Class 7 CS540

Gregg Vesonder

Stevens Institute of Technology

Roadmap- Class 7

- Clarifications from last class
- Log Book volunteer
- Brooks Chapters, 4!
- Systems Engineering
- Architecture
- Mid Term Feedback
- Design
- Reading: Brooks Chapters 7, 8, 9, 10 BY Chapter 5
- Reading next class Brooks, Chapters 11& 12, Bernstein paper - on web

Calendar -Key Dates

- November 7th - second test
- November 21st - log books due
- December 12th - final exam

Clarifications

- Testing in critical applications
  - NATO STANAG 4404
  - RTCA/DO-178B
- UML as documentation and modeling
- Best Common Practices Example
- Seeding vs Mutation Testing -- nuances - both measure test set adequacy but one tries to say something more statistically given distribution assumptions

Logbook

- Your Entry
- Kobayashi Maru

Why did the tower of Babel fall?

- Babel had all the prerequisites for success:
  - Clear mission
  - Adequate manpower
  - Adequate materials
  - Lots of time
  - Appropriate technology
- BUT lacked communication and organization
- How should teams communicate? In as many ways as possible! Informal, regular meetings (group and project) and project work book

The Project Workbook

- (easier with the web and wiki's http://www.wiki.org)
- Structure is imposed on documents that all documents use
- Control material (number it) but make it available to entire team
- A change summary should highlight what is new

The Organization

- Tree like structure of organization
- The producer and the technical/director-architect
  - Producer: assembles team, divides work and establishes schedules
  - Tech/director architect is surgeon - the job worst done by management is to use technical genius not strong on management talent
- Small projects, same person
- Larger projects separate. Producer as boss, Director as right hand man - large projects
  - Establishing director's authority is difficult - symbols / dual ladder
- Director boss, producer right hand man - Brooks okay for small teams - "inhomogeneity of technical understanding" issue

Management is work!

"The techniques of communication and organization demand from the manager as much thought and as much experience and competence as the software itself." - Brooks, p83

Chapter 8, Calling the shot

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

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

Chapter 10, The Documentary Hypothesis

- In praise of certain aspects of bureaucracy
- Engineering manager as a flywheel damping fluctuations from market and management
- Documents for a software project:
  - What1? - goals, constraints priorities
  - What2? - product specification, starts as a proposal, ends as a manual and internal documentation
  - When? - Schedule
  - How Much? - Budget
  - Where? - Space allocation (a killer)
  - Who? - organization chart

