Lecture 12, Part 1: Software Evolution

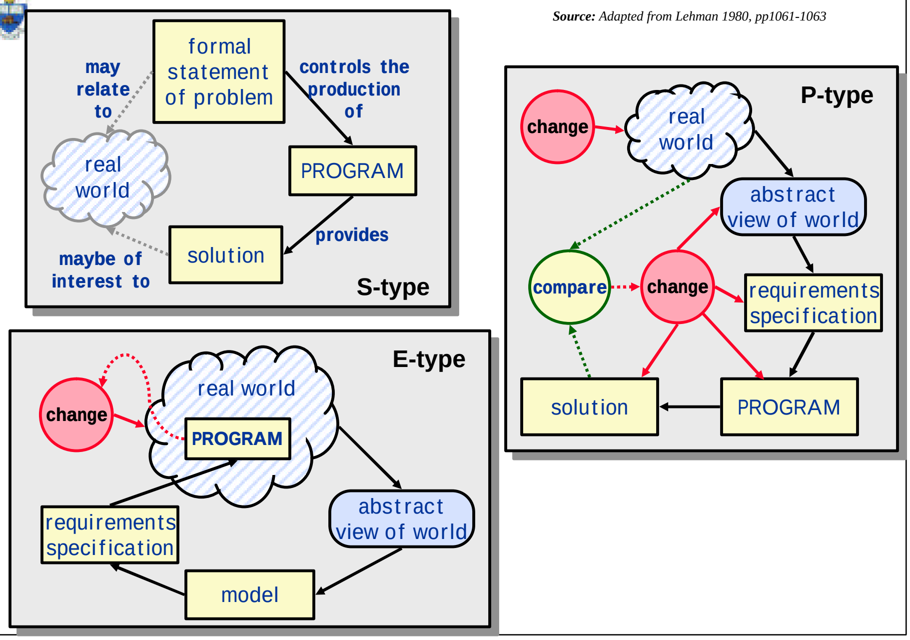
- **Basics of Software Evolution**
  - Laws of software evolution
  - Requirements Growth
  - Software Aging
- **Basics of Change Management**
  - Baselines, Change Requests and Configuration Management
- **Software Families - The product line approach**
- **Requirements Traceability**
  - Importance of traceability
  - Traceability tools

Program Types

Source: *Adapted from Lehman 1980, pp1061-1063*

- **S-type Programs ("Specifiable")**
  - problem can be stated formally and completely
  - acceptance: Is the program correct according to its specification?
  - This software does not evolve.
    - A change to the specification defines a new problem, hence a new program
- **P-type Programs ("Problem-solving")**
  - imprecise statement of a real-world problem
  - acceptance: Is the program an acceptable solution to the problem?
  - This software is likely to evolve continuously
    - because the solution is never perfect, and can be improved
    - because the real-world changes and hence the problem changes
- **E-type Programs ("Embedded")**
  - A system that becomes part of the world that it models
  - acceptance: depends entirely on opinion and judgement
  - This software is inherently evolutionary
    - changes in the software and the world affect each other

Source: Adapted from Lehman 1980, pp1061-1063



Laws of Program Evolution

Source: Adapted from Lehman 1980, pp1061-1063

➲ Continuing Change

- Any software that ***reflects some external reality*** undergoes continual change or becomes progressively less useful
  - change continues until it is judged more cost effective to replace the system

➲ Increasing Complexity

- As software evolves, its ***complexity*** increases…
  - …unless steps are taken to control it.

➲ Fundamental Law of Program Evolution

- Software evolution is self-regulating
  - …with statistically determinable trends and invariants

➲ Conservation of Organizational Stability

- During the active life of a software system, the work output of a development project is roughly constant (regardless of resources!)

