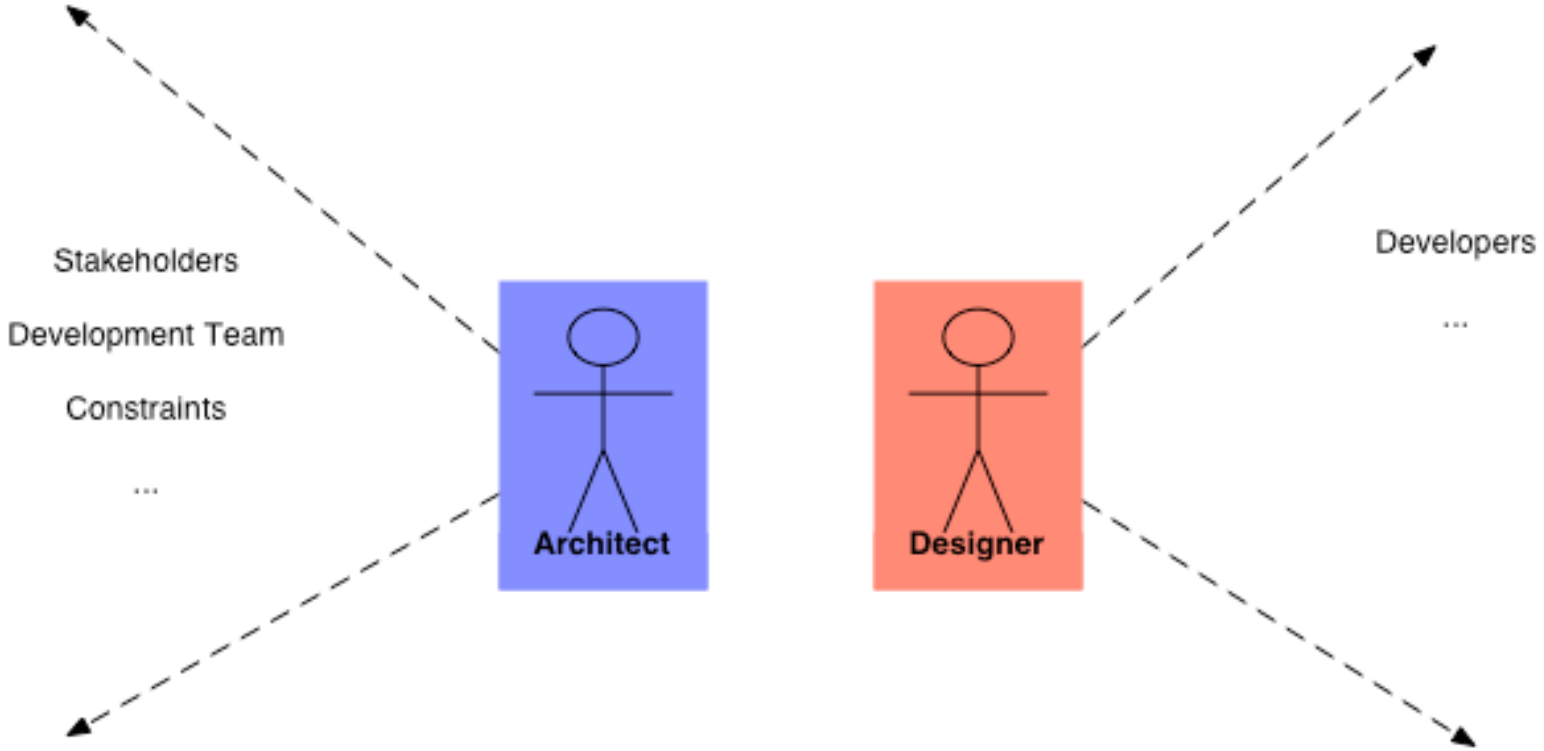
- Harlan Mills: surgical team rather than hog butchers! One does the cutting and the rest do support
- The roles:
  - Surgeon = chief programmer/boss lots of experience (10+ years) and application knowledge
  - Copilot = shares in the design as thinker, discussant and evaluator. Knows all the code. (My buddy system)
  - Administrator- money, people, space, machines, bureaucracy
  - Editor- surgeon generates documentation, ed does rest
  - Secretaries
  - Program clerk- maintains all the records in a programming project
  - Toolsmith - tools/libraries that need to be built or adapted
  - Tester
  - Language lawyer
- 10 people, 1 mind, 200 people only 20 minds have to be coordinated

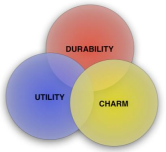


Software Architecture
firmitas, utilitas, venustas

- Vitruvius, good design = durability, utility and charm
- Software architecture = top level decomposition of system into major components and how these components interact
- Much more than high level design!
  - Vehicle for communication among stakeholders - must be understood by all
  - Captures early design decision - structures development (WBS) and testing
  - Transferable abstraction
    - Basis for reuse
    - Essential decisions captured
    - Basis for training

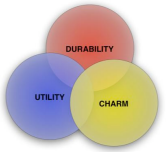
Architecture and Design





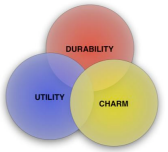
Brooks, Chapter 4

- Conceptual integrity - one set of design ideas
- Ease of use (and understanding) = conceptual integrity
  - A system must have a powerful metaphor that is uniformly applied through out the system (by the architect) .. Attributed to Alan Kay
- Good features and ideas that do not integrate with a system's concepts should be omitted, BUT if this happens too frequently - redesign
- Architect does not steal all the fun, cost/performance is implementers task
- Constraints on the architect are good
- Brooks multimillion dollar mistake - the appeal of putting everyone to work!



