Lecture 22: Moving into Design

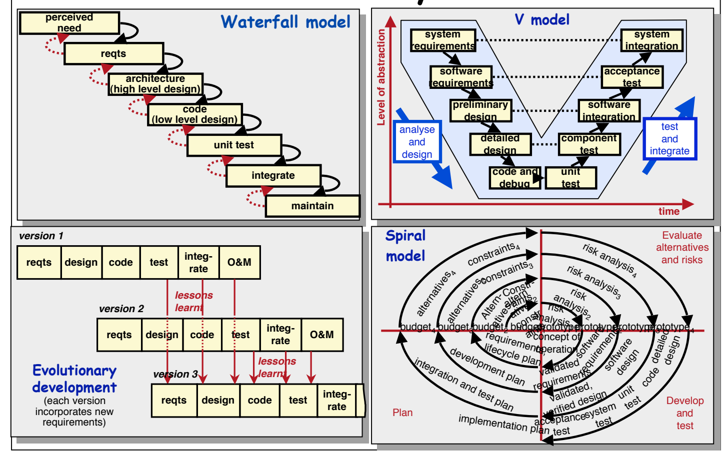
→ **Analysis vs. Design**
- Why the distinction?

→ **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



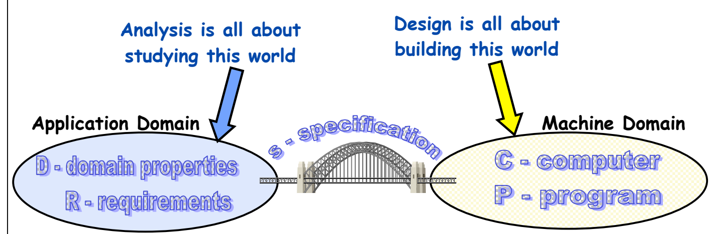
Analysis vs. Design

→ **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



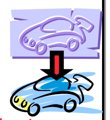
But who builds the bridge?

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

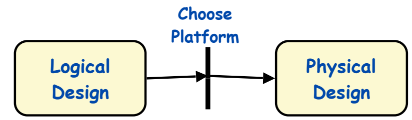
Negotiation

- 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

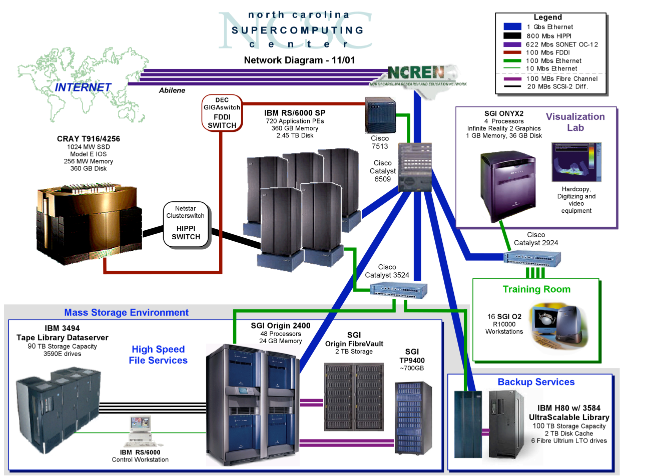
→ **System Design**
- Choose a System Architecture
  - Networking infrastructure
  - Major computing platforms
  - Roles of each node (e.g. client-server; clients-broker-servers; peer-to-peer,…)
- Choose a Software Architecture
  - (see next lecture for details)
- Identify the subsystems
- Identify the 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?

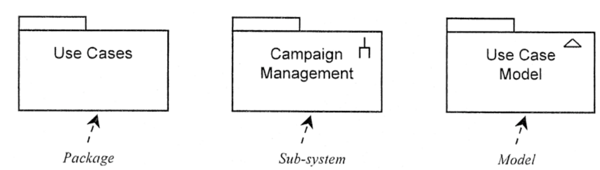
Form	Features	Desirability
Data cohesion	all part of a well defined data abstraction	Very High
Functional cohesion	all part of a single problem solving task	High
Sequential cohesion	outputs of one part form inputs to the next	Okay
Communicational cohesion	operations that use the same input or output 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

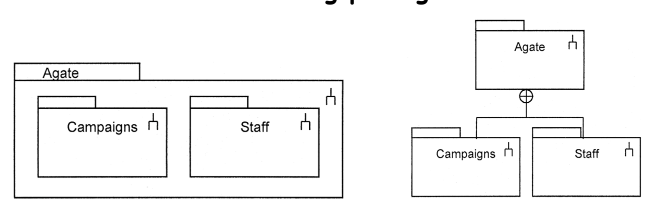
→ We need to represent our architectures
- UML elements can be grouped together in packages
- Elements of a package may be:
  - other packages (representing subsystems or modules);
  - classes;
  - models (e.g. use case models, interaction diagrams, statechart diagrams, etc)
- Each element of a UML model 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

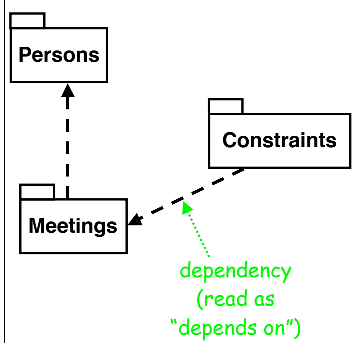
Package notation



→ 2 alternatives for showing package containment:

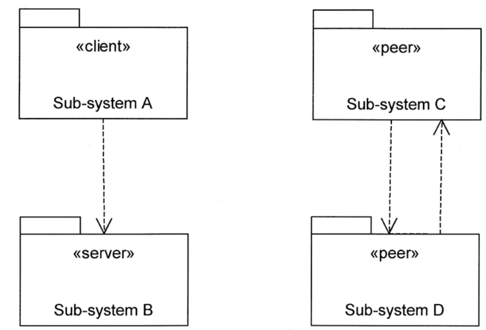


Package Diagrams



- 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:

Presentation Layer
Application Logic Layer
Storage Layer