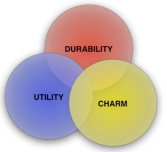
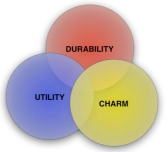


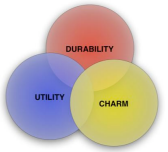
Class 4 CIS 573

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



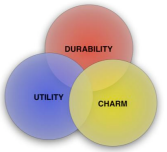
Roadmap

- Continue on 3
- Software Architecture
- Software Reviews
- Design
- Readings this class S Chapter 14; Andersson chapters 1-2
- Readings next class Somerville 27-29, Andersson 3-4



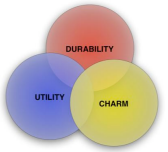
Critical Dates

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



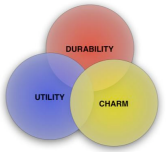
Teams

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



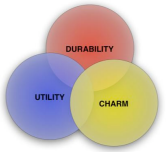
Project Reports

- Presentation each class
  - Green, yellow, red –simplified model + gaps
  - Current pressing issues
  - What was done since last class
  - What will be done before next class
  - Gaps



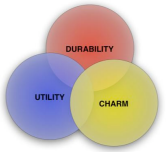
Clarifications

- 9's reliability (365):
  - 3 9's, 99.9%, 525.6 minutes downtime/year
  - 4 9's 99.99%, 52.56 min
  - 5 9's 99.999%, 5.26 min
  - 6 9's, 0.53 min
  - (525,600 min/year)



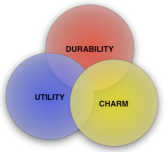
Clarification-2

- PGP vs PHP
  - PGP – PRETTY GOOD PROTECTION
  - PHP – web scripting language



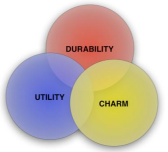
Clarification-3

- Resolution, VESA:
- VGA - 640 X 480, SVGA - 800 X 600, XGA - 1024 X 768, QVGA - 1280 X 960,SXGA+ - 1400 X 1050,UXGA - 1600 X 1200, QXGA - 2048 X 1536Q, SXGA+ - 2800 X 2100, QUXGA - 3200 X 2400
- (4x3 and other aspect ratios) WXGA - 1280 X 800, SXGA - 1280 X 1024, WSXGA+ - 1680 X 1050, WUXGA - 1920 X 1200, QSXGA - 2560 X 2048
- http://www.infocellar.com/hardware/ga.htm



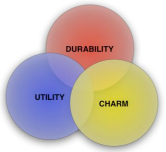
So Far

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



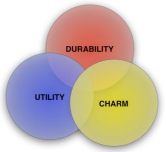
Thought Problems

- 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?



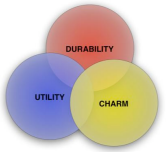
Brooks, Chapter 3

- "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)

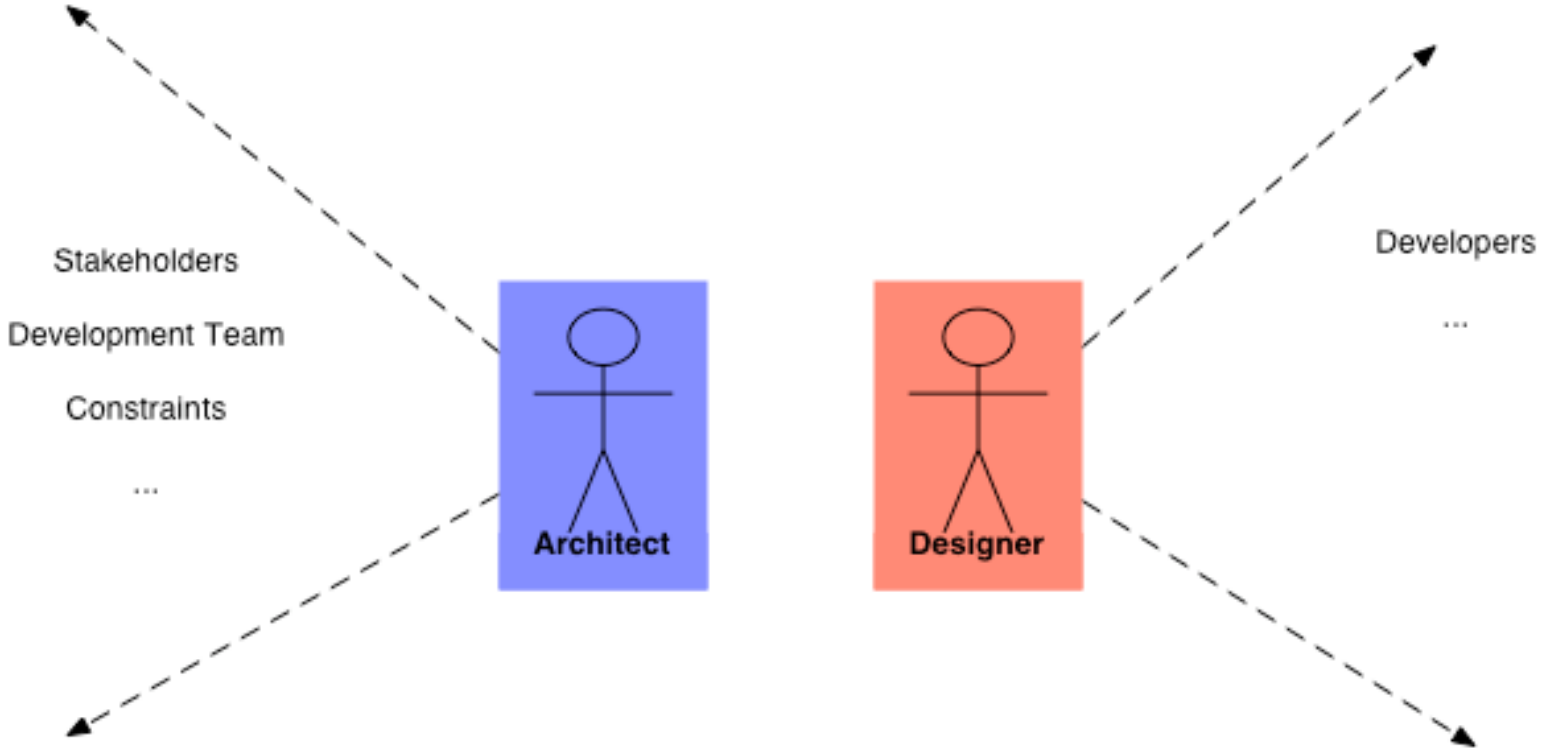
- Harlan Mills: surgical team rather than hog butchers! One does the cutting and the rest do support
- The roles:
  - Surgeon = chief programmer/boss lots of experience (10+ years) and application knowledge
  - Copilot = shares in the design as thinker, discussant and evaluator. Knows all the code. (My 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

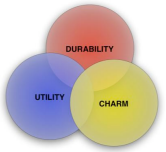


Software Architecture
firmitas, utilitas, venustas

- 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

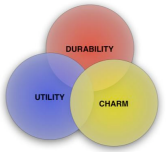
Architecture and Design





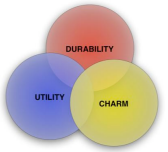
Brooks, Chapter 4

- 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!



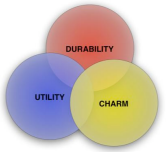
Top-down Design

- 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!



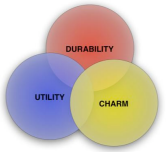
Brooks, Chapter 5

- 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!



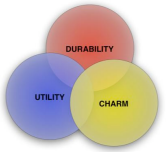
Brooks Chapter 6
On the Life of a Spec

- 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!"



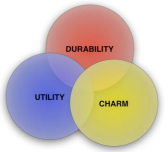
Chapter 9, 10 lbs in a 5 lb Sack

- 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



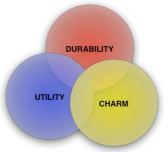
More on Architecture

- 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



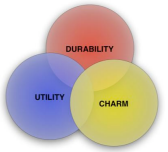
Sommerville Arch ?s

- 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?
- 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?



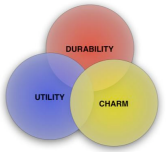
Influences on Architecture

- 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



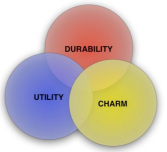
Views of Architecture
(that it must accommodate)