➲ Conservation of Familiarity

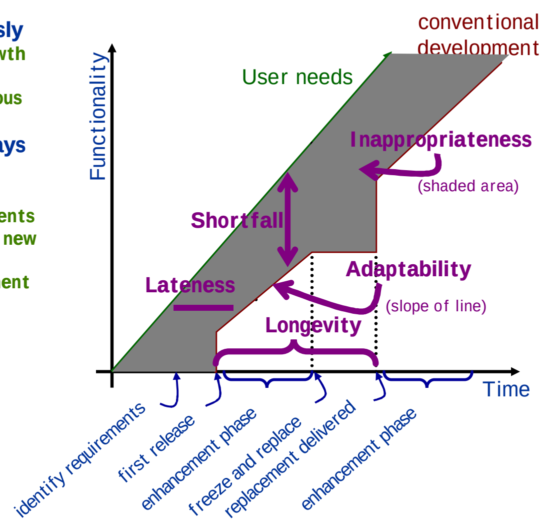
- The amount of change in successive releases is roughly constant

Requirements Growth

Source: Adapted from Davis 1988, pp1453-1455

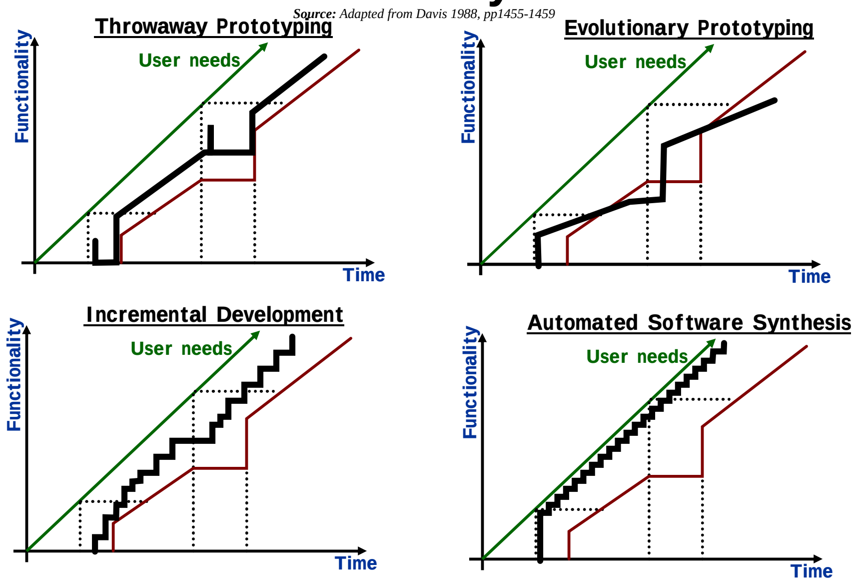
➲ **Davis's model:**

- **User needs evolve continuously**
  - Imagine a graph showing growth of needs over time
  - May not be linear or continuous (hence no scale shown)
- **Traditional development always lags behind needs growth**
  - first release implements only part of the original requirements
  - functional enhancement adds new functionality
  - eventually, further enhancement becomes too costly, and a replacement is planned
  - the replacement also only implements part of its requirements,
  - and so on...



Alternative lifecycle models

Source: Adapted from Davis 1988, pp1455-1459



Software "maintenance"

Source: *Adapted from Blum, 1992, p492-495*

Maintenance philosophies

- "throw-it-over-the-wall" - someone else is responsible for maintenance
  - investment in knowledge and experience is lost
  - maintenance becomes a reverse engineering challenge
- "mission orientation" - development team make a long term commitment to maintaining/enhancing the software

Basili's maintenance process models:

- Quick-fix model
  - changes made at the code level, as easily as possible
  - rapidly degrades the structure of the software
- Iterative enhancement model
  - Changes made based on an analysis of the existing system
  - attempts to control complexity and maintain good design
- Full-reuse model
  - Starts with requirements for the new system, reusing as much as possible
  - Needs a mature reuse culture to be successful

Software Aging

Source: *Adapted from Parnas, 1994*

Causes of Software Aging

- Failure to update the software to meet changing needs
  - Customers switch to a new product if benefits outweigh switching costs
- Changes to software tend to reduce its coherence

Costs of Software Aging

- Owners of aging software find it hard to keep up with the marketplace
- Deterioration in space/time performance due to deteriorating structure
- Aging software gets more buggy
  - Each "bug fix" introduces more errors than it fixes

