Software Architecture and Design I

Instructor: Yongjie Zheng

February 25, 2016

CS 490MT/5555
Software Methods and Tools

Outline

- **What** is software architecture?
- **Why** do we need software architecture?
- **How** do we design software architecture?
  - Object-oriented design and functional design
  - Architecture patterns and styles

What is software architecture? (in traditional software engineering)

- *The top-level decomposition of a software system into major sub-systems together with a characterization of how these sub-systems interact is called software architecture.*
- Software architecture is top-level design or global design!
- Software architecture is the result of the initial design process.

What is software architecture? (in current research)

- *The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.* - [Bass et al., 1998]
- *Software architecture* = {*Elements, Form, Rationale*} - [Dewayne Perry & Alex Wolf, 1992]
- This is so-called *4C* model.
  - **C**omponent, **C**onnector, **C**onfiguration, **C**onstraint.

What is software architecture? (a new definition)

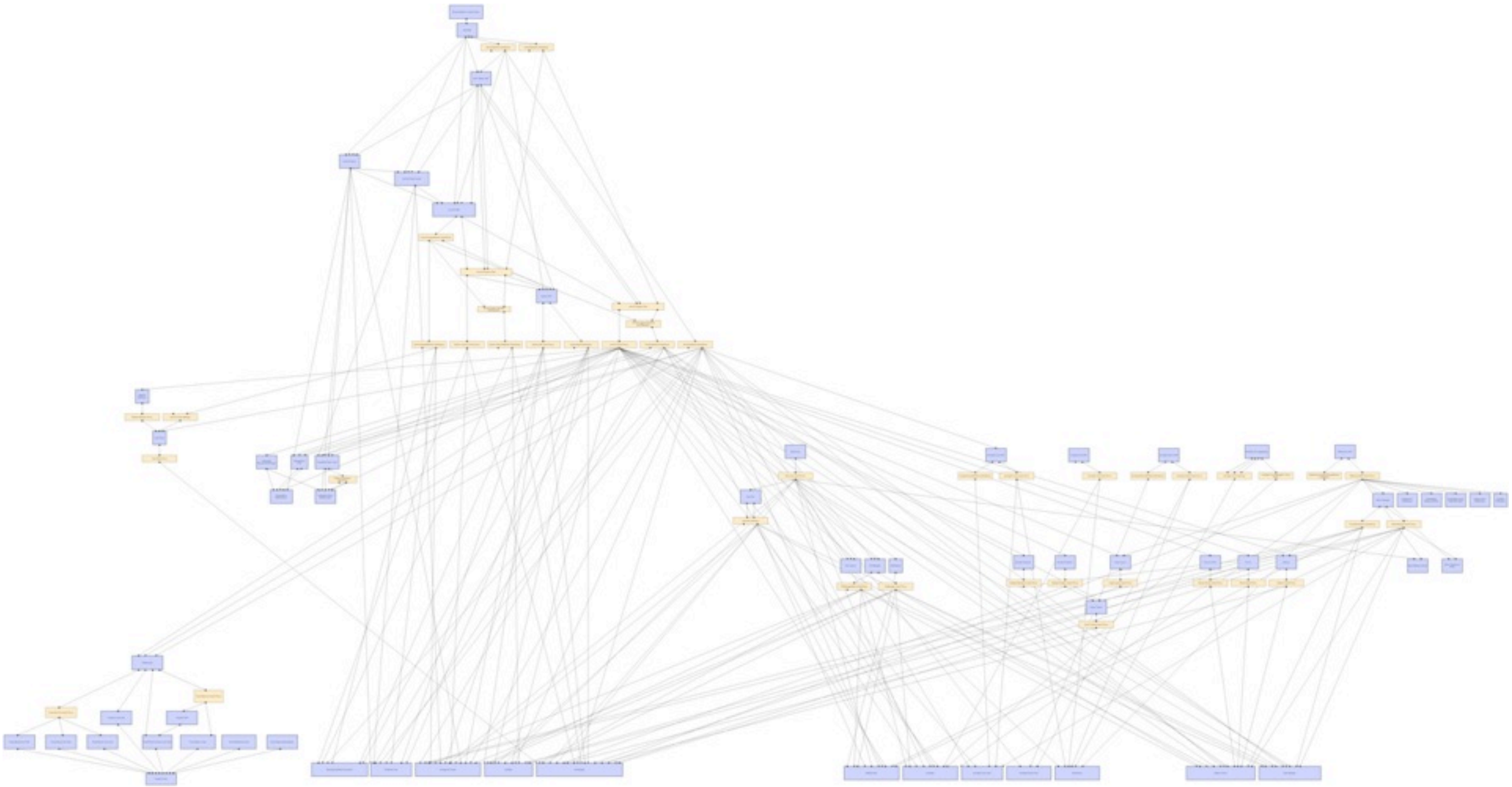
- *A software system's architecture is the set of principal design decisions made about the system.* - [Taylor & Medvidovic & Dashofy]

- This definition particularly emphasizes *extensibility* of software architecture.
  - "principal" is up to stakeholders to decide.
  - Note that the definition does not say anything about what software architecture should look like, and how it should be modeled.