Conway's Law

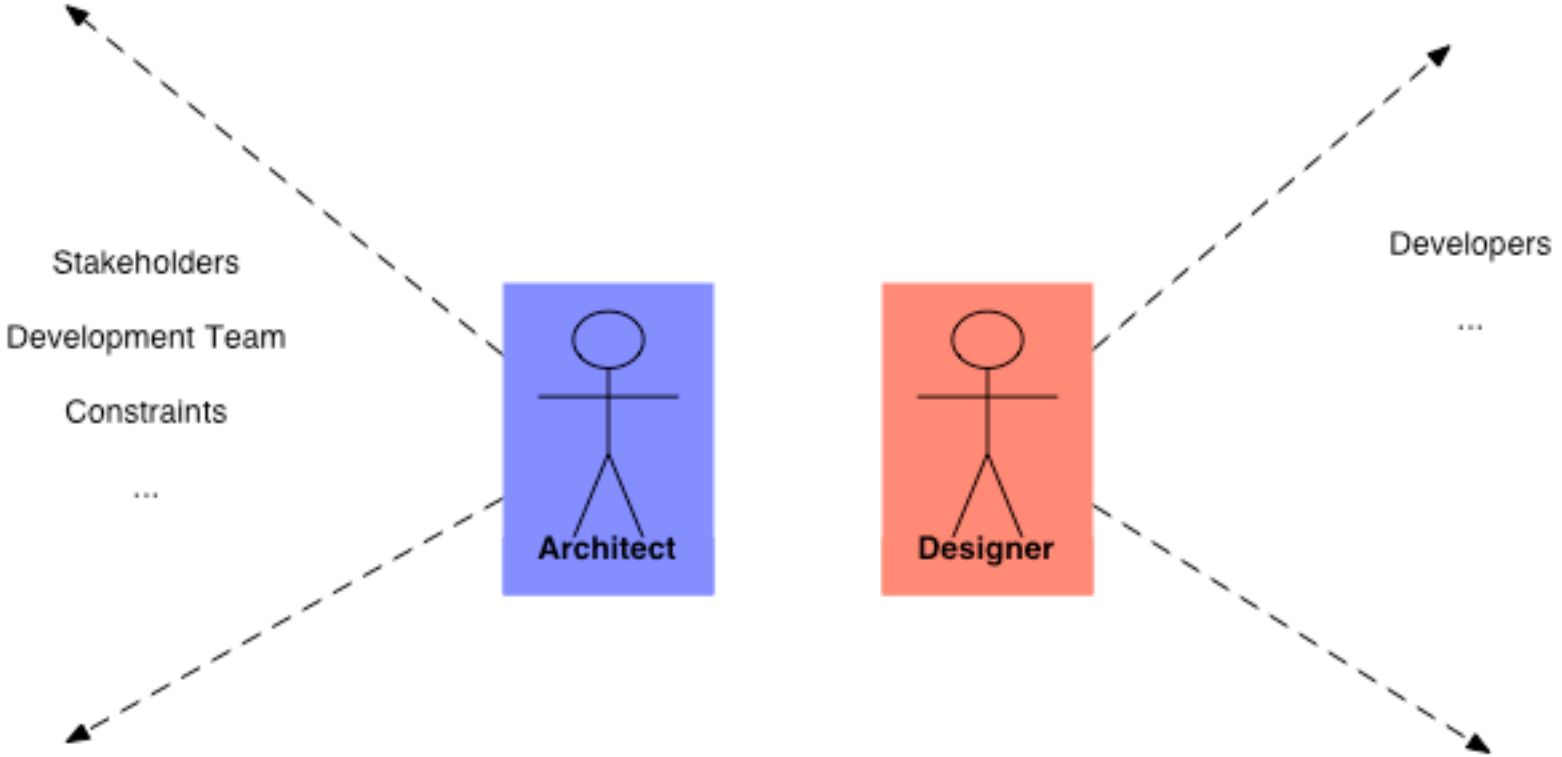
- Organizations must be flexible
- Manager's task - develop a plan and realize it
- Formal Documentation helps:
  - Writing clarifies for all
  - Communicates decisions to others, lightens load for manager whose job is communication

Organizations which design systems are constrained to produce systems which are copies of the communication structure of these organizations.

System Engineering

- Similar to a Knowledge Engineer but generalized to all computer systems
- Domain and technically savvy
- Understands the very large picture at corporate level
- Generates requirements and is user advocate and investigator
- Generates economics of system

Architecture and Design



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

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

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
- 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