Ways of Increasing Longevity

- Design for change
- Document the software carefully
- Requirements and designs should be reviewed by those responsible for its maintenance
- Software Rejuvenation…

Managing Requirements Change

- Managers need to respond to requirements change
  - Add new requirements during development
    - But not succumbing to feature creep
  - Modify requirements during development
    - Because development is a learning process
  - Remove requirements during development
    - requirements "scrub" for handling cost/schedule slippage
- Key techniques
  - Change Management Process
  - Release Planning
  - Requirements Prioritization (previous lecture!)
  - Requirements Traceability
  - Architectural Stability (next week's lecture)

Change Management

- Configuration Management
  - Each distinct product is a Configuration Item (CI)
  - Each configuration item is placed under version control
  - Control which version of each CI belongs in which build of the system
- Baselines
  - A baseline is a stable version of a document or system
    - Safe to share among the team
  - Formal approval process for changes to be incorporated into the next baseline
- Change Management Process
  - All proposed changes are submitted formally as change requests
  - A review board reviews these periodically and decides which to accept
    - Review board also considers interaction between change requests

Towards Software Families

- **Libraries of Reusable Components**
  - domain specific libraries (e.g. Math libraries)
  - program development libraries (e.g. Java AWT, C libraries)
- **Domain Engineering**
  - Divides software development into two parts:
    - domain analysis - identifies generic reusable components for a problem domain
    - application development - uses the domain components for specific applications.
- **Software Families**
  - Many companies offer a range of related software systems
    - Choose a stable architecture for the software family
    - identify variations for different members of the family
  - Represents a strategic business decision about what software to develop
  - Vertical families
    - e.g. 'basic', 'deluxe' and 'pro' versions of a system
  - Horizontal families
    - similar systems used in related domains

Requirements Traceability

- **From IEEE-STD-830:**
  - Backward traceability
    - i.e. to previous stages of development.
    - the origin of each requirement should be clear
  - Forward traceability
    - i.e., to all documents spawned by the SRS.
    - Facilitation of referencing of each requirement in future documentation
    - depends upon each requirement having a unique name or reference number.
- **From DOD-STD-2167A:**
  - A requirements specification is traceable if:
    - "(1) it contains or implements all applicable stipulations in predecessor document
    - (2) a given term, acronym, or abbreviation means the same thing in all documents
    - (3) a given item or concept is referred to by the same name in the documents
    - (4) all material in the successor document has its basis in the predecessor document, that is, no untraceable material has been introduced
    - (5) the two documents do not contradict one another"

Importance of Traceability

- **Verification and Validation**
  - assessing adequacy of test suite
  - assessing conformance to requirements
  - assessing completeness, consistency, impact analysis
  - assessing over- and under-design
  - investigating high level behavior impact on detailed specifications
  - detecting requirements conflicts
  - checking consistency of decision making across the lifecycle
- **Maintenance**
  - Assessing change requests
  - Tracing design rationale
- **Document access**
  - ability to find information quickly in large documents
- **Process visibility**
  - ability to see how the software was developed
  - provides an audit trail
- **Management**
  - change management
  - risk management
  - control of the development process

Source: *Adapted from Palmer, 1996, p365*

Traceability Difficulties

- **Cost**
  - very little automated support
  - full traceability is very expensive and time-consuming
- **Delayed gratification**
  - the people defining traceability links are not the people who benefit from it
    - development vs. V&V
  - much of the benefit comes late in the lifecycle
    - testing, integration, maintenance
- **Size and diversity**
  - Huge range of different document types, tools, decisions, responsibilities,…
  - No common schema exists for classifying and cataloging these
  - In practice, traceability concentrates only on baselined requirements

Source: *Adapted from Palmer, 1996, p365-6*

Current Practice

- **Coverage:**
  - links from requirements forward to designs, code, test cases,
  - links back from designs, code, test cases to requirements
  - links between requirements at different levels
- **Traceability process**
  - Assign each sentence or paragraph a unique id number
  - Manually identify linkages
  - Use manual tables to record linkages in a document
  - Use a traceability tool (database) for project wide traceability
  - Tool then offers ability to
    - follow links
    - find missing links
    - measure overall traceability