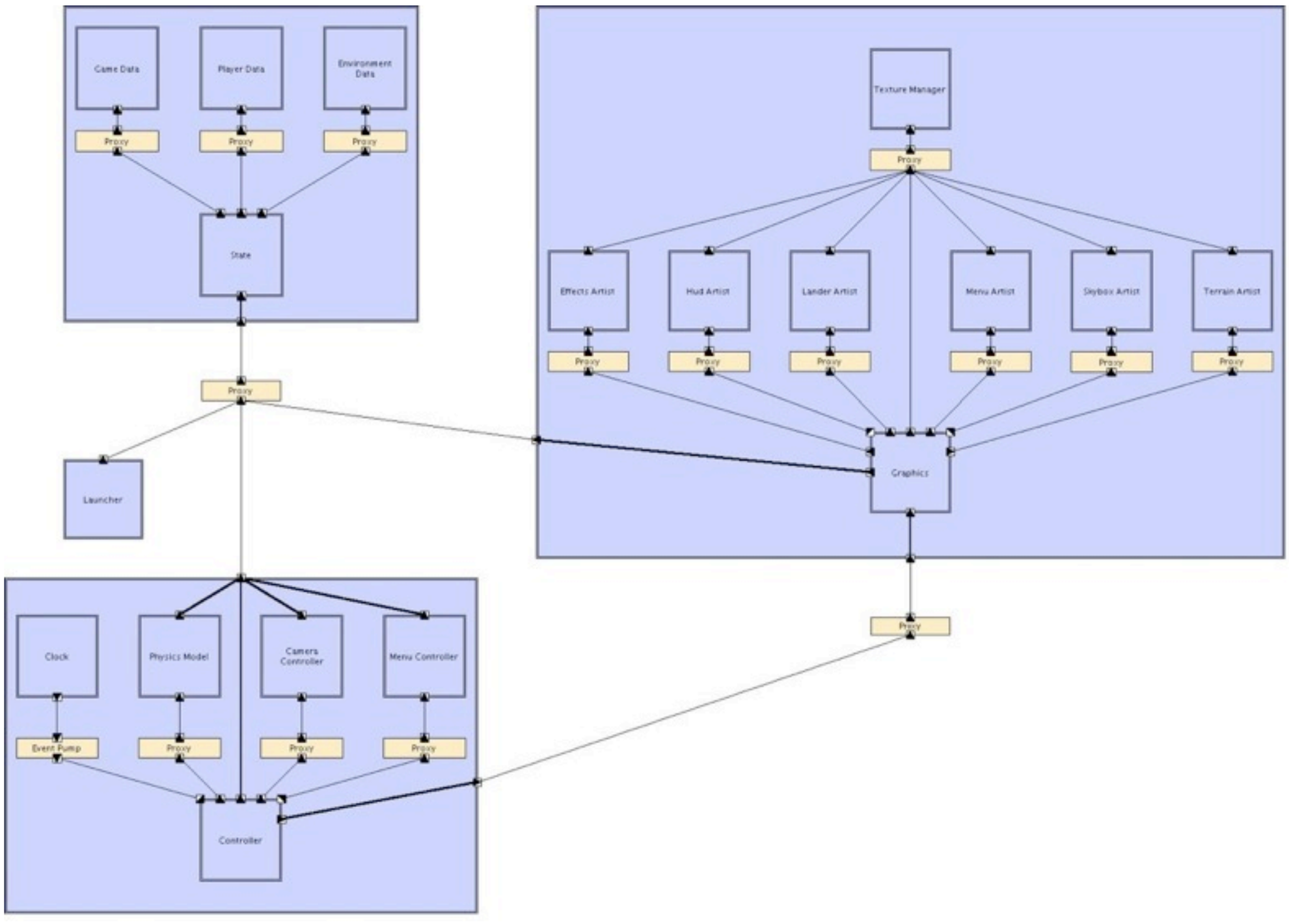
Components and Connectors

- **Component** is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.
- **Connector** is an architecture element tasked with effecting and regulating interactions among components.

