UNIVERSITY OF CAPE TOWN
Department of Computer Science

# Advanced Software Design
# UML and Architecture

Gary Stewart
gstewart@cs.uct.ac.za
slides taken from prof. edwin blake

# Now we move on to Design in the Large

- Architecture
- Architectural Patterns
- Sources:
  - Bennett, McRobb & Farmer, *O-O Systems Analysis & Design*, Chapter 13 “System Design and Architecture”
  - Larman, *Applying UML and Patterns*, Chapter 13 “Logical Architectures and UML Package Diagrams” (+ Ch 17 & Ch 39).

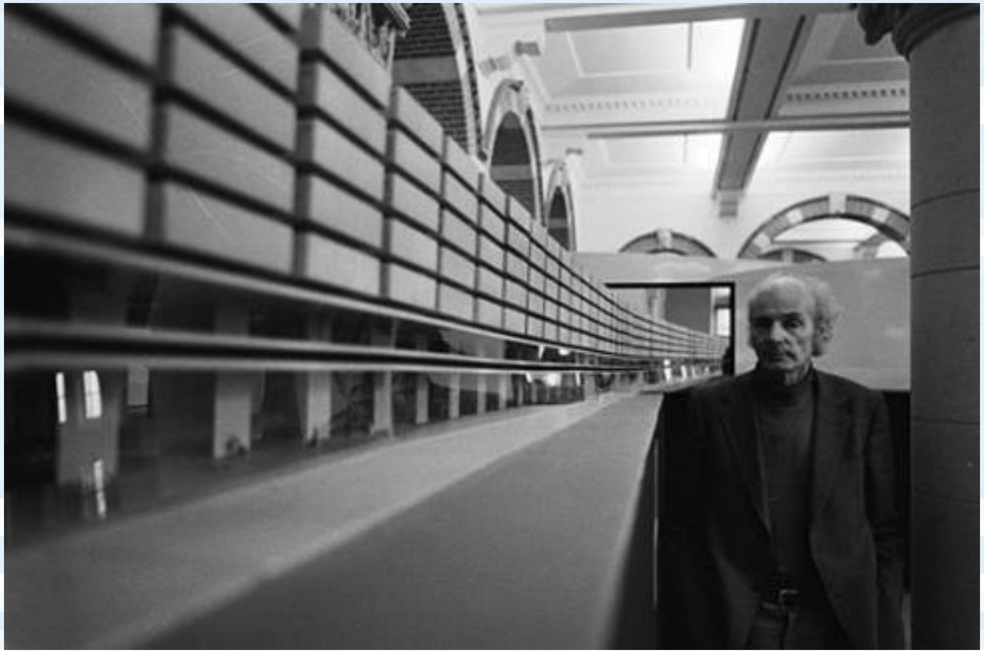
    005. 117 LARM
  - Sommerville, *Software Engineering*, Chapter 6 “Architectural Design”

    005.1 SOMM

# Think Big, Act Small, Fail Fast, Learn Rapidly

- Slogan from Lean Software Development (another Agile method)
  - has the principle of “See the whole” (amongst others)
  - ≠ do the whole design early!
- The remainder of the course is about seeing the whole when building a single system.
- This is architectural thinking
  - It is the translation from the **problem domain** to the **solution concepts**
    - Technology with purpose

# ARCHITECTURE DEFINITION



**Architecture Definition**

UML Views

UML Packages

Architecture Description

Guidelines

# Architecture

**software architecture**

- “the set of principal design decisions about the system”
- the heart software system

Well-engineered software

⇔ good software architecture

⇔ good set of design decisions

**NB**: This is very different from the other meaning of Architecture in CS: hardware and the associated abstractions.

Zeitz Museum of Contemporary Art Africa (Cape Town Waterfront)

# Definition: Architecture

- Basic idea: It is about the **BIG** picture; the large scale:
  - motivations,
  - constraints,
  - organization,
  - patterns,
  - responsibilities,
  - connections of a system (or a system of systems)

# Definition: Architecture I

"An architecture is the set of significant decisions about the *organization of a software system*, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements, the *composition* of these structural and behavioural elements *into progressively larger subsystems*, and the architectural style that guides this organization—these elements and their interfaces, their collaborations, and their composition."

Booch, Rumbaugh, and Jacobson, The UML User Guide, 1999

# Definition: Architecture II

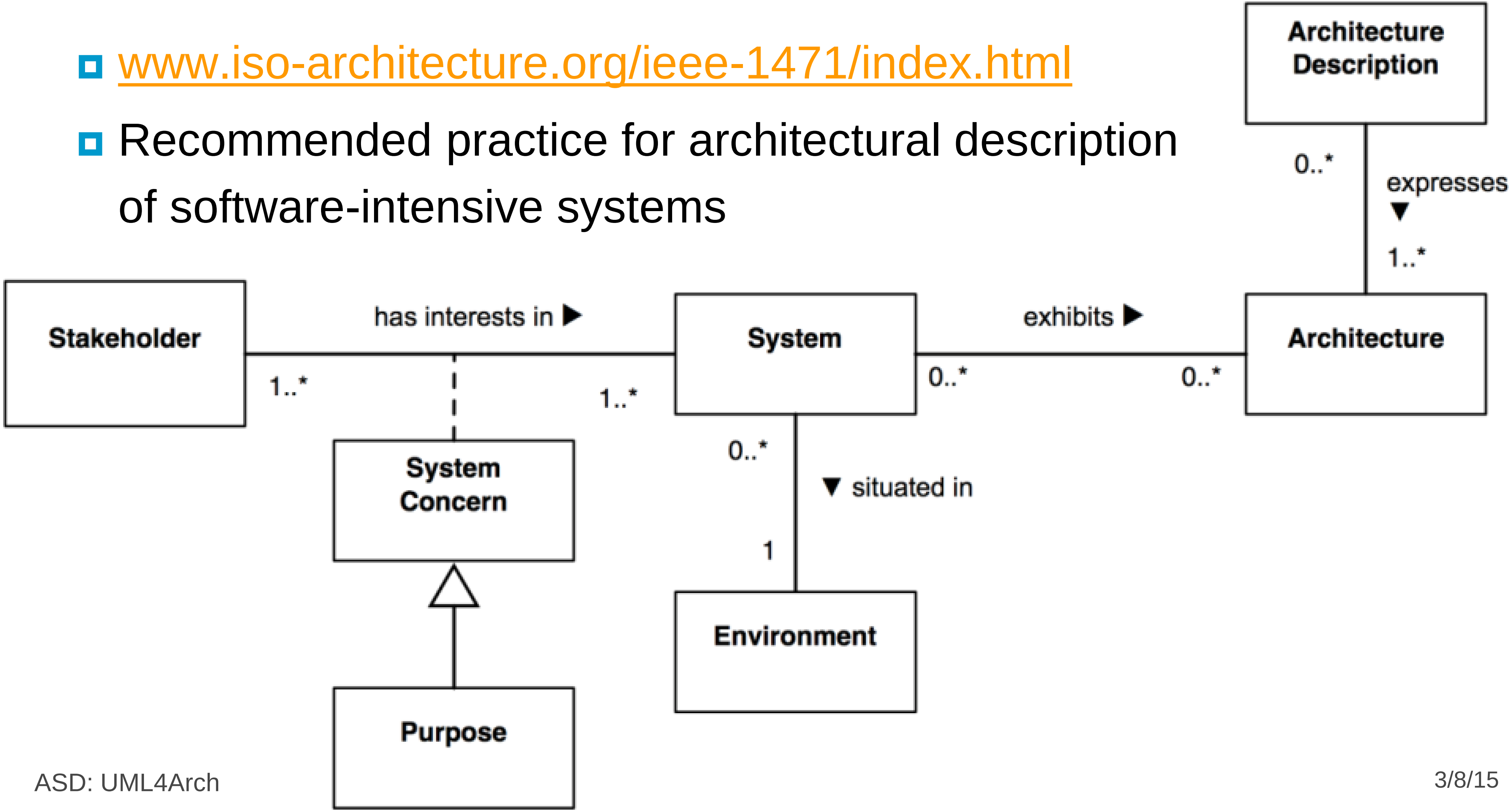
The software architecture of a program or computing system is the structure or structures of the system, which comprise software *components*, the *externally visible* properties of those components, and the *relationships* among them. Architecture is concerned with the public side of interfaces; private details of elements—details having to do solely with internal implementation—are not architectural

By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.

