Components and Deployment Diagrams

Design and Implementation

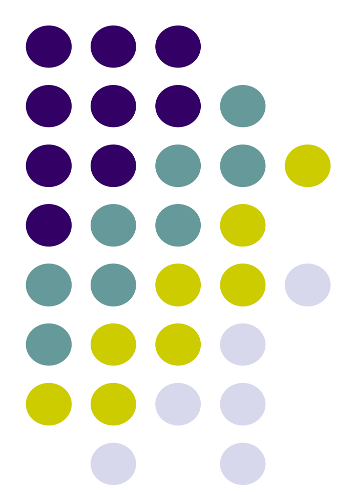
Components and Component Packages

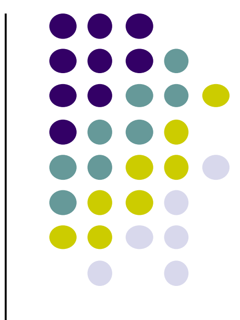
Dependencies

Processors and Devices

Connections

Examples





Bibliography

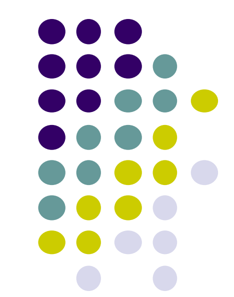
Basic

- Roger S. Pressman. *Software Engineering : A Practitioner's Approach*, 8th edition (2014), McGraw Hill, ISBN-10: 0078022126
- Ian Sommerville. *Software Engineering*, 10th edition (2015), Addison-Wesley Pub Co; ISBN-10: 0133943038
- Dennis A, Wixom BH, Tegarden D. *Systems Analysis and Design, UML Version 2.0*. Wiley; 2009

Additional

- *Software Engineering : Theory and Practice,* by S. Pfleeger and J. Atlee, 4th edition (2009), Pearson International Edition, ISBN-10: 0136061699
- *SDLC (Software Development Life Cycle) Phases, Methodologies, Process, And Models*, https://www.softwaretestinghelp.com/software-development-life-cycle-sdlc/

Stages of a typical Software Development Life Cycle (SDLC)



Planning

Defining

Designing

Building

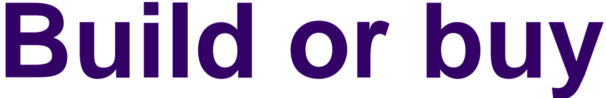
Testing

Deployment

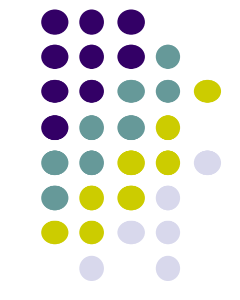
SDLC

Design and implementation

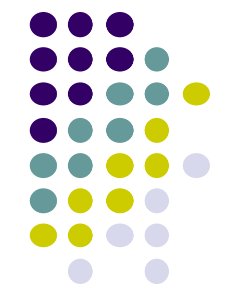
- Software implementation is the stage in the software engineering process at which *an executable software system* is developed.
- Software design and implementation activities are invariably inter-leaved:
  - Software design is a creative activity in which you identify software components and their relationships, based on a customer’s requirements.
  - Implementation is the process of realizing the design as a program.



Build or buy



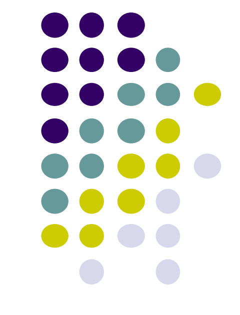
- In a wide range of domains, it is now possible to buy component off-the-shelf systems (COTS) that can be adapted and tailored to the users’ requirements.
  - For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.



An object-oriented design process

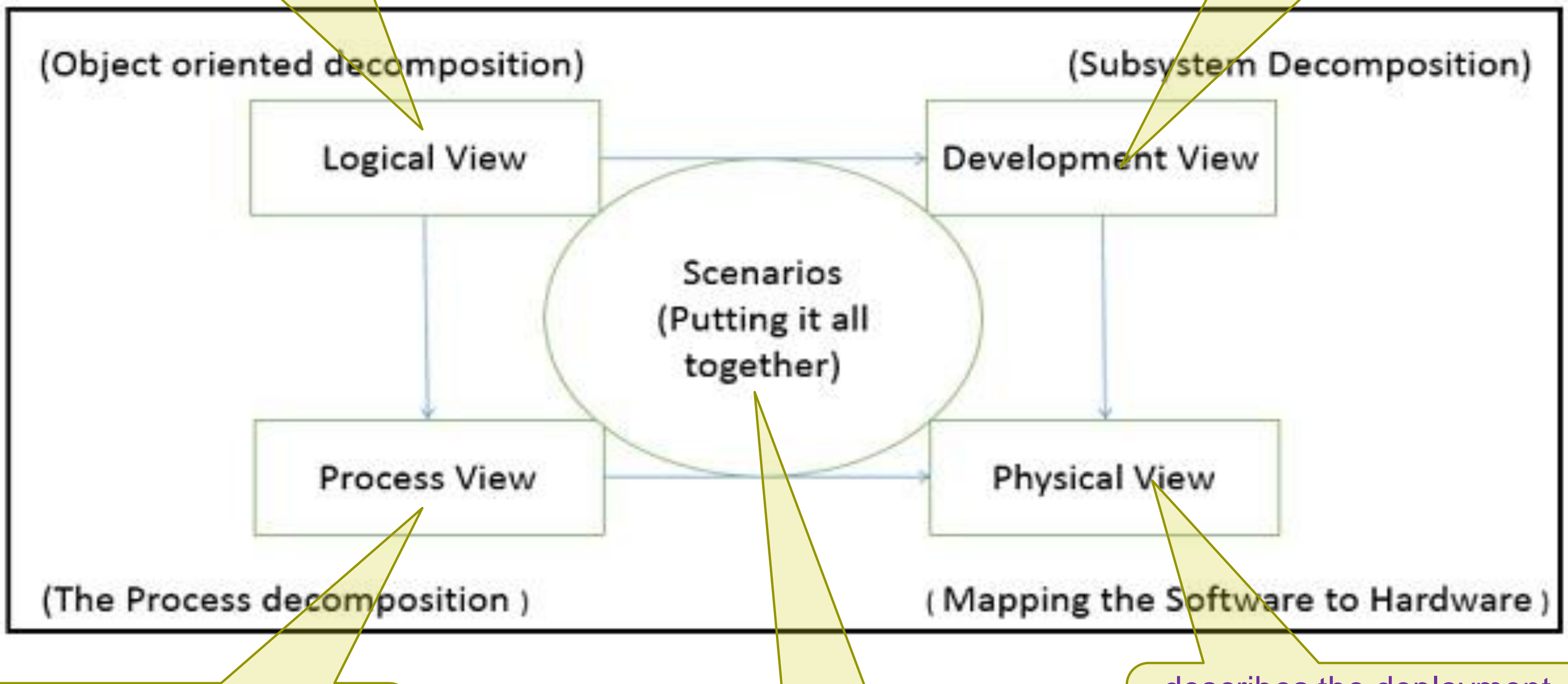
- Structured object-oriented design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an important communication mechanism.

Process stages

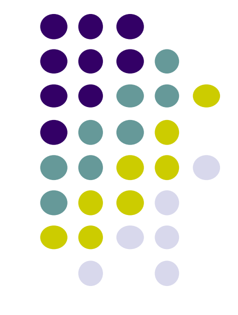


- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.
- Process illustrated here using a design for a wilderness weather station.

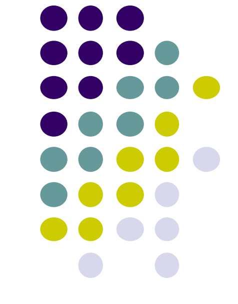
4+1 Software architecture views
(by Kruchten)



System context and interactions



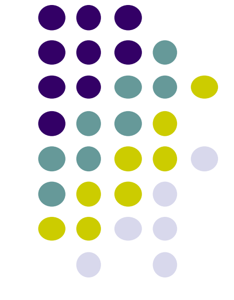
- Understanding the relationships between the software that is being designed and its external environment is essential for:
  - deciding how to provide the required system functionality and
  - how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish the boundaries of the system. Setting the system boundaries helps you decide:
  - what features are implemented in the system being designed and
  - what features are in other associated systems.



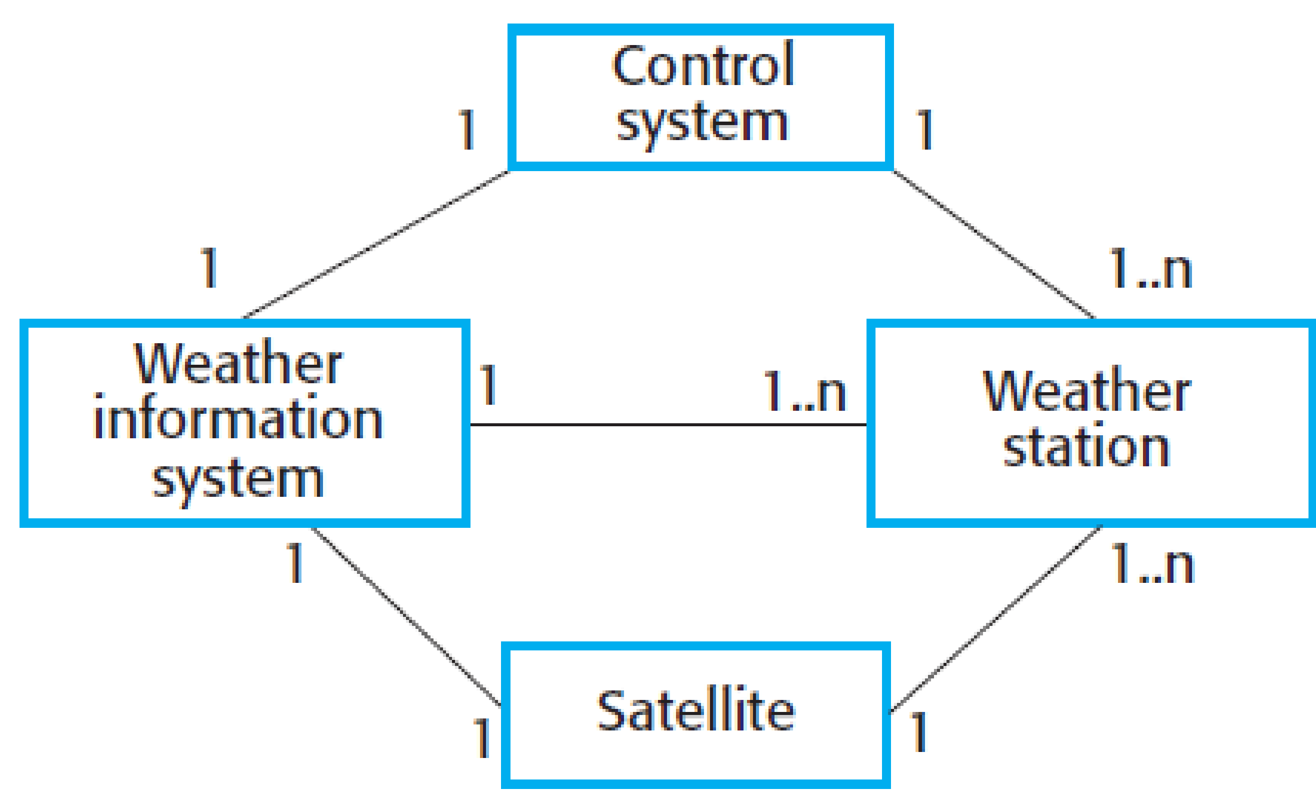
Context and interaction models

- A ***system context model*** is a structural model that demonstrates the other systems in the environment of the system being developed.

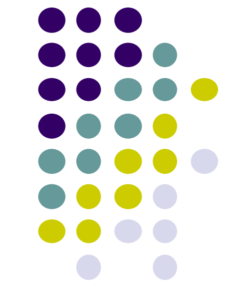
- An ***interaction model*** is a dynamic model that shows how the system interacts with its environment as it is used.

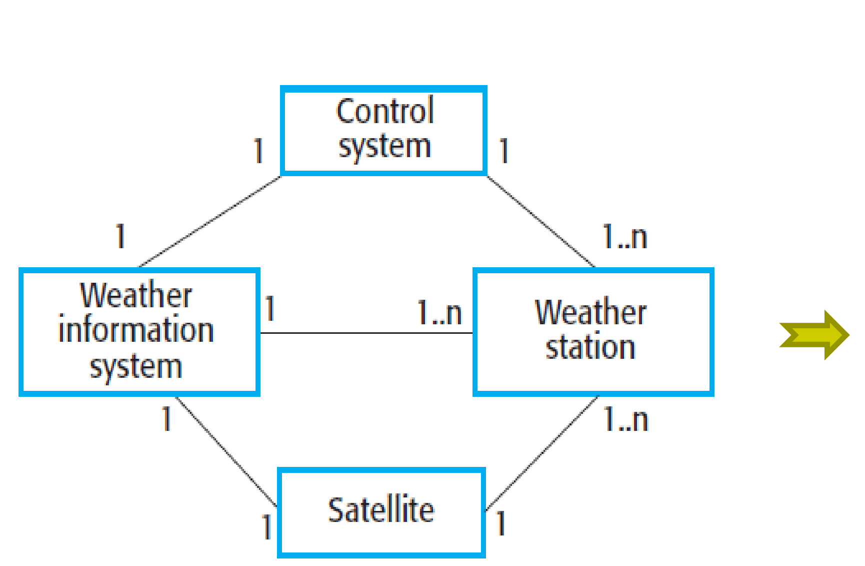


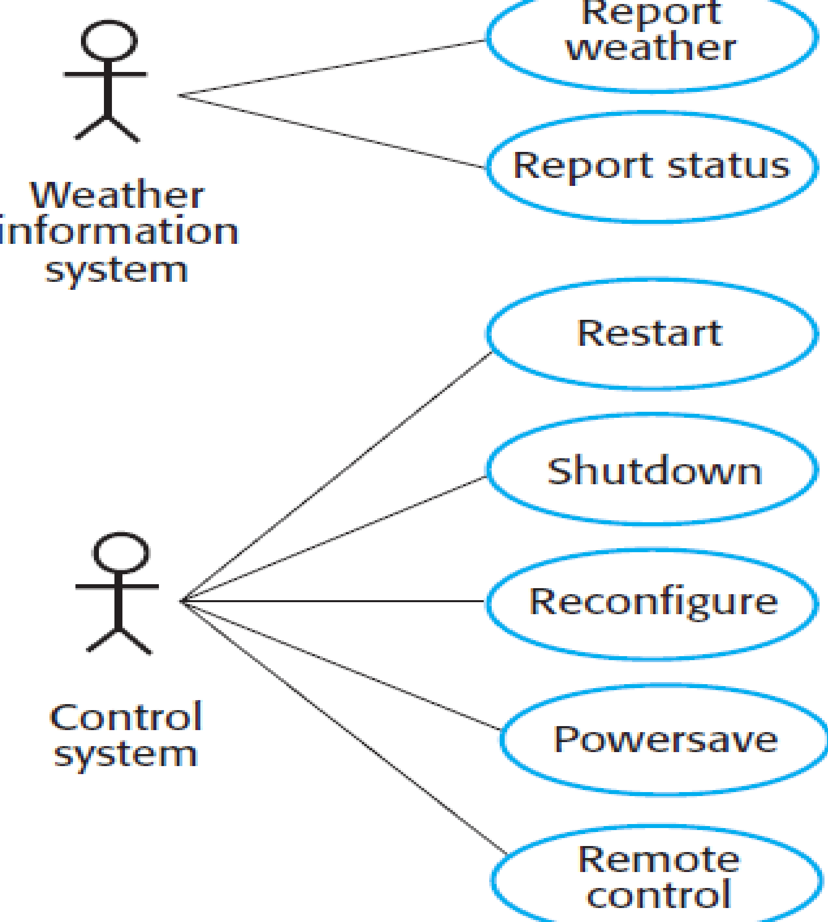
System context for a weather station



Weather station use cases





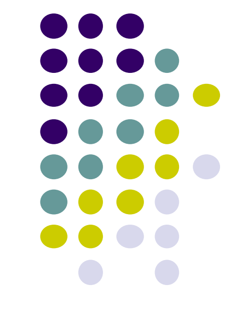


Use case description —Report weather

Weather information system
Report weather
Report status
Control system
Restart
Shutdown
Reconfigure
Powersave
Remote control

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average **ground and air temperatures**; the maximum, minimum, and average **air pressures**; the maximum, minimum, and average **wind speeds**; the total **rainfall**; and the **wind direction** as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

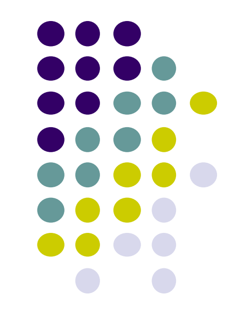
Architectural design



- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture:
  - You identify the major components that make up the system and their interactions, and then
  - You may organize the components using an architectural pattern such as a layered or client-server model.

- The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

High-level architecture of the weather station



«subsystem»
Fault manager

«subsystem»
Configuration manager

«subsystem»
Power manager

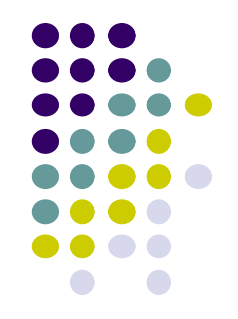
Communication link

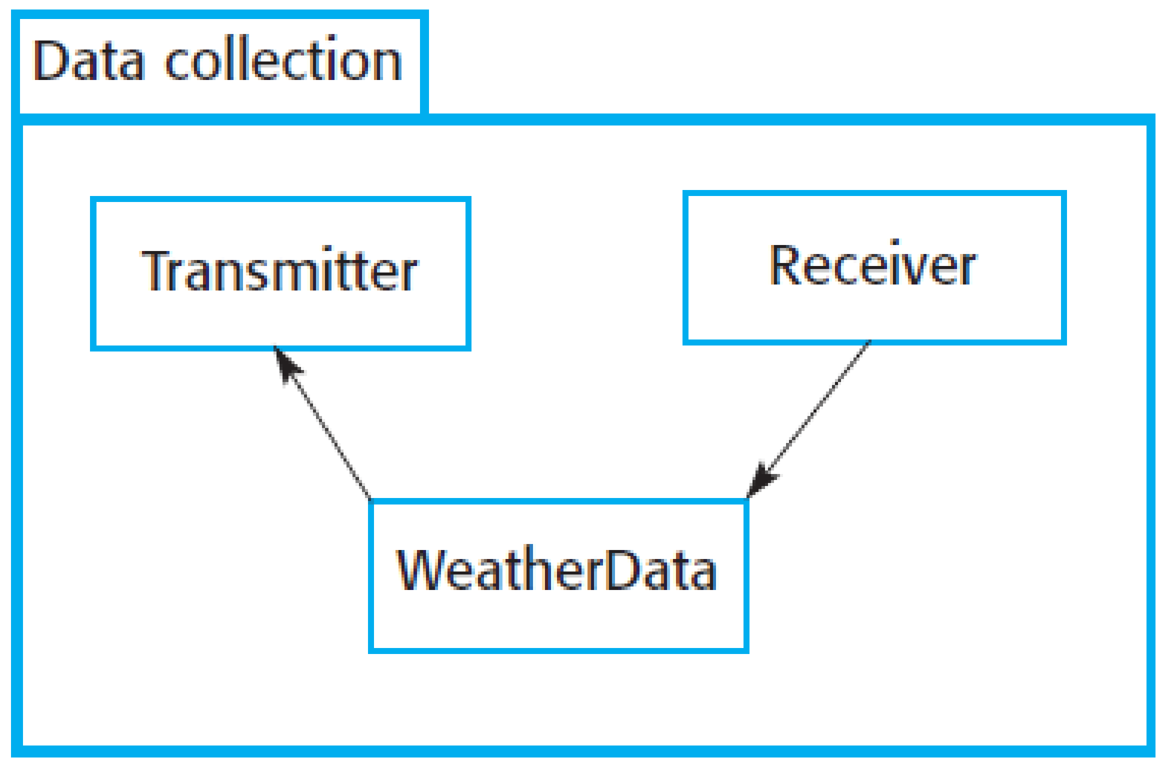
«subsystem»
Communications

«subsystem»
Data collection

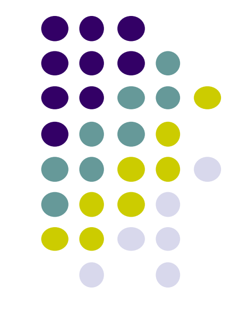
«subsystem»
Instruments

Architecture of data collection system



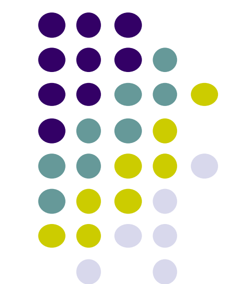


Analysis object/class identification



- Identifying object classes is often a difficult part of object-oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

- Use a grammatical approach based on a natural language description of the system (by a hierarchical decomposition of the software problem).
- Base the identification on tangible things in the application domain.
- Use a behavioural approach and identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.



Weather station description

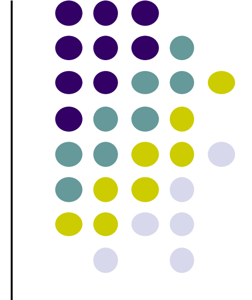
A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing.

The instruments include air and ground thermometers, an anemometer (for measuring the speed of the wind), a wind vane (showing the direction of the wind), a barometer and a rain gauge (pluviometer). Data is collected periodically.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

Weather station measuring devices

Weather station analysis classes



- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - Ground Thermometer, Anemometer, Barometer
    - Application domain objects that are ‘hardware’ objects related to the instruments in the system.
  - Weather station
    - The basic interface of the weather station to its environment. It therefore reflects the interactions identified in the use-case model.
  - Weather data
    - Encapsulates the summarized data from the instruments.

Weather station object classes

WeatherStation
identifier
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

WeatherData
airTemperatures groundTemperatures windSpeeds windDirections pressures rainfall
collect ( ) summarize ( )

Ground thermometer
gt_Ident temperature
get ( ) test ( )

Anemometer
an_Ident windSpeed windDirection
get ( ) test ( )

Barometer
bar_Ident pressure height
get ( ) test ( )

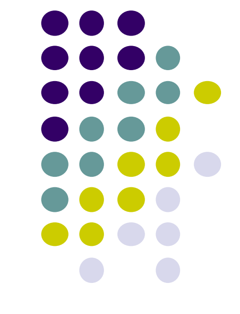
Design models

- Design models show the objects and object classes and relationships between these entities.
- Static models describe the static structure of the system in terms of object classes and relationships.
- Dynamic models describe the dynamic interactions between objects.

Examples of design models

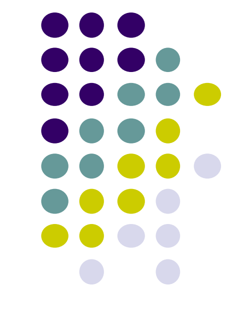
- Subsystem models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

Subsystem models



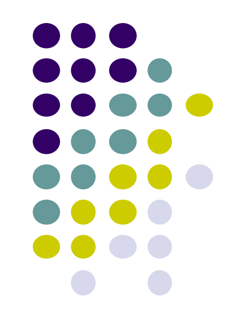
- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

Sequence models



- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Sequence diagram describing data collection



Weather information system

:SatComms

:WeatherStation

:Commslink

:WeatherData

request (report)

acknowledge

reportWeather ()

acknowledge

get (summary)

summarize ()

send (report)

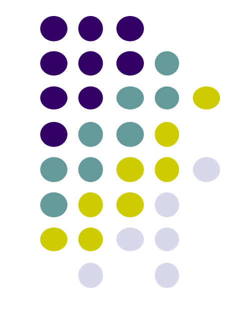
acknowledge

reply (report)

acknowledge

asynchronous

synchronous



State diagrams

- State diagrams are used to show how objects respond to different service requests and the state transitions triggered by these requests.
- State diagrams are useful high-level models of a system or an object’s run-time behavior.
- You don’t usually need a state diagram for all of the objects in the system. Many of the objects in a system are relatively simple and a state model adds unnecessary detail to the design.

Weather station state diagram

Controlled

Operation

shutdown()

remoteControl()

restart()

Shutdown

Running

reportStatus()

Testing

transmission done

test complete

configuration done

reconfigure()
powerSave()

Transmitting

clock

collection
done

reportWeather()

weather summary
complete

Configuring

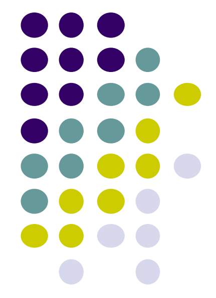
Summarizing

Collecting

Interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

Weather station interfaces



«interface»
Reporting

weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport

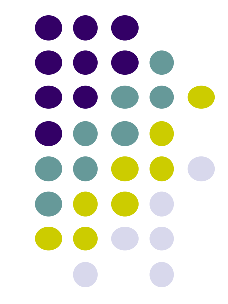
«interface»
Remote Control

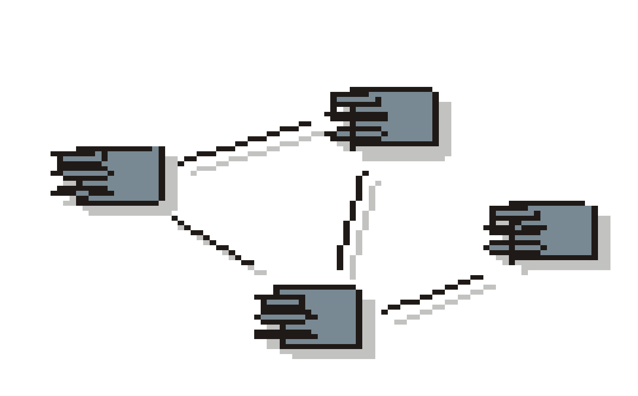
startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

Key points

- Software design and implementation are inter-leaved activities. The level of detail in the design depends on the type of system and whether you are using a plan-driven or agile approach.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the component interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalization models, association models) and dynamic models (sequence models, state machine models).
- Component interfaces (+ their stereotypes) must be defined precisely so that other objects can use them.

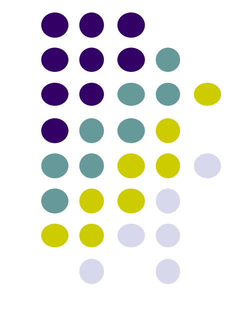
The Implementation Model in the Component View





An **implementation model** is a collection of components, and the implementation subsystems which contain them.
A **component** diagram has a higher level of abstraction than a Class Diagram - usually a **component** is implemented by one or more classes (or objects at runtime).

Components

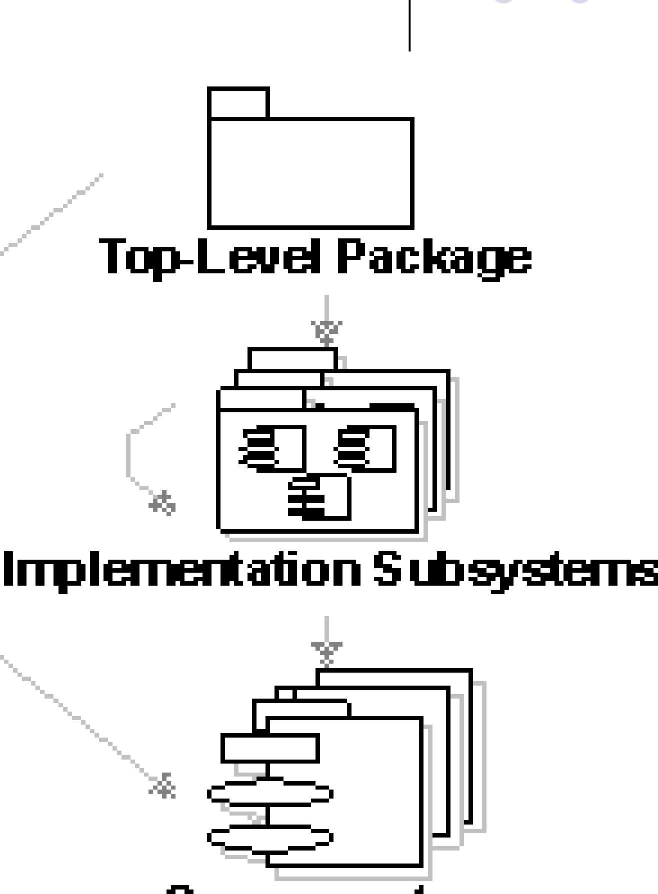


- Components are building blocks so a component can eventually encompass a large portion of a system.
- Components include both deliverable components, such as executables, and components from which the deliverables are produced, such as source code files.

Implementation Model in the Component View

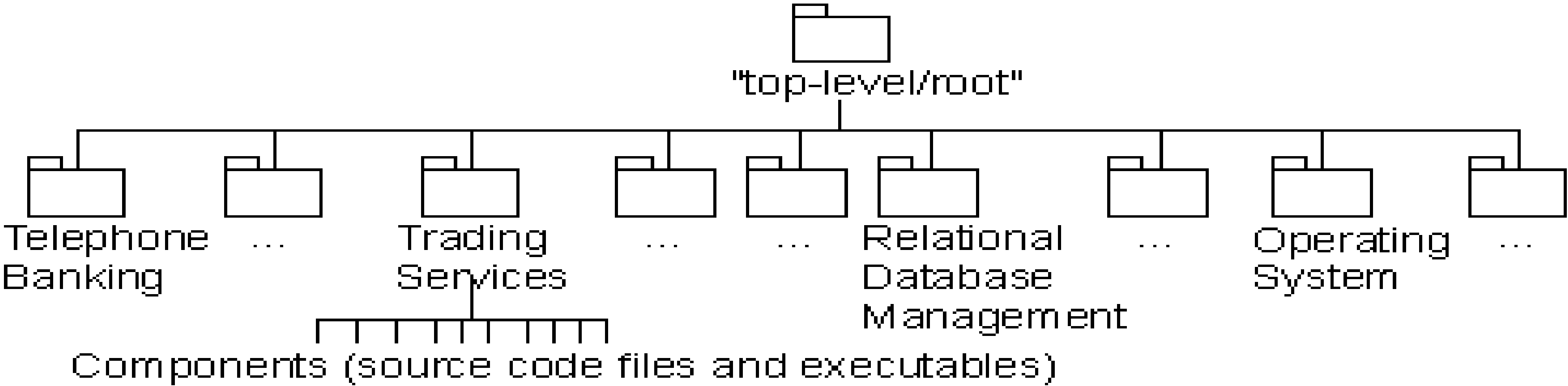
The implementation model is a ***hierarchy of implementation subsystems***, with leaves that are components. There is a package that serves as the top-level (root) node in the implementation model. A subsystem is a collection of components and other subsystems.

The implementation model can be divided into components that are ***deliverables***, such as executables that are delivered to customers; and those ***components from which the deliverables are produced***, such as source code.



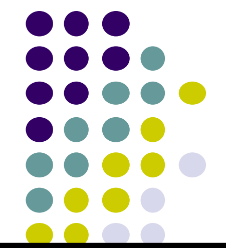
Implementation Model - Example

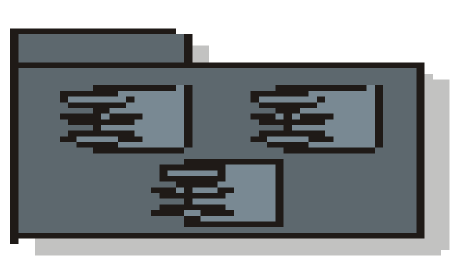
Example: In a banking system the implementation subsystems are organized as a flat structure in the top-level node of the implementation model. Another way of viewing the subsystems in the implementation model is in layers.



The implementation model for a banking system, showing the ownership hierarchy.

Implementation Subsystems. Component Packages.

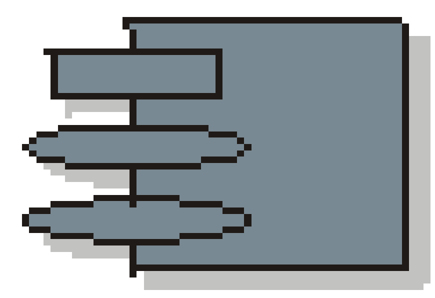


	An **implementation subsystem** is a collection of components and other implementation subsystems that are used to structure the implementation model by dividing it into smaller parts.

Subsystems take the form of directories, with additional structural or management information. For example, a subsystem can be created as a directory or a folder in a file system, or a subsystems in Rational for C++ or Ada, or packages using Java.

Component packages represent clusters of logically related components, or major pieces of your system. Component packages parallel the role played by logical packages for class diagrams. They allow you to partition the physical model of the system.

Components

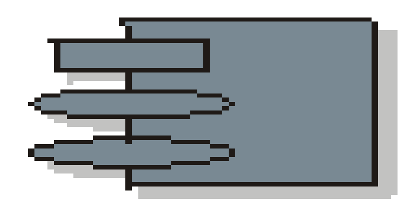


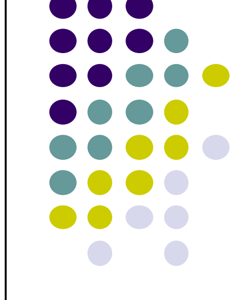
A **component** represents a piece of software code (source, binary or executable, relational schema), or a file containing information.

A component can also be an aggregate of other components (i.e., an application consisting of several executables can be a component).

Components may have stereotypes: <<component>>, <<subsystem>>, ….

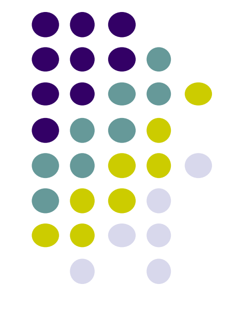
Components - examples





Examples of deliverable components	
Executables	.exe files
Load libraries	.dll files
Applets	.class for Java
Database tables	SQL scripts
Examples of components from which deliverables are produced	
Source code files	.h, .cpp and .hpp files for C++, CORBA IDL, or .java for Java
Binary files	.o files that are linked into executables

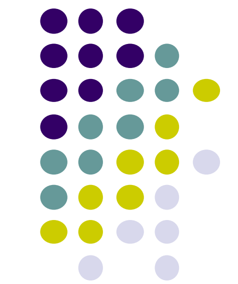
Components and packages



- Components are similar in practice to package diagrams, as they *define boundaries* and are used to *group elements into logical structures*.

- The difference between package diagrams and component diagrams is that Component Diagrams offer *a more semantically-rich grouping mechanism*.

Components – presentation and specification

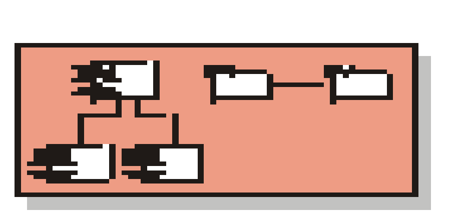
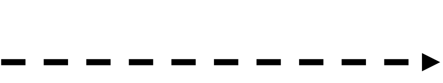


An interface circle attached to the component icon means that the component supports that particular interface. There is no explicit relationship arrow between a component and its interfaces.

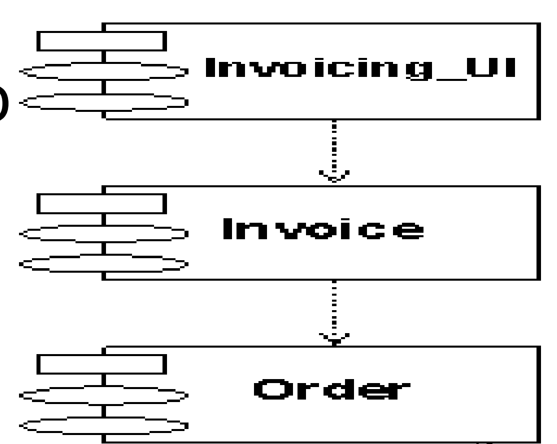
Component Specification contains tabs such as:

- General – ***stereotypes*** (Main Program, Package Body, Package Specification, Subprogram Body, Subprogram Specification, Task Body, and Task Specification) and ***language***
- Detail – ***declarations*** (as **`#Include`**)
- Realizes – classes building the component
- Files - attached files or URLs

Component Diagrams. Dependencies

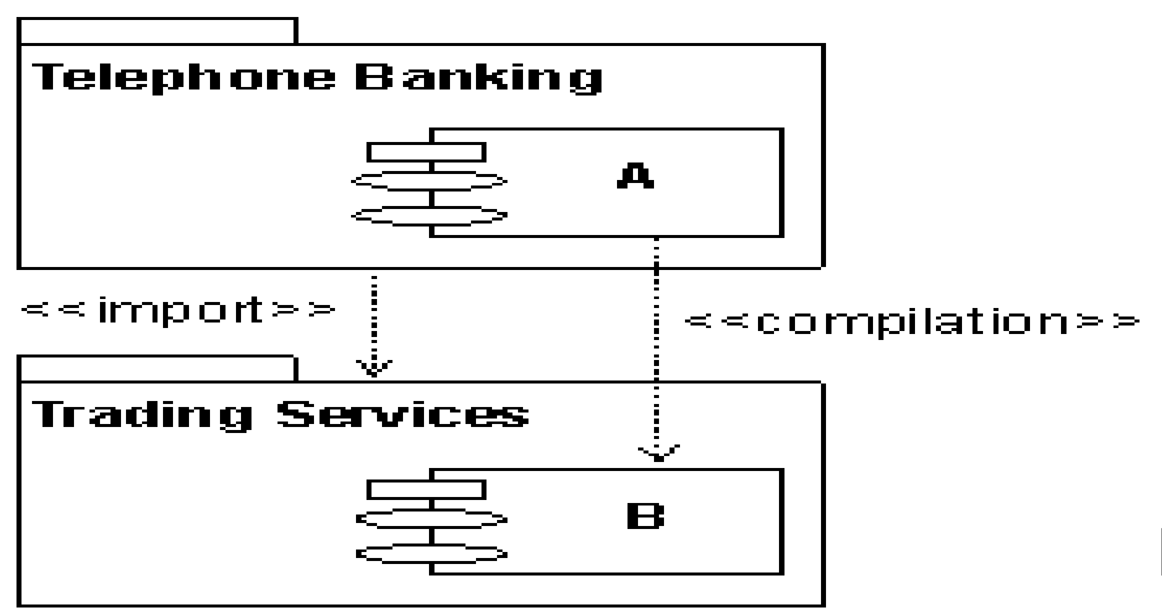
	A **component diagram** shows a collection of declarative (static) model elements, such as components, and implementation subsystems, and their relationships.
	A **dependency** from a component A to a component B indicates component A has a *compilation* dependency, or a *run-time* dependency to B.

A **compilation dependency** exists from one component to the components that are needed to compile the component (i.e., `#include` statements in C++, or `import` in Java).
Example: Invoicing_UI (the top), requires Invoice, which requires Order to compile.



Import Dependency Among Packages

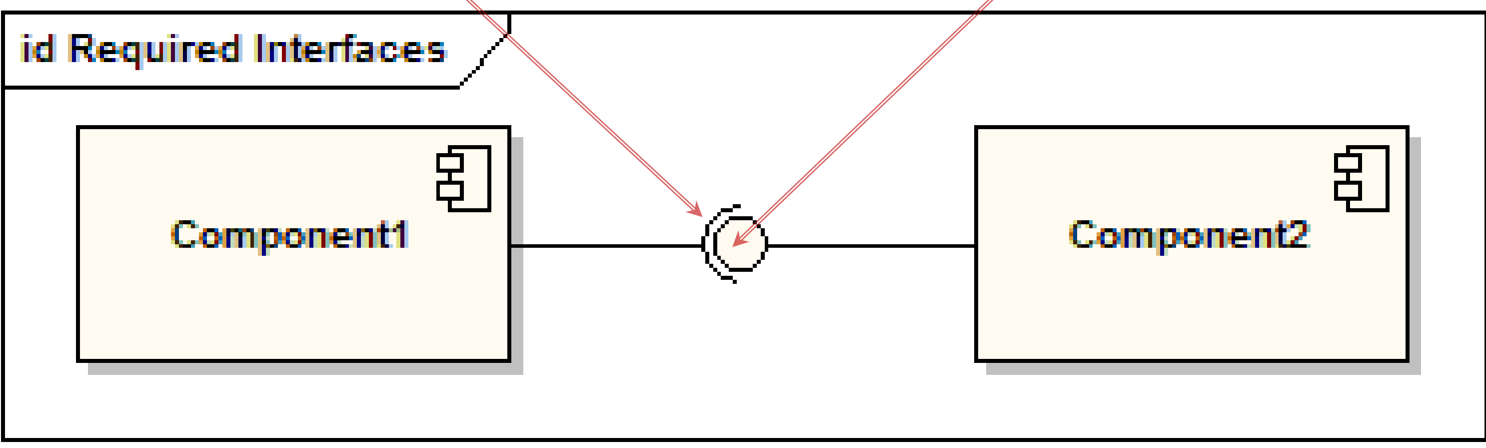
- An **import dependency** in the implementation model is a stereotyped dependency whose **source** is an *implementation subsystem* and whose **target** is another *implementation subsystem*.
- A component in a client subsystem can only compile against components in a supplier subsystem, if the client subsystem imports the supplier subsystem.

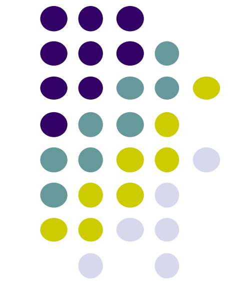


The subsystem Telephone Banking has an import dependency to the subsystem Trading Services, allowing components in Telephone Banking to compile against public (visible) components in Trading Services.

Assembly connectors (UML 2.*)

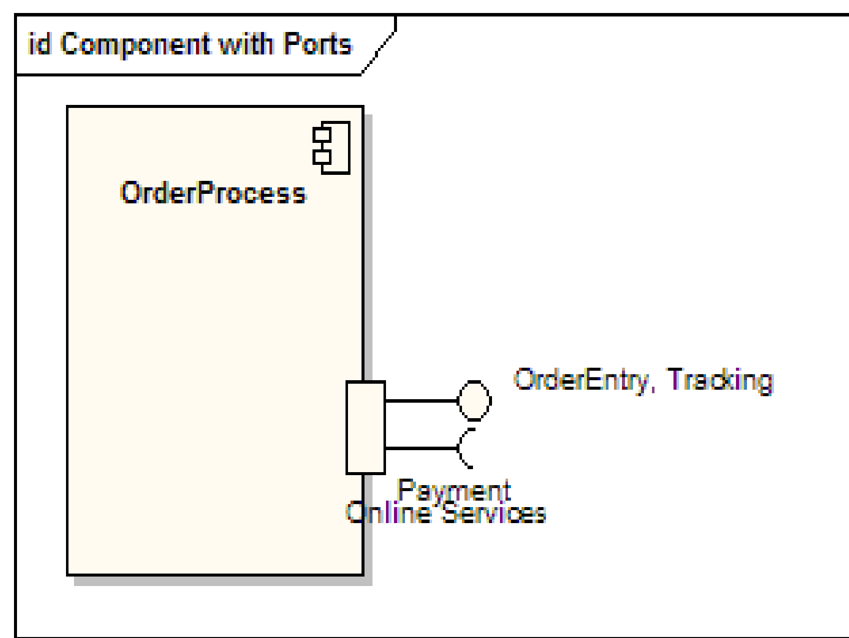
- The assembly connector bridges a component's *required interface* (Component1) with the *provided interface* of another component (Component2);
- The assembly connector allows one component to provide the services (the boll) that another component requires (the socket).



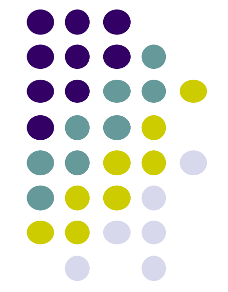


Components with *ports* (UML 2.*)

- ***Ports*** model related interfaces
- They allow for a service or behavior to be specified to its environment as well as a service or behavior that a component requires.
- Ports may specify inputs and outputs as they can operate bi-directionally.
- Example: a component with a port for online services along with *two provided interfaces order entry and tracking as well as a required interface payment.*



Focusing on the key components and interfaces



<<component>>
BroadcastEngine

FeedProvider

FeedProvider

<<component>>
ConversionManagement

DataSource

DataSource

<<component>>
BlogDataSource

<<component>>
BlogViewer

DisplayConverter

DisplayConverter

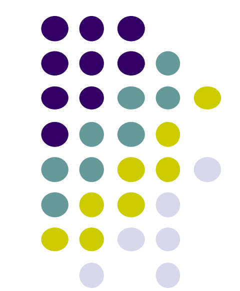
Logger

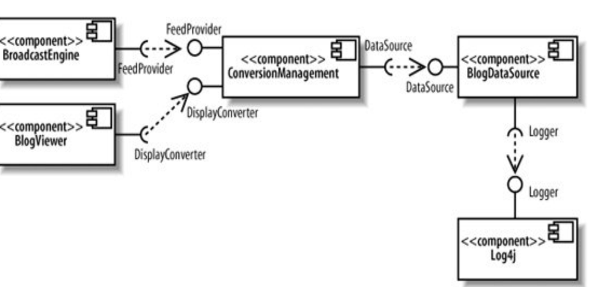
Logger

<<component>>
Log4j

Source: Learning UML 2.0,
by Kim Hamilton and Russell Miles,
O'Reilly 2006.

Focusing on component dependencies and manifesting interfaces





Source: Learning UML 2.0, by Kim Hamilton and Russell Miles, O’Reilly 2006.

<<component>>
BroadcastEngine
<<required interfaces>>
FeedProvider
<<artifacts>>
broadcastEngine.jar

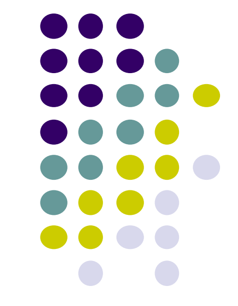
<<component>>
BlogViewer
<<required interfaces>>
DisplayConverter
<<artifacts>>
blogViewer.jar

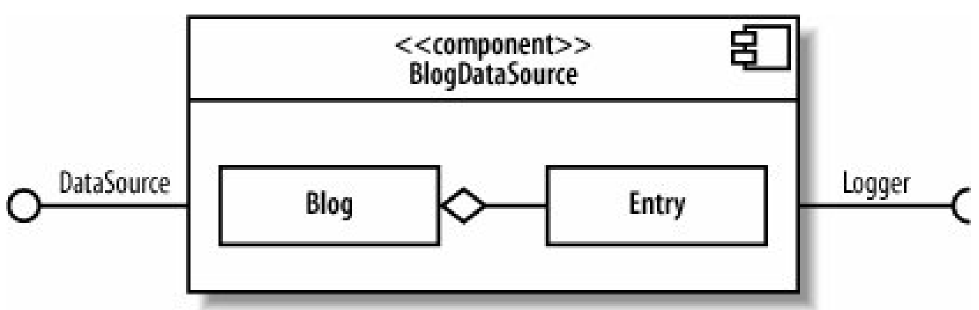
<<component>>
ConversionManagement
<<provided interfaces>>
FeedProvider
DisplayConverter
<<required interfaces>>
DataSource
<<artifacts>>
conversionManagement.jar

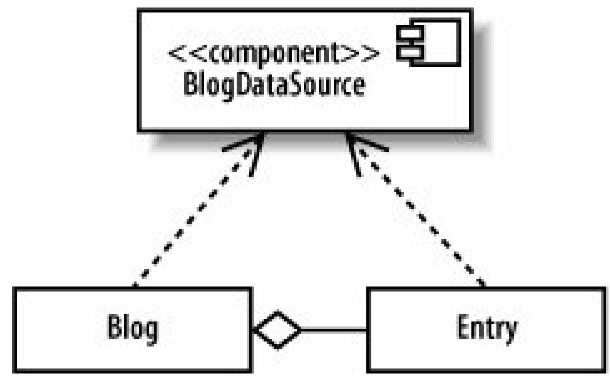
<<component>>
BlogDataSource
<<provided interfaces>>
DataSource
<<required interfaces>>
Logger
<<artifacts>>
blogdata.jar

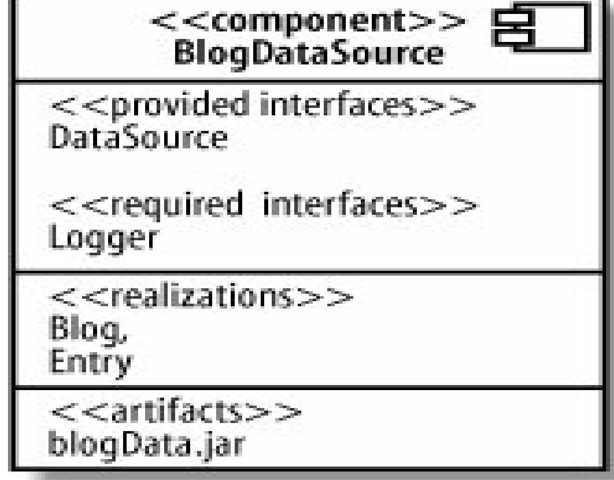
<<component>>
Log4j
<<provided interfaces>>
Logger
<<artifacts>>
log4j.jar

Classes realizing a component – alternative views









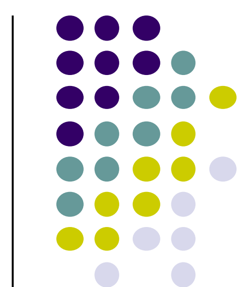
Case Study: ATM example (IBM Rose XDE)

- Use Case View
  - Use-Case Model
    - Actors
      - ATM Network
      - Customer
      - Operator
      - Associations
    - Use Cases
      - Conduct Transactions
        - Conduct Transactions
          - Local View - Conduct Trans
          - theCustomer (Customer)
          - theAuthenticate User (Auth
          - theATM Network (ATM Net
          - theConduct Transactions (
          - theConduct Transactions (
          - theCustomer Profile (Custo
        - Associations
      - Deposit Funds
      - Dispense Cash
        - Dispense Cash
          - Local View - Dispense Cas
          - theDispense Cash (Disper
        - Associations
      - Dispense Fuel
      - Dispense Value
      - Included Use Cases
      - Inquire into Balances
      - Maintain System
      - Transfer Funds
      - Architecturally Significant Use-Ca
      - Associations
    - Global View of Actors and Use Cases

Authenticate User
<<include>>
Customer
(from Actors)
Conduct Transactions
ATM Network
(from Actors)
Inquire into Balances
Deposit Funds
Dispense Value
Transfer Funds
Dispense Cash
Dispense Fuel
This use case is also abstract and represents a framework for exchanging value in an account for something that the system can dispense.
Operator
(from Actors)
Maintain System
ATM Network
(from Actors)

ATM Use Cases and Actors – Global View

ATM Example – Use Cases Local Views



Use Case Diagram: Conduct Transactions / Local View - Conduct Transactions

Created/modified by: <name>
Date: <creation/modification date>

Approved by: <name>
Date: <approval date>

Authenticate User

<<include>>

Customer

Conduct Transactions

ATM Network

Use Case Diagram: Dispense Cash / Local View - Dispense Cash

Created/modified by: <name>
Date: <creation/modification date>

Approved by: <name>
Date: <approval date>

Customer
(from Actors)