Limitations of Current Tools

- **Informational Problems**
  - Tools fail to track *useful* traceability information
    - e.g cannot answer queries such as "who is responsible for this piece of information?"
  - inadequate pre-requirements traceability
    - "where did this requirement come from?"
- **Lack of agreement…**
  - …over the quantity and type of information to trace
- **Informal Communication**
  - People attach great importance to personal contact and informal communication
    - These always supplement what is recorded in a traceability database
  - But then the traceability database only tells part of the story!
    - Even so, finding the appropriate people is a significant problem

Problematic Questions

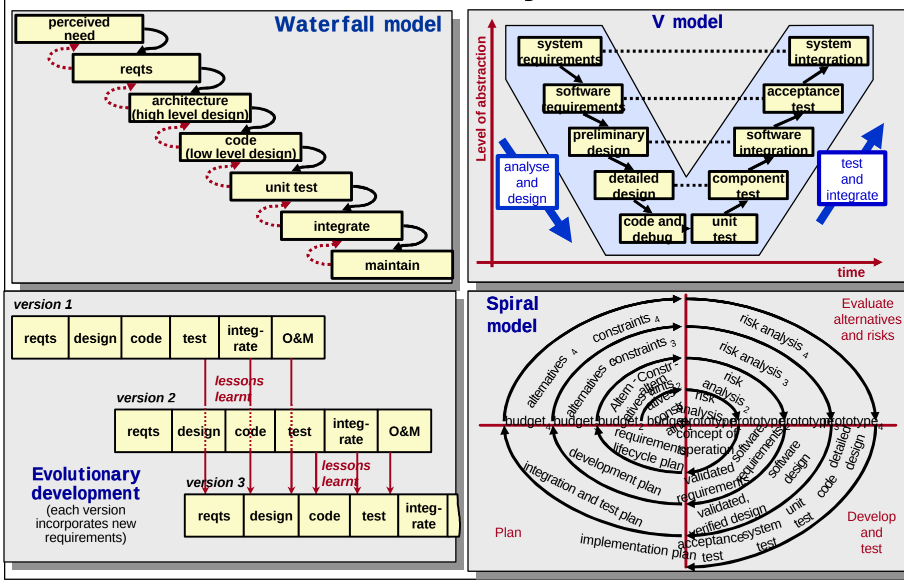
- **Involvement**
  - Who has been involved in the production of this requirement and how?
- **Responsibility & Remit**
  - Who is responsible for this requirement?
    - who is currently responsible for it?
    - at what points in its life has this responsibility changed hands?
  - Within which group's remit are decisions about this requirement?
- **Change**
  - At what points in the life of this requirements has working arrangements of all involved been changed?
- **Notification**
  - Who needs to be involved in, or informed of, any changes proposed to this requirement?
- **Loss of knowledge**
  - What are the ramifications regarding the loss of project knowledge if a specific individual or group leaves?

Source: Adapted from Gotel & Finkelstein, 1997, p100

Lecture 12, Part 2: Moving into Design

- **Analysis vs. Design**
  - Why the distinction?
- **Design Processes**
  - Logical vs. Physical Design
  - System vs. Detailed Design
- **Architectures**
  - System Architecture
  - Software Architecture
  - Architectural Patterns (next lecture)
- **Useful Notation**
  - UML Packages and Dependencies

Refresher: Lifecycle models



Analysis vs. Design

- **Analysis**
  - Asks "what is the problem?"
    - what happens in the current system?
    - what is required in the new system?
  - Results in a detailed understanding of:
    - Requirements
    - Domain Properties
  - Focuses on the way human activities are conducted
- **Design**
  - Investigates "how to build a solution"
    - How will the new system work?
    - How can we solve the problem that the analysis identified?
  - Results in a solution to the problem
    - A working system that satisfies the requirements
    - Hardware + Software + Peopleware
  - Focuses on building technical solutions
- **Separate activities, but not necessarily sequential**

Refresher: different worlds

Analysis is all about studying this world

Design is all about building this world

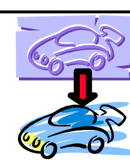
But who builds the bridge?