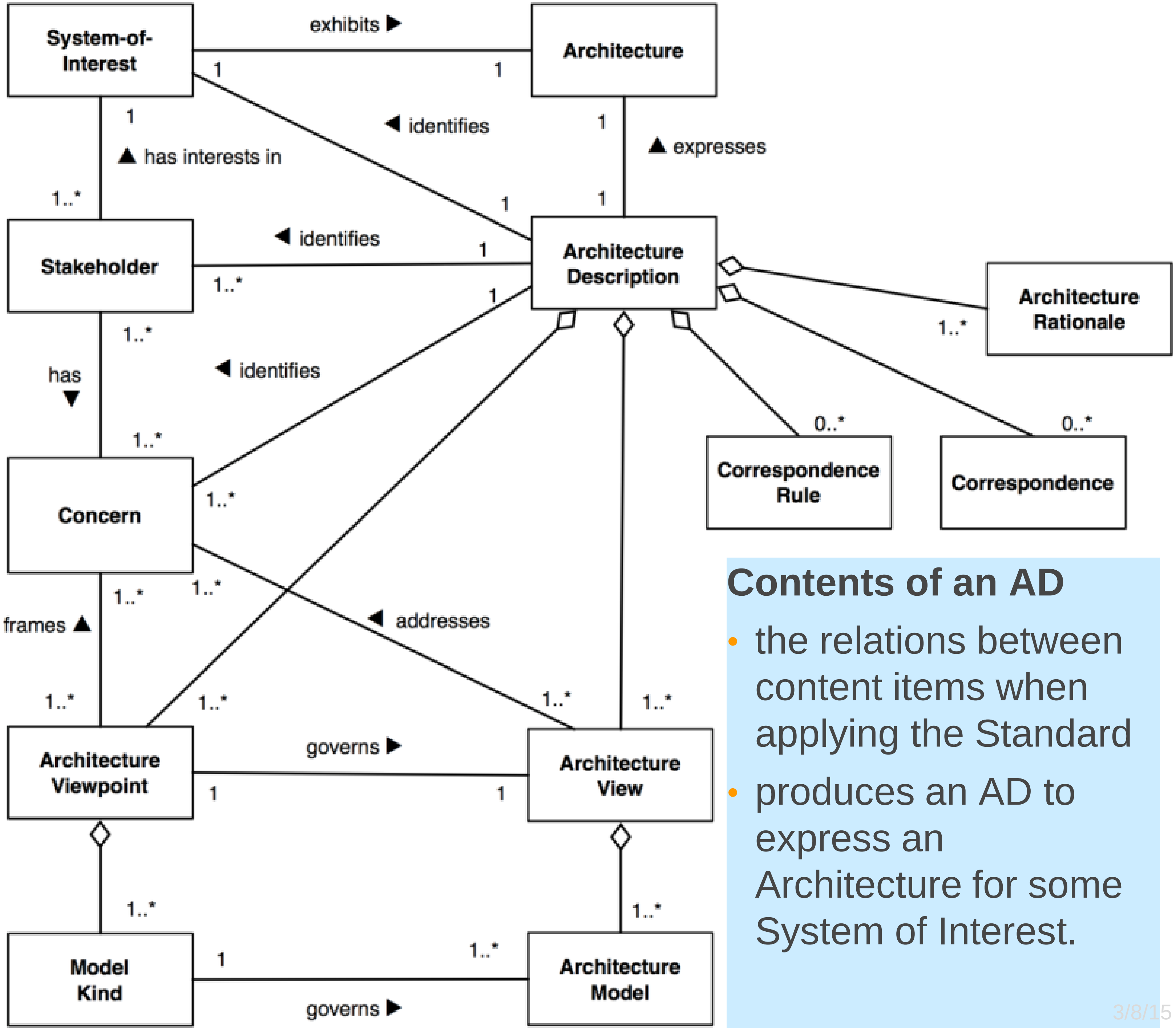
Bass, Clements, and Kazman. Software Architecture in Practice, 2003

# ISO/IEC/IEEE 42010:2011: Architecture Description

- International Standard: ISO/IEC/IEEE 42010:2011
  - [www.iso-architecture.org/ieee-1471/index.html](www.iso-architecture.org/ieee-1471/index.html)
  - Recommended practice for architectural description of software-intensive systems



# Core of Architecture Description



## Contents of an AD

- the relations between content items when applying the Standard
- produces an AD to express an Architecture for some System of Interest.

# ISO/IEC/IEEE 42010 Definition: Architecture

The fundamental concepts or properties of a system in its environment embodied in its **elements**, **relationships**, and in the **principles** of its design and evolution.

- In the standard an architecture is abstract — not an artefact.
  - **Architecture description**: artefacts to express & document architectures
- What is **fundamental** to a system may take several forms:
  - **elements**: the constituents that make up the system;
  - **relationships**: both internal and external to the system; and
  - **principles of its design and evolution**.
- Different architecture communities place varying emphases:
  - Software architecture: focused on software components as elements and their interconnections as a key relationship.
  - System architecture emphasizes sub-system structures and relationships such as allocation.
  - Enterprise architecture emphasizes principles.

# UML VIEWS

Architecture Definition

**UML Views**

UML Packages

Architecture Description

Guidelines

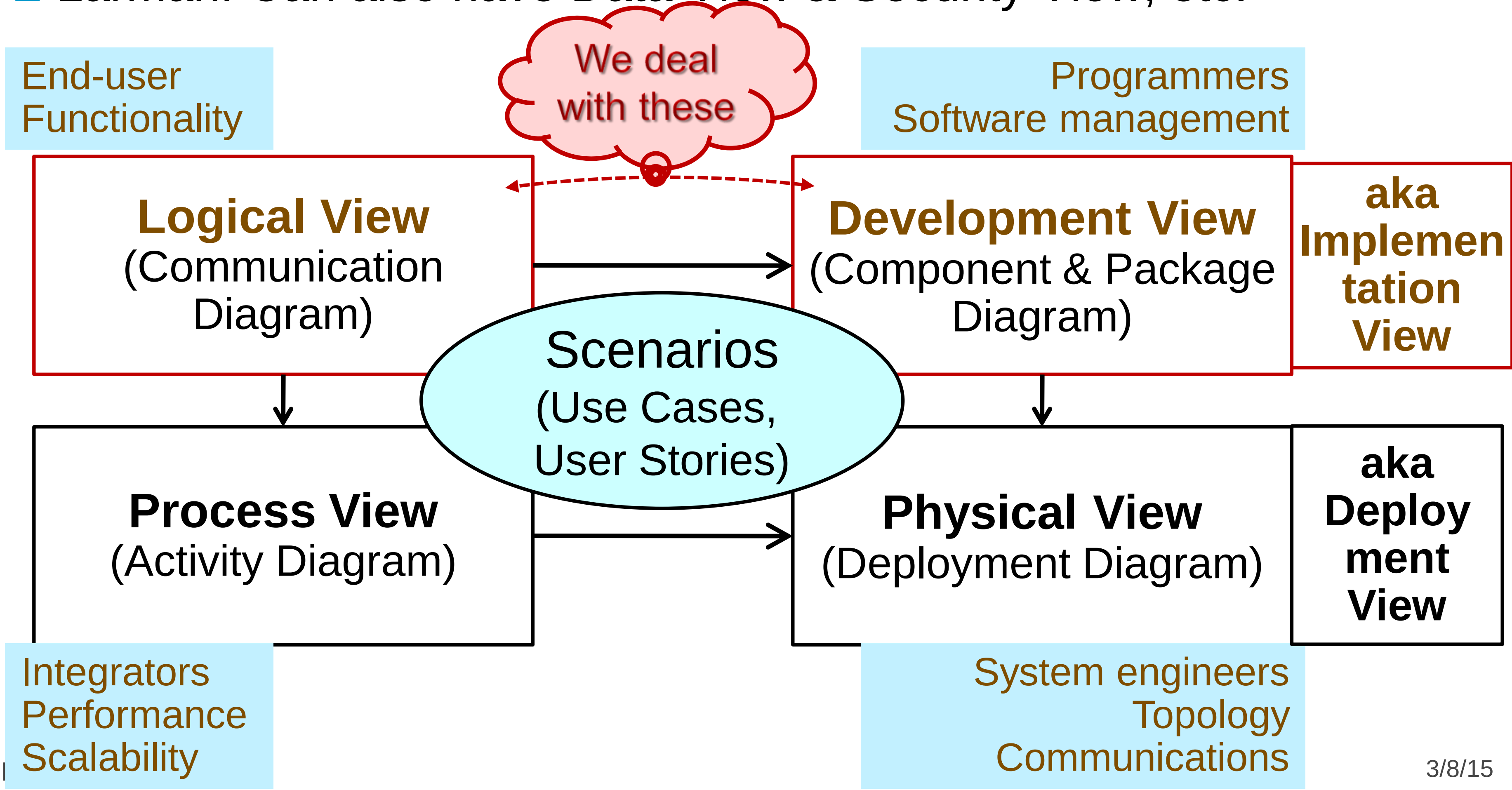
# Why so Many Views and Diagrams?

- Because so many different stakeholders are interested in the overall design ≡ Architecture
- Viewpoints of the different stakeholders may lead to different views of the same system:
  - These views have to be communicated and represented and then integrated.
  - Together they form the complete architectural description of the software system being designed.
- Architectural representation has two objectives:
  - to be able to accommodate different views based on the requirements;
  - integrating of these different views to form the complete architectural representation.

# RUP "4+1" View of Architectures

"4+1" view model of Kruchten together with UML diagrams to use:

- Larman: Can also have Data View & Security View, etc.



We deal with these

End-user Functionality

**Logical View** (Communication Diagram)

Programmers Software management

**Development View** (Component & Package Diagram)

**aka Implementation View**

Scenarios (Use Cases, User Stories)

**Process View** (Activity Diagram)

**Physical View** (Deployment Diagram)

**aka Deployment View**

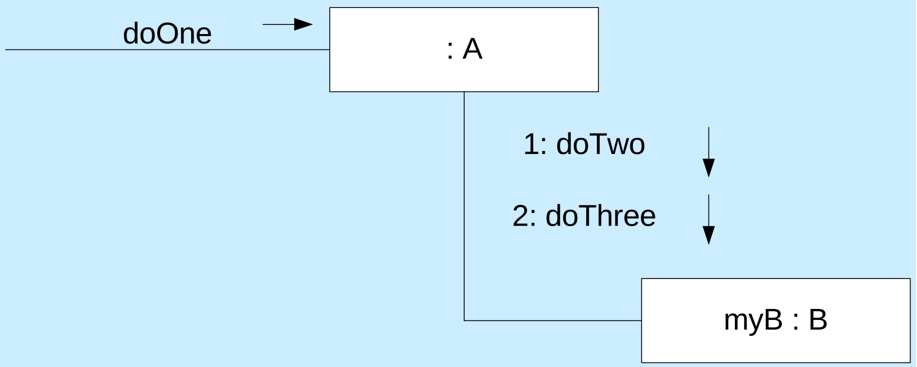
Integrators Performance Scalability

System engineers Topology Communications

# Logical View (Module View)

- The logical view is concerned about the output(s) of the system and how it will affect the end users.
- The logical view splits the system into a set of abstractions, or modules.
- This decomposition serves two purposes:
  - it enables functional analysis
  - it helps in identification of common mechanisms and design elements that are common across the system.

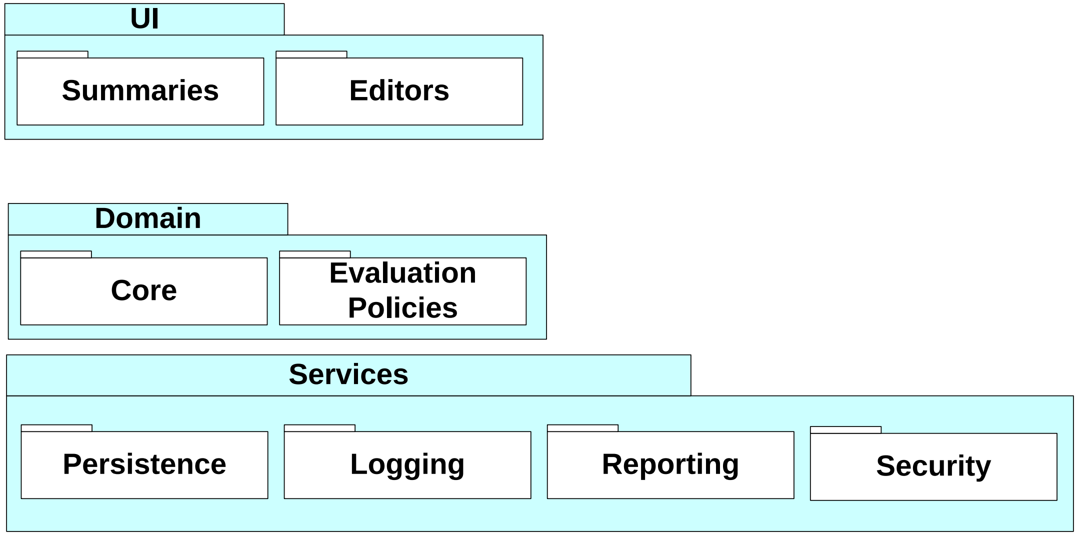
⇨ Communication Diagrams



# Development View (Allocation View)

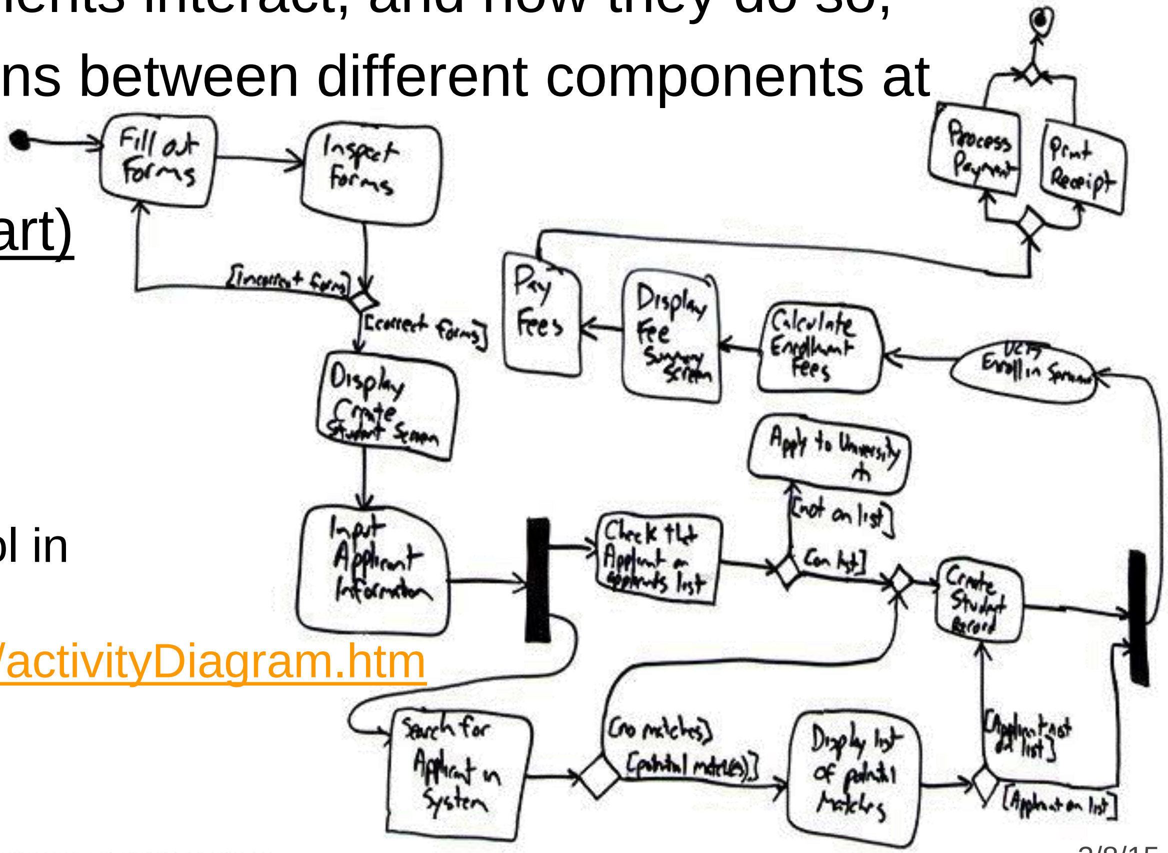
- This view describes the static organization of the software in its development environment.
- It deals with modules, work allocation, costs and planning. It also involves monitoring of project progress, software reuse, and security.

⇨ Component & Package Diagram



# Process View (Component-and-Connector View)

- □ This view deals with concurrency and distribution, system integrity, and fault tolerance.
- □ It explains which components interact, and how they do so;
- □ & the dynamic connections between different components at runtime.
- ⇨ Activity Diagram (flowchart)

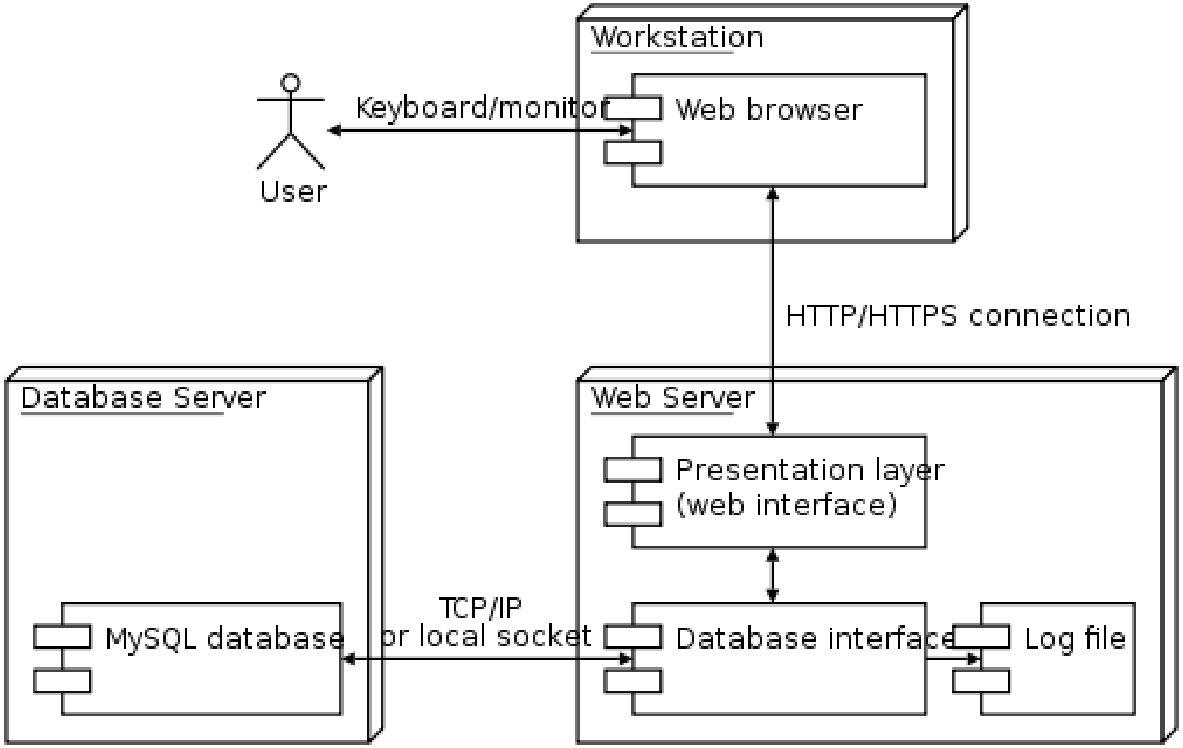


UML activity diagram for the Enrol in University use case. See: www.agilemodeling.com/artifacts/activityDiagram.htm

Copyright 2005 Scott W. Ambler

# Physical View (Deployment View)

- This view describes how the software maps onto the hardware
- It shows networking and distribution.
- It considers system requirements like reliability and performance.
- Deals with the elements identified in the previous three views.
- ⇨ Deployment Diagram



# UML PACKAGES

Architecture Definition

UML Views

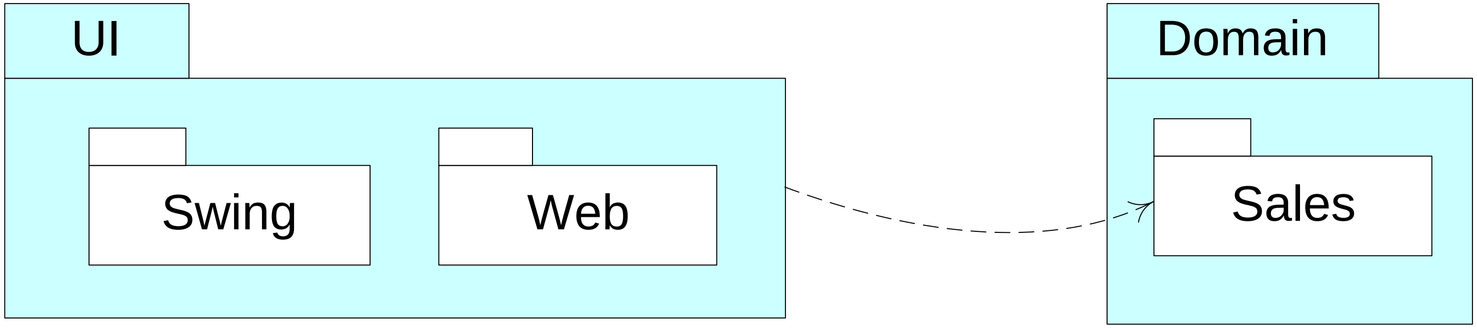
UML Packages

Architecture Description

Guidelines

# UML Packages

- Packages group elements
  - For example, groups of classes in a single namespace
- Drawn as a rectangle with a smaller tab at the upper left
  - If members are shown within the package, name the tab
- Used to show the high level organization of a project
- A dashed arrow between packages indicates a dependency



# Logical Architecture

- Shows large-scale organization of software classes, grouped by
  - Layers (coarse-grained)
  - Packages
  - Subsystems (finer-grained)

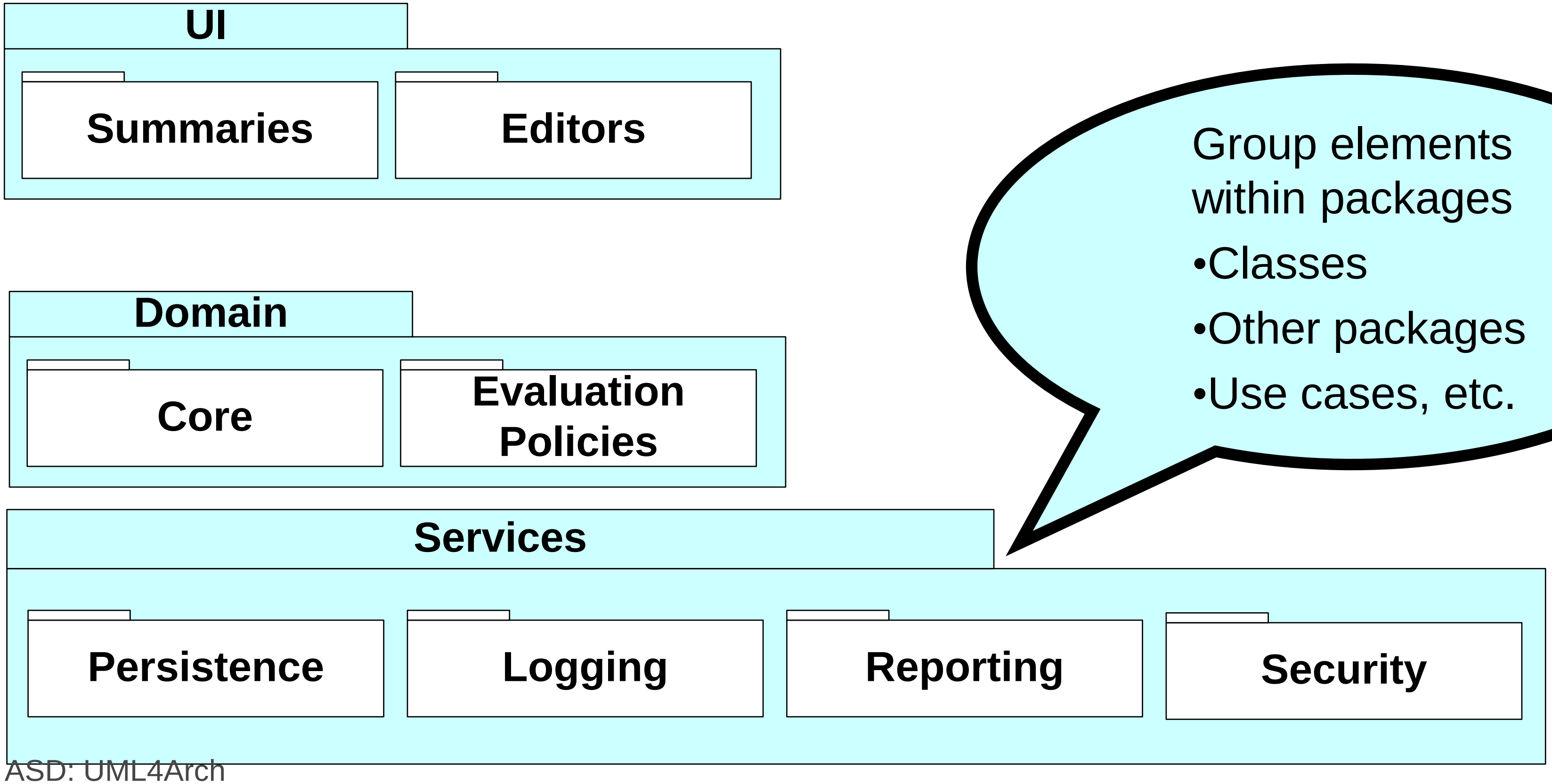
  Cohesive responsibility for a major aspect of the system.

“Logical” ⇒ independent of actual deployment decisions

*See notes provided on Vula:*
*“Larman- Extracts from Chapter 13.pdf”*

# Package Diagrams for Logical Architecture

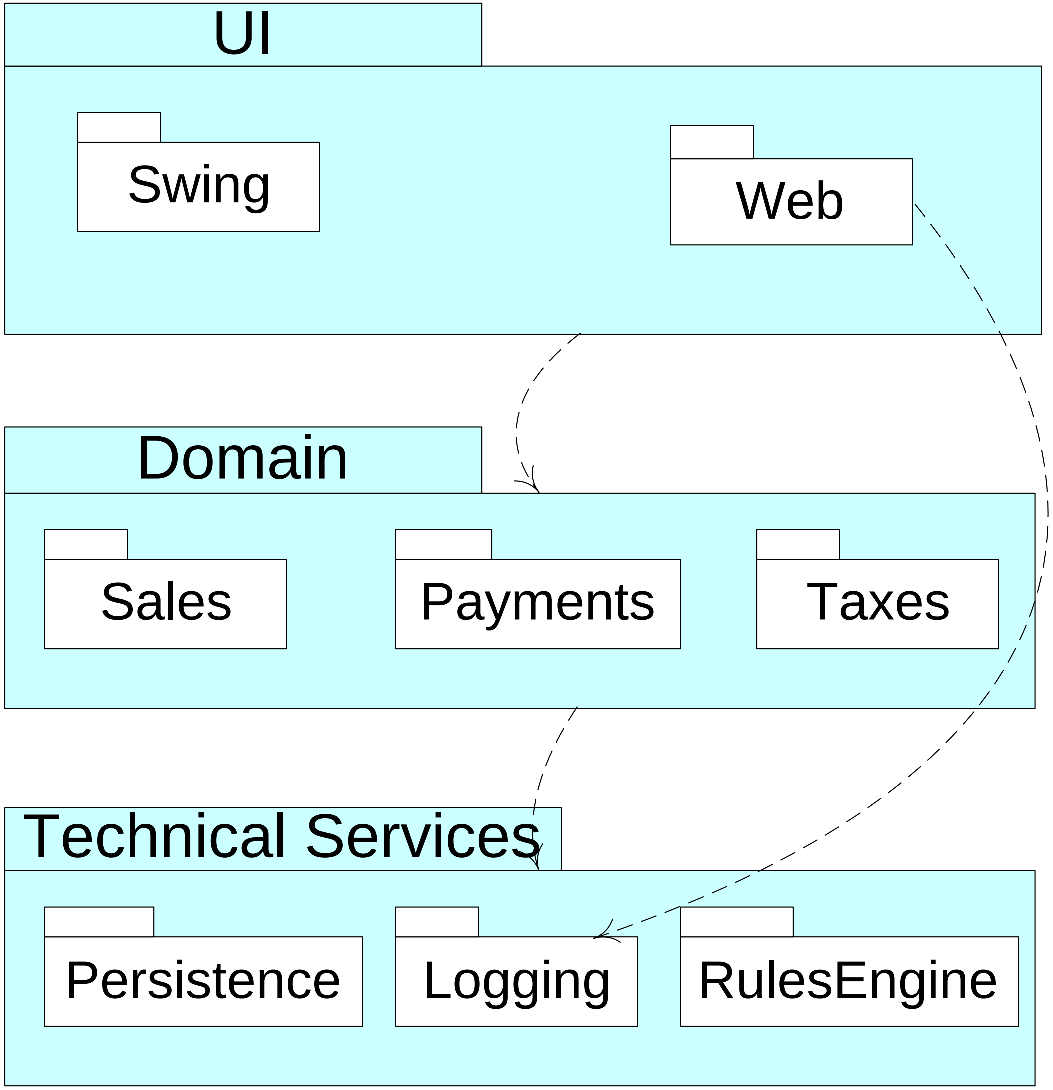
In UML logical partitioning is illustrated with package diagrams.



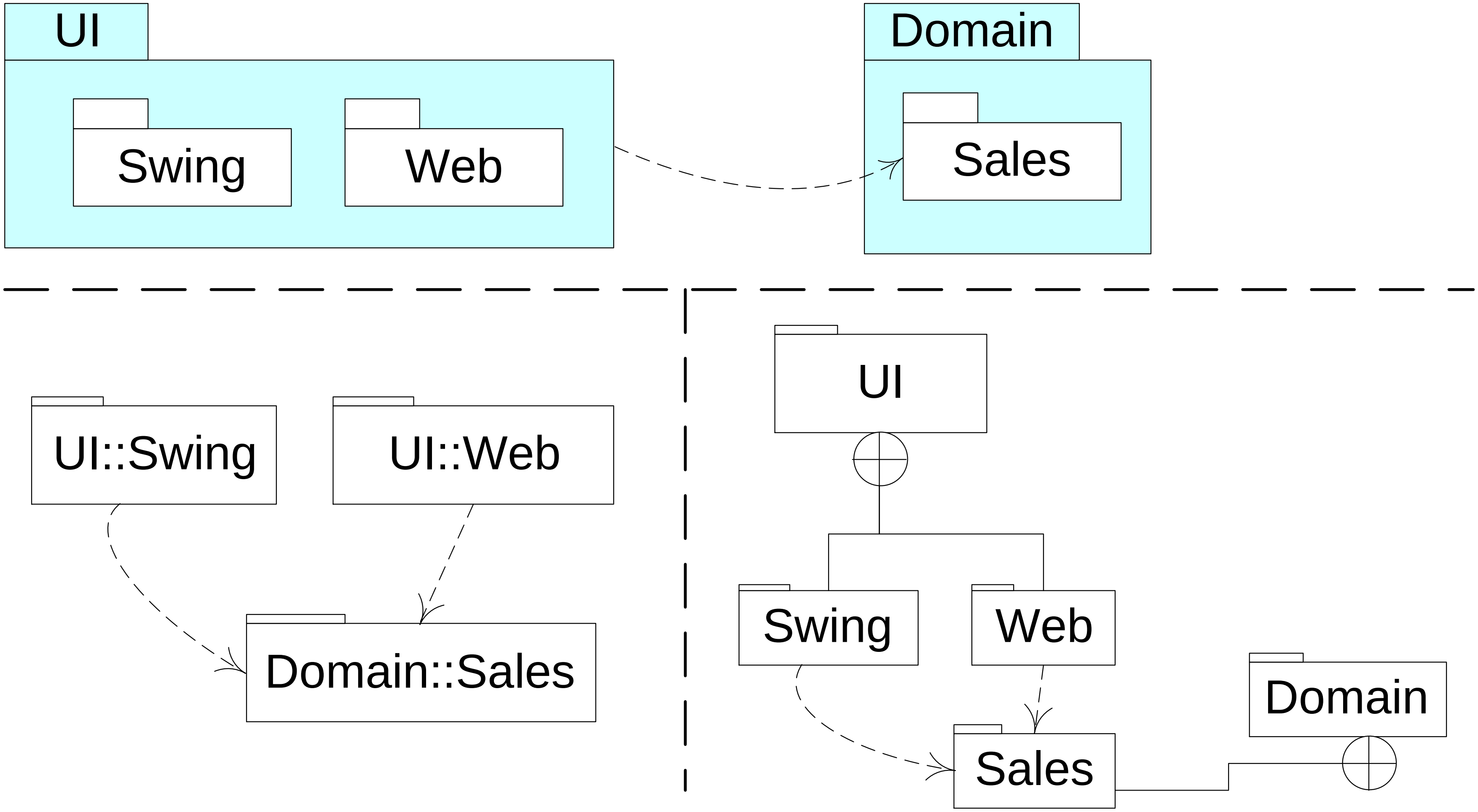
Group elements within packages

- Classes
- Other packages
- Use cases, etc.