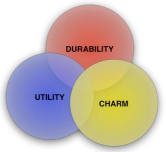
Top-down Design

- Design as a sequence of refinement steps
  - First sketch rough task definition and rough solution, refine
    - Each refinement in task definition results in a refinement in the algorithm
- Avoid bugs by:
  - Clarity of structure and representation makes precise statements about requirements and functioning of the module easier
  - Partitioning and independence
  - Suppression of detail
  - Design can be tested at each of the refinement steps
- On testing - allot for scaffolding - programs and data built for debugging and testing but never intended as a final product - results in 50% more code!



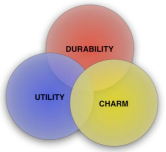
Brooks, Chapter 5

- First system spare and clean with deferred ideas
- Second system dangerous, tendency to overdesign (true for me)
- Leap year example -- 26 bytes permanently resident vs 1 manual op every 4 years
- Overlays versus compilers
- Beware of your second system!



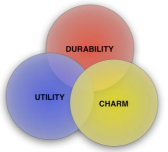
Brooks Chapter 6
On the Life of a Spec

- Must have specs and the changes must be quantized on scheduled dates (and source controlled)
- Be careful about overprescription in any spec -- there is always temptation.
- Spec should be done by 1 or a few -- should have a common, consistent voice
- Should say what it is and what it is not and what constraints it reflects (or imposes)
- If using formal descriptions along with prose, one must be the standard
- "never go to sea with two chronometers, take one or three."
- "If you have multiple standards, there are none!"



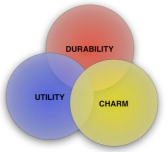
Chapter 9, 10 lbs in a 5 lb Sack

- Space is a cost, especially when memory is expensive - still an issue today
  - Footprint
- Concept of setting memory, cpu, footprint budgets
- "Fostering a total-system, user oriented attitude may well be the most important function of the programming manager" - p.100



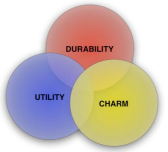
More on Architecture

- Barry Boehm and his students at the USC Center for Software Engineering write that a software system architecture comprises:
  -A collection of software and system components, connections, and constraints.
  -A collection of system stakeholders' need statements.
  -A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements



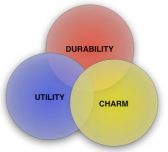
Sommerville Arch ?s

- Is there a generic Application architecture that can be a template
- How will the system be distributed across processors (cores) DISTRIBUTED SYSTEM ARCHITECTURES
- What are the appropriate architectural styles? – see appendix
- What will guide module decomposition? Object, function
- What strategy is used to control the operation of the system units? Centralized, event based, data driven,…
- How will architecture be evaluated?
- How should it be documented?



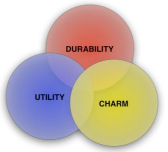
Influences on Architecture

- Requirements
- Development organization - previous architectures influence new ones
- Background, expertise and preferences of architect
- Technical organization and environment
  - Government rules may dictate division of functionality
  - Software engineering techniques of organization
- Architecture is outward looking - how system fits in environment



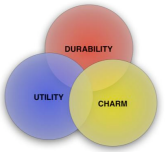
Views of Architecture
(that it must accommodate)

- Conceptual/logical view - major elements and interaction (ADLs)
- Implementation view - modules, packages, layers
- Process view - tasks, communication, allocation of functionality, "alive view," especially with concurrency
- Development view - allocation of tasks to physical nodes
- Augmented by scenarios emphasizing architectural aspects and, in specific instances by other views such as UI, security, …

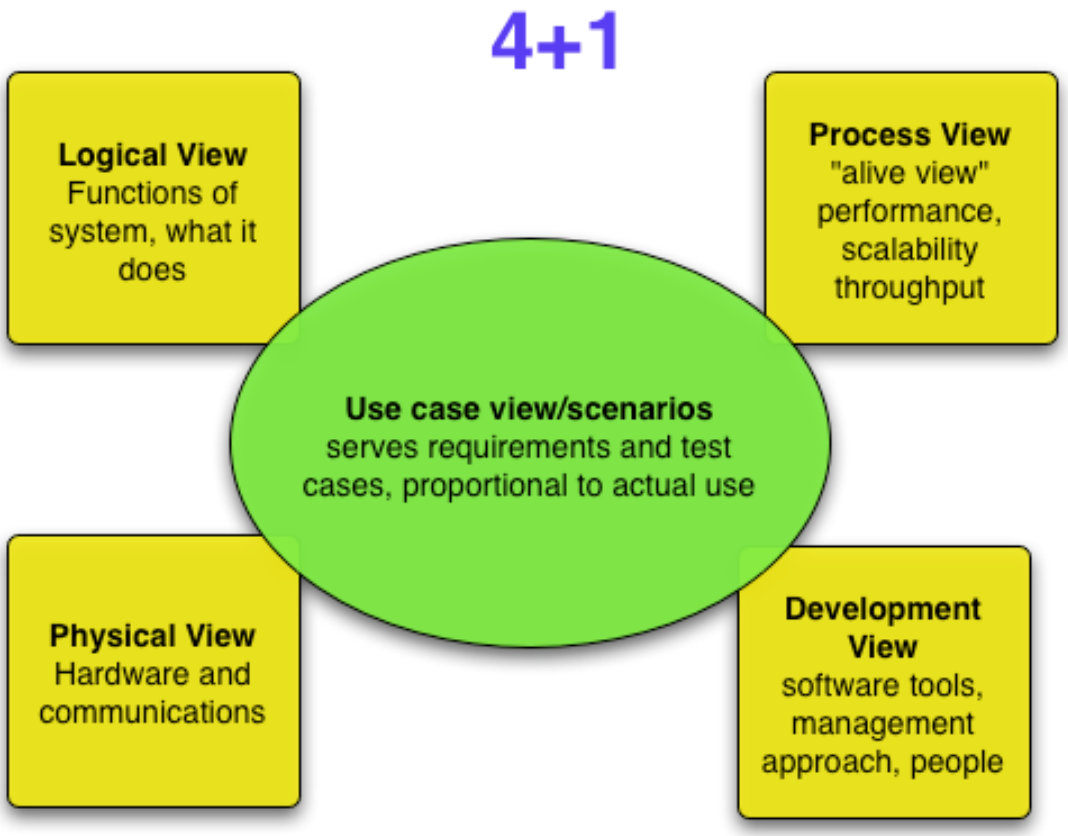


Psychology of the Architect

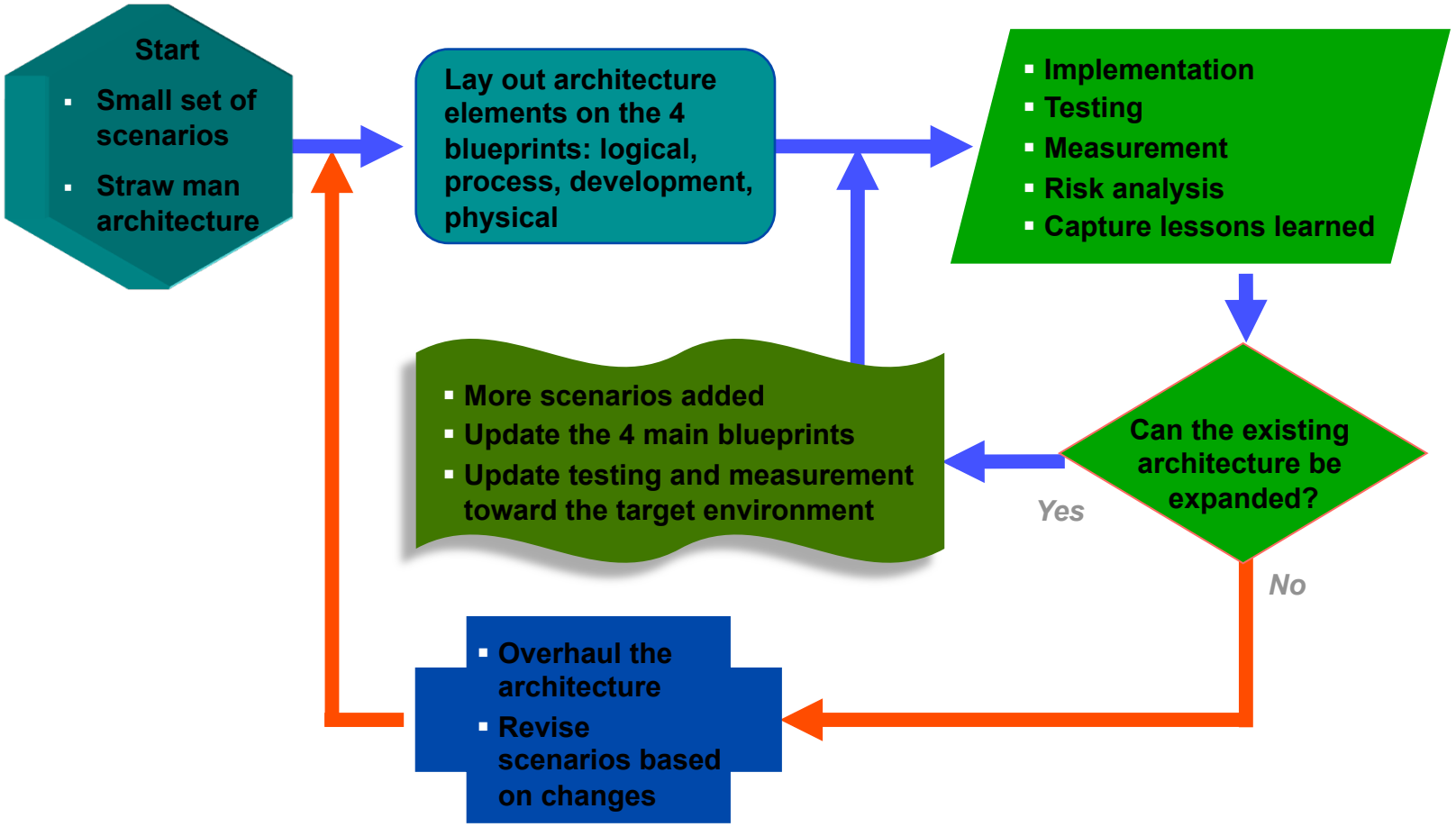
- Architects/developers think of program in chunks
  - Chess/Go/Baseball studies
  - Identifying patterns at a higher level of abstraction
- Design patterns, architectural styles and patterns
- Design pattern is a recurring solution to a standard problem
  - Classic - Model-View-Controller
  - Design patterns = micro architectures

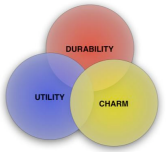


Architecture Approach✓



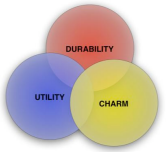
The "4+1" Architecture Model





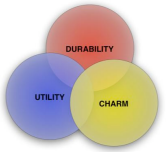
The "4+1" Architecture Model

- Don't Forget to Document It !
  - Key outlines of a software architecture document:
    - Scope
    - References
    - Software Architecture
    - Architecture Goals & Constraints
    - Logical Architecture
    - Process Architecture
    - Development Architecture
    - Physical Architecture
    - Scenarios
    - Size and Performance
    - Quality



KWIC Index
adapted from Parnas

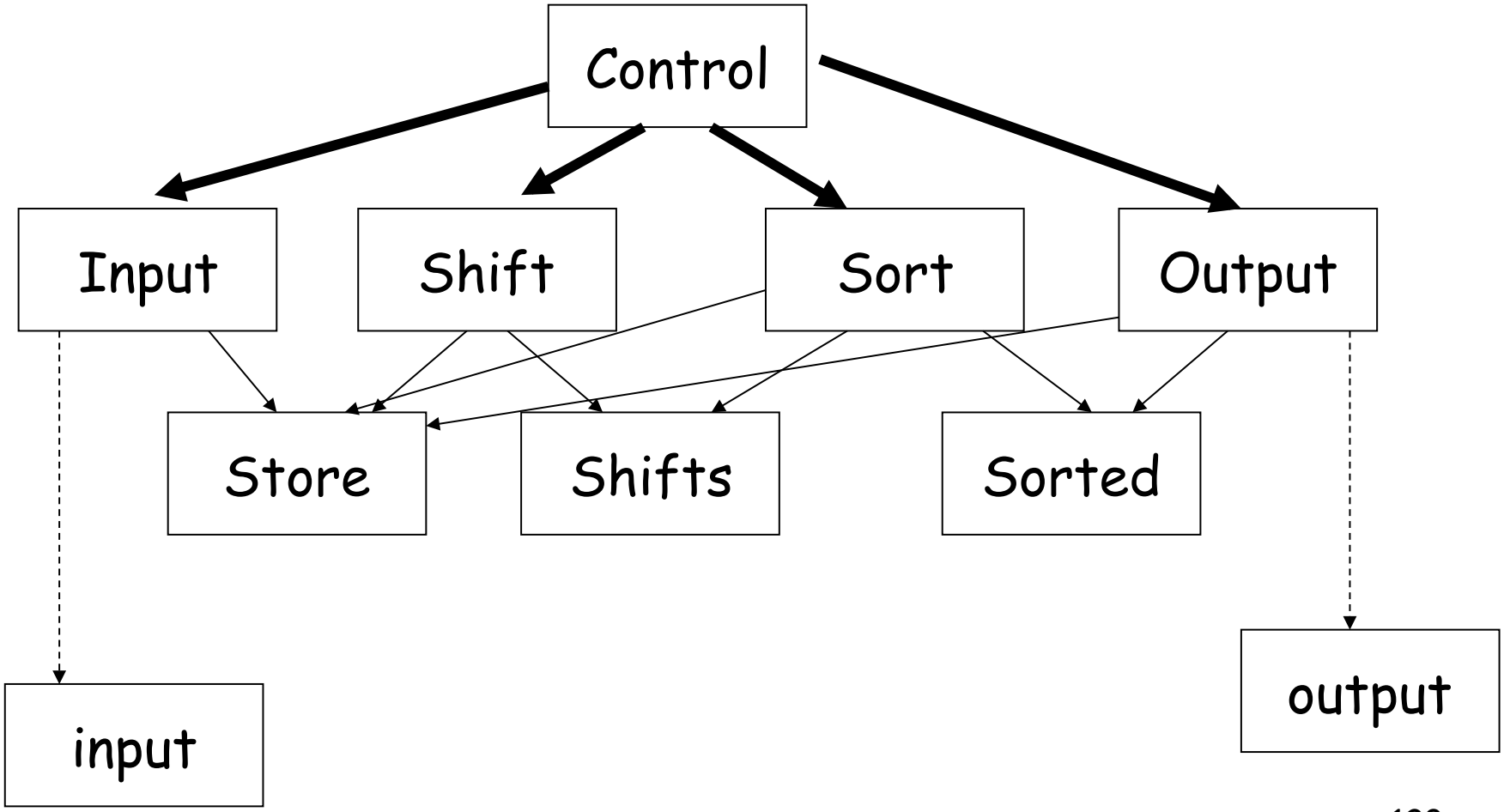
- Key Word In Context = KWIC
- Generate n shifts where n is the number of words in a line to find the key words in a sentence
- E.g., Software engineering for fun and profit, generates:
  - Software engineering for fun and profit
  - Engineering for fun and profit software
  - For fun and profit software engineering
  - …
- Lines then sorted in lexicographic order



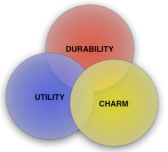
Main Program with subroutines

- Decompose tasks
  - Read & store input
  - Determine all shifts
  - Sort the shifts
  - Write the sorted shifts
- Modules are geared towards actions with respect to time

Main Program with Subroutines

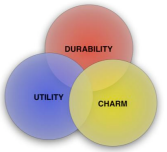


Penn CIS

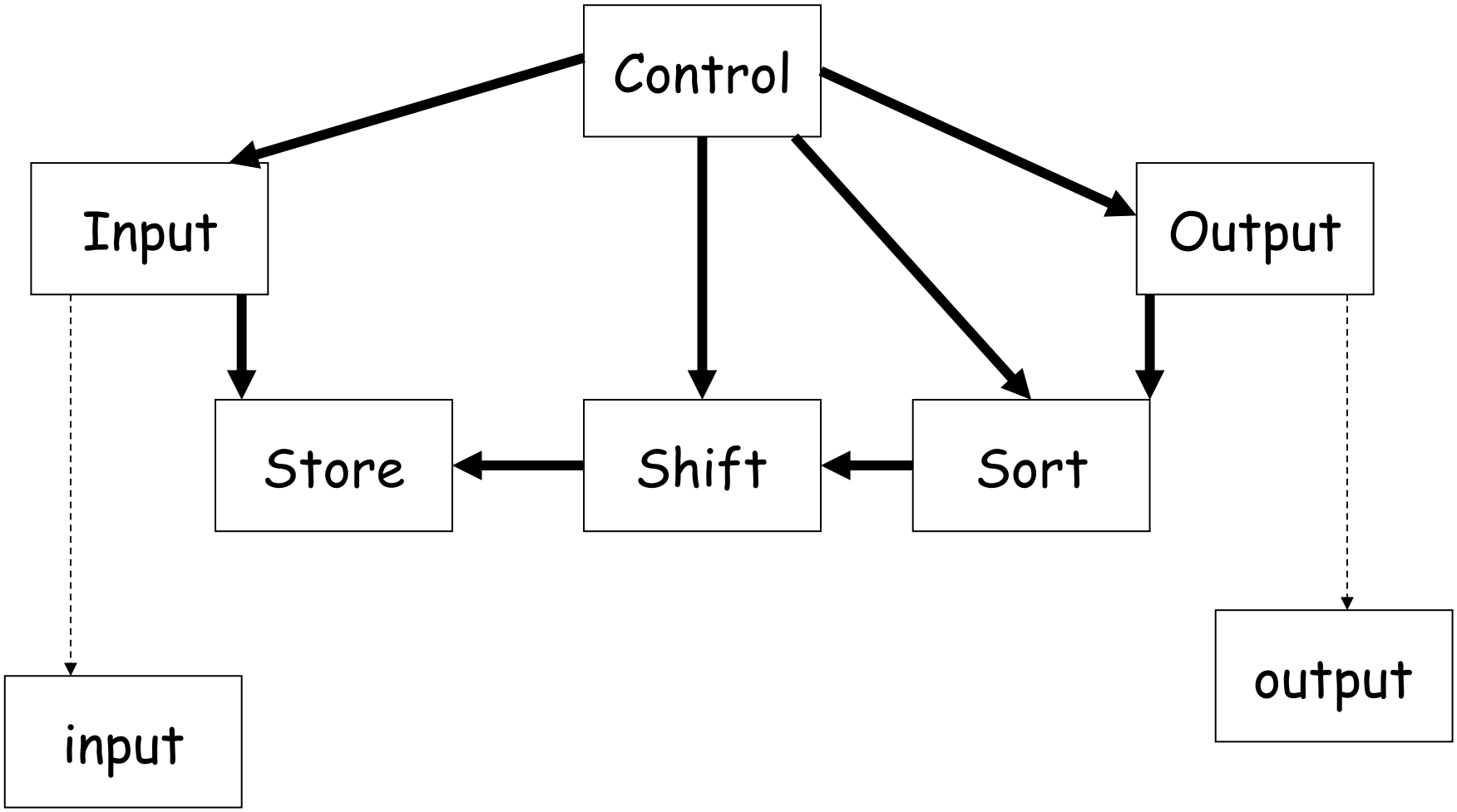


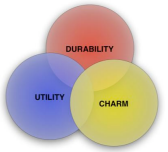
Abstract Data types

- Make type decisions about data representation at an early stage
- Access data through procedures rather than directly
  - Design stage only requires agreement about procedure interface
  - Changes in data can be made easily w/o affecting other modules



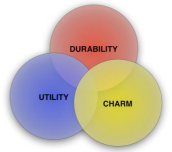
Abstract Data Type



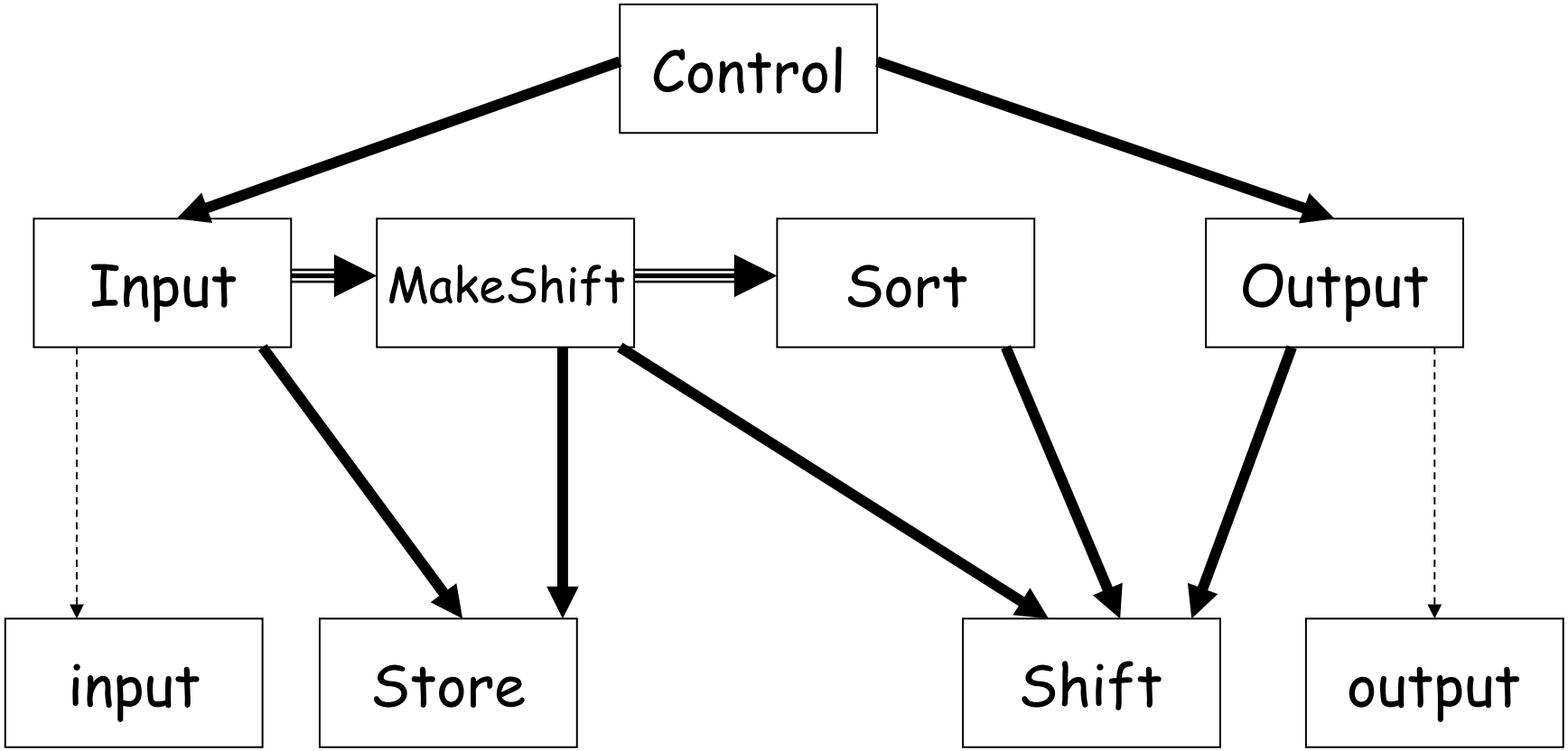


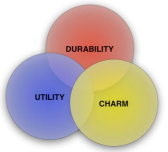
Implicit Invocation

- Want to get rid of uninteresting shifts (or include only interesting shifts)
- Could filter the data but it is inefficient
- Change the shift module - but we may be messing with it too much
- Solution - we do not explicitly call make shift we generate an EVENT.
  - Modules can associate procedures to these events (another level of indirection)



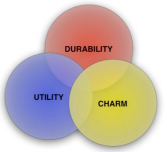
Implicit-invocation



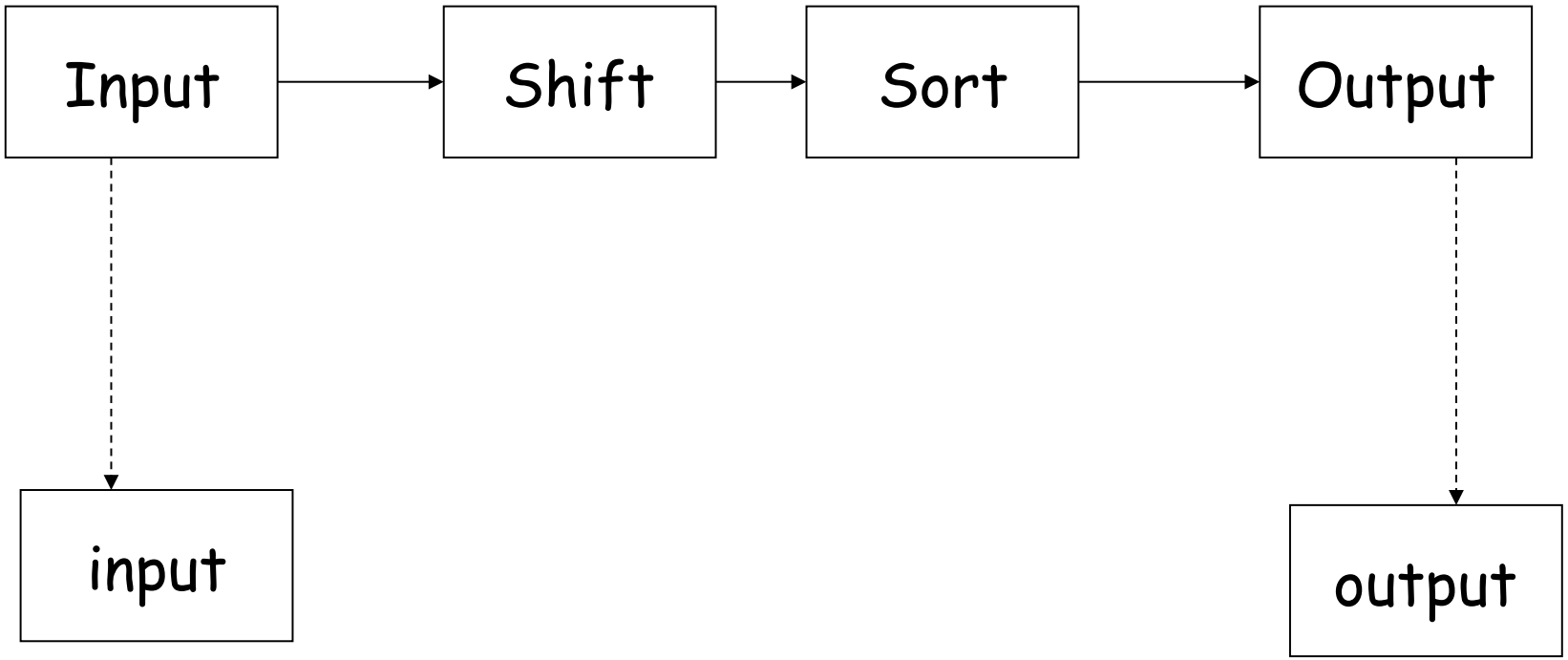


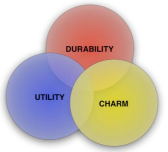
Pipes and Filters

- Good old UNIX
- Since each program reads it input in same order we can use pipes and filters
- But
  - Error handling is primitive, UNIX uses separate error channel
  - Still awesomely convenient and powerful



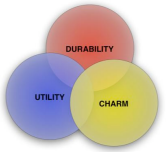
Pipes-and-Filters





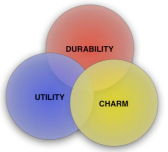
Evaluation of These Architectures

- All work
- Differences become apparent if we evaluate them on some criteria.



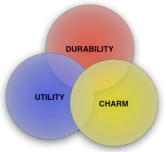
Evaluation

	Shared data	Abstract data type	Implicit invocation	Pipe and filter
Changes data rep	-	+	+	-
Changes algorithm	-	o	o	+
Changes functionality	o	-	+	+
Independent dev	-	+	+	+
Comprehensibility	-	+	o	+
Performance	+	+	-	-
Reuse	-	+	+	+



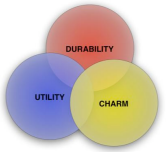
Prototype of Architectural Style

- PROBLEM - type of problem it addresses
- CONTEXT - style imposes requirements on the environment
- SOLUTION - components and connectors
  - component, computational element or procedure
  - connector, how data components interact
- VARIANTS - unique aspects
- EXAMPLES



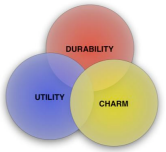
Component Types

Type	Description
computational	Does a computation, I/O simple, local state that persists through computation - math functions
memory	Persistent data structure, shared by components - file
manager	State and associated operators. Operations use or update state, state retained - abstract data types
controller	Governs time sequence of events - scheduler



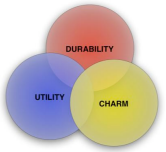
Connector Types

Type	Description
Procedure call	Single thread of control between caller and called. Control transferred to called until done & control returned
Data flow	Processes interact through a stream of data, e.g., pipes. Components are independent
Implicit invocation	Invoked when a certain event occurs rather than by interaction. Invoker and invokee are unaware of each other
Message passing	Independent processes that interact through explicit, discrete transfer of data, e.g., TCP/IP - sync or async
Shared data	Components operate on same data space - blackboard model
Instantiation	Component instantiator, provides space for another component, the instantiated



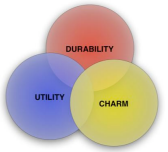
Repository Style

- Manage a richly structured body of information
  - Db schema describing info
  - Relatively independent computational elements
- Examples:
  - Library
  - Compiler - order of invocation matters, different elements enrich internal representation
  - AI blackboard - computational elements triggered by current state



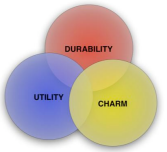
Style: Repository

- PROBLEM: managing a long-lived, richly structured body of information manipulated in many ways
- CONTEXT: requires considerable support, runtime system with a database.
- SOLUTION:
  - System model: major characteristic is a centralized, structured body of information. Independent computational elements act on it
  - Components: one memory component and many computational components
  - Connectors: direct access or procedure call



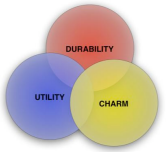
Repository (cont'd)

- SOLUTION (cont'd):
  - Control: input to database functions or in blackboard systems control depends on state of computation
- VARIANTS: database oriented characterized by their transactional nature, blackboards have their origin in AI. Used for complex applications such as speech recognition. Different elements solve part of the problem and update information on blackboard



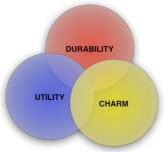
Layered Style

- ISO model for Open System Interconnection, 7 layers: physical, data, network, transport, session. Presentation and application
- Higher layers use functionality of lower layers
- Lower layers cannot use functionality of higher layers



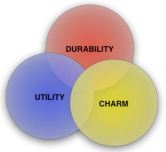
Layered (aka Abstract Machine)

- PROBLEM: services that can be arranged hierarchically and depicted as concentric circles. Often split into 3 layers, one for basic services, one for utilities and one for application specific utilities.
- CONTEXT: each class of service is assigned a layer.
- SOLUTION:
  - System model- a hierarchy of layers and the visibility of inner layers is restricted.



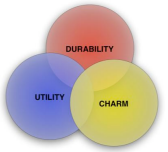
Layered (cont'd)

- SOLUTION (Cont'd):
  - Components- usually a collection of procedures
  - Connectors-interact through procedure calls and the visibility is limited
  - Control structure - single thread of control
- VARIANTS: layer as a virtual machine, offering instructions to next layer. Layering may be used to separate functionality, a UI layer and an application logic layer. Visibility of layers is constrained.



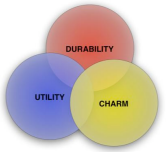
Domain-Specific Software Architectures

- Reference architecture describing general computational framework and is generally a combination of architectural styles without semantic content of components.
- Component library adds application specific semantics representing reusable chunks of domain expertise
- Application configuration method selects and configures components within the architecture to meet specific application needs
- DSSA allows easy instantiation to create actual implementations, it is an application framework. Great for reuse



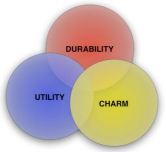
Design Patterns✓

- Recurring structure of communicating components solving a general design pattern within a particular context
- Design patterns can also be termed micro architectures
- Differs from architectural style in scope, not structure of a system but a few (units) interacting components



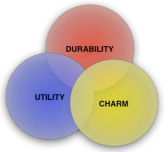
Model-View-Controller

- Classic example of a design pattern. An interactive system with computational elements and elements to display data and handle user input
  - MODEL - system data as well as operations on that data. Independent of how data is represented or input done.
  - VIEW - Displays data of model component. There can be multiple view components and each view has a CONTROLLER
  - CONTROLLER - handles input actions that may cause controller to send request to model to update data or to the view to scroll
- Variant: Document-View pattern where distinction between view and controller has been relaxed



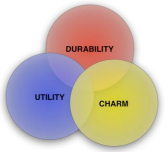
More on Patterns

- Patterns must balance sets of opposing forces, different looks and feels should not affect application code
- Patterns document existing, well-proven design experience and are not invented but evolve, part of best practices
- Patterns identify and specify abstractions above the level of a single component
- Patterns are a means of documentation, describe and prescribe
- Patterns described with schema similar to styles:
  - CONTEXT - the situation
  - PROBLEM - a recurring problem arising in the situation
  - SOLUTION
- More during design



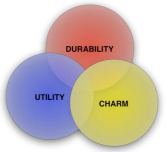
Verification and Validation

- Test the architecture
- Reviews and inspection can be used including the SARB
- Qualitative attributes of maintainability and flexibility are assessed through scenarios
- Skeletal version via proto, story board, incremental development … for testing that later could serve as the environment (test harness) for actual testing.