Four design philosophies

Decomposition & Synthesis

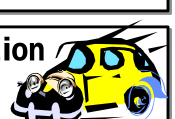
- **Drivers:**
  - Managing complexity
  - Reuse
- **Example:**
  - Design a car by designing separately the chassis, engine, drivetrain, etc. Use existing *components* where possible

Search



- **Drivers**
  - Transformation
  - Heuristic Evaluation
- **Example:**
  - Design a car by *transforming* an initial rough design to get closer and closer to what is desired

Negotiation

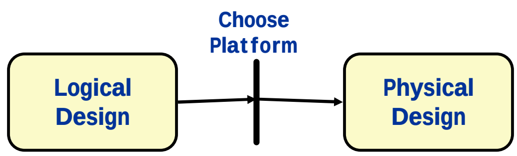


- **Drivers**
  - Stakeholder Conflicts
  - Dialogue Process
- **Example:**
  - Design a car by getting *each stakeholder* to suggest (partial) designs, and them compare and discuss them

Situated Design

- **Drivers**
  - Errors in existing designs
  - Evolutionary Change
- **Example:**
  - Design a car by observing what's wrong with existing cars *as they are used*, and identifying improvements

Logical vs. Physical Design



➲ Logical Design concerns:

- Anything that is platform-independent:
  - Interactions between objects
  - Layouts of user interfaces
  - Nature of commands/data passed between subsystems
- Logical designs are usually portable to different platforms

➲ Physical Design concerns:

- Anything that depends on the choice of platform:
  - Distribution of objects/services over networked nodes
  - Choice of programming language and development environment
  - Use of specialized device drivers
  - Choice of database and server technology
  - Services provided by middleware

System Design vs. Detailed Design

➲ System Design

- Choose a System Architecture
  - Networking infrastructure
  - Major computing platforms
  - Roles of each node (e.g. client-server; clients-broker-servers; peer-to-peer,…)
- Choose a Software Architecture
  - (see next lecture for details)
- Identify the subsystems
- Identify the components and connectors between them
  - Design for modularity to maximize testability and evolveability
  - E.g. Aim for low coupling and high cohesion

➲ Detailed Design

- Decide on the formats for data storage
  - E.g. design a data management layer
- Design the control functions for each component
  - E.g. design an application logic layer
- Design the user interfaces
  - E.g. design a presentation layer

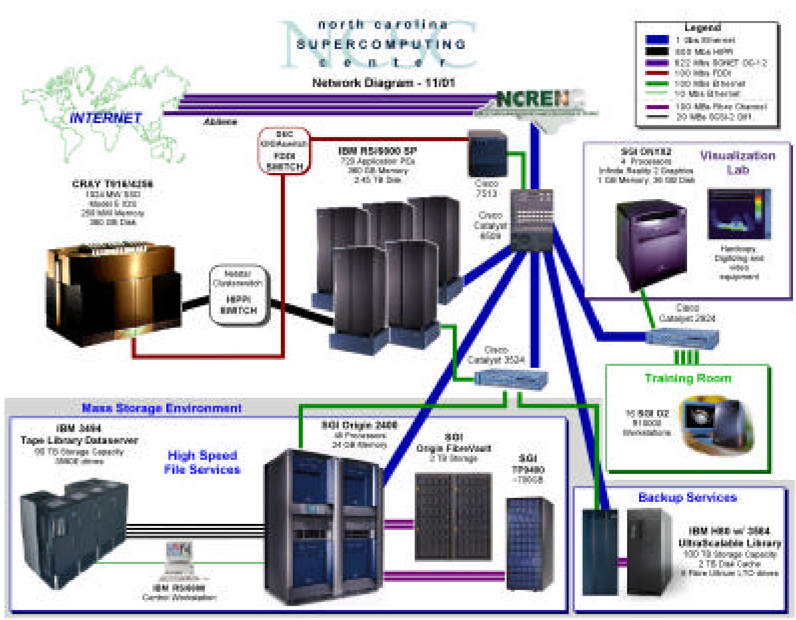
Global System Architecture

➲ **Choices:**

- Allocates users and other external systems to each node
- Identify appropriate network topology and technologies
- Identify appropriate computing platform for each node

➲ **Example:**

- See next slide...