# Layers shown with UML package diagram.

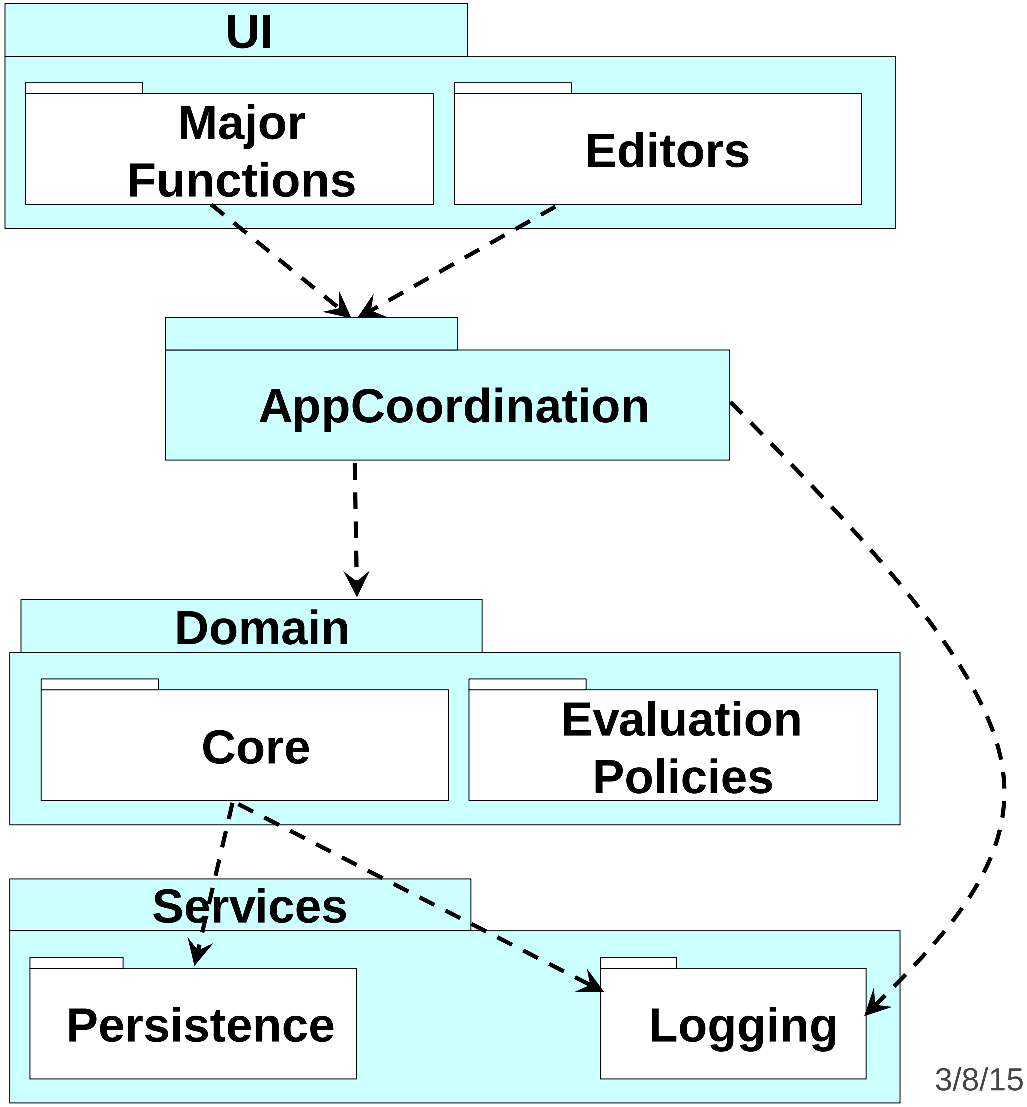


# Various UML notations for package nesting



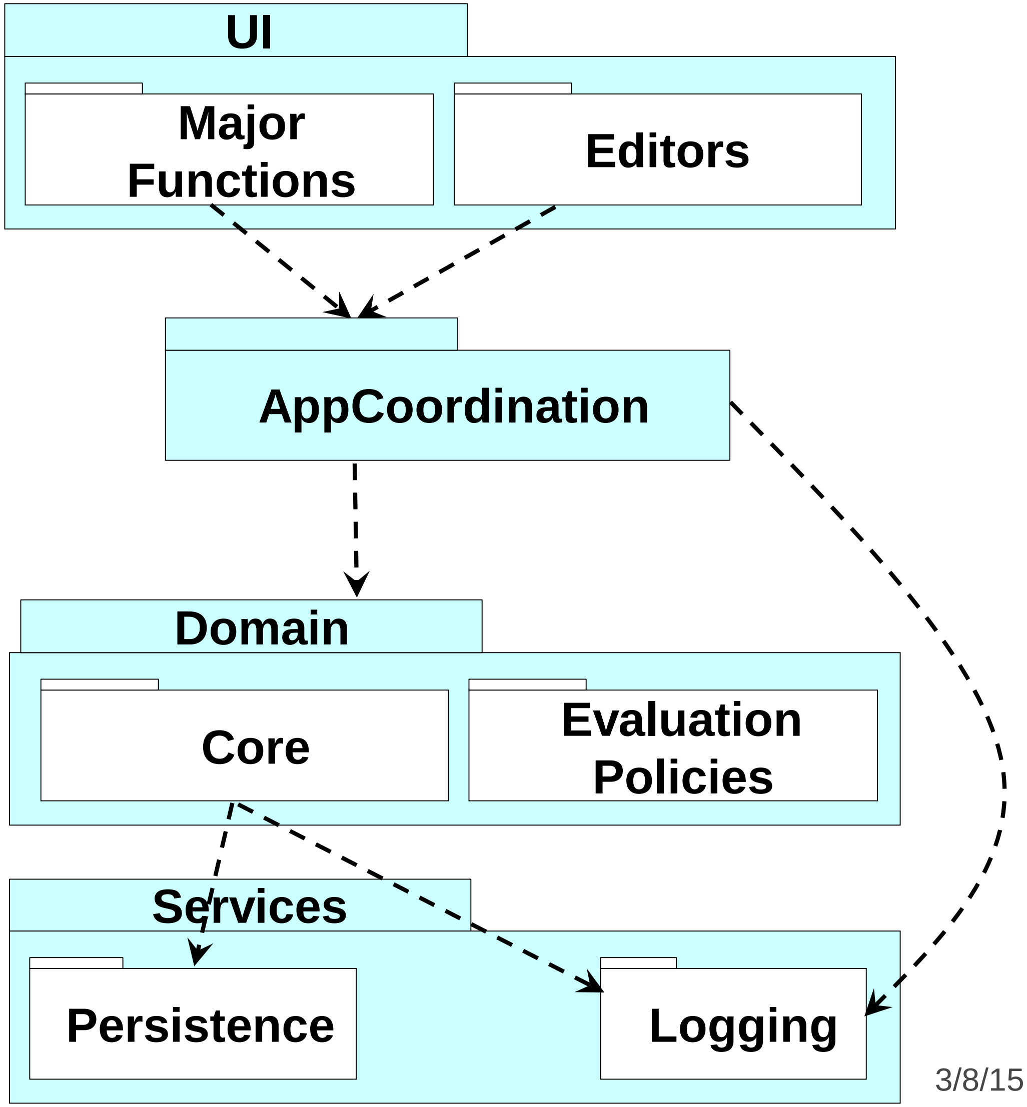
# Package Dependencies

- Dependency line indicates coupling of packages
  - Arrow points to the depended upon package
  - Implies change to depended upon package likely impacts dependent package
  - Robust architectures minimize dependencies



# Ordering Work

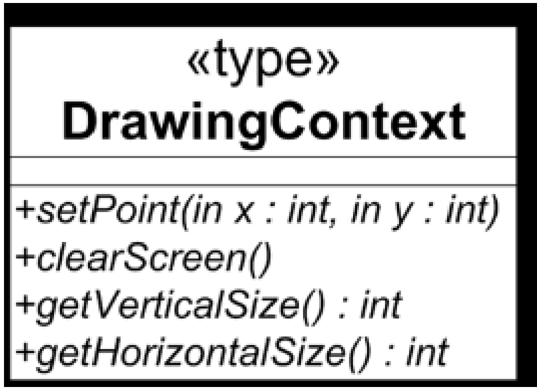
- What can we start with?
- What can we do in parallel?
  - How?
    - Developers can work independently on different layers simultaneously



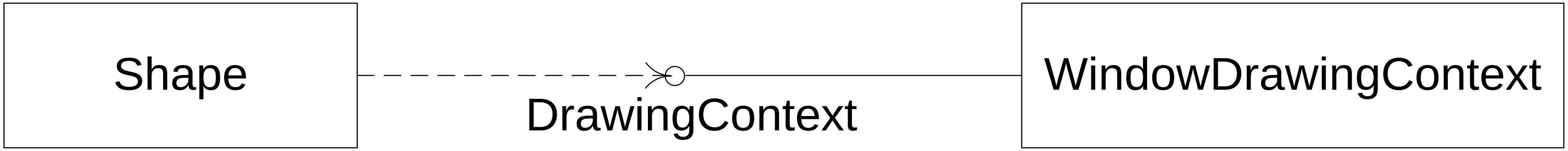
# Type and Lollipop

- The «type» stereotype indicates that the class is an interface

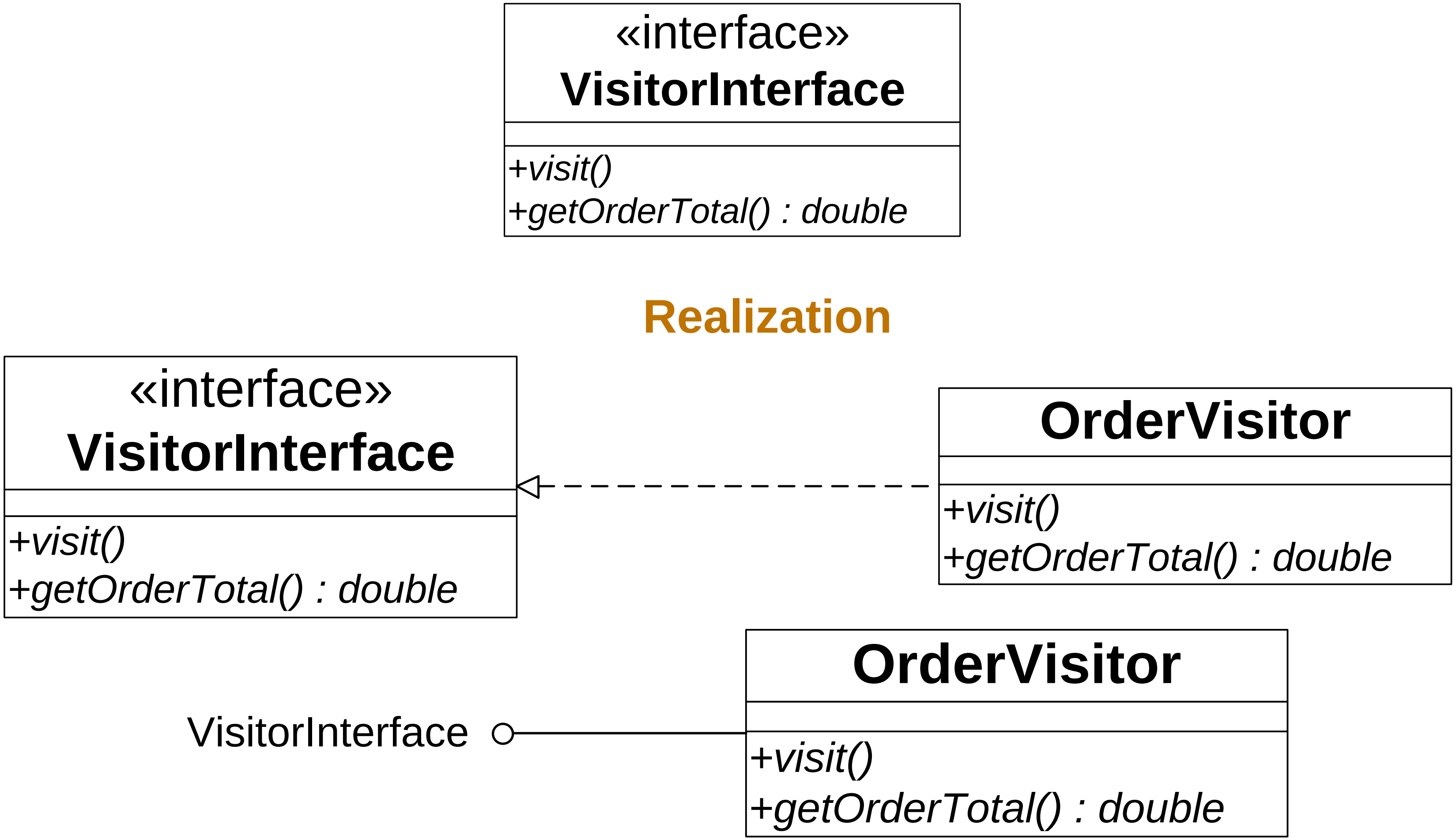
⇒ it has no member variables, and all of its member functions are pure virtual.



- A shortcut for «type» classes is the “lollipop” notation to represent an interface.
  - Shape depends on DrawingContext as shown by the dashed arrow (as usual)
  - The class WindowsDrawingContext is derived from, or conforms to, the DrawingContext interface



# Can use «interface» instead of «type»



«interface»
**VisitorInterface**

*+visit()*
*+getOrderTotal() : double*

**Realization**

«interface»
**VisitorInterface**

*+visit()*
*+getOrderTotal() : double*

**OrderVisitor**

*+visit()*
*+getOrderTotal() : double*

VisitorInterface

**OrderVisitor**

*+visit()*
*+getOrderTotal() : double*

# Component: a Design Level Perspective

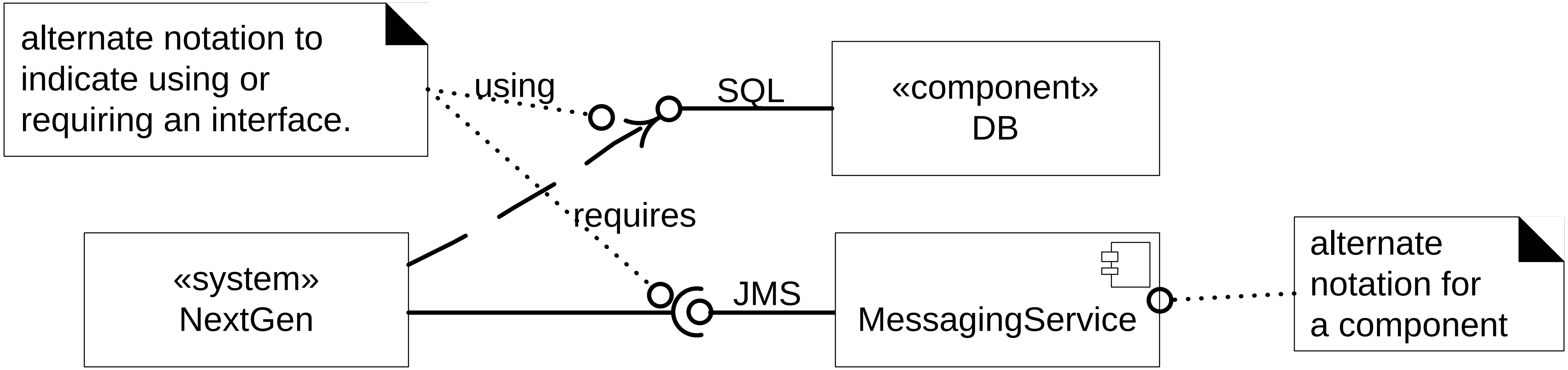
- A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
  - defines its behaviour in terms of provided and required interfaces
  - serves as a *type* defined by these provided & required interfaces
  - can be composed of multiple classes, or components
- Intent of using components is to emphasize
  - that the interfaces are important, and
  - it is modular, self-contained and replaceable.
    - it is a (relatively) stand-alone module.
- Components does not represent concrete software
  - Can map to concrete artefacts such as a set of files.

# Components: UML Example

- □ At a large-grained level, a SQL database engine can be modelled as a component

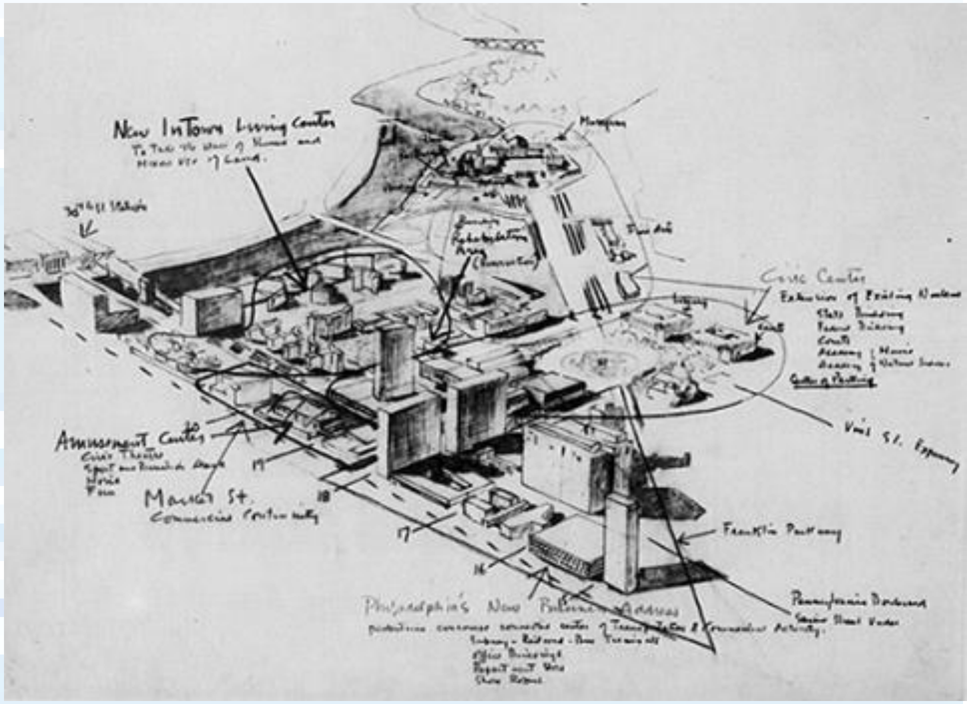
⇒ any database that understands the same version of SQL and supports the same transaction semantics can be substituted.

- □ At a finer grained level, any solution that implements the standard Java Message Service API can be used or replaced in a system.


alternate notation to indicate using or requiring an interface.
using
SQL
«component»
DB
requires
«system»
NextGen
JMS
MessagingService
alternate notation for a component

# ARCHITECTURE DESCRIPTION

- Architecture Definition
- UML Views
- UML Packages
- **Architecture Description**
- Guidelines



# Architecture Diagram

- The Architecture Diagram provides a graphical view of the major components in the system, and the relationships between them.
- Conceptual architecture diagram communicates with various stakeholders (e.g., management, project managers for team/individual work assignments, developers and customers or users).
  - Provides a high-level view useful to non-technical audiences;
  - Summarizes the entire system for technical audiences.
- Use any appropriate UML subset (even class diagrams).
- Remember the point is capturing and conveying the information; not providing perfect UML

# Subscription-Based Sensor Collection Service I

- The “hello world” equivalent of an architecture description conforming to ISO/IEC 42010.
  - www.iso-architecture.org/ieee-1471/docs/SBSCS-AD-v02.pdf

# Subscription-Based Sensor Collection Service II

| **Version:** | **v02** |
|---|---|
| **Date of issue and status:** | 15 April 2010, approved |
| **Issuing organization:** | Dunder Mifflin and Associates, Inc. |
| **Change history:** | Version v02 was updated to reflect requirements and numbering changes between WD4 and CD1 of ISO/IEC 42010. |
| **Summary:** | This architecture provides a subscription-based service of providing access to a widely-distributed set of sensors. |
| **Scope:** | Includes only weather sensors. Does not consider acquisition or maintenance issues. |
| **Context:** | Gore and Associates commissioned this architecture study. |
| **Glossary:** | Not applicable. |
| **Results from evaluations:** | The SBSCS AD was reviewed on 6 Nov 2009 and 14 February 2010. The results of evaluations can be obtained at: https://dunder-mifflin.com/sbscs-eval |
| **References:** | Technical Memo, SCS Architecture Study, 12 March 2010 |

# SBSCS — system stakeholders and concerns

- The following stakeholders were considered and identified:
  - users of the system
  - operators of the system
  - developers of the system
- The system concerns were considered, and the following concerns were identified for SBSCS:

| System Concerns | Stakeholders |
|---|---|
| Return on investment | Operators |
| Timely delivery of sensor data | Users |
| Understanding of interactions between system elements | Developers |

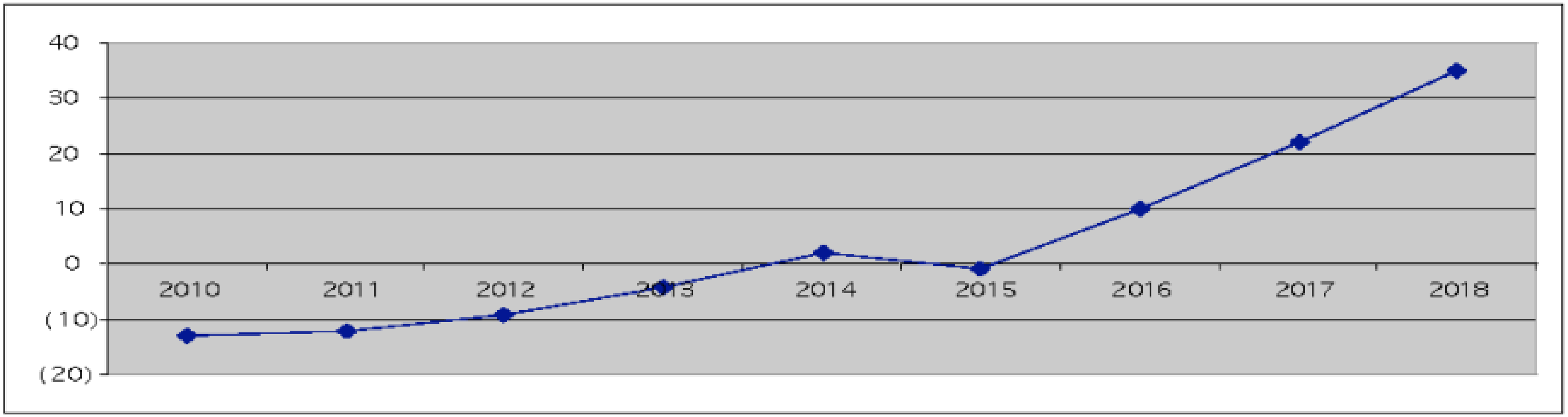
- Architecture Description uses three viewpoints: a financial viewpoint (FVP), an operational viewpoint (OVP) and a system viewpoint (SVP)

# SBSCS — Financial view

- Overview: This view projects that SBSCS will achieve breakeven after five years of system operation.
- Models:
  - Model ID: SCS profit statement; Version: v1.1; Model kind: cash flow statement. Shown in figure 1.
  - Model ID: SCS profitability curve; Version: v1.4; Model kind: ROI curv. Shown in Figure 2.

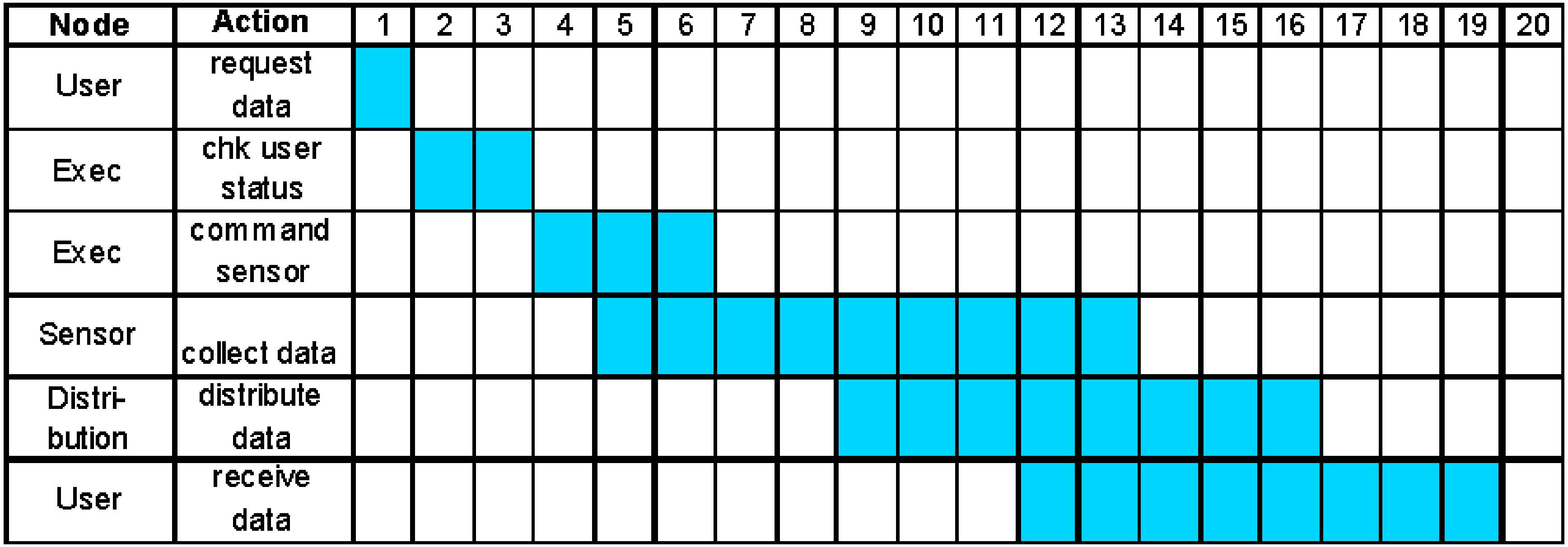
# SBSCS — Financial view: profit statement & profitability curve

| Year | 2010 | 2011 | 2012 | 2013 | 2014 | 2015 | 2016 | 2017 | 2018 |
|---|---|---|---|---|---|---|---|---|---|
| Income | 0 | 2 | 4 | 6 | 8 | 12 | 14 | 16 | 18 |
| Expenses | (13) | (1) | (1) | (1) | (2) | (15) | (3) | (4) | (5) |
| Profit | (13) | 1 | 3 | 5 | 6 | (3) | 11 | 12 | 13 |
| ROI | (13) | (12) | (9) | (4) | 2 | (1) | 10 | 22 | 35 |



# SBSCS — Operational View

- This view shows that a typical user request will be satisfied within 20 seconds.
- Model ID: Collection TLD; Version: v2.4; Model kind: Timeline diagram.



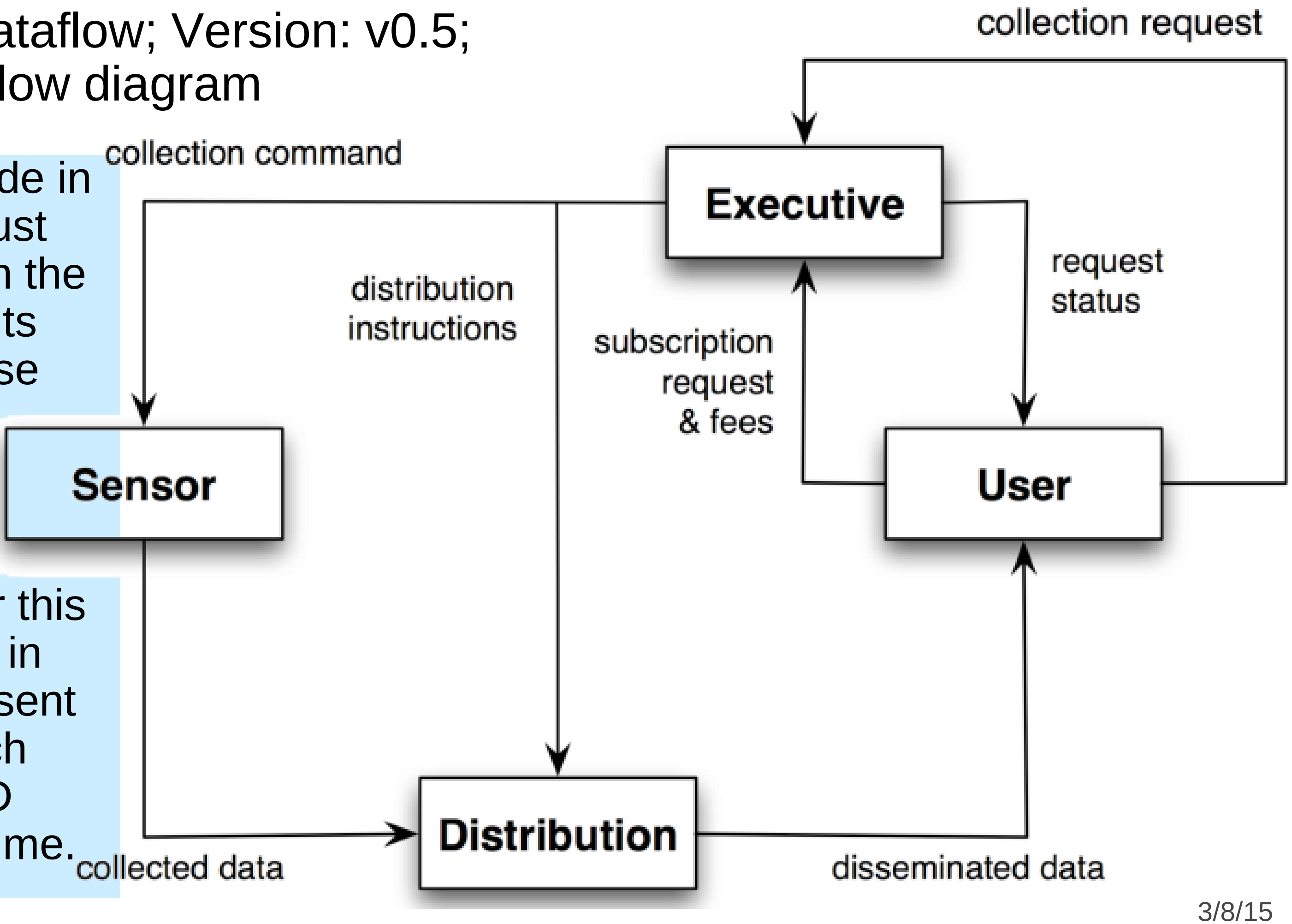
| Node | Action | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| User | request data | ■ | | | | | | | | | | | | | | | | | | | |
| Exec | chk user status | | ■ | ■ | | | | | | | | | | | | | | | | | |
| Exec | command sensor | | | | ■ | ■ | ■ | | | | | | | | | | | | | | |
| Sensor | collect data | | | | | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | | | | | |
| Distri-bution | distribute data | | | | | | | | | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | | | | |
| User | receive data | | | | | | | | | | | | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ■ | |

# SBSCS — System View

- This view shows system nodes and dataflow between nodes
- Model ID: SCS Dataflow; Version: v0.5;
  Model kind: Dataflow diagram

**NodeCheck**: Each node in a dataflow diagram must appear at least once in the timeline diagram with its corresponding response time for that node.

Assessment of **NodeCheck**:
This rule holds true for this SBSCS AD. All nodes in SCS Dataflow are present in Collection TLD. Each entry in Collection TLD specifies a response time.



# GUIDELINES

Architecture Definition

UML Views

UML Packages

Architecture Description

**Guidelines**

# How to partition the domain model

- Place elements together that
  - Are in the same subject area – closely related by concept or purpose
  - Are in a class hierarchy together
  - Participate in the same use cases
  - Are strongly associated

# Architecture Analysis

- Start architectural analysis before the first cycle
  - Can start early iterations before architectural analysis is complete
- It is mainly concerned with non-functional requirements
  - *quality attributes* (Bennett)
  - e.g., security
- within the context of the functional requirements
  - e.g., processing sales
- Examples of issues to be identified and resolved:
  - How do reliability and fault-tolerance requirements affect the design?
  - How do the licensing costs of purchased subcomponents affect profitability?
  - How do the adaptability and configurability requirements affect the design?

# Common steps in architectural analysis

1. Identify and analyze the non-functional requirements that have an impact on the architecture
   - architectural factors (or drivers)
2. For those requirements with a significant architectural impact, analyze alternatives and create solutions that resolve the impact
   - architectural decisions

# Identification and analysis of architectural factors

- Quality Scenarios
  - Form: <stimulus> <measurable response>
  - Record non-functional architectural factor in a measurable form
- Example:
  - When the completed sale is sent to the remote tax calculator to add the taxes, the result is returned within 2 seconds “most” of the time, measured in a production environment under “average” load conditions
  - When a bug-report arrives from a test volunteer, reply with a phone call within 1 working day.
- No point in describing scenarios that will never be tested before shipping.

# Conclusion: the basic architectural design principles

- **Low coupling**
- **High cohesion**
- **Separation of concerns** and **localization of impact**
  - One *could* design persistence support such that each object also communicated with a database to save its data
    - the concern of persistence is then mixed in with the concern of application logic
    - and same with security, etc.

  ⇒ Cohesion drops and coupling rises.
  - Recommend: factor out persistence, security, …
    - object with application logic just has application logic
    - persistence subsystem focuses on the concern of persistence,
    - security subsystem doesn't do persistence.