KWIC Index

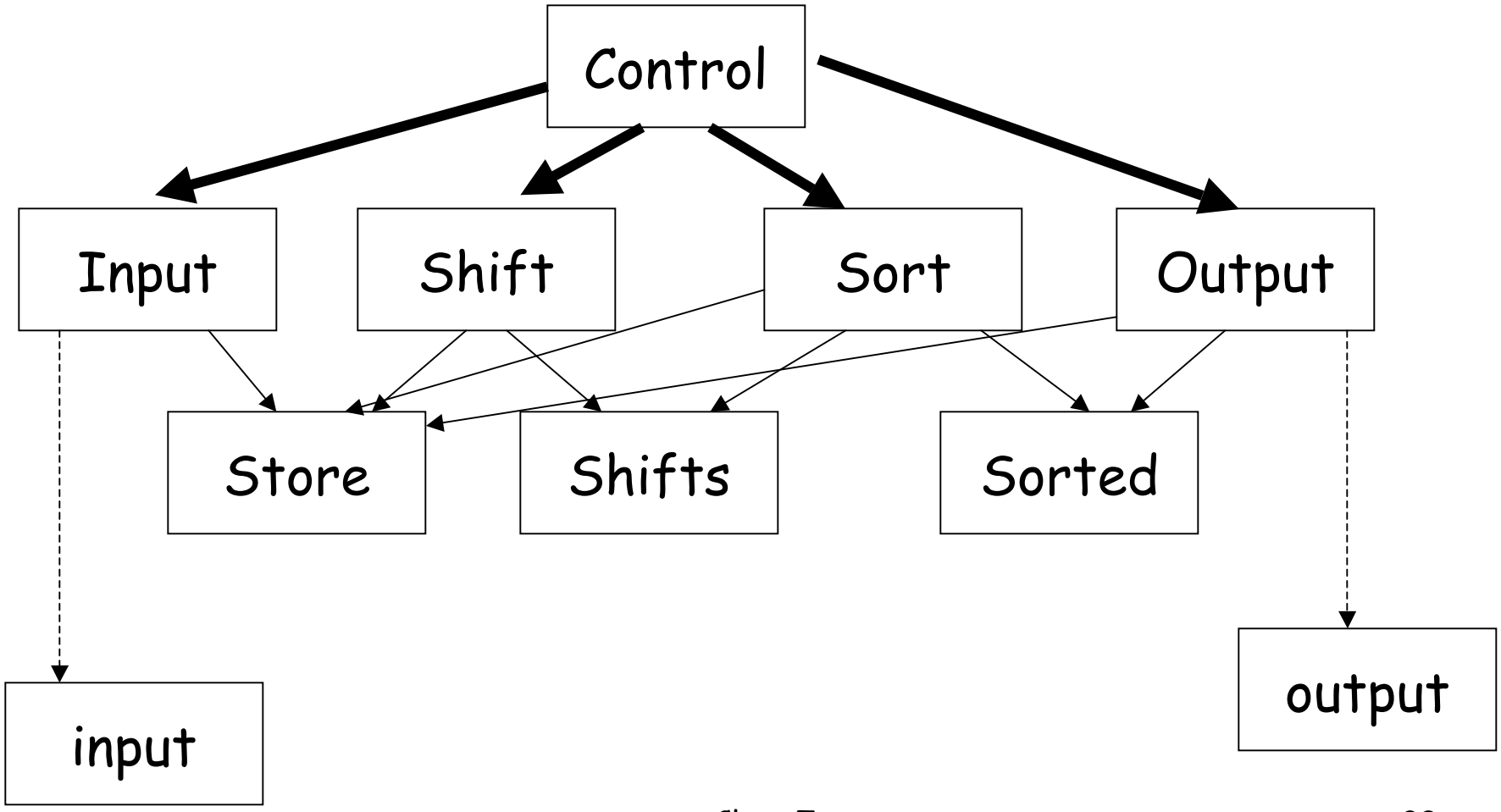
adapted from Parnas

- 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

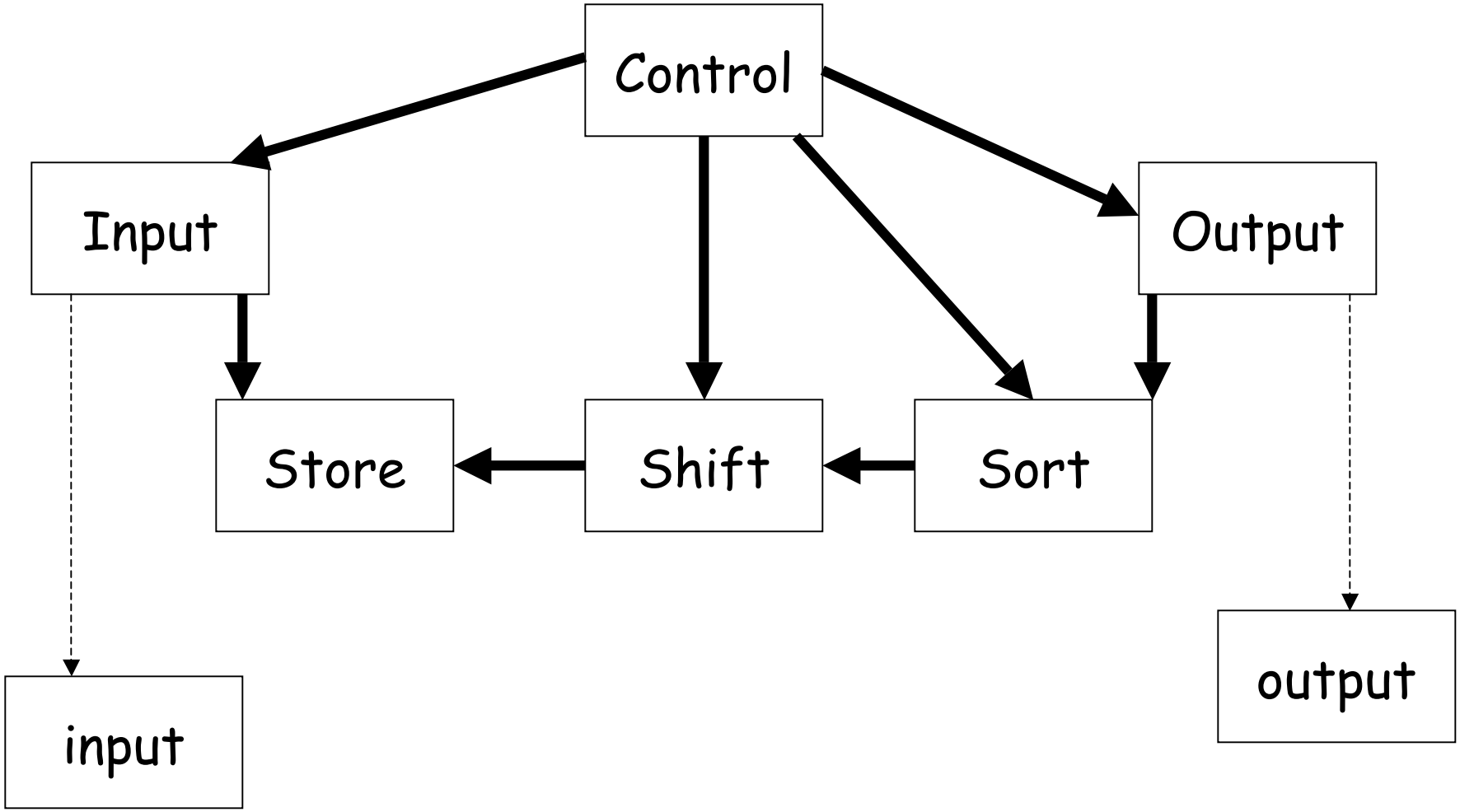
Main Program with Subroutines



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

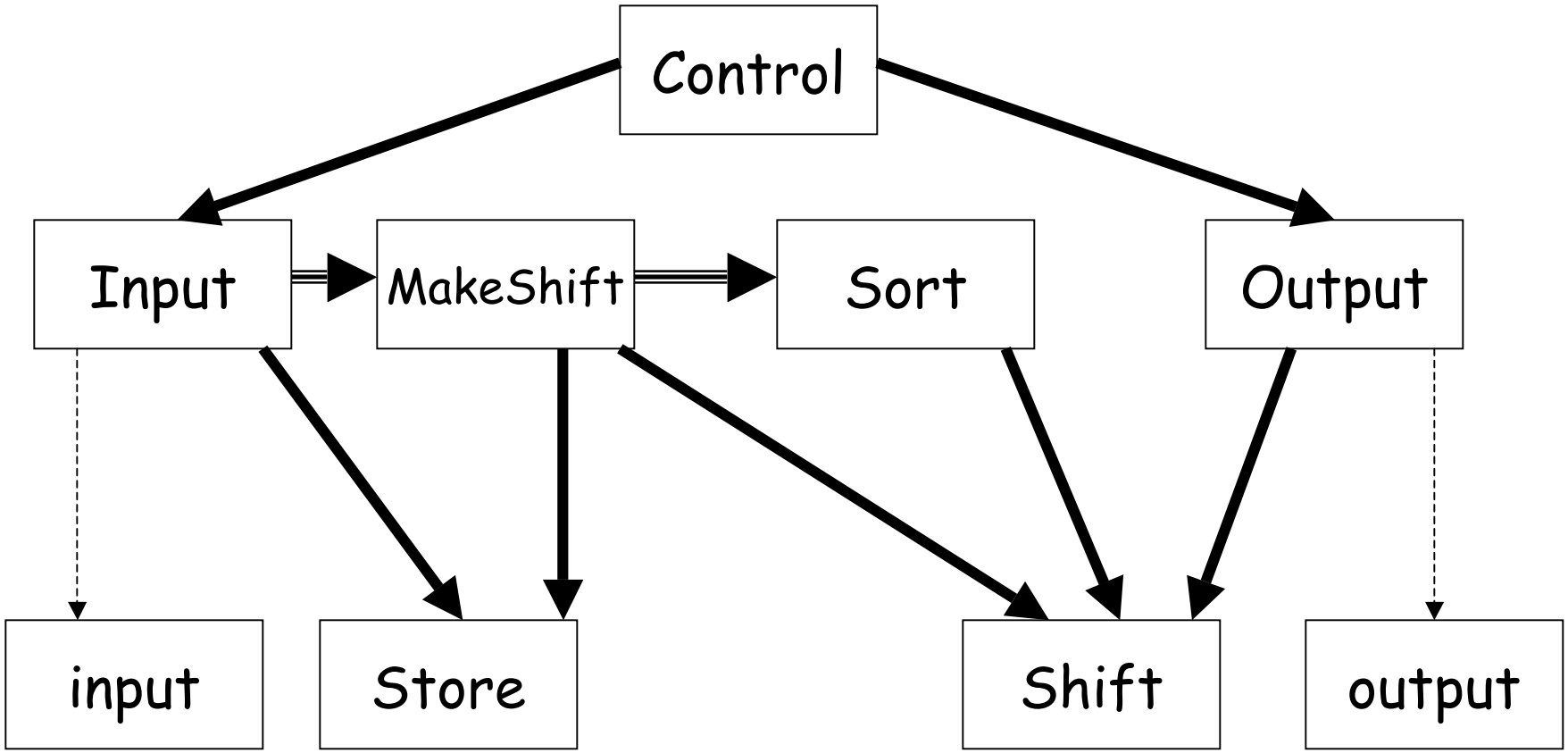
Abstract-data-type



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)

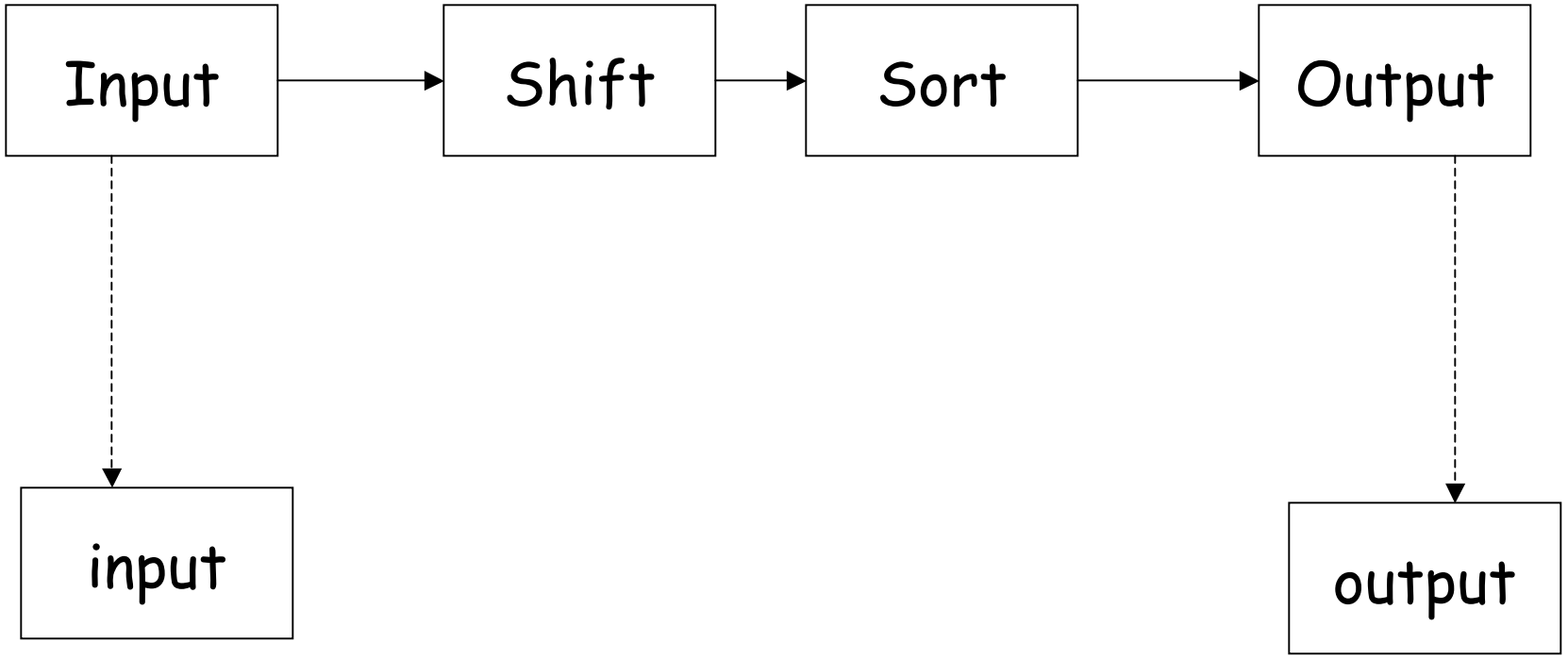
Implicit-invocation



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

- 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

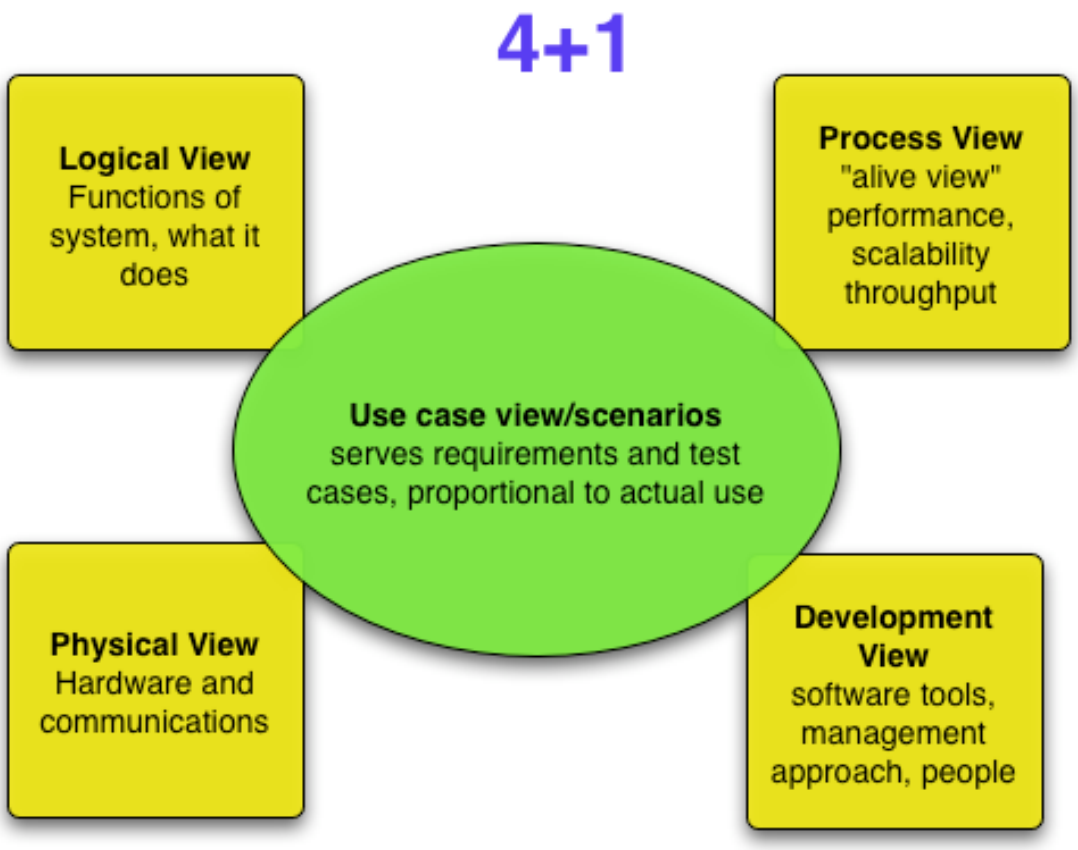