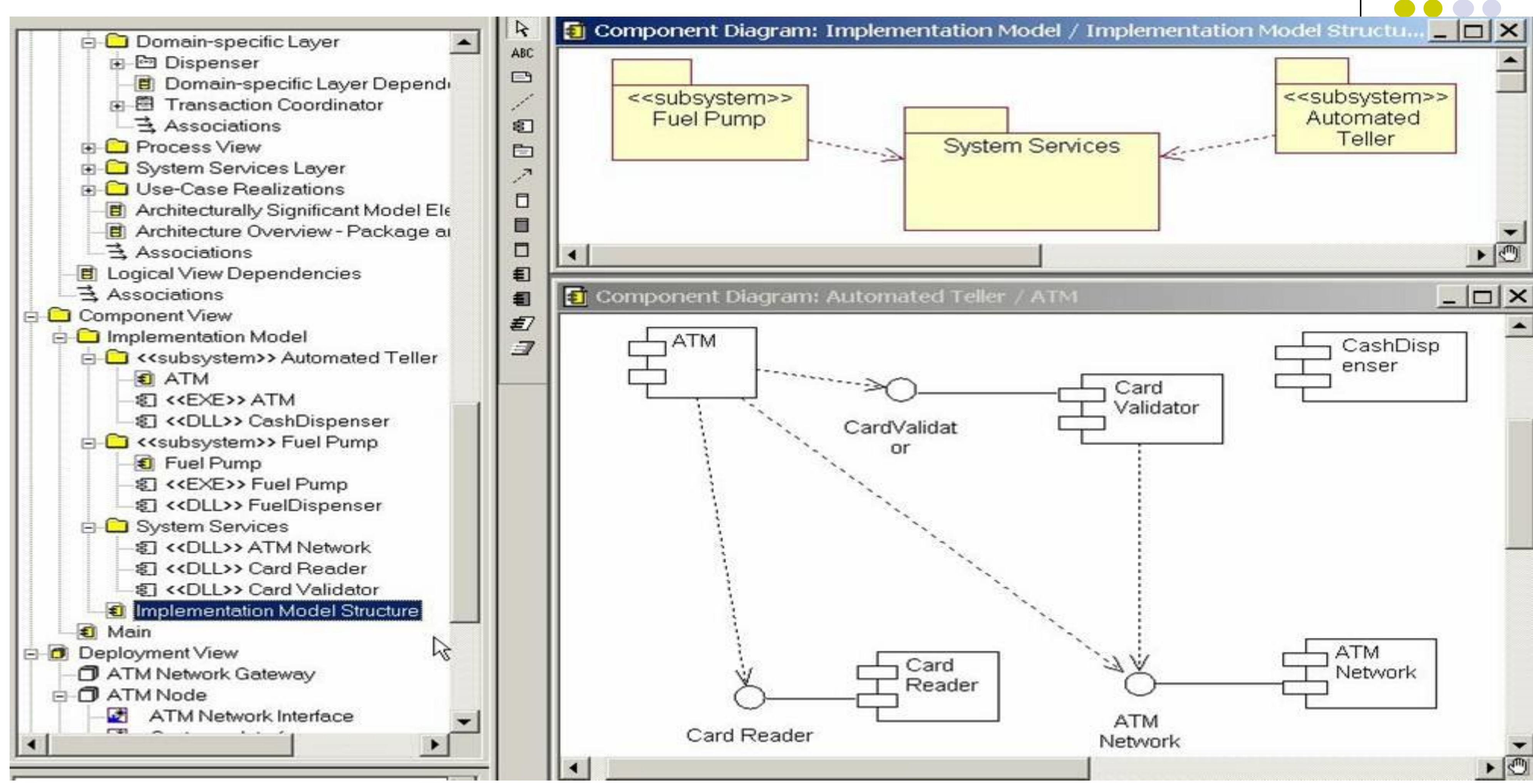
Conduct Transactions
(from Conduct Transactions)

ATM Network
(from Actors)

Dispense Value
(from Dispense Value)

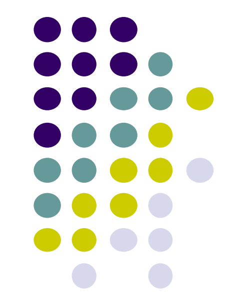
Dispense Cash

ATM Example – Component View



The implementation model is built by three subsystems

ATM Example: Component Specification



Component Diagram: Implementation Model / Implementation Model Structu...

Component Specification for Card Reader

General | Detail | Realizes | Files

Name: Card Reader

Stereotype: DLL

Language: Java

Documentati

DLL
EXE
Generic Package
Generic Subprogr
Main Program
Package Body
Package Specific

OK | Cancel | Apply | Browse | Help

<<subsystem>>
Automated
Teller

CashDisp
enser

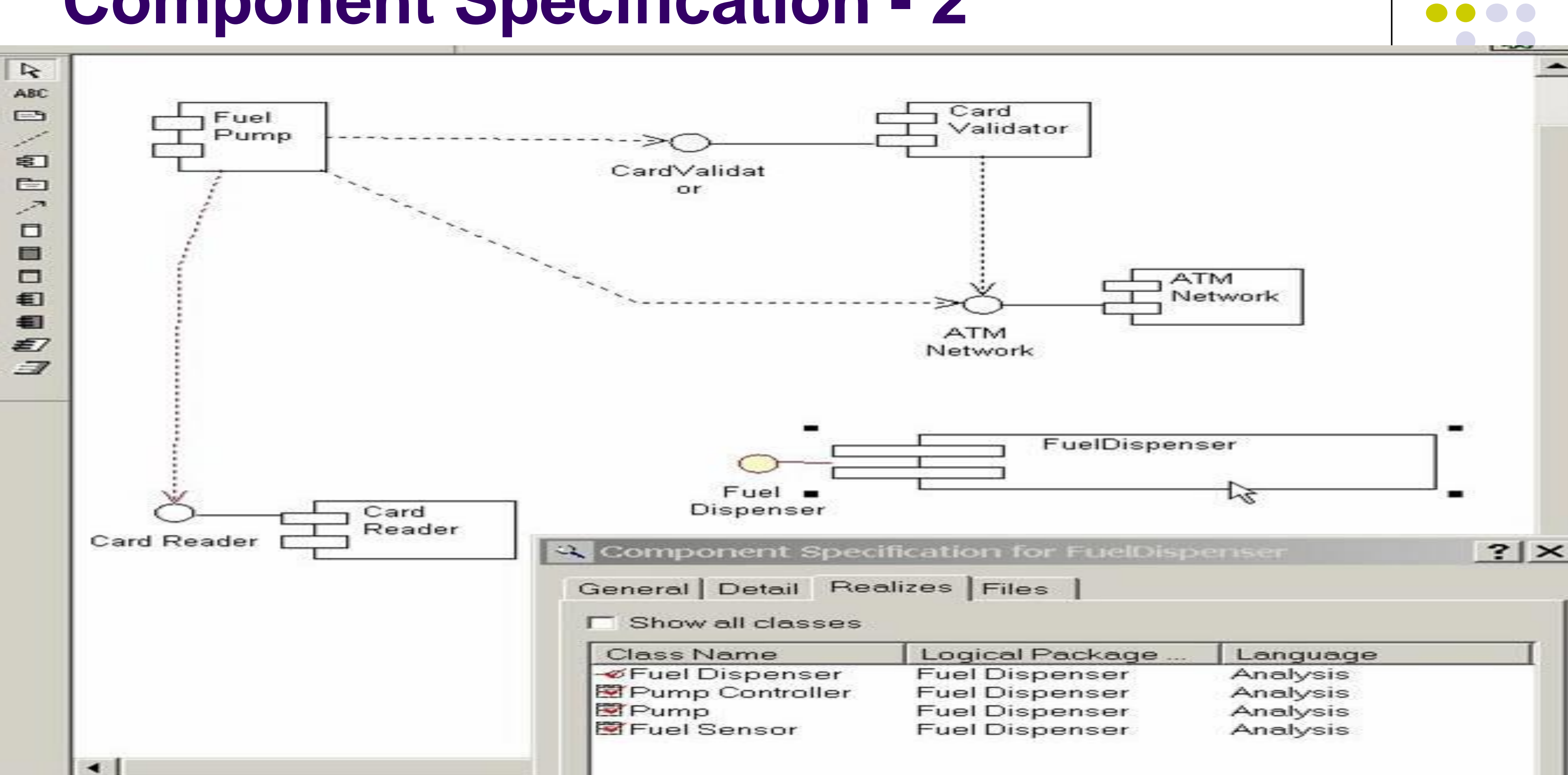
Card
Reader

Card Reader

ATM
Network

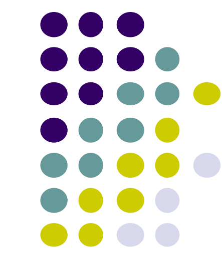
ATM
Network

ATM Example: Component Specification - 2



The Fuel Dispenser component realizes one interface and three classes

ATM Example: Class Specification shows Built Components



Fuel Sensor
fuelLevel : Number = 0
tankEmpty()
overFlow()

sensor
1..1

<<realize>>
controller

Pump Controller
pumpOn()

pump

Pump
on()
off()