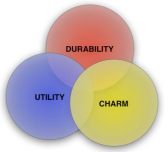
Architecture Reviews and Discoveries✓

- Adapted from Starr & Zimmerman(2002) and personal experience
- These reviews have been around for at least 11 years/ 500+ reviews at AT&T and Lucent (SARB, Systems Architecture review Board)
- Stresses architecture and evaluating it early in the project
- Architecture in this context is viewed as a solution to a problem for a client and should cover a broad range from cost to client needs
- After you establish an architecture:
  - Decide what and when to test
  - Establish success criteria
  - Make decisions based on your findings



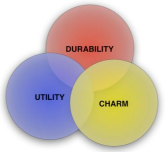
How and what do you address?

- Code inspections examine source code, Design reviews examine models, modules and interfaces
- However a single document or artifact rarely captures architecture
- Architecture is reviewed as an interactive oral presentation which varies from a chalk talk to a few overheads
- Review should be done before contracts are signed or announcements made



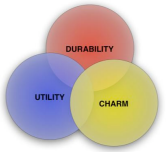
The Architecture Problem Statement

- 1-4 pages in length
- It is criteria to test the architecture and includes:
  - What project must do (functional aspects)
  - What it must be (constraints)
  - Economic constraints (time, schedule, money)
  - How system relates to past (backward compatibility), present and future (evolvability)
  - Unique aspects of the problem (e.g. risks)
- NOT requirements document



Preparing for the Review

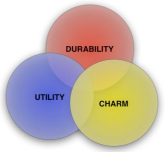
- Problem statement should be approved by architect, client, development manager and other stakeholders (customer representatives, system engineers/analysts). Also should iterate with review coordinator.
- Review team selected (review team << project team):
  - Review leader - technical person trained in SARB and facilitation and responsible for writing review report
  - Angel - manager not associated with project, interface between review team and project management
  - Reviewers - several internal tech experts or outside consultants with relevant expertise, including: design, technology, management, domain knowledge, wild card reviewer



Review process

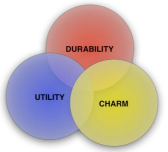
- Pre-review meeting a week before review
- Review meeting (2-3 days):
  - Begins with problem statement - what success means
  - Overview of proposed solution
  - Presentation on "ilities" reliability, maintainability, …
  - Reviewers ask questions and record issues and strengths on snow cards "out in the open" "reconfigurable"
  - Review team has caucus for several hours
    - Snow cards (50-200) arranged by functional categories (requirements, management, interfaces, …) and severity
    - Process provides overall assessment of chances for success and key messages

Snow Cards



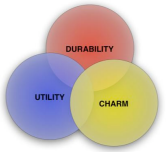
Review Process (cont'd)

- Readout:
  - Encourage project team to invite clients and stakeholders
  - Provides overall assessment, strengths and issues
  - Elicits feedback for Project team
- Followup:
  - Within 2 days snow cards are numbered, recorded and sent to project team
  - If there are immediate issues angel and leader compose letter to client
  - Within 2 weeks detailed writeup of key issues and strengths
  - Within 2 months review leader provides report to SARB, assessment, key issues and review critique
- SARB creates annual report of key issues and trends



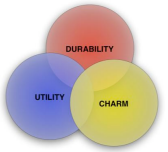
SARB Heuristics

- Requires cultural change:
  - Cost to project, reviewers
  - Project team safety
  - Ubiquity
- Key steps:
  - Combine executive and grass roots support
  - Establish and maintain self reinforcing support for experts
  - Recognize contributors
  - Maintain safety for team and reviewers
  - Start small



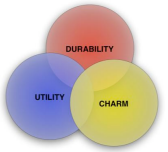
On team safety - key element

- Protect individuals from blame
- Keep no secrets from project team
- Treat all findings as confidential:
  - Specific review findings only reported to team and board
  - Written report is property of project team
  - Only general/anonymous findings are part of annual report
- Review team are partners and colleagues do not have role as policeman



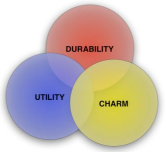
Arch Review Payoff

- Average review pays back 12 times its cost
- Reduced development effort and interval - find defects early
- Higher product quality
- Lower product cost
- Faster less costly product evolution (planned)
- Company wide learning - annual report
- Yes, projects were canceled after reviews and the attitude of the project team was often surprising



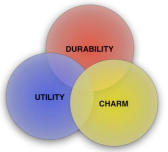
More General Review Heuristics

- When do you review -- early and often, note arch review is very early
- Roles in the review:
  - Author - responsible for updating entity afterwards (e.g., actual architect, developer)
  - Moderator - enforces roles and responsibilities -- manages meeting and is "neutral"
  - Scribe - accurately documents review points (active role, sometimes shared role with monitor), distributes drafts and with moderator and author publishes final review report
  - Reviewers - find issues, try not to solve at that point (can volunteer to help) constructive
- ALL IN THE TONE



Active Reviews✓

- Parnas suggestion to get folks engaged:
  - Ordinary assumption, smaller number of comments during design review indicates higher quality work - NOT!
  - Parnas suggests it indicates a superficial review, besides effective review is not reading a document end to end but using it to
    - Evaluate correctness of document
    - Usefulness of it for future tasks
  - Give reviewers questions on the Design Document so they have to use information - Active Review
  - And a rigorous active design review process should include questionnaires written by all groups needing information, not just authors



References

- Futrell, Shafer & Shafer, Quality software project management, Prentice Hall, 2002, ISBN 0-13-091297-2
- Robertson, S. and Robertson, J., Mastering the requirements process, 1999, Addison-Wesley.
- Endres, A. and Rombach, D. A handbook of software and systems engineering. 2003, Addison-Wesley.
- Wirfs-Brock and Schwartz - http://www.wirfsbrock.com/pages/resources/pdf/the_art_of_writing_use_cases_slides_and_notes.pdf
- Others embedded in text