System Architecture Questions

- **Key questions for choosing platforms:**
  - What hardware resources are needed?
    - CPU, memory size, memory bandwidth, I/O, disk space, etc.
  - What software/OS resources are needed?
    - application availability, OS scalability
  - What networking resources are needed?
    - network bandwidth, latency, remote access.
  - What human resources are needed?
    - OS expertise, hardware expertise,
    - system administration requirements,
    - user training/help desk requirements.
  - What other needs are there?
    - security, reliability, disaster recovery, uptime requirements.
- **Key questions constraining the choice:**
  - What funding is available?
  - What resources are already available?
    - Existing hardware, software, networking
    - Existing staff and their expertise
    - Existing relationships with vendors, resellers, etc.

Data Management Questions

- **How is data entry performed?**
  - E.g. Keyless Data entry
    - bar codes; Optical Character Recognition (OCR)
  - E.g. Import from other systems
    - Electronic Data Interchange (EDI), Data interchange languages,…
- **What kinds of data persistence is needed?**
  - Is the operating system's basic file management sufficient?
  - Is object persistence important?
  - Can we isolate persistence mechanisms from the applications?
- **Is a Database Management System (DBMS) needed?**
  - Is data accessed at a fine level of detail
    - E.g. do users need a query language?
  - Is sophisticated indexing required?
  - Is there a need to move complex data across multiple platforms?
    - Will a data interchange language suffice?
    - E.g. HTML, SGML, XML…
  - Is there a need to access the data from multiple platforms?

Software Architecture

- A software architecture defines:
  - the components of the software system
  - how the components use each other's functionality and data
  - How control is managed between the components
- An example: client-server
  - Servers provide some kind of service; clients request and use services
  - applications are located with clients
    - E.g. running on PCs and workstations;
  - data storage is treated as a server
    - E.g. using a DBMS such as DB2, Ingres, Sybase or Oracle
    - Consistency checking is located with the server
  - Advantages:
    - Breaks the system into manageable components
    - Makes the control and data persistence mechanisms clearer
  - Variants:
    - Thick clients have their own services, thin ones get everything from servers
  - Note: This is a SOFTWARE architecture
    - Clients and server could be on the same machine or different machines…

Coupling

Given two units *(e.g. methods, classes, modules, …)*, A and B:

Form	Features	Desirability
Data coupling	A & B communicate by simple data only	High (use parameter passing & only pass necessary info)
Stamp coupling	A & B use a common type of data	Okay (but should they be grouped in a data abstraction?)
Control coupling (activating)	A transfers control to B by procedure call	Necessary
Control coupling (switching)	A passes a flag to B to tell it how to behave	Undesirable (why should A interfere like this?)
Common environment coupling	A & B make use of a shared data area (global variables)	Undesirable (if you change the shared data, you have to change both A and B)
Content coupling	A changes B's data, or passes control to the middle of B	Extremely Foolish (almost impossible to debug!)

Cohesion

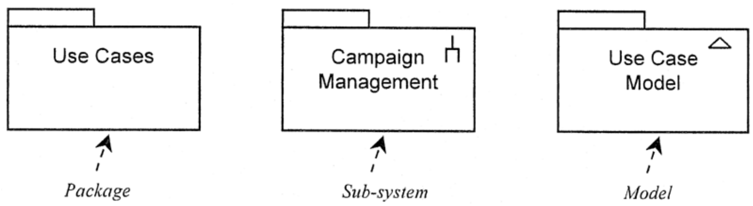
How well do the contents of an object *(module, package,…)* go together?