More on Patterns

- 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

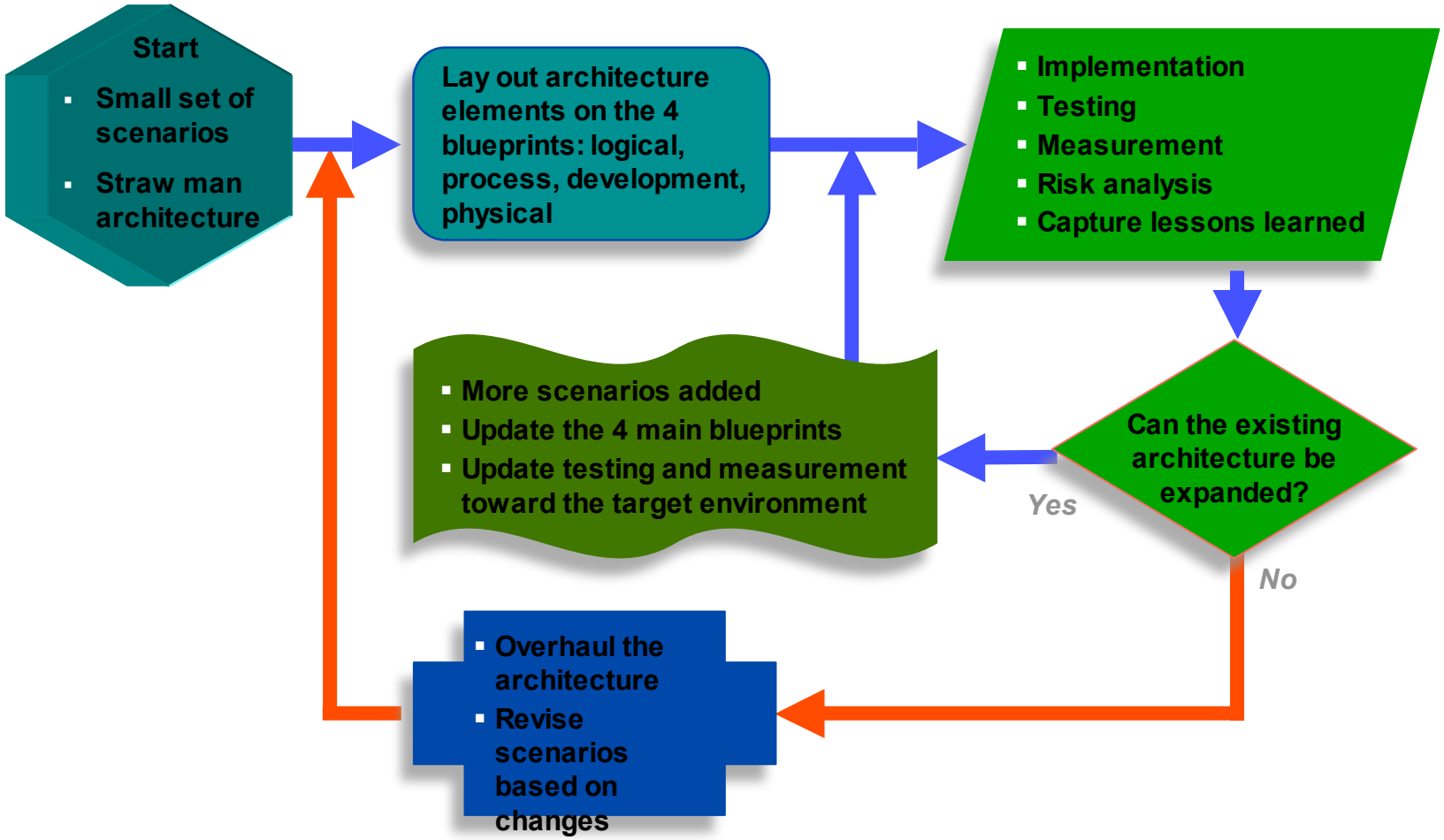
Verification and Validation

- 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.

Modern 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

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

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

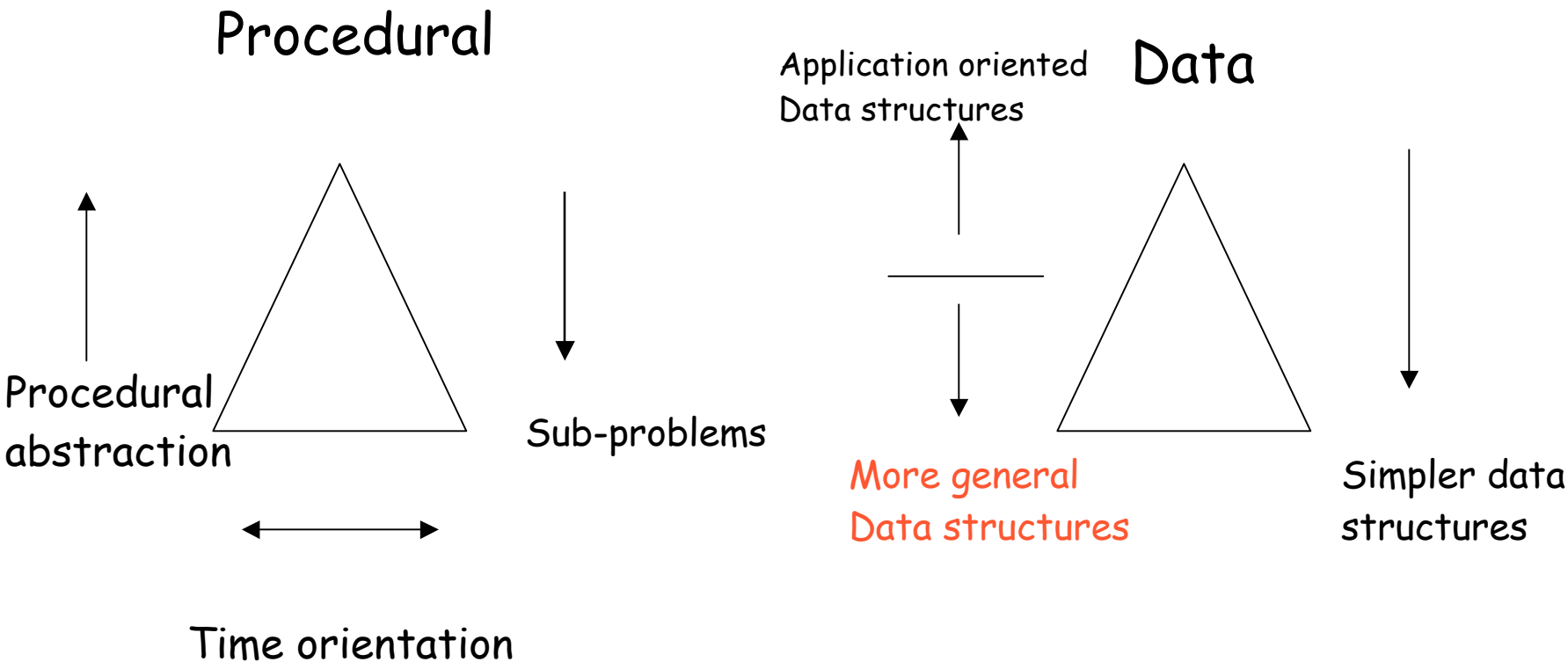
Design - 2

- 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

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

Essence of procedural and data abstraction



Modularity

- Modules and their interaction
- Compare designs by considering both a typology for the individual modules and connections between them
- 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

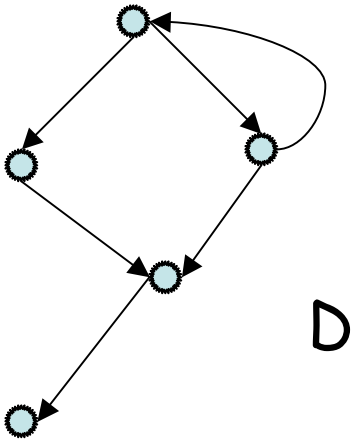
Complexity

- 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
- Halstead and McCabe

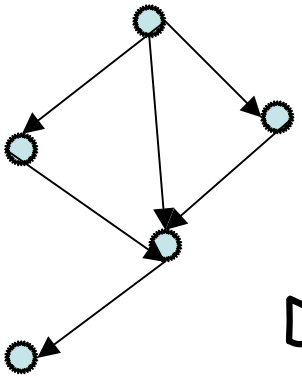
System Structure

- 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

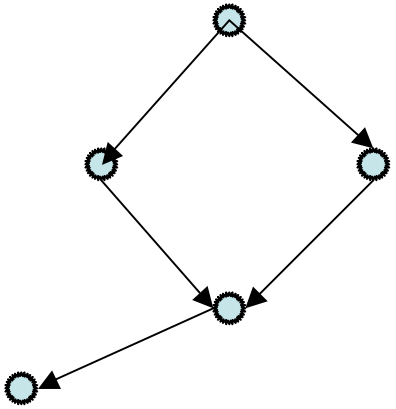
Module Hierarchies



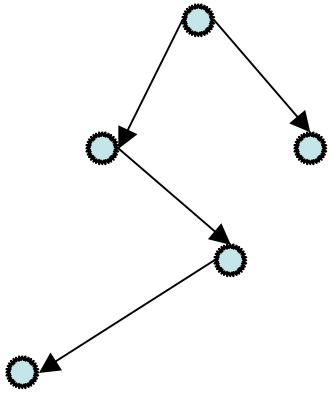
Directed graph



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

More on Design

- In the 9th lecture we will discuss design in the context of OO
- OO dominates the design community these days

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?

So Far

- Software Process Models, Software Project Planning (woosh!), Requirements, Estimation, Risk Analysis, Multics case study, Architecture Reviews, Questionnaire Design
- Software Quality Assurance
- Configuration Management and Testing
- This Time: Architecture and Design
- Next Time: Software Engineering skills: Problem Solving, meeting, stat, …

In Summary

- Covered a huge area = 2/3's of CS 565
- Architecture - outward looking, sets the stage
- Design - inward looking
- Next week a bit of a hiatus then OO