- 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, …

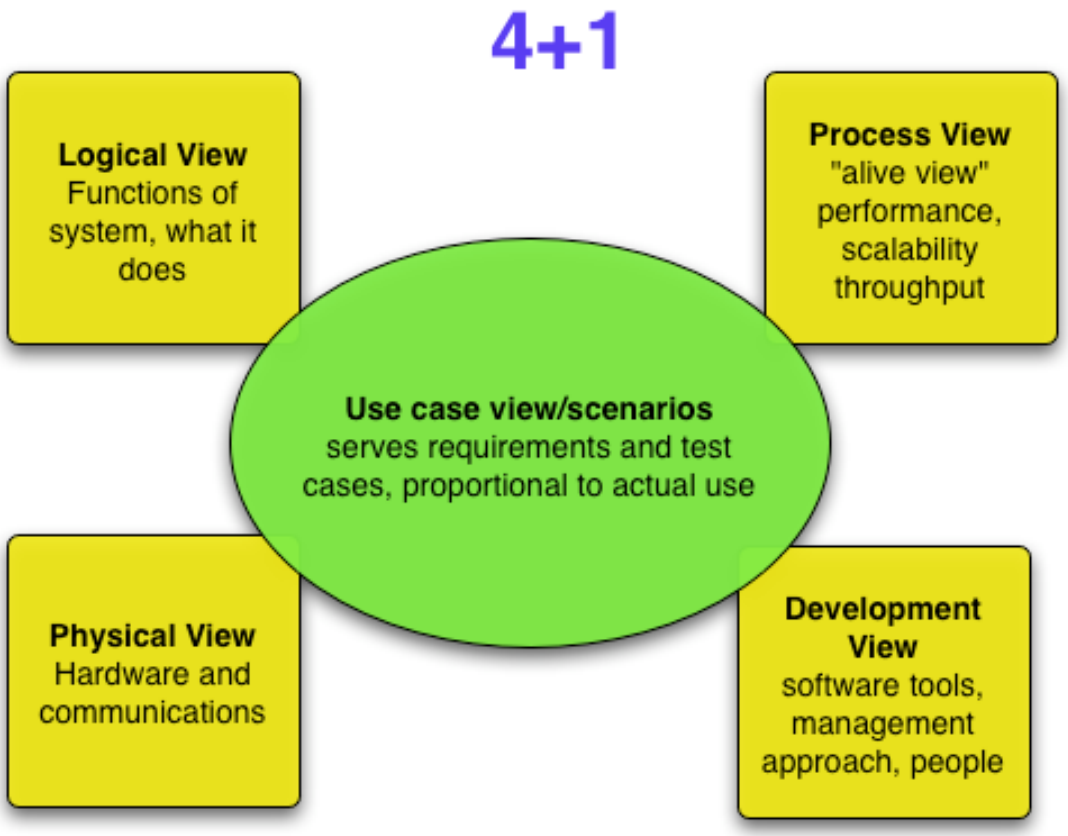


Psychology of the Architect

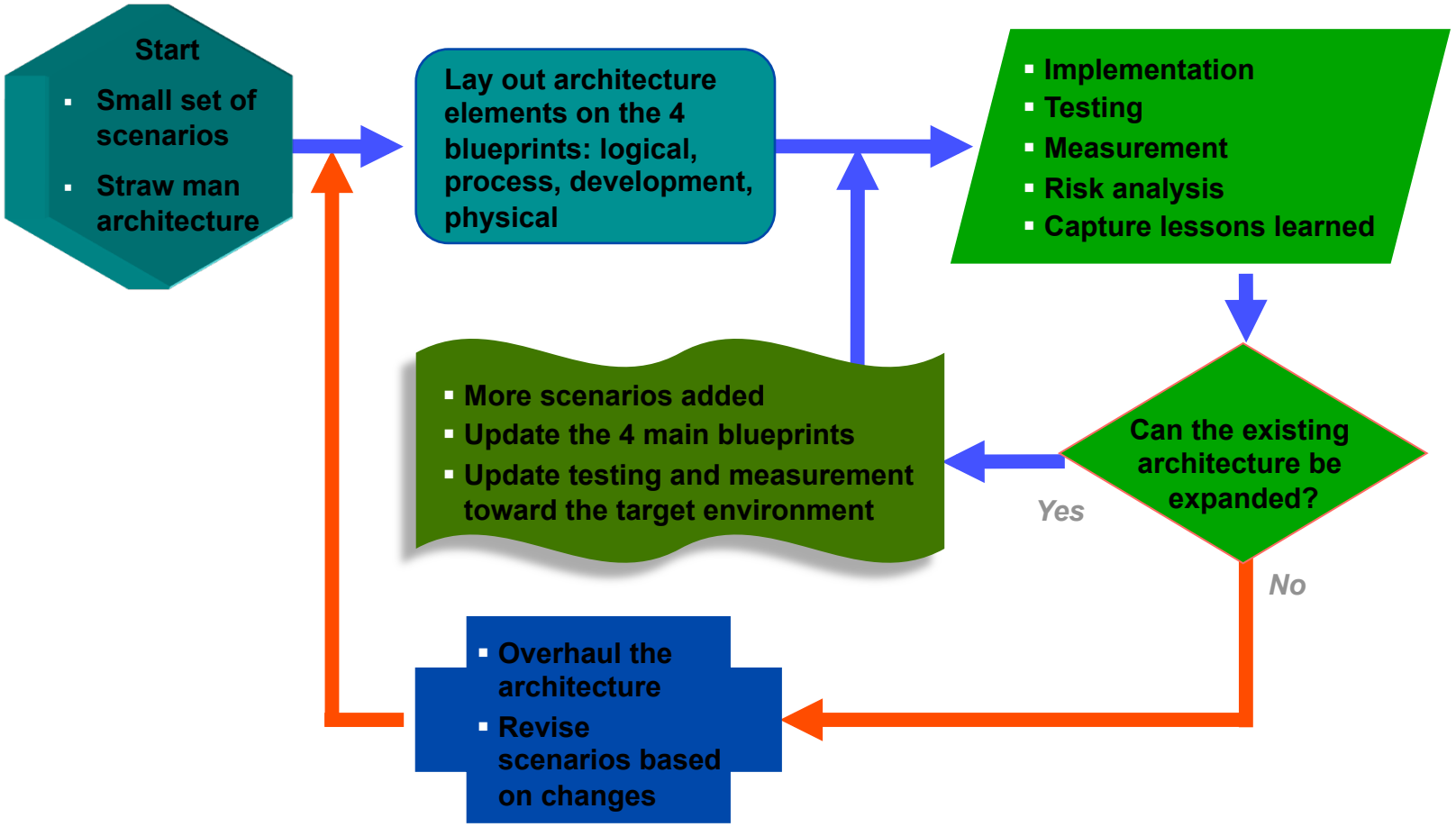
- Architects/developers think of program in chunks
  - Chess/Go/Baseball studies
  - Identifying patterns at a higher level of abstraction
- Design patterns, architectural styles and patterns
- Design pattern is a recurring solution to a standard problem
  - Classic - Model-View-Controller
  - Design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

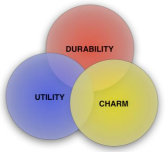


Architecture Approach✓



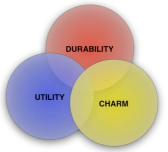
The "4+1" Architecture Model





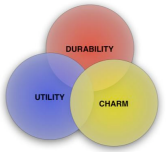
The "4+1" Architecture Model

- 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

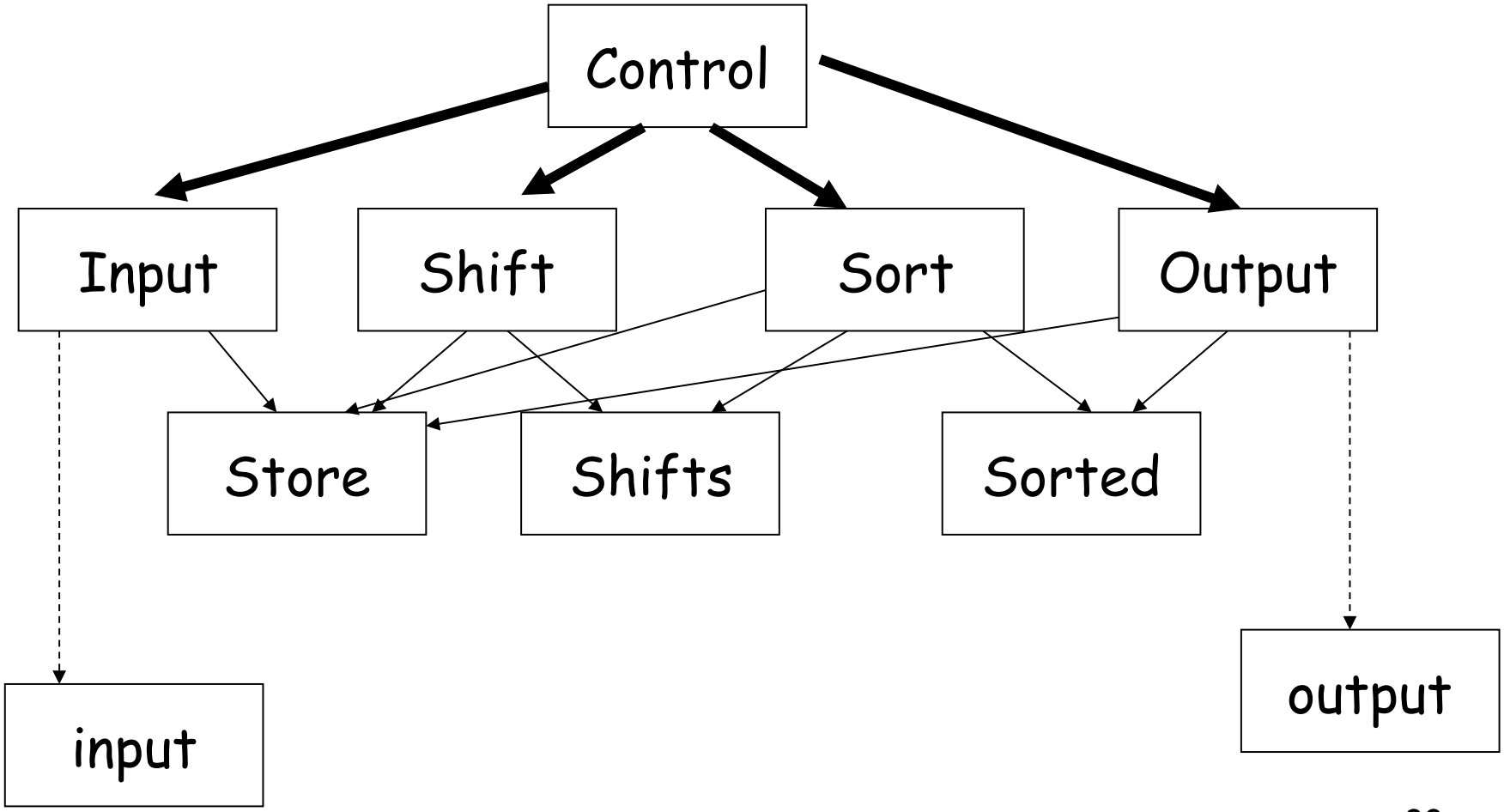
- Key Word In Context = KWIC
- Generate n shifts where n is the number of words in a line to find the key words in a sentence
- E.g., Software engineering for fun and profit, generates:
  - Software engineering for fun and profit
  - Engineering for fun and profit software
  - For fun and profit software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