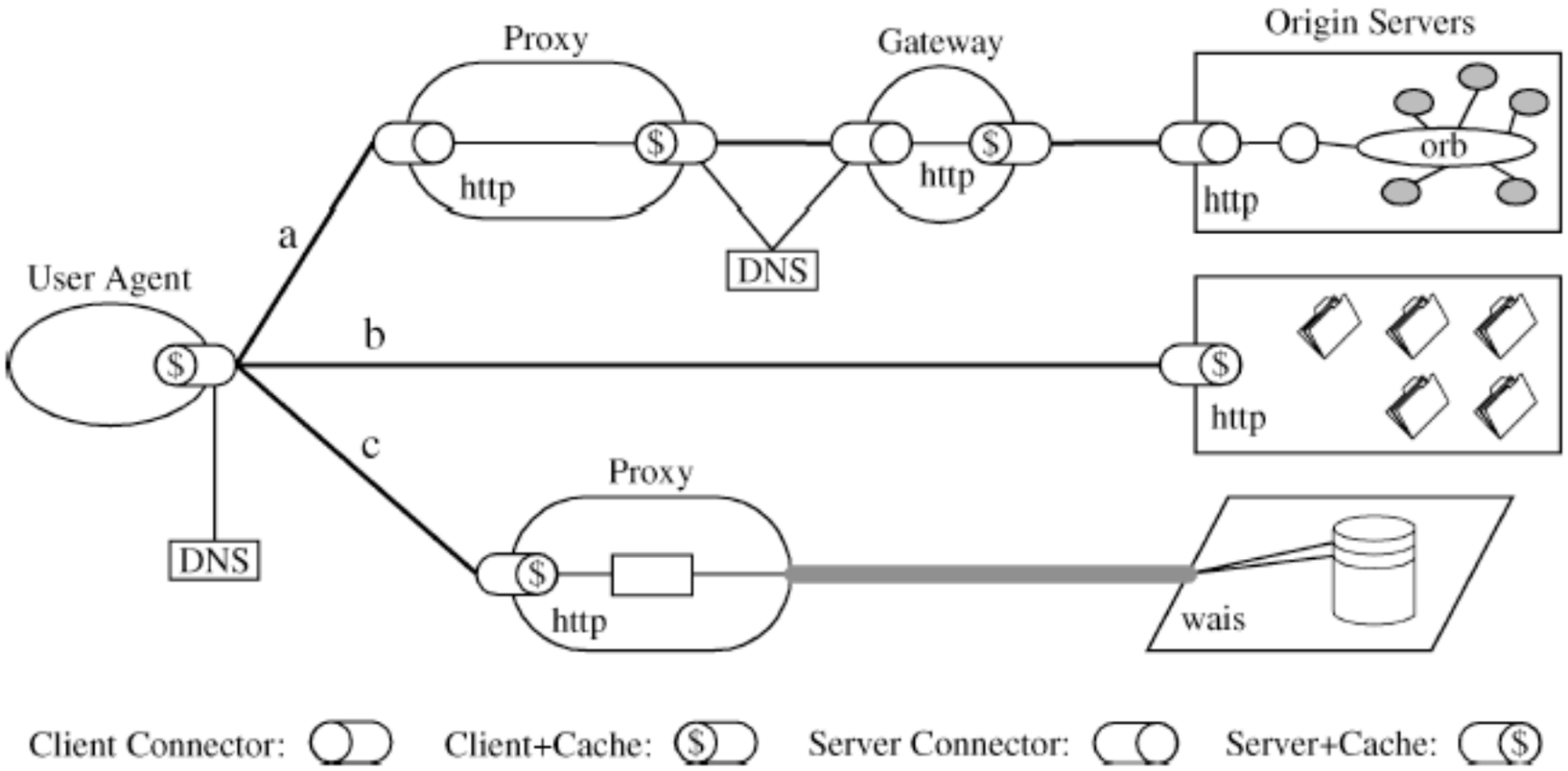
An example of software architecture



A less complicated example



Another example



Why do we need software architecture? (architecture in context)

- Requirements
  - Existing architectures provide vocabulary for requirements.
- Design
  - Software architecture is the outcome of software design.
- Implementation
  - High-level reuse
- Testing
  - Early system analysis
- Maintenance
  - Open architecture

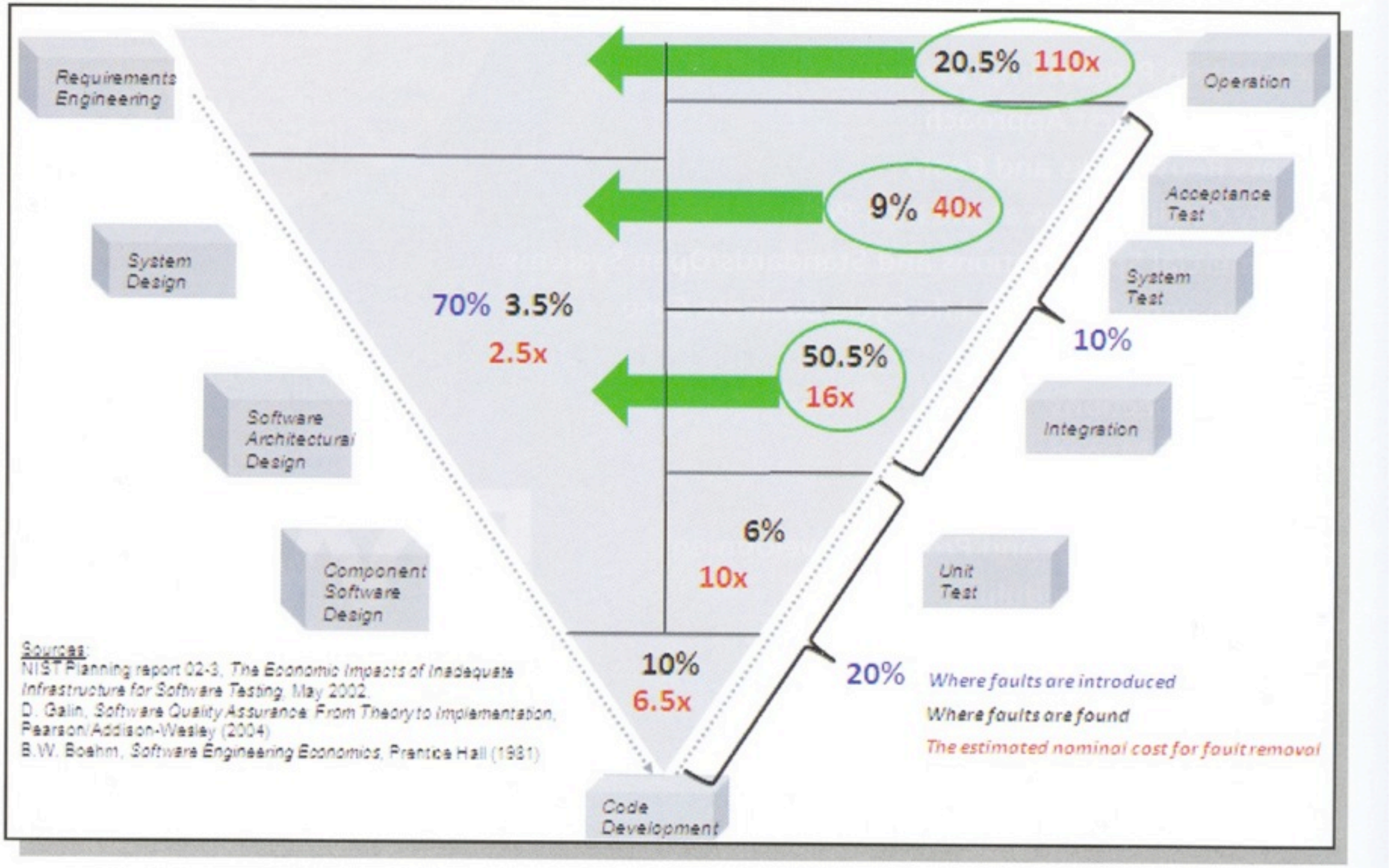


Figure 1: Fault Introduction, Discovery and Cost Factors

How do we design architecture?

- Creativity
  - This requires extensive experience, broad training, ...
- Principles, process, and methods
  - Goals, activities, and principles
  - Design methods: object-oriented design, functional design, and quality-driven design
- Reuse
  - Horizontal reuse: architecture patterns and styles
  - Vertical reuse: product-line architectures

Goals (Considerations) of Architecture Design

Conceptual Integrity

The fact that a software product presents to each of its users a coherent mental model of the application, of strategies for doing the application, and of the user-interface tactics to be used in specifying actions and parameters. The conceptual integrity of the product is the most important fact in ease of use. - [Fred Brooks]

Conceptual integrity implies that the similar or same design decisions are made to solve a collection of similar problems for the same goal.

Activities of Architecture Design

- Analyzing and refining requirements
- **Decomposing** the system into components
- Selecting protocols for communication, synchronization, and data access
- Developing global structures
- Designing component internal structures
- Selecting among design alternatives (often needs to consider non-functional properties)
- Dealing with deployment issues
- Making stakeholder related decisions

Design Principles

- **Abstraction**: we concentrate on the essential features and ignore, abstract from, details that are not relevant at the level we are currently working.
- **Modularity**: the degree (cohesion, coupling) to which a system is partitioned into components (or modules).
  - Cohesion: a measure of the mutual affinity of the elements of a component. E.g. functional cohesion, data cohesion.
  - Coupling: a measure of the strength of the inter-component connections. E.g. control coupling, data coupling.
  - High-cohesion and loose-coupling are generally preferred.

Design Principles, cont.

- **Information hiding** (i.e. encapsulation): one begins with a list of difficult design decisions or design decisions which are likely to change. Every module (component) is designed to hide such a decision from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings. [Parnas]

Design Methods

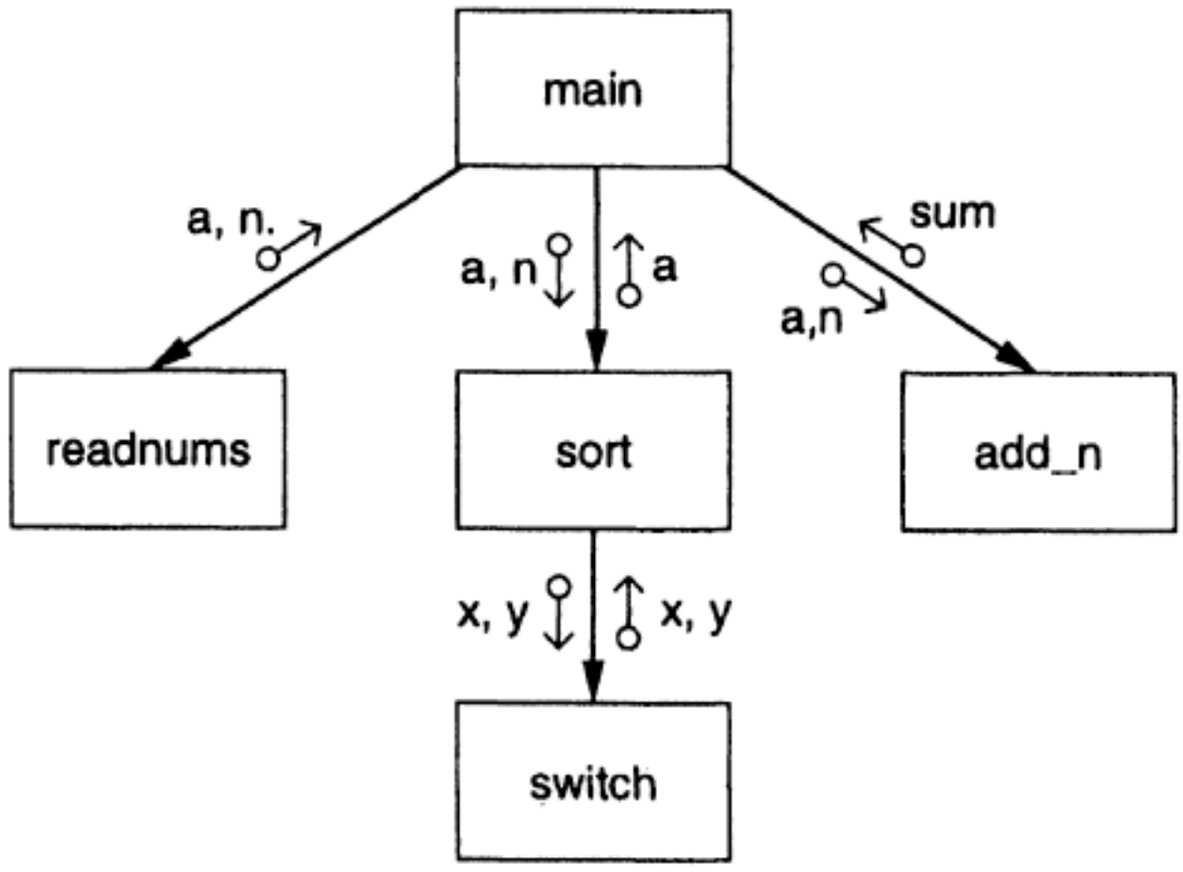
- Function-Oriented Design
  - Views a system as a transformation function that transforms specified input to specified output.
  - Separates data and procedures, and models them separately.
    - The state information is usually in centralized data stores.
- Object-Oriented Design
  - Views a system as a group of objects that interact to satisfy system requirements.
  - Combines data and methods into a cohesive entity.
  - The state information is distributed among system objects.

Function-Oriented Design

- Design notation: structure charts
- Design method: structured design method

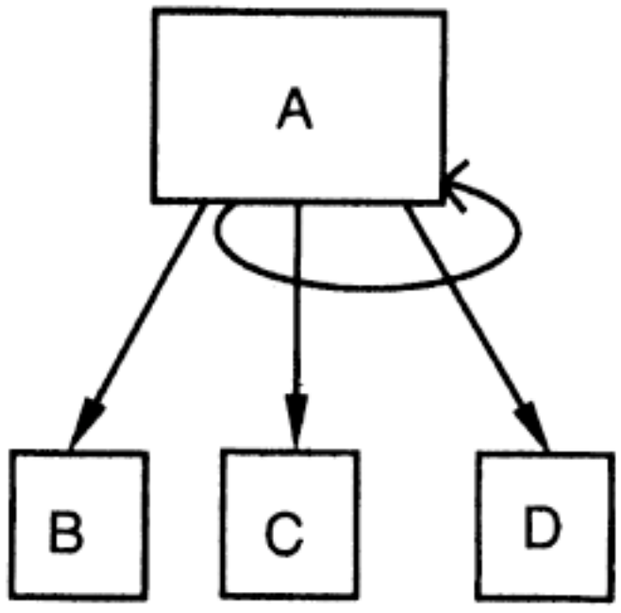
Structure Charts

- In a structure chart, a module is represented by a box with the module name written in the box.
- An arrow from module A to module B represents that module A invokes module B.
  - B is called the *subordinate* of A;
  - A is called the *superordinate* of B.
- The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows.

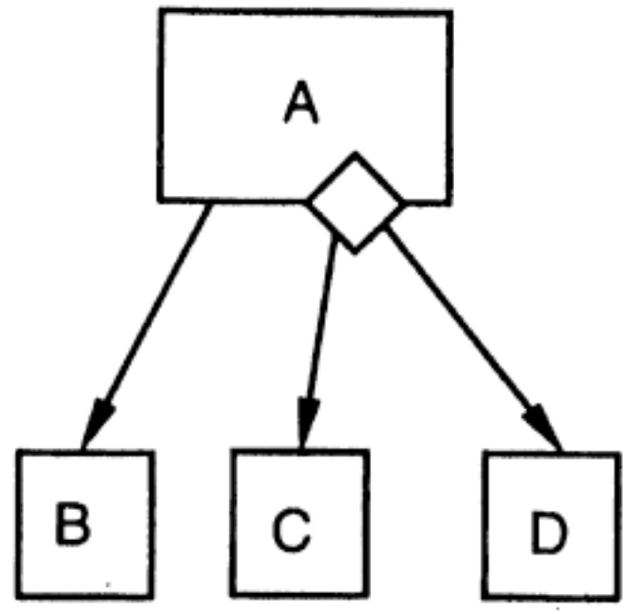


An example of structure charts

Iteration and decision can be explicitly represented.



