Implementation Diagrams

Topic # 15

Chapter 37 – UML Deployment and component Diagrams – Craig Larman

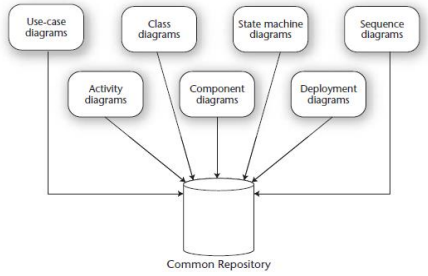
UML Digarams

i. Static
- a. Use case diagram
- b. Class diagram

ii. Dynamic
- a. Activity diagram
- b. Sequence diagram
- c. Object diagram
- d. State diagram
- e. Collaboration diagram

iii. Implementation
- a. Component diagram
- b. Deployment diagram

UML Diagrams



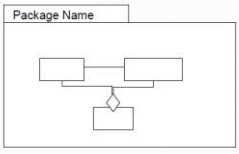
Implementation Diagrams

Implementation Diagrams

- Both are structural diagrams
- **Component Diagrams:**
  - set of components and their relationships
  - Illustrate static implementation view
  - Component maps to one or more classes, interfaces, or Collaborations
- **Deployment Diagrams:**
  - Set of nodes and their relationships
  - Illustrate static deployment view of architecture
  - Node typically encloses one or more components

Package

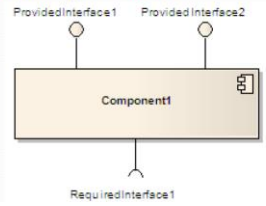
- General purpose mechanism for organizing elements into groups
- Can group classes or components.



Component in UML

- A **component** in the Unified Modeling Language "represents a modular part of a system, that encapsulates its content and whose manifestation is replaceable within its environment. A component defines its behavior in terms of *provided* and *required* interfaces".[1]

Manifestation : an event, action, or object that clearly shows or embodies something abstract or theoretical.



What is Component?

- A component is an autonomous unit within a system.
- A component is a self-contained unit that encapsulates the state and behavior of a set of classifiers.
- All the contents of the components are private—hidden inside.
- Also unlike a package, a component realizes and requires interfaces.
- The components can be used to describe a software system of arbitrary size and complexity.

Component

- A component may be **replaced** by another if and only if their provided and required interfaces are identical.
- This idea is the underpinning for the **plug-and-play** capability of component-based systems and promotes **software reuse**.
- UML places **no restriction on the granularity** of a component. Thus, a component may be as small as a *figures-to-words converter*, or as large as an entire *document management system*.
- Such assemblies are illustrated by means of **component diagrams**.

Component Diagram

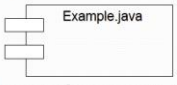
- A component diagram breaks down the actual system under development into various high levels of functionality.
- Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.

Component Diagram

- Models the physical implementation of the software
- Models the high-level software components, and their interfaces
- Elements of component diagram are not much different than what we have seen in class diagram(Classes, interface, relationships)
- Dependencies are designed such that they can be treated as independently as possible
- So, Special kind of class diagram focusing on system's Components.
- Components to use with Component Diagram are:
  - Components required to run the system(library file, etc.)
  - Source code file, and data file
  - Executable file (.exe)

Component Diagram

- Classes
- Interfaces
- Dependency, generalization, association, and realization relationships



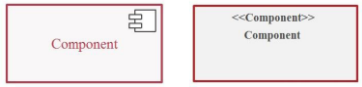
Special kind of class diagram focusing on system's components.

Component Diagram Elements

- Component diagram shows
  - components,
  - Provided interface
  - required interfaces,
  - ports, and
  - Relationships between them.

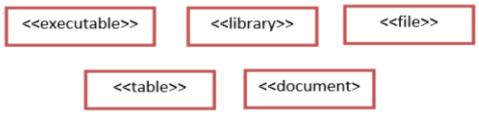
Component Notation

- A component is shown as a rectangle with
  - A keyword <<component>>
  - Optionally, in the right hand corner a component icon can be displayed.
    - A component icon is a rectangle with two smaller rectangles jutting out from the left-hand side
    - This symbol is a visual stereotype
  - The component name
- Components can be labelled with a stereotype
- There are a number of standard Stereotypes.



Component Stereotypes

- Components stereotype provides visual cues about roles played by components in a system. Some of component stereotype are as follows.



- **<<executable>>: executable file (.exe)**
- **<<library>>: references resources (.dll)**
- **<<file>>: text file, source code file, etc.**
- **<<table>>: database file, table file, etc.**
- **<<document>>: document file, web page file, etc.**

Common Stereotypes

Stereotype	Indicates
<<application>> ASP/JSPs	A "front-end" of your system, such as the collection of HTML pages and that work with them for a browser-based system or the collection of screens and controller classes for a GUI-based system.
<<database>> database.	A hierarchical, relational, object-relational, network, or object-oriented
<<document>>	A document. A UML standard stereotype.
<<executable>> stereotype.	A software component that can be executed on a node. A UML standard
<<file>>	A data file. A UML standard stereotype.
<<infrastructure>> audit logger.	A technical component within your system such as a persistence service or an
<<library>>	An object or function library. A UML standard stereotype.
<<source code>>	A source code file, such as a .java file or a .cpp file.
<<table>>	A data table within a database. A UML standard stereotype
<<web service>>	One or more web services.
<<XML DTD>>	An XML DTD.

Interfaces

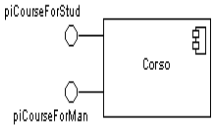
- **An interface**
  - Is the definition of a collection of one or more operations
  - Provides only the operations but not the implementation
  - Implementation is normally provided by a class/ component
  - In complex systems, the physical implementation is provided by a group of classes rather than a single class
- A class can implement one or more interfaces
- An interface can be implemented by 1 or more classes

Interfaces

- May be shown using a rectangle symbol with a keyword <<interface>> preceding the name.
- For displaying the full signature, the interface rectangle can be expanded to show details
- Can be
  - Provided
  - Required
- The purpose is:
  - To control dependencies between components
  - To make components swappable

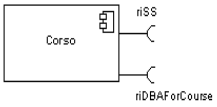
Provided Interfaces

- **Provided Interface:**
- Characterize services that the component offers to its environment
- Is modeled using a ball, labelled with the name, attached by a solid line to the component. Also known as Lollipop notation.

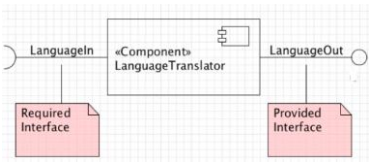


Required Interfaces

- **Required Interface:**
- Characterize services that the component expects from its environment
- Is modeled using a socket, labelled with the name, attached by a solid line to the component
- In UML 1.x were modeled using a dashed arrow

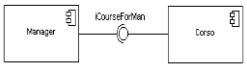


Interface Example



Interfaces

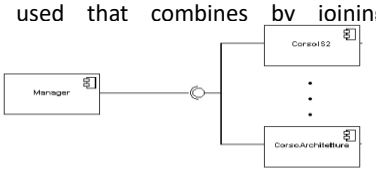
- Where two components/classes provide and require the same interface, these two notations may be combined.



- The ball-and-socket notation hint at that interface in question serves to mediate interactions between the two components
- If an interface is shown using the rectangle symbol, we can use an alternative notation, using dependency arrows

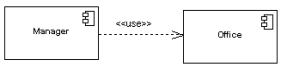
Interfaces

- In a system context where there are multiple components that require or provide a particular interface, a notation abstraction can be used that combines by joining the in



Dependencies

- Components can be connected by usage dependencies.

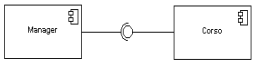


- **Usage Dependency**
- A usage dependency is relationship which one element requires another element for its full implementation
- Is a dependency in which the client requires the presence of the supplier
- Is shown as dashed arrow with a <<use>> keyword
- The arrowhead point from the dependent component to the one of which it is dependent

Connectors

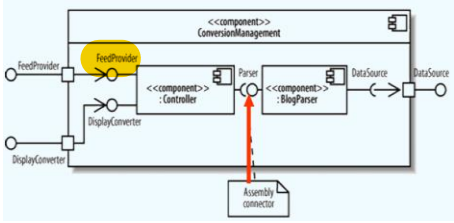
- Two kinds of connectors:
  - Delegation
  - Assembly
- **ASSEMBLY CONNECTOR**
  - A connector between 2 components defines that one component provides the services that another component requires
  - It must only be defined from a required interface to a provided interface
  - An assembly connector is notated by a "ball-and-socket" connection

This notation allows for succint grafical wiring of components



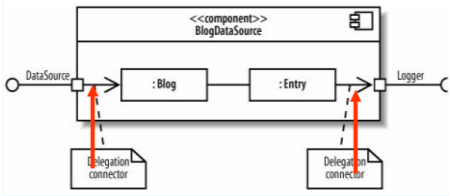
Assembly Connector

- Used to show components within another component working together through interfaces.



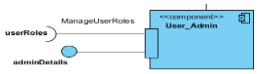
Delegation Connector

- Used to show that internal parts realize or use the component's interfaces.



Port

- Specifies a distinct interaction point between that component and its environment –
- Between that component and its internal parts – Is shown as a small square symbol –
- Ports can be named, and the name is placed near the square symbol – Is associated with the interfaces
- Library Services class has port searchPort.



Internal Realization

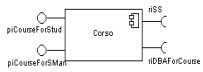
- **A component often contains and uses other classes to implement its functionality.**
- **These classes realize the component**

Views of a Component

- **A component have an**
  - **external view and**
  - **an internal view**

EXTERNAL VIEW

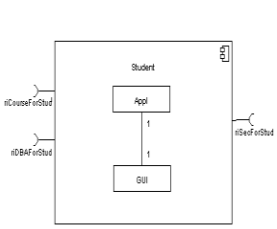
- An external view (or black box view) shows publicly visible properties and operations
- **An external view of a component is by means of interface symbols sticking out of the component box**
- **The interface can be listed in the compartment of a component box**

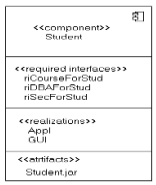


Internal View

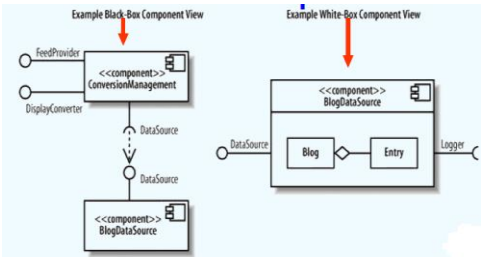
- An internal, or white box view of a component is where the realizing classes/components are nested within the component shape
- The internal class that realize the behavior of a component may be displayed in an additional compartment
- Compartments can also be used to display parts, connectors or implementation artifacts
- An artifact is the specification of a physical piece of information

Internal View



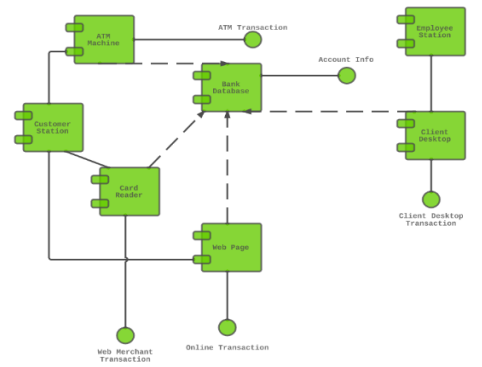


Black-Box and White-Box Views



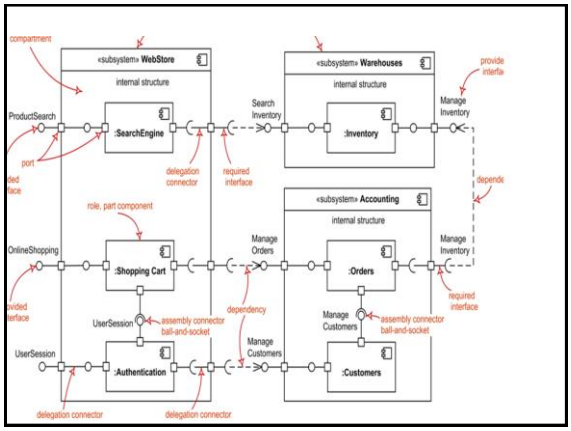
Component Diagram Guidelines

- **Use Descriptive Names for Architectural Components**
  - Use Environment-Specific Naming Conventions for Detailed Design Components
  - Apply Textual Stereotypes to Components Consistently
- **Interfaces**
  - Prefer Lollipop Notation To Indicate Realization of Interfaces By Components
  - Prefer the Left-Hand Side of A Component for Interface Lollipops
  - Show Only Relevant Interfaces
- **Dependencies and Inheritance**
  - Model Dependencies From Left To Right
  - Place Child Components Below Parent Components
  - Components Should Only Depend on Interfaces



Case Study

- The internal structure of online shopping consists of three related subsystems - WebStore, Warehouses, and Accounting. – WebStore subsystem contains three components related to online shopping - Search Engine, Shopping Cart, and Authentication. Search Engine component allows to search or browse items by exposing provided interface Product Search and uses required interface Search Inventory provided by Inventory component. Shopping Cart component uses Manage Orders interface provided by Orders component during checkout. Authentication component allows customers to create account, login, or logout and binds customer to some account. – Accounting subsystem provides two interfaces - Manage Orders and Manage Customers. Delegation connectors link these external contracts of the subsystem to the realization of the contracts by Orders and Customers components. – Warehouses subsystem provides two interfaces Search Inventory and Manage Inventory used by other subsystems and wired through dependencies.



Deployment Diagrams

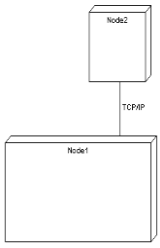
- Deployment diagrams are parts of the Physical view.
- This view is concerned with the physical elements of your system, such as executable software files and the hardware they run on.
- Deployment diagrams bring the software into real world by showing how software gets assigned to hardware and how the pieces communicate.

Deployment Diagrams

- There is a strong link between components diagrams and deployment diagrams
- Deployment diagrams show the physical relationship between hardware and software in a system
- Hardware elements:
  - Computers (clients, servers)
  - Embedded processors
  - Devices (sensors, peripherals)
- Are used to show the nodes where software components reside in the run-time system

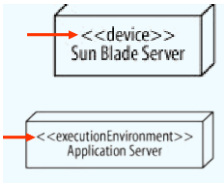
Deployment Diagrams

- Shows the configuration of:
  - Run time processing nodes &
  - The components that live on them
- Contains **nodes** and **connections**
- A node usually represent a piece of hardware in the system.(represented by three dimensional box)
- A connection depicts the communication path used by the hardware to communicate.
- Usually indicates the method such as TCP/IP
- **Graphically:** collection of vertices & arcs

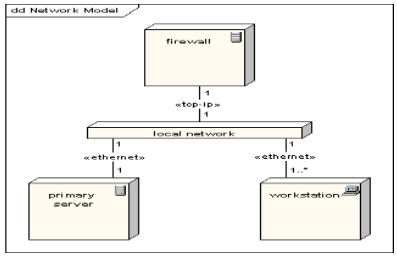


Nodes

- Nodes can be:
- Hardware nodes:
  - Server
  - Desktop PC
  - Disk drive
- Execution nodes:
  - Operating system
  - J2EE container
  - Web server
  - Application server

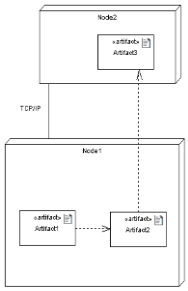


Nodes and Connections



Deployment Diagrams

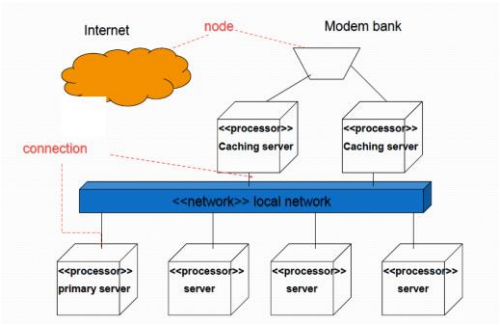
- Deployment diagrams contain **artifact**
- An artifact is the specification of a physical piece of information
  - Ex: source files, binary executable files, table in a database system, executable files, configuration files,....
- Artifacts are physical files that execute or are used by your software
- An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon,



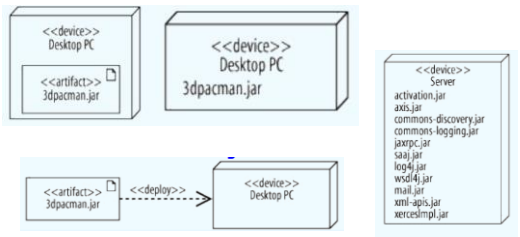
Modeling Client server architecture

- Identify nodes that represent system's client and server processors
- Highlight those devices that are essential to the behavior
  - E.g.: special devices (credit card readers, badge readers, special display devices)
- Use stereotyping to visually distinguish

A deployment diagram

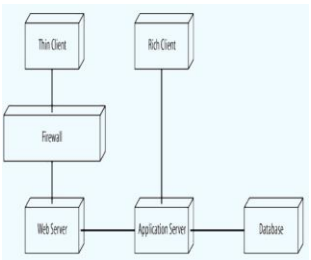


Artifacts



When to Use?

- At the early stage: helps figuring out the general configuration.
- Example: a web application will include:
  - A web server, application server and database
  - Clients access the application through browsers
  - The web server should have a firewall

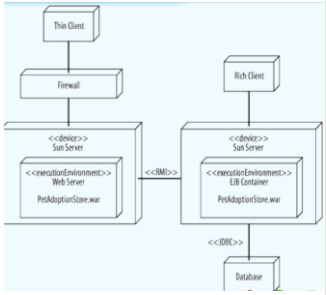


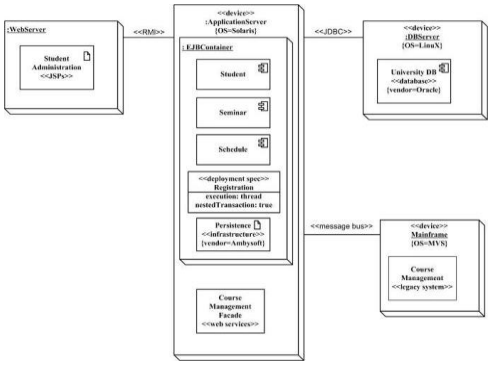
Common stereotypes for communication associations

- Asynchronous: An asynchronous connection, perhaps via a message bus or message queue.
- HTTP: HyperText Transport Protocol.
- JDBC: Java Database Connectivity, Java API for DB access.
- ODBC: Open Database Connectivity, (MS API for DB).
- RMI: Remote Method Invocation, (Java comm. Protocol).
- RPC: Communication via remote procedure calls.
- Synchronous: A synchronous connect where the senders waits for a response from the receiver.
- web services: Communication is via Web Services protocols such as SOAP and UDDI

When to Use?

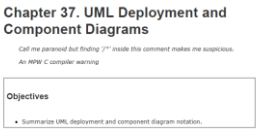
- At the later stages: the deployment diagram will come into details about the system architecture.
  - Which technology is used
  - What communication protocols are used





READING

Chapter 37 – Applying UML and Pattern by Craig Larman 3rd Edition

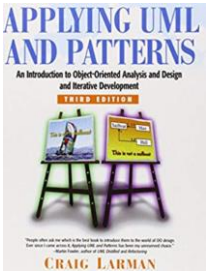

Chapter 37. UML Deployment and Component Diagrams

Call me paranoid but finding ';'' inside this comment makes me suspicious.

An HPW C compiler warning

Objectives

- Summarize UML deployment and component diagram notation.



END OF TOPIC 15

-COMING UP!!!!!!
-RUP
-4+1 model
-Design Patterns