Form	*Features*	*Desirability*
Data cohesion	all part of a well defined data abstraction	Very High
Functional cohesion	all part of a single problem solving task	High
Sequential cohesion	outputs of one part form inputs to the next	Okay
Communicational cohesion	operations that use the same input or output data	Moderate
Procedural cohesion	a set of operations that must be executed in a particular order	Low
Temporal cohesion	elements must be active around the same time (e.g. at startup)	Low
Logical cohesion	elements perform logically similar operations (e.g. printing things)	No way!!
Coincidental cohesion	elements have no conceptual link other than repeated code	No way!!

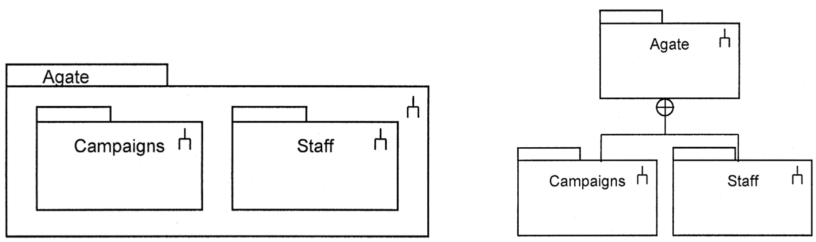
UML Packages

- **We need to represent our architectures**
  - UML elements can be grouped together in packages
  - Elements of a package may be:
    - other packages (representing subsystems or modules);
    - classes;
    - models (e.g. use case models, interaction diagrams, statechart diagrams, etc)
  - Each element of a UML model is owned by a single package
  - Packages need not correspond to elements of the analysis or the design
    - they are a convenient way of grouping other elements together
- **Criteria for decomposing a system into packages:**
  - Ownership
    - who is responsible for working on which diagrams
  - Application
    - each problem has its own obvious partitions;
  - Clusters of classes with strong cohesion
    - e.g., course, course description, instructor, student,…
  - Or use an architectural pattern to help find a suitable decomposition

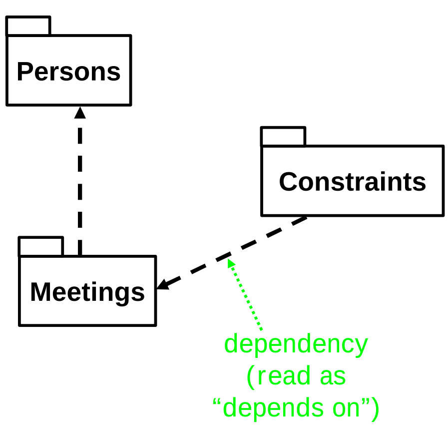
Package notation



➲ **2 alternatives for showing package containment:**



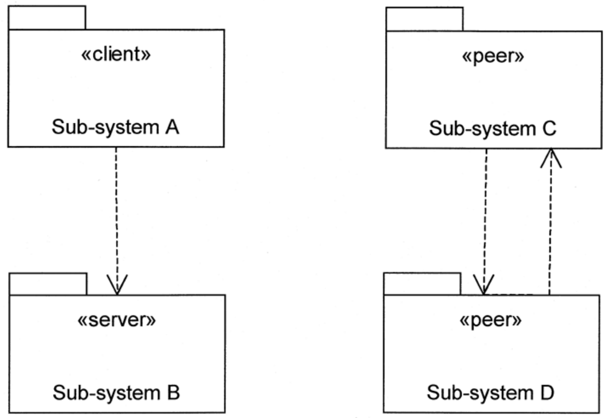
Package Diagrams



➲ **Dependencies:**
- Similar to compilation dependencies
- Captures a high-level view of coupling between packages:
  - If you change a class in one package, you may have to change something in packages that depend on it

➲ **A good architecture minimizes dependencies**
- Fewer dependencies means lower coupling
- Dependency cycles are especially undesirable

...Dependency Cycles



The server sub-system does not depend on the client sub-system and is not affected by changes to the client's interface.

Each peer sub-system depends on the other and each is affected by changes in the other's interface.

Architectural Patterns

E.g. 3 layer architecture: