What is Software Architecture
Is this diagram an architecture? (ATM Software)
What are ambiguities in the previous diagram?

- Nature of the elements (process, class, object, module, function, processor, or etc)
- Responsibility of elements
- Type of connections (calls, invokes, uses, signals, sends data, controls, sub-class)
- Significance of layout
- Run-time operation of system
What’s Missing?

• What is the nature of the elements?
• What are the responsibilities of the elements?
• What is the significance of the connections?
• What is the significance of the layout?

• Unless we know precisely what the elements are & how they cooperate to accomplish the purpose of the system, this diagram is unhelpful!
Definition

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships between them.

- We see the elements (CP, MODP, MODR, MODN), but what properties do they or their relationships have?
Definition Again

The software architecture of a program or computing system is the **structure** or structures of the system, which comprise software **elements**, the **externally visible properties** of those elements, and the **relationships** among them.
Externally Visible vs. Internal Properties of Component

Externally visible properties are what assumption other elements can make of an element

– Provided services (and interface to access those services)
– Performance
– Fault handling
– Shared resource usage

...

SA intentionally abstracts away internal properties of elements (to better encounter complexity)
Some Points

• Every software system has an architecture (SA ≠ specification of SA)
• Specification of architecture can comprise more than one structure
• Behavior of elements and relationships are defined in SA (abstractly)
• The definition do not talk about Good and Bad architectures: SA evaluation methods
• In the literature:
  – Component = Element
  – Connector = Relationship
What issues are architectural?

Architectural issues are those issues that are important to us at the SA abstraction level.

An issue is architectural if it is not internal to any element. e.g.:

– Performance is an architectural quality attribute
– Behavior of an element is architectural to the extent that influences how other elements must be developed
Architectural Pattern

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.

- example: client-server, layered, data-centered
- unresolved issues of a pattern:
  - Exact number of elements and relations during application
  - Behavior of elements during application
  - Configuration (Topology) during application

Patterns define constraint on architecture but are not architecture themselves. Patterns are abstraction for a set of architectures.
Reference Model

A reference model is a division of functionality together with data flow between pieces.

- A standard division of a known problem (a mature domain) into parts.
- e.g. compiler and DBMS
- Reference model is not architecture
Reference Architecture

A reference architecture is a reference model mapped onto software elements and data flows between the components.

- Elements cooperatively implements the functionality defined in the reference model.
- Reference architecture is not the final architecture but is not far from it.
Relationship of the previous concepts

Reference Model

Architectural Pattern

Reference Architecture

Software Architecture

System
Why is architecture important?

• Handling complexity

• Communication among stakeholders
  – Requirements and concerns of stakeholders
  – Time
  – Budget
  – Other Resources
Why is architecture important? (cont)

- Early Design Decisions
  - Constraints implementation and implementers
  - Organizational structure
  - Enables predicting and ensuring quality attributes
  - Makes it possible to reason about and manage change
  - Helps evolutionary prototyping (risk reduction)
Why is architecture important? (cont)

• SA is a transferable, reusable model
  – Software product lines
  – Component-based development
  – Automatic generation of lower-level models
  – A basis for training

• A run-time model in self-adaptive and reconfigurable systems
Hazards

With regards to SA changes are categorized to

– Local (a single component)
– Non-local (a few components)
– Architectural (architectural style)

• Once decided architecture is extremely hard to change

• It impossible to reach to some quality attribute if architecture disallows
Software Arch. vs. System Arch.

• System Arch. is the overall architecture of system including **hardware** and **software** architecture
• In assuring quality attributes the architect needs to think about system architecture too (e.g. performance or reliability)
• But architect has more freedom in software architecture than hardware (hardware choices is less under the architects control)
Architectural Structures and Views

In construction, there are blueprints of
- Plan
- Different sides of construction
- Electrical wiring
- Plumbing

Each of these views specifies a single entity (i.e. the construction) from a different perspective (used by a different person, for a different goal).

Similarly there are different structures and views in SA.
Structures and Views (cont)

• Structures is a set of coherent elements and the relations among them. For each structure these we can specify:
  – Types of elements
  – Types of relations
  – A set of syntactic constraints
  – Semantics of the diagram
  – Rationale, principles, and guidelines
  – For what purposes it is useful

• View is a representation of software architecture based on an structure as written by the architect and read by stakeholders (an instance of the structure)
• SA is documented by a number of views.
Categorization of Structures

1. Module Structures
2. Component and Connector Structures
3. Allocation Structures

Categories are orthogonal
1 Module Structures

• Elements: modules (units of implementation). Modules are a code based way of considering the system

• Specifies:
  – Functional responsibility of modules
  – Other elements a module is allowed to use
  – Generalization and specialization relations

• Run-time operation of software is not a concern from this view
1.1 Decomposition Structure

- Elements: modules in a hierarchy
- Relations: is a sub-module of, shares secret with

Function Examples:

- Contributes to system's modifiability, by ensuring that likely changes fall within the scope of at most a few small modules.

- Often used as the basis for the development project's organization: the structure of the documentation, and its integration and test plans.
1.2 Uses Structure

- Elements: modules, procedures, or resources on the interfaces of modules
- Relations: uses: one unit uses another if the correctness of the first requires the presence of a correct version (not a stub or of) of the second.

- Function Example:
  - Allows incremental development
1.3 Layered Structure

• Is a subclass of uses structure
• Elements: layers: a coherent set of related functionality
• Relations: uses (ideally layer $n$ may only use the services of layer $n-1$), provides abstraction to

• Function Example:
  – Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.
1.4 Class Structure

- Elements: **classes**
- Relations: **inherits from, is an instance of**

- Function Example:
  - Allows us to reason about **reuse** and the incremental addition of functionality
2 Component and Connector Structures

• Elements: run-time components (principal units of computation) and connectors (communication vehicle among components.)

• Specifies:
  – Major executing components and how they interact
  – Major shared data-stores
  – Which part of system is replicated
  – Flow of data through the system
  – What parts can run in parallel
  – How can system structure change as it executes
2.1 Process Structure

- Elements: processes or threads
- Relations: attachment (that allow communication, synchronization, and/or exclusion operations)

- Function Example:
  - Engineering a system's execution performance and availability.
2.2 Shared Data or Repository Structure

- Elements: data stores, data producers, and data consumers
- Relations: data-flow

- Function Example:
  - To ensure good performance and data integrity.
2.3 Client-Server Structure

• Elements: clients and servers
• Relations: protocols and message passing infrastructure.

• Function Example:
  – Separation of concerns (supporting modifiability)
  – Load balancing (supporting runtime performance)
3 Allocation Structures

• Show the relationship between the software and the elements in one or more external environment in which software is created and executed.

• Specifies:
  – The processor that executes each software element
  – The file that stores each software element during development
  – Assignments of software to development team
3.1 Deployment Structure

- Shows how software is assigned to hardware
- Elements: software (usually a process from a component and connector view), hardware entities, and communication pathways
- Relations: is-allocated-to and migrates-to (for dynamic allocations)

- Function Example:
  - Allows reasoning about performance, data integrity, availability, and security.
3.2 Implementation Structure

• Shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration management environments.

• Elements: any logical unit (e.g. module)

• Relations: implemented in

• Function Example:
  – management of development activities and build process
3.3 Work Assignment Structure

- Assigns responsibility for implementing and integrating the modules to the appropriate development teams
- Elements: any logical unit (e.g. module)
- Relations: is assigned to

- Function Example:
  - The architect will know the expertise required on each team
  - The means for factoring functional commonalities and assigning them to a single team, rather than having them implemented by everyone who needs them.
Categorization of Structures (summary)
Notes

• Each structure is useful on its own right but not all structures are used in all projects.

• Structures are not independent and must be considered together
  – e.g. relationship of modules with components (many to many)
  – Some structures may be the same in some systems
  – Some structures may be combined (e.g. all component and connector structures may be combined in a single structure)
Relating Structures to Each Other

- Although the structures give different system perspectives, they are not independent.
- Elements of one structure are related to elements in another, and we need to reason about these relationships.
  - For example, a module in a decomposition structure may map to one, part of one, or several, components in a component-and-connector structure at runtime.
- In general, mappings are many-many.
4 + 1 View Model of Architecture

- **Logical**: objects and classes, elements are “key abstractions” that are objects or classes in OO. (a module view)
- **Process**: addresses concurrency & distribution of functionality. (a component and connector view)
- **Development**: shows organization of software modules, libraries, subsystems, and units of development. (an allocation view)
- **Physical**: maps other elements onto processing & communication nodes, also an allocation view, but usually referred to specifically as the deployment view. (an allocation view)
- **Scenarios (Use-cases)** view is not itself architectural.