What you've seen so far …

- Software architecture concepts
- Quality attributes and scenarios
- Styles, patterns and tactics
- The ADD method

… and you have an assignment to complete (due this week)
Agenda

- Recap of software architecture and quality attributes
- What information should be captured
  - e.g., in ADD
- Main types of architectural views
- Guidelines and notations
- Structure, behavior and interfaces

For more information
Why architectures are important?

- Plan before building
- A manageable model of a complex system
- Communication among stakeholders
- Prescriptions for the implementation

Requirements and implementation

The gap between requirements and implementation is usually too big
**The role of software architecture**

Key activities in the software architecture lifecycle

- **BUSINESS GOALS**
  - derive
- **REQUIREMENTS**
  - design
  - satisfy
  - adjust
- **SOFTWARE ARCHITECTURE**
  - conform
- **SYSTEM (CODE)**
  - implement

**The definition (again)**

“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”

*Software Architecture in Practice (Bass, Clements and Kazman, 1998)*

A given architecture is essentially determined by its quality attributes (rather than by its functional requirements)
Documenting is a concrete task!

- It produces a **software architecture document**

Why documenting architectures?
- It is the primary carrier of quality attributes
- It is the best artifact to support early analysis
- It defines the work assignments for the project
- It helps post-deployment maintenance

The architecture document is both
- **Descriptive**
- **Prescriptive**

Documentation contents

- Features and functional requirements
- Quality-attribute requirements (drivers)
- Constraints
- Architecturally-significant decisions
  - Architectural structures
    - captured by means of **architectural views**
  - Architecture behavior
  - Interfaces
  - Traceability
Architecture = design decisions?

- Architecture is design, but not all design is architecture

  “Do not dilute the meaning of the term architecture by applying it to everything”

Decisions
“Design” decisions
Architecturally-significant decisions
“Requirements and constraints”

Architecturally-significant decisions

- Those decisions (with respect to structure and behavior) that have a great impact on the system, its quality attributes, or are difficult/costly to change in the future

Decisions
“Design” decisions
Architecturally-significant decisions
“Requirements and constraints”
Multiple structures

- An architecture is a multi-dimensional entity, which is usually complex to be seen/comprehended all at once

- Systems are composed of many structures
  - Static aspects
  - Dynamic aspects
  - Correspondence between static and dynamic aspects

Architectural structures

- modules, showing composition/decomposition and mapping to code units
- processes and how they synchronize
- programs, and how they call or send data to each other
- how software is deployed on hardware
- how teams cooperate to build the system
- how components and connectors work at runtime
- many others
**Views (versus structures)**

A view is a **representation** of a set of system elements and the relations associated with them.

- **In more detail**
  - A **structure** is the set of elements/relations itself, as they exist in software or hardware.
  - A **view** is a representation of a coherent set of elements, as written by and read by **stakeholders**.
    - A view can be thought as capturing a structure

**View-based documentation**

- **SEI’s Views and Beyond approach**: Views give us our basic principle of architecture documentation:

  *Documenting a software architecture is a matter of documenting its **relevant views** and then adding information that applies to more than one view.*
What views are available?

- Three basic types:
  1. **module views** *(module viewtype)*
     How is it structured as a set of code units?
  2. **component-and-connector views** *(C&C viewtype)*
     How is it structured as a set of elements that have runtime behavior and interactions?
  3. **allocation views** *(allocation viewtype)*
     How does it relate to non-software structures in its environment?

Main viewtypes

1. Views in the **module viewtype** show elements that are units of implementation
2. Views in the **component-and-connector (C&C) viewtype** show elements that have runtime behavior and interaction
   (There is often, but not always, a direct correspondence between module and C&C views)
3. Views in the **allocation viewtype** show how software structures are allocated to non-software structures
Modules, C&C and allocation

- Suggested relationships between views

The view zoo

- **Module viewtype**
  - Decomposition style
  - Uses style
  - Generalization style
  - Layered style

- **Component-and-connector viewtype**
  - Pipe-and-Filter style
  - Shared-Data style
  - ...

- **Allocation viewtype**
  - Deployment style
  - Implementation style
  - Work assignment style
The view zoo

- **Module viewtype**
  - Decomposition style
  - Uses style
  - Generalization style
  - Layered style
- **Component-and-connector viewtype**
  - Pipe-and-Filter style
  - Shared-Data style
  - …
- **Allocation viewtype**
  - Deployment style
  - Implementation style
  - Work assignment style

The module viewtype

- **Elements:** modules
  - A module is a code unit that implements a set of responsibilities
- **Relations:** relations among modules include
  - **A is part of B.** This defines a part-whole relation among modules
  - **A depends on B.** This defines a dependency relation among modules
  - **A is a B.** This defines specialization and generalization relations among modules
- **Properties:** Properties of modules include a *name*, *responsibilities*, and the *visibility* of the module and its interface. Other properties are style-specific
What are modules used for?

- **Construction**
  - Modules are the blueprints for the code
  - Modules are assigned to teams for implementation and are often the basis for subsequent design (e.g., of interfaces)
  - Subsets and deployment packages are built using module styles

- **Analysis**
  - Traceability and impact analysis rely on implementation units.
  - Project management, budgeting, planning, and tracking often use modules

- **Education**
  - A software developer can learn the development project's structure by understanding module views

Notation for module styles

**Informal:** box-and-line; nesting can represent “is part of” relations

**Semi-formal:** UML, class diagrams
Example: UML Components

- **FLABot tool**
- **Clarification:**
  - UML component = module
  - FLABot includes a stereotyped dependency relation

![UML Component Diagram]

Styles in the module viewtype

- **Decomposition style:** Documents how system responsibilities are partitioned across modules and how those modules are decomposed into sub-modules

- **Uses style:** Tells the developer what other modules must exist for this portion of the system to work correctly

- **Generalization style:** Documents the “is a” relations among elements of the system

- **Layered style:** Documents the “allowed to use” relations among elements of the system
Decomposition style

- **Elements:** modules
- **Relations:** *Is part of*
  - Criteria for decomposition vary:
    - achievement of modifiability
    - build vs. buy
- **Topology:** A child can have only one parent

What it’s for:
- starting point for assigning responsibilities to modules as a prelude to detailed design
- change/impact analysis
- basis for work assignments
- basis for unit testing

Decomposition style: Notations

- **Semi-formal:** UML

- **Informal:**
  - box and line, nested boxes
  - textual outline
  - table
Uses Style

- **Elements:** modules
- **Relations:** uses, a refined form of “depends on” relation
  - A uses B if A depends on the presence of a correctly functioning B to satisfy its (A’s) own requirements.
- **Topology:** no constraints
  - (However, loops can cause problems with incremental system delivery)
- **What it’s for:**
  - planning incremental development
  - system extensions and subsets
  - debugging and testing
  - gauging the effects of specific changes

Uses Style: UML Example

“User Interface” and “Administrative System” each use a different interface of the Database package
Generalization style

- **Elements:** modules
- **Relations:** generalization, an “is a” relation
- **Properties:** abstract?
  - (i.e., is this an interface without an implementation?)
- **Topology:**
  - A module can have multiple parents
  - Cycles are prohibited

- **What it’s for:**
  - basis for object-oriented designs
  - incremental description for evolution and extension
  - capturing commonalities, with variations as children
  - support for reuse

Generalization Style: Notations

- Formal, semi-formal:
  - A module can have multiple parents
  - UML
Generalization style: Examples

(a) A inherits the implementation of B and realizes the same interface as B

(b) Interface inheritance: A realizes the same interface as B

(c) A inherits the implementation of B but realizes its own interface

Layered style - 1

- **Elements**: layers, a virtual machine
- **Relations**: “allowed to use,” a specialization of the “depends on” relation
  - Recall that A uses B if A’s correctness depends on the presence of a correct B
  - The relation is not necessarily transitive
- **Topology**:
  - Every piece of software is assigned to exactly one layer.
  - Software in a layer is allowed to use software in {any lower layer, next lower layer}
  - Software in a layer (is, is not) allowed to use other software within the same layer
Layered style - 2

What it’s for:
- portability
- fielding of subsets, incremental development
- separation of concerns
- promotes reuse

Variations:
- segmented layers: dividing a layer into segments (or sub-modules), with “allowed to use” relations between those segments and segments from other layers

Layers: Informal notations - 1

What do these diagrams mean?

<table>
<thead>
<tr>
<th>A</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>C</th>
</tr>
</thead>
</table>
Layers: Informal notations - 2

Layers in UML: Packages
Layered style: Example

<table>
<thead>
<tr>
<th>KEY</th>
<th>Behavior-hiding module</th>
<th>Software decision-hiding module</th>
<th>Hardware-hiding module</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Function Driver</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Shared Services</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Banker</td>
<td>Physical Models</td>
<td>Filter Behaviors</td>
<td></td>
</tr>
<tr>
<td>Device Interfaces</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Application Data Types</td>
<td>Software Utilities</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended Computer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The view zoo

- **Module viewtype**
  - Decomposition style
  - Uses style
  - Generalization style
  - Layered style
- **Component-and-connector viewtype**
  - Pipe-and-Filter style
  - Shared-Data style
  - ...
- **Allocation viewtype**
  - Deployment style
  - Implementation style
  - Work assignment style
Component-and-Connector (C&C) viewtype - 1

- **Elements:**
  - components: principal units of runtime interaction and data stores
  - connectors: interaction mechanisms

- **Relations:**
  - attachment of components’ ports to connectors’ roles (interfaces with protocols)

- **Properties:**
  - name
  - quality attributes (to help analysis)
  - others, depending on style

What are C&C used for?

- **Construction**
  - Specifying the behavior that elements must exhibit

- **Education and analysis**
  - A starting point for the architect to show how the system works
  - To reason about runtime system quality attributes such as performance, security, and reliability
  - To answer questions such as:
    - What are the major executing components and how do they interact?
    - What are the major shared-data stores?
    - Which parts of the system are replicated?
    - How does data progress through the system?
    - Which parts of the system can run in parallel?
    - How can the system’s structure change as it executes?
The Pipe-and-Filter style

- **Elements:**
  - component type: filter, which transforms data
  - connector type: pipe, a unidirectional data conduit that preserves the order and value of data

- **Relations:** attachment of pipes to filters

- **Topology:** Pipes connect filters. Further specializations of the style may prohibit loops or branching.

- **What it’s for:**
  - good choice for systems in which data is transformed serially
  - supports functional composition data analysis

Pipe-and-Filter style: Example
Shared-Data style

- **Elements:**
  - component types: data stores and accessors
  - connector types: data reading and writing

- **Relations:** attachment

- **Properties:** can include type of data, data-related performance properties, and data distribution

- **Topology:** The data store is attached to the data accessors via connectors

- **What it’s for:** when there are multiple accessors to persistent data items

- **Variations:** Data store is a blackboard that announces updates to interested subscribers

---

**Shared-Data style: Example**
Other C&C styles

- Client-server
- Peer-to-peer
- Publish-subscribe
- Communicating-processes

Each is a specialization of the C&C viewtype, restricting element types and topologies

Each may be further specialized.
- For example, Shaw and Clements identify five sub-styles of the communicating-processes style and three sub-styles of the client-server style

Mixed C&C styles

Which styles do you see in this diagram?

- This is the norm for non-trivial systems. It is less common to see individual C&C views created separately
Special considerations for C&C

- **Components:**
  - Often hierarchical
    - Represents a subsystem with a finer-grained C&C substructure
    - Might not be in the same style
  - Interfaces represent points of interaction.
    - Might have several interfaces with the same “signature”

- **Connectors:**
  - Might represent complex forms of interaction
    - Often not directly supported by programming language
    - Require complex runtime support

C&C versus module views

- The relationship between C&C elements and module elements is often not one to one
Notations for C&C styles

- Informal:
  - box-and-line diagrams
  - Most box-and-line diagrams showing runtime behavior are in fact attempting to be C&C views.

- Formal:
  - architecture description languages such as Acme, Wright, UniCon, MetaH, and Rapide

- Semi-formal:
  - UML
    - not a straightforward mapping
    - three major strategies are available

The view zoo

- Module viewtype
  - Decomposition style
  - Uses style
  - Generalization style
  - Layered style

- Component-and-connector viewtype
  - Pipe-and-Filter style
  - Shared-Data style
  - ...

- Allocation viewtype
  - Deployment style
  - Implementation style
  - Work assignment style
Allocation viewtype

- **Elements:**
  - software elements (as defined in module or C&C styles)
  - environment elements

- **Relations:** “allocated to”

- **Properties:** various, according to style

Allocation viewtype: Software elements and environment elements

[Diagram showing computing platform, development environment, development organization, and module and/or C&C styles of software architectures]
Styles of the allocation viewtype

- **Deployment style:** Allocates software elements to processing and communication nodes
  - Properties include those necessary to calculate (and achieve) performance and availability.

- **Implementation style:** Allocates software elements to structures in the development environment’s file systems
  - Properties include files and capacities.

- **Work assignment style:** Allocates software elements to organizational work units
  - Properties include skill sets

---

**Deployment style - 1**

- **Elements:**
  - software element - usually processes from the C&C viewtype
  - environmental element - computing hardware

- **Relations:**
  - “allocated to” - physical elements on which software resides
  - “migrates to,” “copy migrates to,” and/or “execution migrates to” with dynamic allocation
Deployment style - 2

- **Properties:**
  - required property for software element
  - significant features required from hardware
  - provided property of environment elements
  - significant features provided by hardware

- **Topology:** unrestricted

- **What it's for:** analysis of
  - performance
  - bandwidth utilization
  - availability
  - security
  - purchasing options for hardware

Deployment style: Example
Implementation style - 1

- **Elements:**
  - software element - a module
  - environmental element - configuration item; for example, a file or directory

- **Relations:**
  - containment - One configuration item is contained in another.
  - “allocated to”—allocation of a module to a configuration item

Implementation style - 2

- **Properties:**
  - required property for software element—usually requirements on the development environments; for example, Java
  - provided property of environment elements—characteristics provided by development environments

- **Topology:** hierarchical configuration items, “is contained in”

- **What it’s for:**
  - managing and maintaining files that correspond to software elements
  - analyzing purchasing options for development environments
**Implementation style: Example**

```
A
make

src A.cc
include A.h
doc A.doc
config A.conf
```

**KEY:**
- `A` Folder with name of module
- `A.conf` File
- `containment`

---

**Work assignment style - 1**

- **Elements:**
  - software element - a module
  - environmental element - organizational unit; for example, a person, team, or subcontractor
- **Relations:** “allocated to”—software element allocated to a development team
- **Properties:**
  - required property for software element - skills set
  - provided property of environment elements - skills set
Work assignment style - 2

- **Topology:** unrestricted in general;
  - however, in practice, usually restricted such that one module is *allocated to* one organizational unit

- **What it’s for**
  - shows major units of software necessary for a system and who will produce (i.e., develop, test, and integrate) those units
  - used for planning/managing team resource allocations
  - used to explain the structure of a project
  - can be used to assign responsibilities for builds
  - early on, can be used to help write RFPs

---

**Work assignment style: Example**

<table>
<thead>
<tr>
<th>ECS Element (Module)</th>
<th>Organizational unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Segment</strong></td>
<td><strong>Subsystem</strong></td>
</tr>
<tr>
<td>Science Data</td>
<td>Client</td>
</tr>
<tr>
<td>Processing Segment</td>
<td></td>
</tr>
<tr>
<td>(SDPS)</td>
<td>Interoperability</td>
</tr>
<tr>
<td></td>
<td>Ingest</td>
</tr>
<tr>
<td></td>
<td>Data Management</td>
</tr>
<tr>
<td></td>
<td>Data Processing</td>
</tr>
<tr>
<td></td>
<td>Data Server</td>
</tr>
<tr>
<td></td>
<td>Planning</td>
</tr>
<tr>
<td>Flight Operations</td>
<td>Planning and Scheduling</td>
</tr>
<tr>
<td>Segment (FOS)</td>
<td>Data Management</td>
</tr>
<tr>
<td></td>
<td>User Interface</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

---
Documenting an architectural view

- The primary presentation is generally graphical, it shows the elements and their relations
- The element catalog explains each element (and relations) in the primary presentation
- Also, it must capture design rationale and references to related views

SA Document

- The System Overview section describes the context of the system and its purpose
- The Requirements section lists the main functional and quality-attribut requirements, as well as constraints
- The Mappings between Views section shows how elements in one view are related to elements in other view(s)
Behavior: Beyond structure

- **Structural diagrams**
  - show all potential interactions among software elements

- **Behavioral diagrams**
  - describe specific patterns of interaction
    - the system’s response to stimuli

Why behavioral documentation?

How well can this system do what we want it to?

**Structural diagrams support static analysis**
- Is there a source for this needed input?
- Is there a component that does this?

**Behavioral diagrams support dynamic analysis**
- throughput
- deadlock
- response time
Uses of behavioral documentation: Communication among stakeholders

- Describe the planned behavior of the system, especially to users and developers
  - For example, “This is the way I expect the system to respond.”
- Discuss alternatives to that behavior
  - For example, “If it worked like this, would we get better throughput?”
- Give developers their mission statement for the software they’ve been assigned to produce

Uses of behavioral documentation: Analysis

- System-wide analysis looks at properties for all potential uses
  - For example, “Can the system ever deadlock?”
- Simulation-based analysis looks at special cases;
  - For example, “What happens if this input is received?”
- Behavioral documentation lets you test the architecture.
  - For example, simulating scenarios
    - supports analysis of the architecture’s tradeoffs
    - checks the architecture’s potential to support quality requirements
Where should behavior be documented?

- At least three places are available:
  - In a view template: we document behavior in the element catalog to help describe an element.
  - In an interface specification: we use behavior to
    - explain a resource's semantics
    - provide a usage guide, to show the element's reaction to the use of its resources
  - Where we capture the design rationale: we use behavior to
    - put the results of the analysis in context
    - show the satisfaction of requirements

Two classes of languages

- **Trace-oriented languages**
  - describe how the system reacts when a specific stimulus arrives and the system is in a specific state
  - are easy to use because of their narrow focus
  - offer an incomplete view of the system
  - allow the complete capture of behavior through the collection of all possible traces
  - usually include only parts of elements

- **Static languages**
  - show the complete behavioral structure of a system
  - are usually state based (e.g., statecharts)
  - can infer all traces using a static model - even impossible traces
  - support documentation of alternatives and variable repetition
**Which is which?**

- **Trace-Oriented**
  - use cases
  - collaboration diagrams
  - sequence diagrams
  - message sequence charts
  - activity diagrams
  - ...

- **Static**
  - Statecharts
  - SDL diagrams
  - Z specifications
  - some ADLs
  - ...

**Example: Use-case Maps**

- Use case maps (UCM) is an intuitive notation that helps communicate how a system works or is supposed to work, without getting lost in much details

- UCM concentrate on visualizing execution paths through a set of elements from a bird's-eye view

- FLABot tool

"causal paths cutting across organizational structures"
Documenting an interface

An interface specification is a statement of element properties that its architect chooses to make known.

Element Interface Specification

- Section 1. Interface identity
- Section 2. Resources provided
  - Section a. Resource syntax
  - Section b. Resource semantics
  - Section c. Resource usage restrictions
- Section 3. Locally defined data types
- Section 4. Error handling
- Section 5. Variability provided
- Section 6. Quality attribute characteristics
- Section 7. What the element requires
- Section 8. Rationale and design issues
- Section 9. Usage guide

Notation for showing the existence of interfaces - 1

- Elements can be adorned with circles or "lollipops" showing one or more interfaces.

- An interface shown by itself emphasizes that there are many elements that can realize it.
Stakeholders’ needs

- Different stakeholders use/consume/are interested in different perspectives of the software architecture
- The documentation should be customized to satisfy the stakeholders

Take-Aways

- **Fundamental principle:** Documenting a software architecture is a matter of documenting the relevant views and then adding information that applies to more than one view

- Always use a notation key!

- 3 viewtypes that categorize architectural information
- A catalog of styles, with notational approaches for each style
- Templates for views, documentation beyond views, and software interfaces
What versus when: Process

- In this presentation, we have concentrated on what:
  - what artifacts should be produced
  - what information they should contain
  - what uses can be made of them

- We have not concentrated on when because that tends to be process-specific
  - Nevertheless, you cannot produce an architecture document unless you can answer the following question:

  When is the architecture documentation produced?

Thank you!

Andrés Díaz Pace
adiaz@exa.unicen.edu.ar
o cualquier miembro de la cátedra Diseño
disenio@exa.unicen.edu.ar