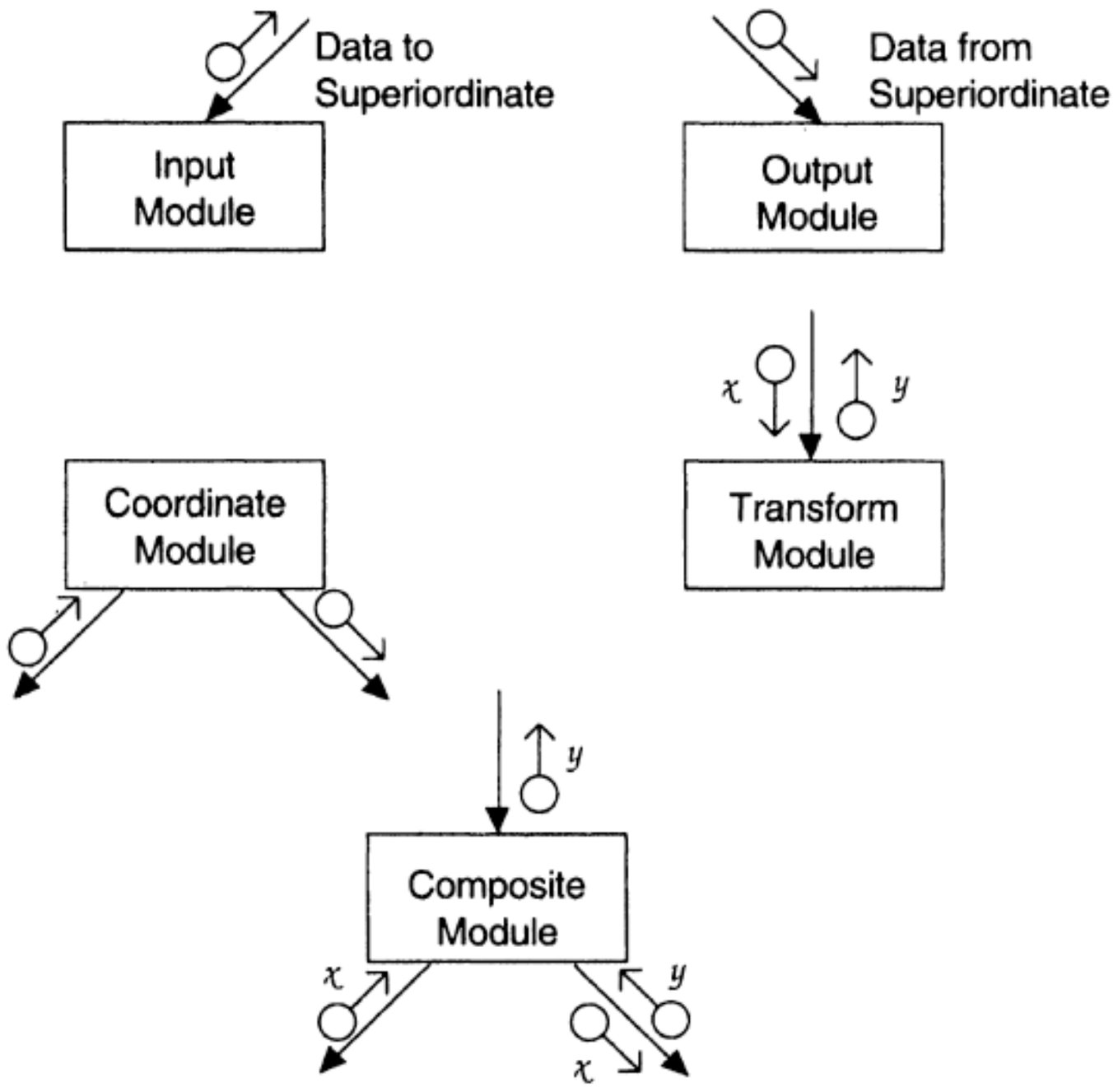
A repeatedly calls modules C and D.



The invocation of modules C and D in module A depends on the outcome of some decision.

Types of modules

- **Input module**: gets the data from the sources and gets it ready to be processed.
- **Output module**: takes the output produced and prepares it for proper presentation to the environment.
- **Transform module**: transforms data into some other form.
- **Coordinate module**: manages the flow of data to and from different subordinates.
- **Composite module**: performs functions of more than one type of module


Data to
Superiordinate
Data from
Superiordinate
Input
Module
Output
Module
x
y
Coordinate
Module
Transform
Module
y
Composite
Module
x
y
x

Structured Design Method (SDM)

SDM is a function-oriented design method that views every software system as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function.

Overall strategy: identify the input and output streams and the primary transformations that have to be performed to produce the output. High-level modules are then created to perform these major activities, which are later refined.

Structured Design

Factoring: the process of decomposing a module so that the bulk of its work is done by its subordinates.

A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination.

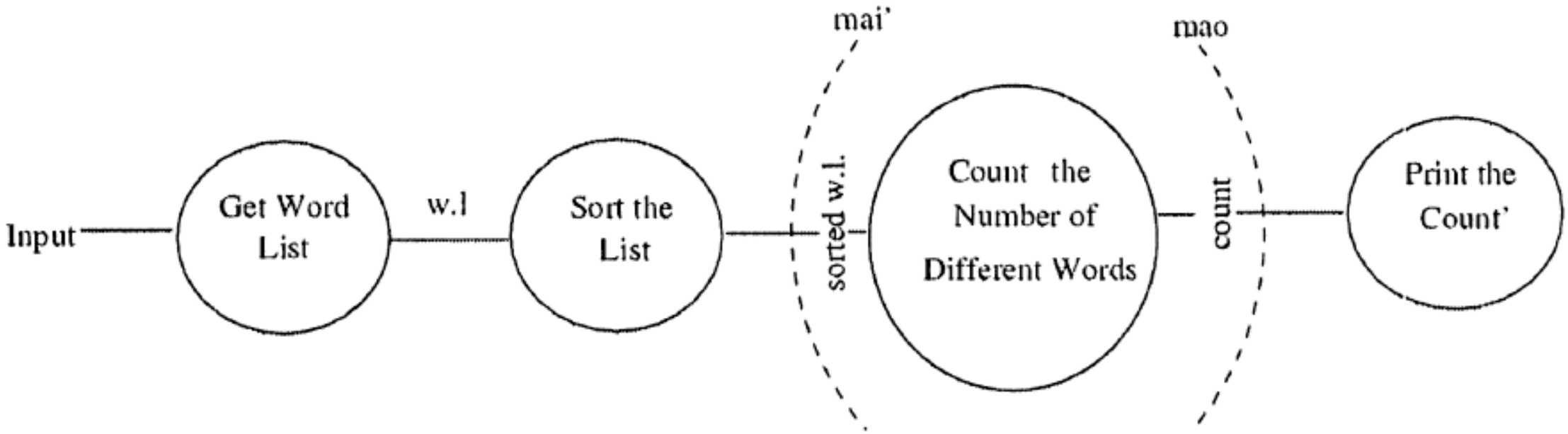
Structured design attempts to achieve a structure that is close to being completely factored.

Four steps of structured design

1. Restate the problem as a data flow diagram.
2. Identify the input and output data elements.
3. First-level factoring.
4. Factoring of input, output, and transform branches.

Step 1: Restate the Problem as a Data Flow Diagram (DFD)

A DFD shows the major transforms that the software will have and how the data will flow through different transforms.



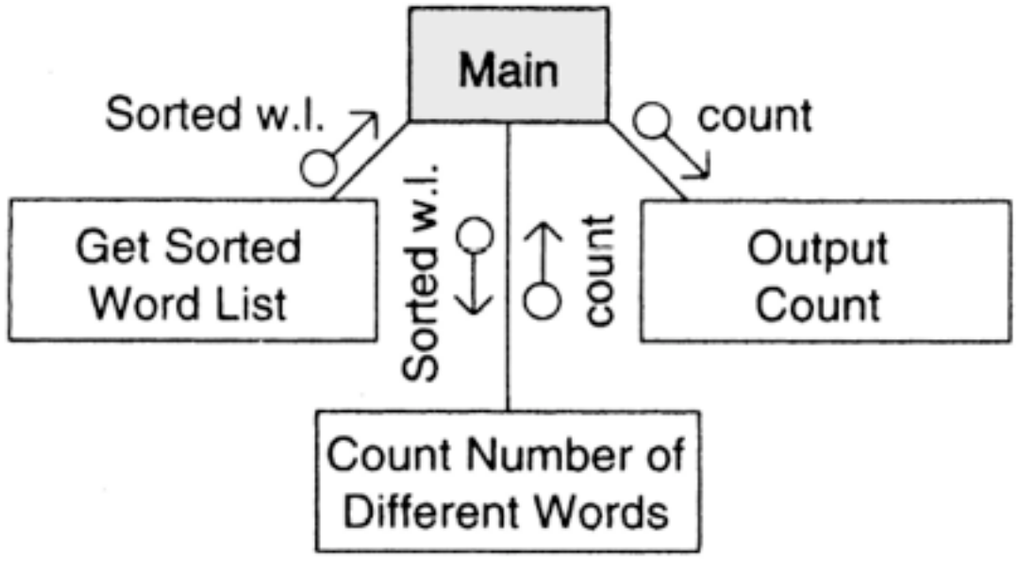
Example: count the number of different words in an input file.

Step 2: Identify the Most Abstract Input and Output Data Elements

- **Most Abstract Input** (MAI): the data elements in the data flow diagram that are farthest from the physical inputs but can still be considered inputs to the system.
- **Most Abstract Output** (MAO): the data elements that are farthest from the actual outputs but can still be considered outgoing.
- **Central Transform**: the transforms left between MAI and MAO that take MAI and transform it into MAO.

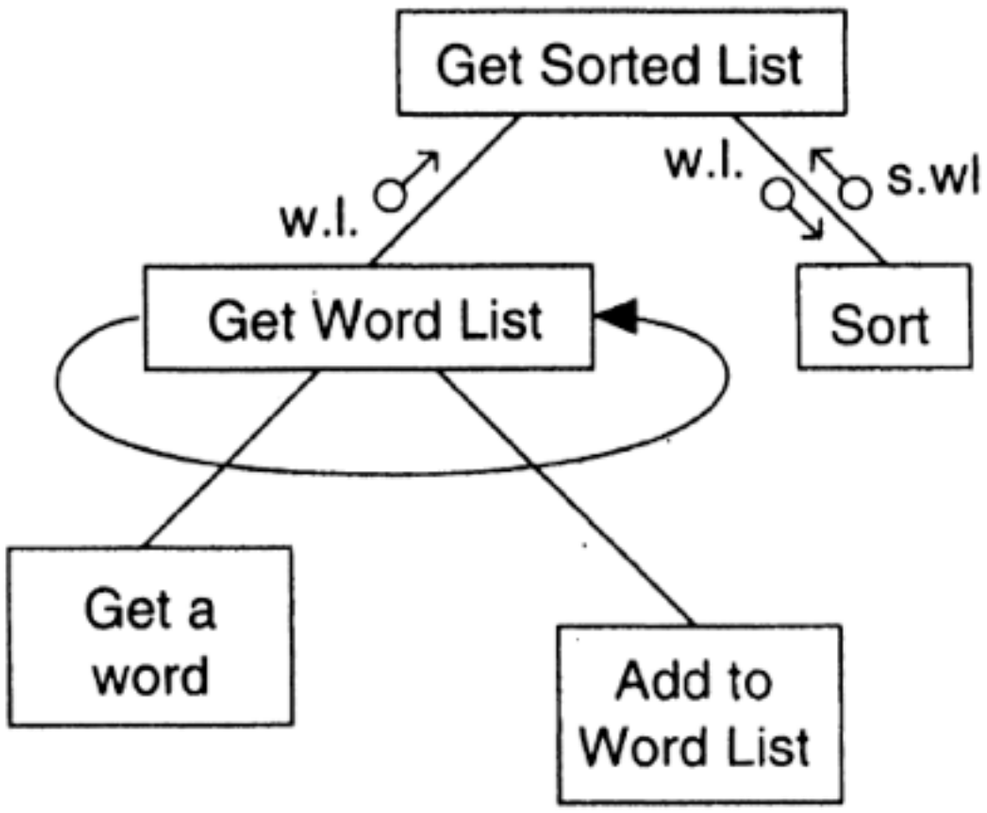
Step 3: First-Level Factoring

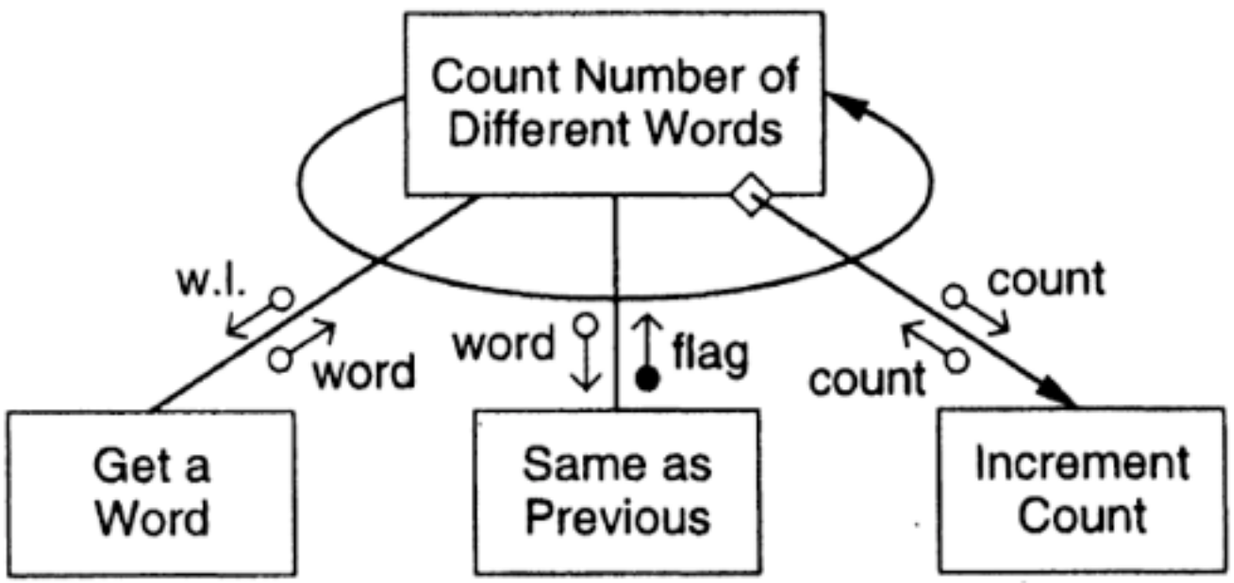
- First, create a main module.
- For each MAI data item, create an immediate subordinate module that delivers data to the main module.
- For each MAO data item, create an immediate subordinate module that accepts data from the main module.
- For each central transform, create an immediate subordinate module that accepts data from the main module, and then return the appropriate data back to the main module.



Step 4: Factoring the Input, Output, and Transform Branches

- For an input module, we look at the previous transform applied to the input to bring it closer to MAI. This now becomes the central transform, and an input module is created for each data stream going into this transform.
- Similar strategy applied to output modules.





Additional Considerations

- Module size
- "Fan-in" and "Fan-out" of modules
  - Fan-in of a module is the number of arrows coming in the module, or the number of its superordinates.
  - Fan-out of a module is the number of arrows going out of that module, or the number of its subordinates.
  - A high fan-in means hat the module is used by many modules either because it includes different functions (bad) or because it contains a common function (good).
  - A high fan-out means that the module coordinates too many modules and may be too complex.

Object-Oriented Design

- Objects: an entity that has a state and a defined set of operations that operate on that state. Objects are created according to an object class definition.
- Object-Oriented Analysis
  - The emphasis is on finding and describing the objects – or concepts – in the problem domain.
- Object-Oriented Design
  - Design the system architecture
  - Identify objects in the system
  - Describe the design using different object models
  - Document the object interfaces

Object-Oriented Design

- Principles
  - Modularization
  - Encapsulation
- Techniques
  - Rational Unified Process (RUP)
  - UML
  - Design patterns
- Methods
  - OMT, OOSE, Booch, etc.

Reference

- Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. Software Architecture: Foundations, Theory, and Practice. John Wiley and Sons. ISBN-10: 0470167742; ISBN-13: 978-0470167748. 2010.
- The materials about function-oriented design is based on Chapter 6 of the book "An Integrated Approach to Software Engineering" by Pankaj Jalote.