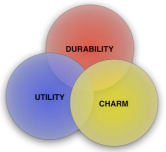


Main Program with subroutines

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

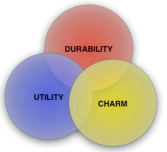
Main Program with Subroutines



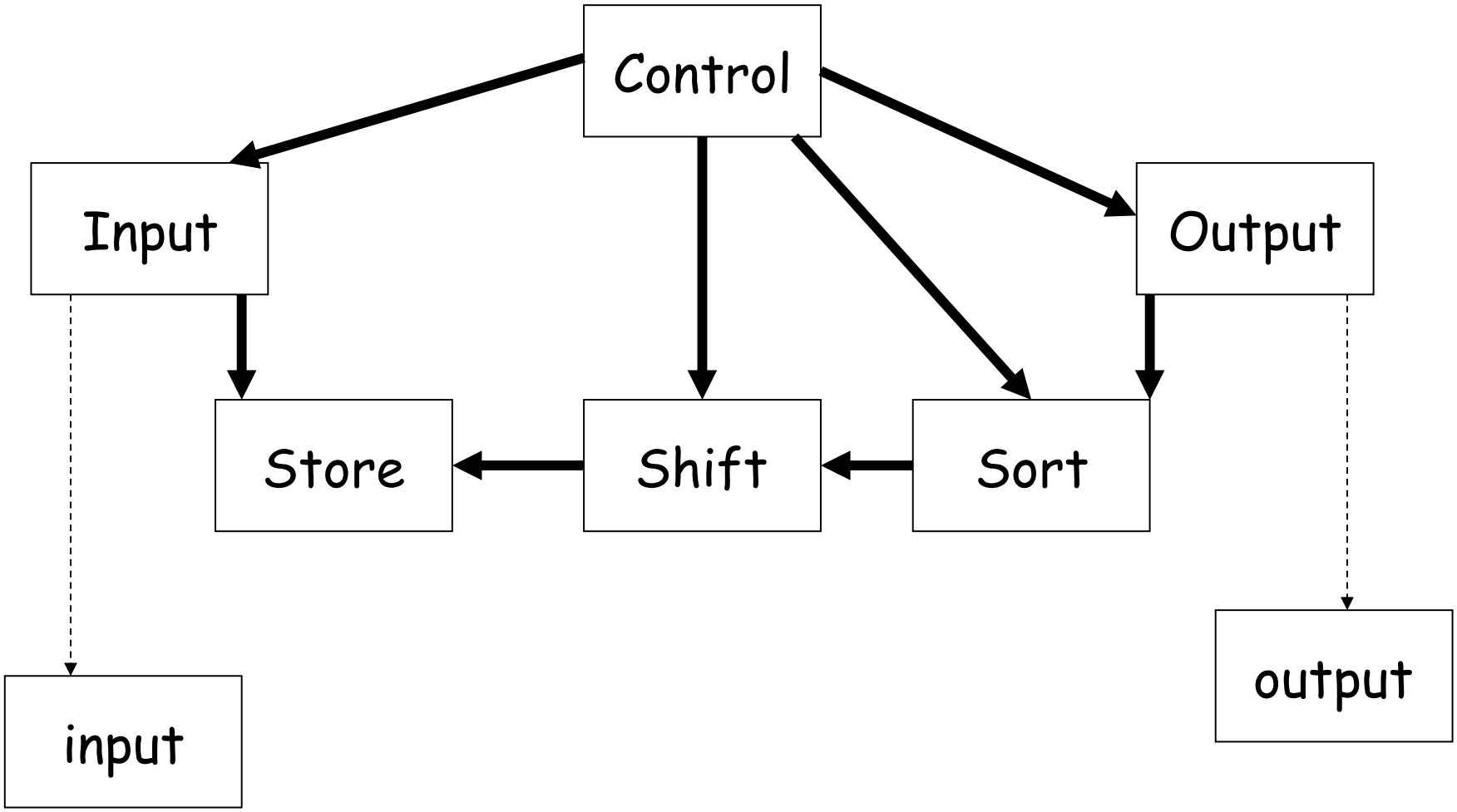


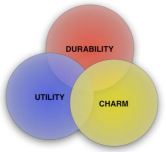
Abstract Data types

- 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

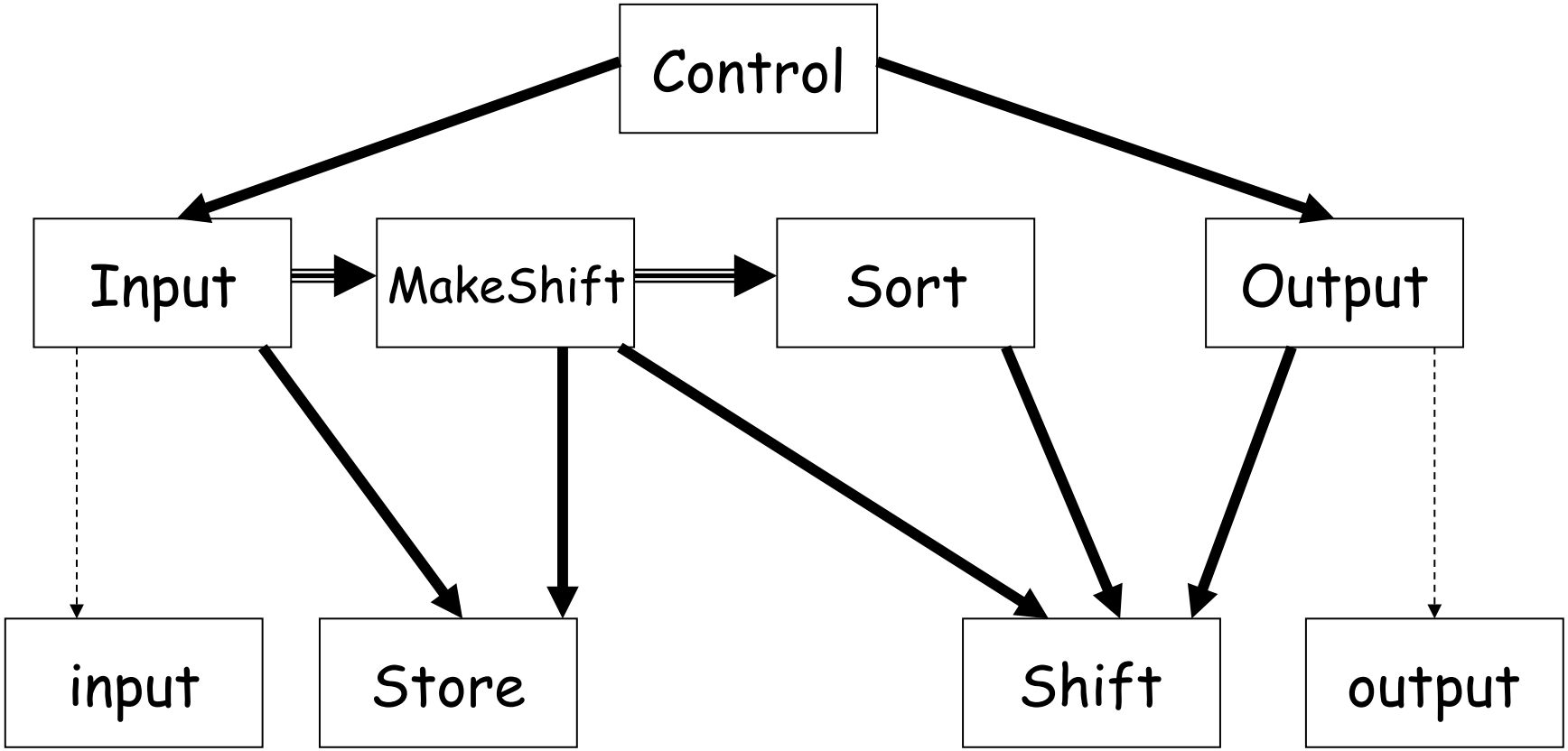


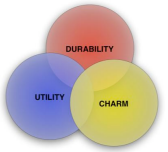


Implicit Invocation

- 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)

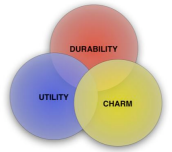
Implicit-invocation



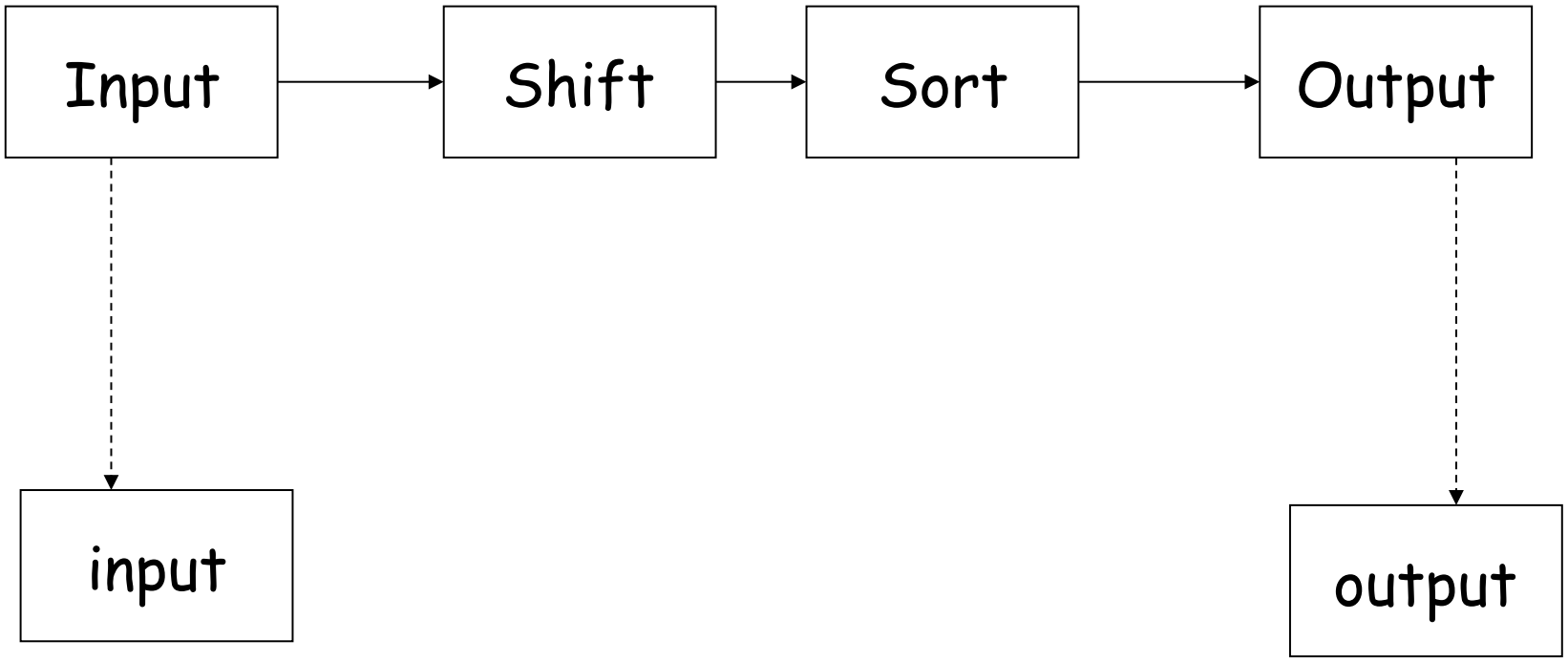


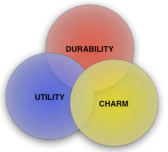
Pipes and Filters

- 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



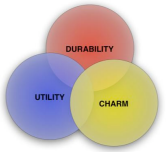
Pipes-and-Filters





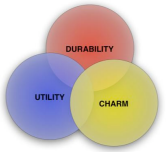
Evaluation of These Architectures

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



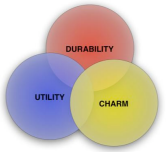
Evaluation

	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	-	+	+	+



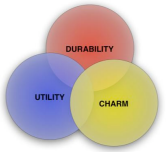
Prototype of Architectural Style

- 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



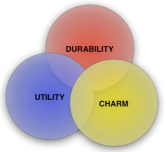
Component Types

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



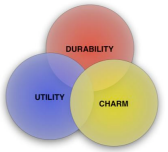
Connector Types

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



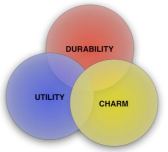
Repository Style

- 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



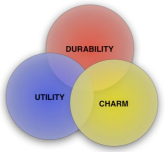
Style: Repository

- 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



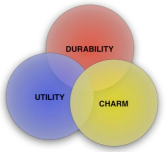
Repository (cont'd)

- 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



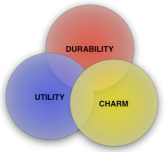
Layered Style

- 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



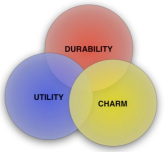
Layered (aka Abstract Machine)

- 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.



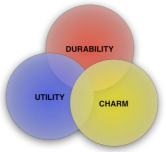
Layered (cont'd)

- 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.



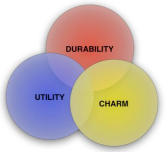
Reality

In practice there is usually a mixture of Architectural styles. Many can be characterized as a mixture of repository and layered. But still a lot of main program with subroutine hanging around!



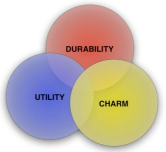
Domain-Specific Software Architectures

- 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



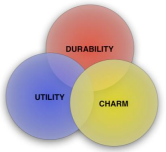
Design Patterns✓

- 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



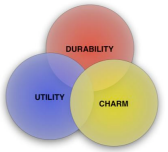
Model-View-Controller

- 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



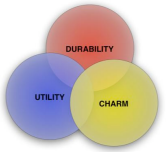
More on Patterns

- 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



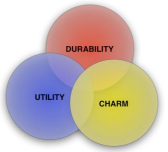
Verification and Validation

- 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.



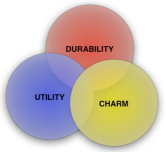
Architecture Reviews and Discoveries✓

- 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



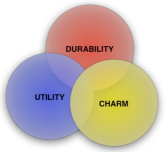
How and what do you address?

- 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



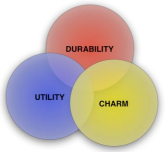
The Architecture Problem Statement

- 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

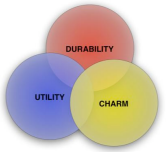
- Problem statement should be approved by architect, client, development manager and other stakeholders (customer representatives, system engineers/analysts). Also should iterate with review coordinator.
- Review team selected (review team << project team):
  - Review leader - technical person trained 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

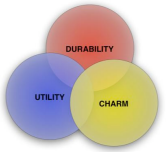
- Pre-review meeting a week before review
- Review meeting (2-3 days):
  - Begins with problem statement - what success means
  - Overview of proposed solution
  - Presentation on "ilities" reliability, maintainability, …
  - Reviewers ask questions and record issues and strengths on snow cards "out 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



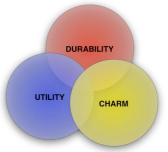
Review Process (cont'd)

- 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



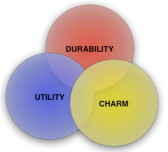
SARB Heuristics

- 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



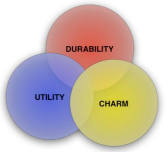
On team safety - key element

- 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



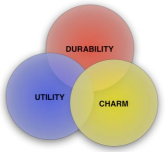
Arch Review Payoff

- 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



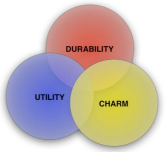
More General Review Heuristics

- 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✓

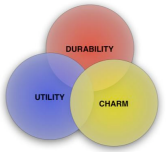
- Parnas suggestion to get folks engaged:
  - Ordinary assumption, smaller number of comments during design review indicates higher quality work - NOT!
  - Parnas suggests it indicates a superficial review, besides effective review is not reading a document end to end but using it to
    - Evaluate correctness 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



SOFTWARE DESIGN: A wicked problem

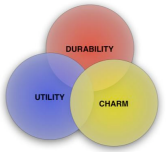
- No definite formulation of a wicked problem - design overlaps other stages
- No stopping rule - dangerous cycle
- Not true or false - no 8 ball, but satisficing
- Every wicked problem is a symptom of another problem - resolving one may result others
- May also term this ill-formed problems - Newell and Simon
- Suggests that we pay equal attention to the human system - the Scandinavian school

Penn CIS



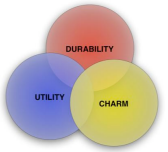
On Design

- Design Problem- decompose a system into parts, each part having a lower complexity than the system as a whole and the interaction between the parts is not complicated. These parts and their interaction solve the user's problem .
- Architecture is the characterization of the design process.
- Five elements affect the quality of the design: abstraction, modularity, information hiding, complexity and system structure



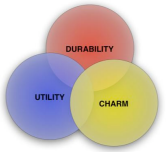
Design - 2

- A module is an identifiable unit in design
- Design features we are most interested in are those that facilitate maintenance and reuse:
  - Simplicity
  - Clear separation of concepts into different modules
  - Restricted visibility, locality of information, "secrets of the module"



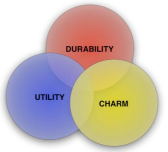
Abstraction

- Concentrate on essential issues and ignore (abstract from) details irrelevant at this stage
- A problem is decomposed to sub-problems with major tasks recognized by their description and the verb (read, sort, process)
- Essence of main program with subroutine architecture style
- Procedural abstraction is the name of the procedure designating the actions
- Data abstraction- finding a hierarchy in the program's data and data typing set of objects and operations on them



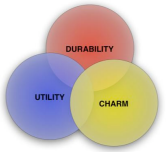
Modularity

- Modules and their interaction
  - Not considered to be an independent (sub-)system, depends on other modules
- Compare designs by considering both a typology for the individual modules and connections between them. Two potential strategies:
  - OO decomposition - set of communicating objects
  - Function oriented pipelining (pipes and filters) receives input data, transforms, outputs
- 2 structural design criteria, cohesion and coupling
- Strive for high cohesion, low coupling



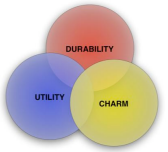
Yourdon and Constantine, 7 (+1) levels of Cohesion

- Levels are of increasing strength:
  - Coincidental- modules grouped in haphazard way, no relation
  - Logical - logically related tasks that do not call each other or pass data between each other - all output routines
  - Temporal - various independent components activated at same time - initialization
  - Procedural- group of components executed in a set order
  - Communicational - components operate on the same temporal data
  - Sequential- output of one serves as input to another
  - Functional all components contribute to a single function in the module
  - + data - modules that encapsulate an abstract data type



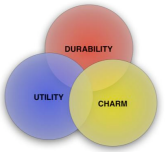
Coupling

- Tightest to loosest (worst to best):
  - Content- one module directly affects working of another
  - Common- 2 modules have shared data
  - External modules communicate through external media, a file
  - Control - one module directs execution of another by passing necessary control information via flags
  - Stamp - complete data structures are passed from one to another
  - Data - only single data passed between modules



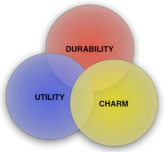
Coupling and Cohesion

- Advantages of low coupling, high cohesion:
  - Communication between developers is easier
  - Correctness proofs are easy to derive and sustain
  - Changes will not affect other modules, lower maintenance costs
  - Reusability is increased
  - Understanding increase
  - Empirical data shows less errors.



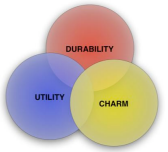
Information Hiding

- Most important aspect
- If a module hides some secret, it does not permeate the module's boundary
- Decreases, coupling, increases cohesion
- But should only hide one secret



Complexity

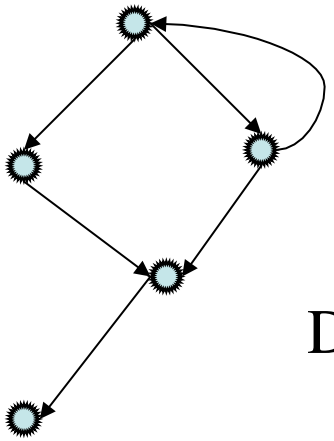
- Attributes of the software that affect effort needed to construct or change a piece of software
- 2 classes of complexity metrics
  - Size based - KLOC
  - Structure based - complicated control or data structures



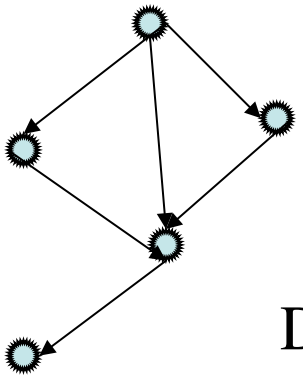
System Structure

- Types of intermodule relations such as A contains B, A follows B, A delivers data to B, A uses B
- The amount of knowledge each uses about the other should be kept to a minimum
  - Information flow should be limited to procedure calls - no Common data structures
- Graph depicting procedure calls is a call graph - we can measure attributes related to the shape of the call graph

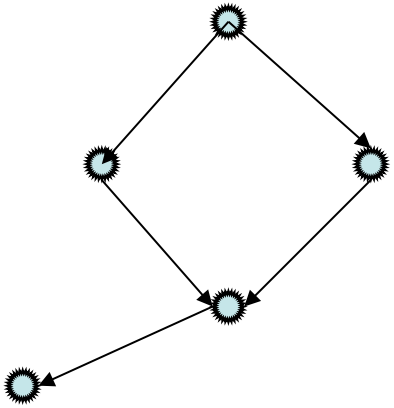
Module Hierarchies



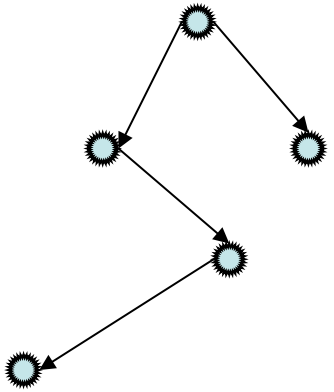
Directed graph



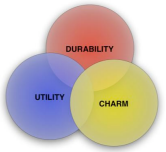
Directed acyclic graph



Layered graph

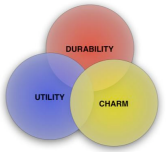


tree



Graph analysis

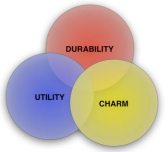
- Some measures:
  - Size - number of nodes and edges
  - Depth - longest path from root to leaf
  - Width - maximum nodes at some level
- A good design should have a tree like call graph
  - One measure of complexity is to assess tree impurity
  - Remove edges until you get a tree
  - Tree impurity = # of extra edges / maximum # of extra edges
  - If 0 graph is a tree, if 1 it is a complete graph
  - But trees are not always desirable, does not permit reuse
- Fan in /fan out measures indicates spots deserving attention, e.g., if a module has high fan in it may indicate little cohesion, excessive increase in information flow from one level to next may indicate missing level of abstraction



Design Heuristics for Modularity

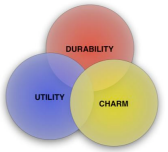
(Pressman) - start here

- Evaluate the first iteration of a program structure to reduce coupling and improve cohesion
- Attempt to minimize structures with high fan out; strive for high fan in as depth increases
- Keep the scope of effect of a module within the scope of control of that module
- Evaluate module interfaces to reduce complexity and redundancy and improve consistency
- Define modules whose function is predictable, but avoid modules that are overly restrictive - balance!
- Strive for controlled entry modules by avoiding pathological conditions (branches or references into the middle of a module)



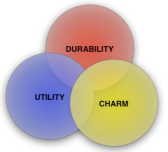
Design Methods

- Functional decomposition - next slide
- Data flow design - functional decomposition with respect to flow of data. Component is a black box transforming some input stream to an output stream
- Design based on data structures - given a correct model of data structures, design of the program is straightforward
- Object-oriented design - later



Functional Decomposition

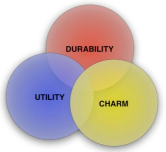
- Intended function decomposed into sub functions and continues downward
- Start from user end it is top-down, primitives, bottom-up
- Parnas method:
  - Identify sub systems, start with a minimal subset and define minimal extensions (incremental development)
  - Apply information hiding principles
  - Define extensions step by step
  - Apply uses relation and try to develop a hierarchy
  - Layered approach, use only components at the same or lower level



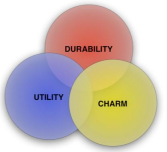
Design Documentation

- IEEE 1016
- Seven user roles for the design documentation:
  - Project manager
  - Configuration manager
  - Designer
  - Programmer
  - Unit tester
  - Integration tester
  - Maintenance programmer

View on Design

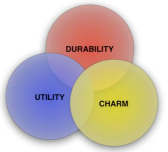


Design View	Description	Attributes	User Roles
Decomposition	Decomp of system into modules	Identification, type, purpose, function, subcomponents	Project manager
Dependencies	Relations between modules and resources	Identification, type, purpose, dependencies, resources	Configuration manager, maintenance programmer, integration tester
Interface	How to use modules	Identification, function, interfaces	Designer, Integration tester
Detail	Internal details of modules	Identification, computation, data	Module tester, programmer



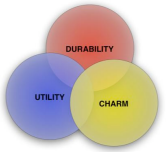
Verification and Validation

- Inspection and walk throughs, reading and critiquing text
- Formal techniques for problem areas
- Prototypes
- Test cases on each of the modules



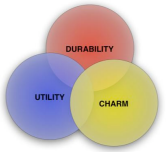
Thought Problem

- You want to nurture architects in your organization, what is a plan to do that and what styles will you encourage?
- You are asked to decide on a design strategy for your company - will you go OO?



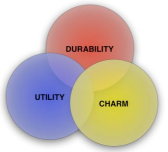
Some Preliminaries

- Object (state (variables), behavior (methods))
  - Instance, instance variables, instantiated, encapsulation
- Message - everything an object can do is represented by its message interface
- Class - software blueprint including common elements of objects that need not be repeated
  - Class variables
- Inheritance
  - States and methods, override
- Data containing instances and function containing classes
- Polymorphism



OO

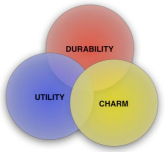
- Traditional techniques focus on functions of the system
- OO focuses on identifying and interrelating the objects that play a role in the system
- Convergence to the UML, Unified Modeling Language, Booch, Jacobson and Rumbaugh
- Heuristic thoughts- keep objects simple and each method should send messages to objects of a very limited set of classes (more when we explore OO metrics)
  - Cohesion and coupling



OO Analysis and Design

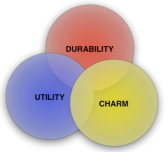
- OO Analysis = Requirements analysis + Domain class selection
  - Product = Complete requirements document + domain class model + basic sequence diagrams
  - Domain classes obtained via use cases -> sequence diagrams and brainstorming/editing process
  - Use domain classes to organize requirements
- OO Design = all other activities except coding
  - Product = complete detailed design ready for coding

Eric Braude, Software Design, John Wiley, 2004.



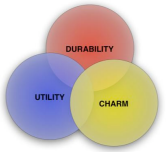
"Schools" of OO

- European school, influenced by the Scandinavian school of Programming, OO analysis and design is modeling real world objects both animate and inanimate
- American school, OO focuses on data abstraction and component reuse - identifying reusable components and building an inheritance hierarchy.
  - "What matters is not how closely we model today's reality but how extensible and reusable our software is"



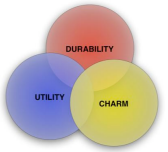
OO Viewpoints

- Modeling (European) viewpoint - conceptual model of some part of a real or imaginary world.
  - Each object has identity, is unique
  - Objects have substance, properties that hold and can be discovered
  - Objects are implementations of abstract data types
    - Mutable state, variables of abstract data type
    - Operators to modify or inspect the state
      - Only way to access object
      - Interface to object
  - Object = identity + variables + operators or
  - Object = identity + state + behavior



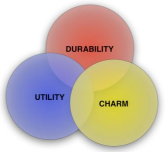
OO Viewpoints - 2

- Philosophical view - objects as existential abstractions, the unifying notion underlying all computation
  - Beginning and end to objects
  - Eternal objects, e.g., integers
    - Not instantiated, cannot be changed
- Software Engineering view - data abstractions encapsulating data and operations
  - Object based languages encapsulates abstract data types in modules whereas
  - Object oriented also includes inheritance



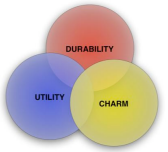
OO Viewpoints - 3

- Implementation view
  - Continuous structure in memory, a record of data and code elements
- Formal view
  - Object viewed as a state machine with a finite set of states and a finite set of state functions. State functions map old states and inputs to new states and inputs
- While modeling conceptual viewpoint is stressed
- Tensions between a problem oriented (analysis) vs. solution oriented viewpoint (design)



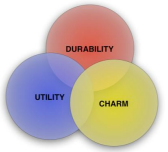
Objects

- Characterized by a set of attributes or properties
  - Attributes originate from Entity-Relationship Modeling
    - In ERM attributes represent intrinsic properties that do not depend on other entities
    - Shared properties among objects are denoted as relationships
- OO modeling uses attributes to denote any field in the underlying data structure
  - State includes intrinsic and shared properties
    - State includes set of structural attributes and operations = behavioral attributes.
  - Identity is an attribute



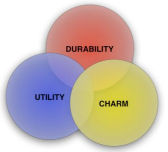
More on Objects

- Programming level;
  - Objects having same set of of attributes belong to the same class
  - Individual objects of the class are called instances, when they are created they are instantiated
  - Objects not only encapsulate state but also behavior - the way it is acted upon and acts upon other objects
  - Behavior of an object is described as the services provided by that object
  - Services are invoked by sending messages from the requestor to the object acted upon
  - Client server model of objects, client object requests services from server object, services also are referred to as responsibilities



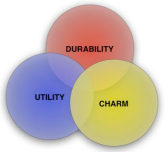
Relations Between Objects

Relationship	Example
Specialization/ generalization, isa	Table isa furniture
Whole-part, has	Table has tabletop
Member-of, has	Library has member



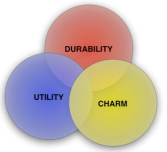
More on Relations

- Generalization-specialization can be expressed as a hierarchy
  - Single inheritance, tree
  - Multiple inheritance, directed acyclic graph
  - Define common attributes at a higher level and let descendants inherit the attributes (abstraction)
  - Object hierarchy can be viewed as a type hierarchy, chair and table are subtypes of furniture



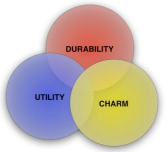
More on Relations - 2

- Part of relationship aggregates components into a whole
  - It is a transitive relationship
- Member-of relationship represents the relation of a set and its neighbors
  - It is not transitive



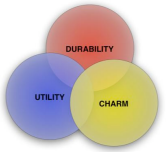
OO Analysis

- Not considered with instances - concerned with object types, classes
- Major goal- identify set of objects (classes) with their attributes (states) and their services (behavior)



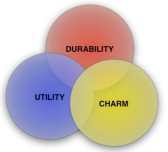
OO Analysis and Design Schemes

- Common notations of the schemes:
  - Class diagram - static depiction of objects as nodes and their relations as edges
  - State diagram - models dynamic behavior of single objects using a variant of a finite state machine representation. Nodes in state diagram represent state of object, edges possible transitions between states
  - Interaction diagrams - model sequence of messages in an interaction among objects
    - Sequence diagrams emphasize time orderings
    - Collaboration diagrams emphasize objects and their relationships relevant to a particular interaction



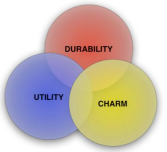
OO Analysis and Design Assumptions

- Assumes a stable problem statement
  - Not strong on elicitation (but not weak either)
  - A collection of use cases are one view of the software architecture of a system



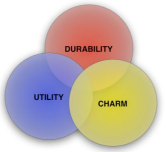
CRC Cards✓

- Class Responsibility Collaborator cards
- Documents collaborative decisions - usually done in a group, but useful individually
- Very helpful in early stages of software development
- Nice for small to medium size projects
- One of the techniques to use, very useful - a winner!
- Wilkinson, N. Using CRC Cards: An informal approach to Object-Oriented development, Cambridge University Press, 1995, ISBN: 0133746798



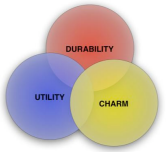
CRC Card

Class Name: Superclasses: Subclasses:	
Responsibilities	Collaborators



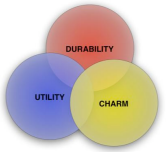
Analysis and Design Methods

- Approach:
  - Identify the objects
  - Determine attributes and services
  - Determine relationships between objects
- Variant of a class diagram provides static map of objects and their interrelationships
- State chart is a dynamic model in addition to scenario analysis through use cases



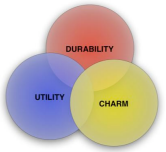
Identify the Objects

- Look for important concepts from the application domain
  - Domain specific entities are prime candidates for objects
    - Real world objects - books
    - Roles played - customer
    - Organizational units - department
    - Locations
    - Devices
- Look at existing classifications and assembly (whole, parts relationship)
  - Sometimes listing most of the nouns in the requirements specification or the problem statement
  - Eliminate from the noun list implementation constructs
  - Vague terms replaced by concrete terms or eliminated
  - Eliminate synonymous terms
  - Demote some terms to attributes
- Some information is from statement, some from tacit knowledge
- Diagram starts with relationships and evolves to more detailed descriptions (cardinality constraints and inheritance)



Identify Attributes and Services

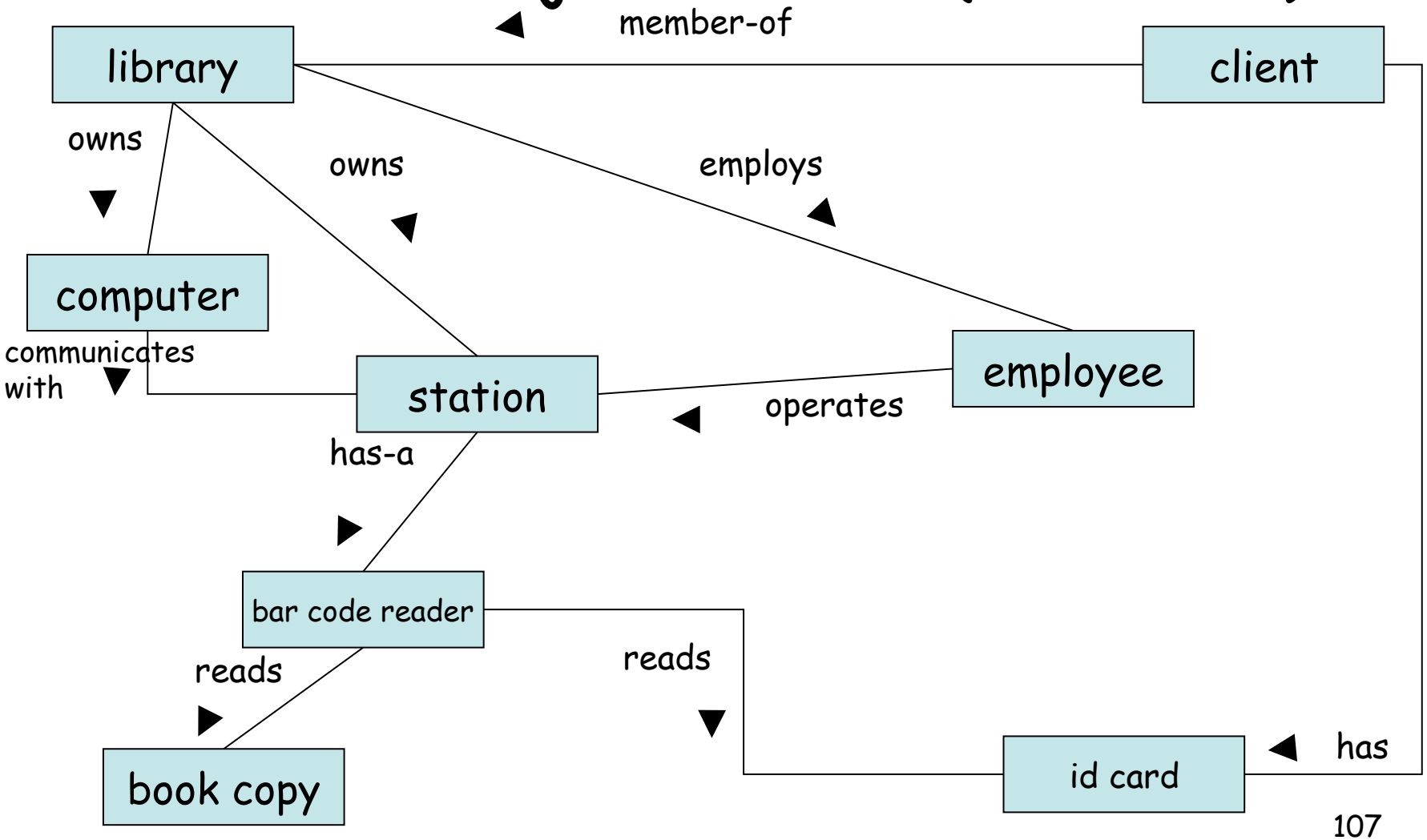
- Describe an instance of the object
  - The state of the object
  - Consider characteristics that distinguish instances but are common properties of the objects
    - Look for atomic rather than composite attributes - compute from atomic to composite if necessary
  - Services are related to life cycle and are usually verbs in the description, e.g., book is acquired, borrowed, returned, retired
    - Concern state of the object
    - Usage scenarios aid in discovery

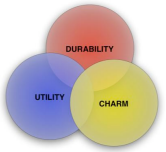


Identify Relationships

- Services are one way objects can be related
  - OO flavor comes from whole-part, gen/spec relationships
  - Consider similarities of objects as basis for specification of a more general object
  - Publication is an abstract object, one that has no instances
    - Attributes and services defined at publication level constitute a common interface for its descendants
  - Generalization/Specialization relationship can lift services to higher levels in the hierarchy and they are represented by virtual functions - services for which a default implementation is provided and can be redefined by specializations

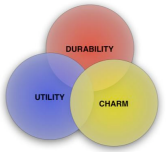
Initial Object Model (van Vliet)





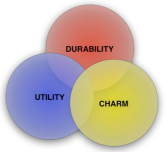
Comments on OO Analysis and Design

- Instances of an object should have common attributes if not repartition or reconsider
- Over evolution/time, object hierarchy should remain stable but attributes and services may change
- OO can be considered a middle-out design method!
  - Set of objects constitute the middle design level
- OO could prosper if there were a future where collections of domain specific classes become available - domain libraries - DIFFICULT



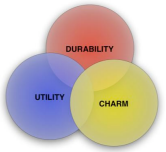
Design Patterns✓

- Portland Design Repository - http://c2.com/ppr/
- Motivation came from a "real" architect, Christopher Alexander-
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution over a million times without ever doing it the same way twice."
  - Referring to buildings and towns, e.g. couple's realm, children's realm, sleeping to the east, …



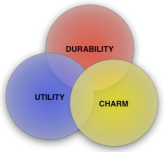
Couple's Realm

- Consists of following centers:
  - Bed Alcove
  - Couple's Realm Main Area
    - Fireplace
    - Connects to Bed alcove, dressing room, Porch/balcony, still pond, bathing room and its child centers
  - Dressing Room
  - Porch/Balcony
- Connects to Bathing Room and Child Centers
- http://www.simplybuilding.net/center/view/39



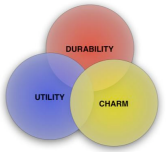
Categories of Patterns

- Creational - abstract instantiation, strive for independence - problem is creating a complex object.
  - Used to control creation of objects, including families of objects. Often replace constructors.
- Structural Patterns - how classes and objects are composed to form larger structures--uses inheritance - problem is representing a complex structure
  - Lists, collections and trees with convenient interfaces
- Behavioral- algorithms and assignment of responsibility, describe objects and communication - problem is representing behavior
  - -application behavior options at runtime
- Pros, more flexibility
- Cons, increased complexity and potential performance issues



OO Metrics✓

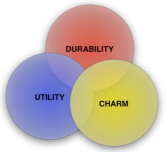
Source	Metric	OO Construct
Traditional	Cyclomatic Complexity	Method
Traditional	Lines of Code	Method
Traditional	Comment Percentage	Method
OO	Weighted Methods per Class (WMC)	Class/Method
OO	Response for a Class (RFC)	Class/Message
OO	Lack of Cohesion of Methods (LCOM)	Class/Cohesion
OO	Coupling between Objects (CBO)	Coupling
OO	Depth of Inheritance Tree(DIT)	Inheritance
OO	Number of Children(NOC)	Inheritance



OO Metrics

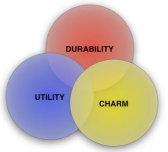
- WMC (Weighted Methods per Class) is a measure of size of class, assumes larger classes are less desirable - usually a count of the number of methods
- RFC (Response For a Class) is number of methods in class + number of methods called by each of these class methods where each method is counted once
- LCOM (Lack of Cohesion of Methods)is number of disjoint sets of methods of a class, any 2 methods in the same set share at least one local state variable, preferred value is 0, cohesion metric
- CBO (Coupling Between Objects) is coupling metric, 2 classes are coupled if a method of one class uses a method or state variable of the other class, high values = tight bindings, undesirable
  - Gradations
  - Law of Demeter - methods of a class should only depend on top level structure of own class
- DIT (Depth of Inheritance Tree) is the distance of the class from the root of the tree. Language dependent
  - Strive for inheritance trees of medium height, not narrow and deep, not shallow and broad
- NOC (Number of Children)is number of immediate descendants of a class, large number suggests improper abstraction

Penn CIS



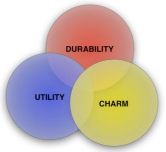
OO: Hype or Hammer?

- OOA and OOD are similar with OOD adding implementation specific classes, OOA should be problem oriented, OOD should be solution oriented
- Transition to OO takes time
- Issues: handling of real time requirements, less mature, measuring progress is hard, no good cost models, scalability and interoperability with non OO systems pose problems
- Traditional functional models are easier to understand by the customer
  - Users do not think in objects they think in tasks - use cases as a way to bridge this
- BUT…



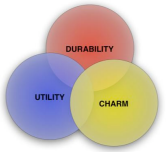
Thought Problems

- You are beginning an OO project, what analysis style would you use, e.g., the UML, CRC cards, (your favorite method)? Do you think it would depend on the size of the project?
- What so you think are relevant criteria for deciding if you should use OO methodology?



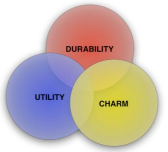
References

- 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



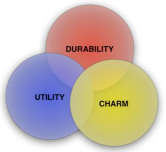
Appendix

- Summary of Software Design Patterns from:
  - Gamma, et.al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 978-0201633610



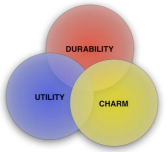
Abstract Factory

- Pattern Name - abstract factory, aka kit
- Problem - There is a model of a generic product and a specification of all the necessary members (parts or products if it is a product family) but this generic product has product families or there is a need to separate the specification from the actual implementation. The client initiating the process stays independent of the concrete implementation.
- Solution - define abstract factory with abstract products that are needs of the client. Concrete versions of factory are accessed by client through abstract factory. Abstract products define what concrete classes must be developed for the concrete factory to meet the needs of the client.



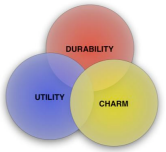
Abstract Factory-2

- Consequences - ability to develop code once that relies on abstract factory with knowledge that same set of products, as defined by abstract products, will be created for all concrete instances but client code is isolated from these details. Note that client calls only one thing, the abstract factor, so the concrete factory can be switched. It also enforces the fact that only this combination of product objects works together --- important in look and feel. The bummer is that if you want to add a new product to the set you have to change ALL versions of concrete factory since abstract factory defines what are the set of products for any concrete factory derived from it. Therefore if you want to add a product all concrete factories have to be changed (by adding a new product). Why? Because when the client calls the abstract factory it expects a certain, consistent set of products, so whatever concrete factory, abstract factory points to bette have those products.



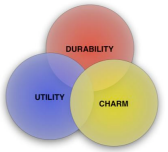
Builder

- Pattern Name - Builder
- Problem - Separate construction from representation.
- Solution - Builder pattern constructs the product step by step under the Director's control. Focuses on building a complex object step by step. Go over maze implementation.
- Consequences - vary a product's internal representation, better control over the construction process and therefore the internal structure of the product.



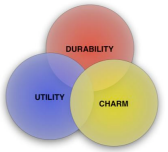
Factory Method

- Pattern Name - Factory Method, aka Virtual Constructor
- Problem - Do not create a family of products (as in Abstract Factory), but a product. Define what an object needs to do (its interface) but separate what it needs to do from its implementation. One variant can specify default implementations for the product.
- Solution - create abstract factory methods that specify what the creator of an object should define (and in some cases define a concrete default implementation). For instance the abstract factory method for door must also state that it needs two places to connect.



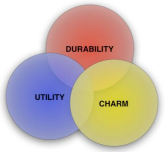
Factory Method -2

- Consequences - sometimes subclassing the creator class just adds another level of indirection w/o adding anything to the design. But using factory method can make explicit connections between parallel class hierarchies. For example if a class of object created consistently needs a class of manipulator that relationship is indicated in the abstract factory method since it states that that class of manipulators is necessary and there likely to be a one to one correspondence between the object class in its hierarchy and the manipulator class in its hierarchy.



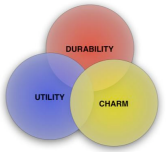
Prototype

- Pattern Name - prototype
- Problem- need to define "classes" at runtime, dynamic creation or classes do not have state. Want to avoid creating class hierarchies.
- Solution - provide the effect of different classes by cloning, making copies of objects with different parameters (state). These parameters could even include bitmaps or fonts or … . Great when "classes" are instantiated at runtime or when you do not want to build class hierarchies or when there is not to many combinations of state. In effect the combination of parameters, when registered as a prototype becomes a new classes that you can freely create more instances of. These prototype instances become "classes"



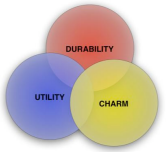
Prototype-2

- Consequences - again hides concrete classes from client. Can add products at runtime by combinations of parameters and these instances become prototypes. For example you can refer to a combination of parts dynamically as a prototype and this becomes a new class that you can instantiate. Must be able to support a deep - true copy that is cloning all instance variables too rather than sharing them.



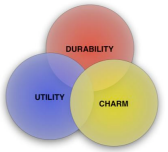
Singleton

- Pattern Name - Singleton
- Problem - situations where a class can only have one instance, for example a thread pool manager or print spooler.
- Solution - devise a class that is responsible for keeping tracking and providing access to one and only one instance. This instance needs to be extensible (be modified) through subclassing.
- Consequences - access is controlled to instance, it avoids using global variables, subclass permits evolution. You can modify to provide a variable number of instances. More general than language provided features (e.g., static member functions in C++).



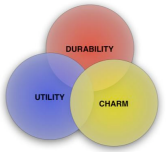
Adapter

- Pattern Name - adapter aka wrapper (but not quite the same)
- Problem - interface to a class is not interface client expects, but you want to use it. You want to create a class that works with classes that have not been designed but may have different interfaces. Translates.
- Solution - two ways, class adapter uses multiple inheritance to adapt one interface to other or create a parent class that defines interface for subclasses, known as object adapter. Basically whether the adapter interface is defined high in the class hierarchy or low in the class hierarchy.



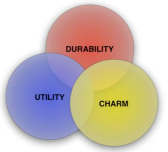
Adapter - 2

- Consequences - for class adapter works only for a class since you commit to an Adaptee class. Lets adapter override adaptees class, introduces only one object. For object adapter a single adapter works with a lot of class that inherit the interface. Overriding is harder.



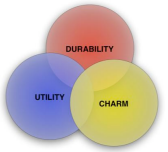
Bridge

- Pattern Name - bridge aka handle, body
- Problem - separate abstraction from implementation by defining what needs to be available (the abstraction) but not the implementation. Therefore we need to bridge the abstraction from the implementation for a particular use.
- Solution - define the abstraction that has a reference to the implementation, a class called implementor that refers to the necessary concrete implementations.



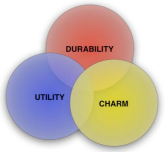
Bridge - 2

- Consequences - implementation is not bound permanently to interface and therefore can be changed, binds one of a few or many implementations, eliminating compile time dependencies. Hides implementation details since the clients refer to abstract interface.



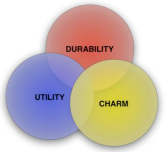
Composite

- Pattern Name - composite
- Problem - need to represent part whole hierarchies and you do not want to deal with all of the objects (parts) in the composition.
- Solution - component handles interactions with composite structure. The component transmits operations to either a composite or and actual part (aka leaf) and if part does it or if composite, composite transmits to its parts.
- Consequences - client code does not need to know if it is a part or a composite, interacts same way, so makes the client simple. This makes it easier to add composites or parts. Sometimes is to general and does not provide the ability to restrict what parts can be used.



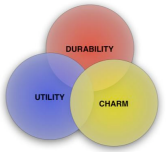
Decorator

- Pattern name: Decorator, a.k.a. wrapper, adorner (MacApp)
- The problem: want to add additional responsibilities to an object not to the entire class. The usual way is to use inheritance, but inheritance is one size fits all every object of the new class will have the responsibilities. Alternative is to enclose component in another object that adds feature.
- The solution - So decorator wraps object in another object and this can be done recursively to add an unlimited number of responsibilities (features). It still is transparent to all the wrapped objects responsibilities.



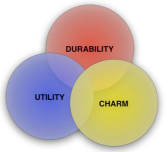
Decorator-2

- The Solution (cont'd)
  - When to use: 1) to add responsibilities to individual objects w/o affecting other objects, 2) withdrawable responsibilities 3)when extension would cause and explosion of subclasses to support every combination -- borders are a good example
  - Changes the objects skin
- The consequences
  - You can "snap-on" responsibilities - even duplicate (e.g.,for double borders)
  - Unlike inheritance, objects do not pay for features they do not use
  - But a "decorated" object and an object it wraps are not identical and you cannot rely on object identity
  - Lots of little objects that look alike.



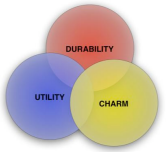
Decorator-3

- Other
  - Provides a way to emulate pipe and filter with each wrapper potentially becoming another filter on a base stream class.
  - Related … especially adapter which does some traditional aspects of wrapper, I.e., changing an objects interface



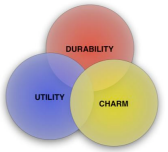
Facade

- Pattern Name - façade (I sometimes call it veneer)
- Problem - need a simpler interface, a unified interface to a subsystem. Example is a single call to compiler versus calling all of its necessary subtasks, scanner, parser, … Note that the classes still exist so that other clients can use them if need be. Also when there is a need for a novice versus expert interface.
- Solution - A single class façade that is interface to all the subsystem classes
- Consequences - makes it easier for client and provides weak coupling so you can change the elements of the subsystem that are being used. At the same time the subsystem components are still there to be used.



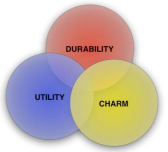
Flyweight

- Pattern Name - flyweight
- Problem - some applications would benefit from creating many objects for maxim flexibility but the resulting overhead would be difficult to manage for the system and the resources.
- Solution - provides a scheme to share objects allowing the benefit of many objects without the cost. The key is differentiating between intrinsic and extrinsic state. The intrinsic state is the information that does not vary by context, e.g. a character code. However its position in the document and font, size, … does vary and are represented in the extrinsic component.



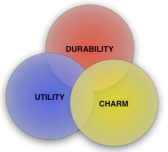
Flyweight - 2

- Solution (cont'd) The extrinsic component provides the context of that particular use of the object. Note this only works if there are core features that can be intrinsic and the bulk of the object definition is extrinsic. Note also that you cannot use object identity since the core object is reused/shared. Client has the burden of maintaining all the extrinsic information
- Consequences - savings are the key and depends on amount of intrinsic state, reduction in total instances by using this scheme, whether extrinsic state is computed or stored, where computed is definitely better (no space is required).



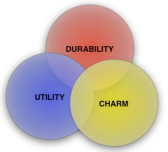
Legacy Wrapper - gtv

- Pattern Name - legacy wrapper, aka wrapper
- Problem - cost of reimplementing legacy systems is prohibitive yet there is a need to make these systems compatible with technology of newer systems.
- Solution - create a wrapper that invokes all the methods of the legacy API. It does not implement the API, it calls it. Alternative is implementing a number of wrappers each encapsulating a particular service of the API. Both support incremental replacement of legacy services.



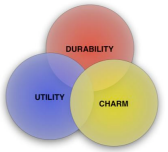
Legacy Wrapper - 2

- Consequences - depends on proper functioning and support of the legacy system(s) since they are doing the until they are replaced. Supports the philosophy of the pattern community, "an aggressive disregard for originality>"



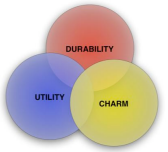
Proxy

- Pattern Name -proxy, aka surrogate, (similar to lazy evaluation)
- Problem - creating an object is expensive, yet you need to manipulate the object.
- Solution - create a stand-in that acts like the object but only actually creates the full blown object when necessary. There are different types: remote provides a local representative, virtual is a placeholder and creates it only on demand when absolutely needed, protection provides access according to access rights and



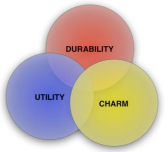
Proxy - 2

- Solution (cont'd) - smart reference is a smart pointer that can load a persistent object, reference count or maintain locks to manage access. Proxy is placed in front of the actual object and has the same interface. It controls access and can be responsible to create or delete it.
- Consequences - can hide the fact that a real object exists in a different address space. It can also do additional housekeeping duties.



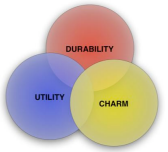
Chain of Responsibility

- Pattern Name - chain of responsibility
- Problem - object that provides services isn't known explicitly by requesting object because object servicing it depends on context and availability of resources, information, … need to present multiple objects that can potentially handle request
- Solution - handler defines an interface for handling requests and pointing to the chain of handlers (successor chain) that the request propagates through until an object handles request.
- Consequences - clearly reduces coupling of a sender to a particular receiver, but no guarantee that request will be handled (it falls off the chain). Adds flexibility in design of receivers



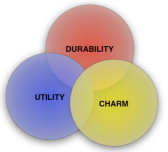
Command

- Pattern name: Command, a.k.a action, transaction
- The problem - lets toolkit objects make requests of unspecified app objects by turning the request into an object -- do not know receiver or operations it will do --- e.g. menu class
- The solution - bare bones is a class with interface for executing operations. For sequence of commands it is a for loop enclosing the execute.



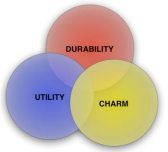
Command-2

- (cont-d) Uses:
  - Specify, queue and execute requests later-can have lifetime longer than original request
  - Can store state to support undo, create persistent log of changes for recovery
  - Support transactions
- The consequences
  - Separates request from actual command (object that knows how to do request)
  - Can manipulate commands, so therefore you can create a macro-command that does a sequence of commands
  - New commands are just added
  - Issue of how intelligent a command should be
  - Beware of side affects in undo/redo



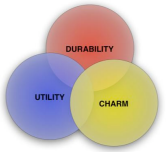
Interpreter

- Pattern Name - interpreter
- Problem - need to define a grammar for simple languages that would be frequently used. This includes representing sentences and interpreting these sentences. Need to represent a BNF. The grammar must be fairly simple and there must not be efficiency concerns.
- Solution - define a syntax tree of terminal and non terminal expressions. Terminal expressions implement specified operator, non terminal calls itself recursively until reaching a terminal expression. Context is also passed for information (such as variable bindings) that are global to the interpreter.



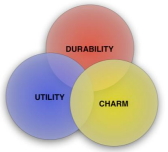
Interpreter - 2

- Consequences - easy to implement, change and extend grammar. Larger grammars are hard to maintain. Can add new ways to interpret the expression for type-checking, pretty-printing …



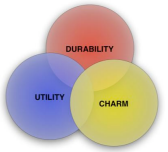
Iterator

- Pattern Name - iterator
- Problem - need to sequentially access elements of an aggregate object (e.g., object representing a list) without exposing underlying representation. Provides a uniform interface for traversing different aggregate structures.
- Solution - define interface for accessing list's elements and keeping track of current element, implementing first(), next(), CurrentItem() and IsDone().
- Consequences - supports variations of traversal algorithm, simplifies the aggregate since iteration is defined outside the aggregate. Because of this more than one traversal can be in progress on the same aggregate object.



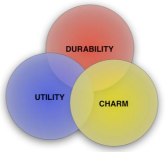
Mediator

- Pattern name: mediator
- The problem: OO encourages distributing behavior among objects but it can result in a lot of highly interconnected objects that become monolithic and hard to change (high coupling), common and not a good thing! Also since responsibilities are distributed, behavior can be difficult to change because it is distributed among so many objects.
- The solution - the mediator object controls and coordinates object interactions, keeping objects from referring to each other and providing developer/designer with a single place to change things if interaction changes. Acts as a communication hub for the objects.



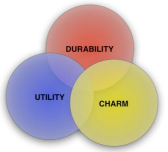
Mediator-2

- The solution (cont'd) uses:
  - Objects communicate in well-defined but complex ways
  - When reuse of an object is hindered because it refers to and communicates with many objects
  - Behavior used by several classes should be tweakable w/o subclassing
- The consequences:
  - Limits subclassing by localizing behavior that can be distributed across objects
  - Promotes loose coupling
  - Changes many to many to one to many
  - Separates object interaction from their behavior
  - Mediator can become a monolith that is hard to maintain -- complexity has to go somewhere



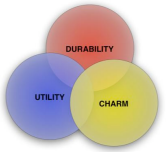
Memento

- Pattern name: Memento a.k.a as token (definition a keepsake)
- The problem: in cases where it is necessary to record internal state of an object (fault tolerance), e.g., undo, rollback state. Objects encapsulate the state that needs to be saved and exposing it would violate encapsulation. Also useful for reflection.
- The solution: stores a snapshot of the original object, the memento's originator that initializes the memento. Only the originator can store and retrieve, thus preserving encapsulation.



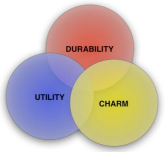
Memento-2

- The solution (cont'd) Uses:
  - Snapshot of all or portion of an object should be saved to restore later
  - Direct interface would expose implementation details and break encapsulation
- The consequences:
  - Preserves encapsulation
  - Simplifies originator … does not need to store the state
  - May be expensive if tons of state must be saved or if the originator needs to checkpoint often due to its frequent use
  - Impacts caretaker which garbage collects



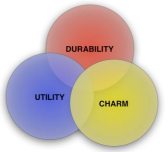
Observer

- Pattern Name - observer, aka dependents, publish-subscribe
- Problem - Problem of maintaining consistency between/among related objects.
- Solution - a subject (object being observed) and one or more observers are implemented and all observers are notified when subject changes state, and then each observer synchronizes with subject. Subject is publisher, observers subscribe to the notifications.



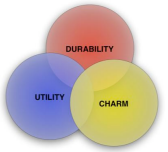
Observer - 2

- Consequences - can vary subjects and observers independently, loosely coupled, support broadcast. As more observers subscribe to a subject, it can be costly changing the subject. Also if it is a simple broadcast (stating simply that a change has occurred but not specifying the change), observers may have significant work finding what has changed.



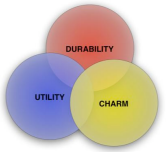
State

- Pattern Name - state
- Problem - need to alter an objects behavior when its state changes. Need to establish different operational modes for each state. For example an object representing a phone call.
- Solution - implement subclasses that provide state specific behavior.
- Consequences - localizes state specific behavior and partitions behavior for different states, permitting new states to be added easily. Makes state transitions explicit and enables state objects to be shared and they act like flyweights.



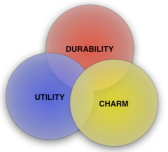
Strategy

- Pattern Name - strategy, aka policy
- Problem - different variants of an algorithm are needed, need to hide data an algorithm uses from clients.
- Solution - define classes for each strategy and an interface that is common to all supported strategies.
- Consequences - provides a way to represent hierarchies of related algorithms, provides a way to vary an algorithm supporting a class dynamically. Clients must be aware of the strategies and could increase number of objects unless you implement as stateless objects using a flyweight approach.



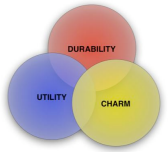
Implementing abstract classes

- Captures common behavior
- 3 kinds
  - Base method - behaviors generally useful to subclasses, implement in one place behavior used by all subclasses, e.g., error message for dividing by 0
  - Abstract methods - default behavior that subclasses are expected to override, placeholders, meant to specify classes responsibilities
  - Template methods - step by step algorithms, may include abstract methods that need to be defined, base methods, other template methods or a combo. Purpose is to abstract steps of an algorithm. First draw border then draw interior.



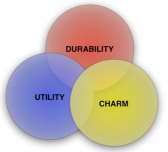
Template method

- Pattern name: template method
- The problem: define skeleton of an algorithm, deferring some definition to subclasses.
- The solution: template method defines an algorithm in terms of abstract operations that subclasses override. It fixes the ordering of the algorithm and the minimum that needs to be there.



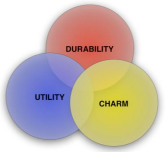
Template Method-2

- Solution (cont'd): Uses:
  - Implement the invariant parts of an algorithm
  - To localize common behavior among subclasses, generalizing
  - A fundamental technique for code reuse, especially in clas libraries
- The consequences:
  - The Hollywood principle - don't call us, we will call you. A parent class calls operations of subclass
  - Differentiate between operations that are hooks (may be overridden) and operations that are abstract and must be overridden



Visitor

- Pattern Name - visitor
- Problem - operations are required on the elements of an object without changing the classes of these elements and these elements are heterogeneous, requiring potentially different operations for each type. Also need to define additional operations on the elements, yet they rarely change.
- Solution - create a visitor for these elements and two hierarchies one for the elements acted on and one for the visitors that define operations on the elements. A new operation is created by adding a new subclass.



Visitor - 2

- Consequences - makes adding new operations easy, simplifies object class by using a visitor to gather related operations and separate unrelated rather than spreading it around original object structure. Visitors can also easily accommodate state in a well defined place. Adding new elements is hard since it requires new implementations in each visitor class. It also breaks encapsulation since an objects internal structure must be made available to the visitor