1..1

1..1

Fuel Dispenser

Class Specification for Fuel Sensor

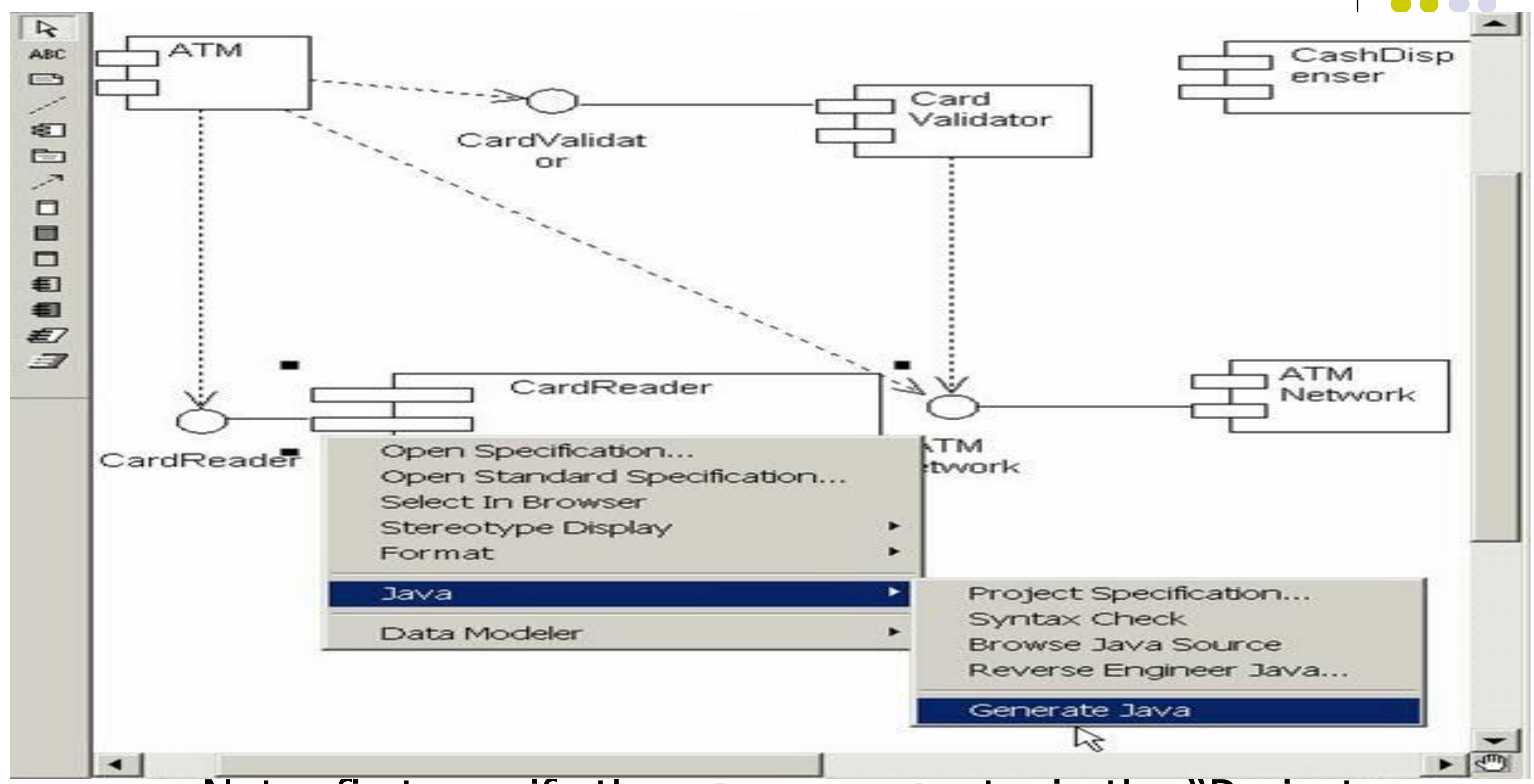
General | Detail | Operations | Attributes
Relations | Components | Nested | Files

☑ Show all components

Component Name	Package Name	Language
FuelDispenser	Fuel Pump	Analysis
ATM	Automated Teller	Analysis
CashDispenser	Automated Teller	Analysis
Fuel Pump	Fuel Pump	Analysis
Card Validator	System Services	Analysis
ATM Network	System Services	Analysis
Card Reader	System Services	Analysis

The Fuel Sensor is one of the classes building the Fuel Dispenser component

ATM Example: Code Generation



Note: first specify the `Classpath` etc. in the "Project Specification…" menu

Deployment Diagrams

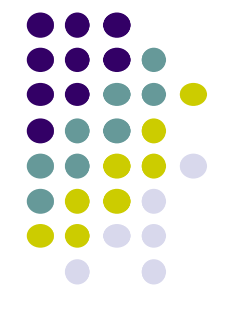
The **deployment architectural view** shows the configuration of run-time processing elements and the software processes living in them. **Deployment diagrams** are created to show the different nodes along with their connections in the system. They represent system topology and mapping executable subsystems to processors.

Issues concerned:

- processor architecture
- speed
- inter-process communication and synchronization
- etc.

A deployment diagram shows processors, devices, and connections. Each model contains a single deployment diagram which shows the connections between its processors and devices, and the allocation of its processes to processors.

Nodes

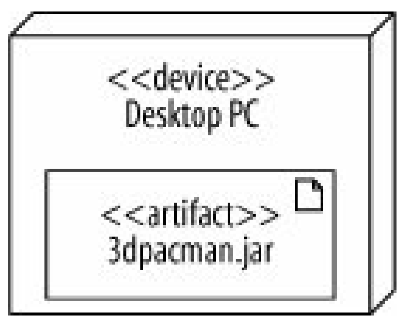


A *node* is a hardware or software resource that can host software or related files. You can think of a software node as an application context; generally not part of the software you developed, but a third-party environment that provides services to your software

hardware nodes	execution environment nodes
Server Desktop PC Disk drives	Operating system J2EE container Web server Application server

Artifacts within nodes

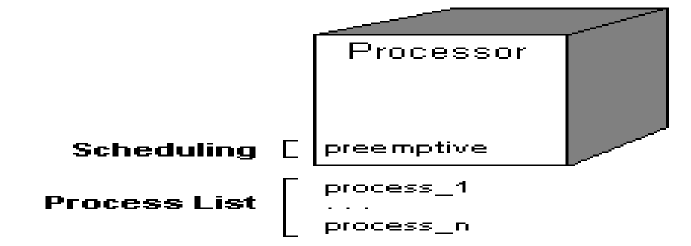
- Drawing an artifact inside a node shows that the artifact is deployed to the node
- But where is JVM? ->



- Your deployment diagrams should contain details about your system that are important to your audience. If it is important to show the hardware, firmware, operating system, runtime environments, or even device drivers of your system, then you should include these in your deployment diagram.

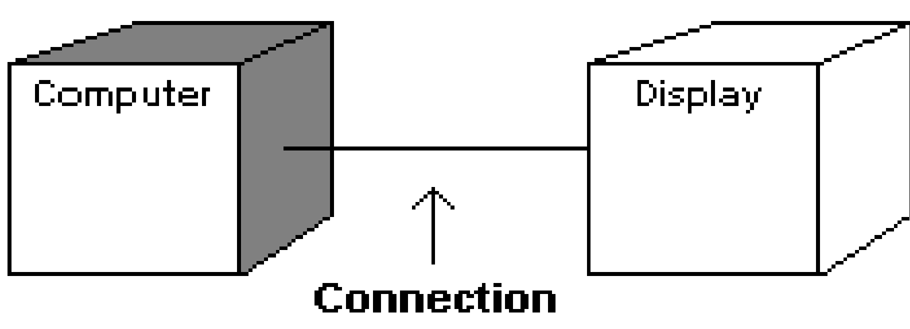
Processors, Devices and Connections

Processor - identify its processes and specify the type of process scheduling (preemptive, non-preemptive, cyclic, executive, manual).



Device – in some models: a hardware component with no or restricted computing power (i.e., "modem" or "terminal"); in others: specialization of node.

Connection - represents some type of hardware coupling between two entities. An entity is either a processor or a device. The hardware coupling can be direct, such as an RS232 cable, or indirect, such as satellite-to-ground communication.



ATM Example: Deployment Diagram

<<subsystem>> Auto
ATM
<<EXE>> ATM
<<DLL>> CashDis
<<subsystem>> Fuel
Fuel Pump
<<EXE>> Fuel Pur
<<DLL>> FuelDisp
System Services
<<DLL>> ATM Ne
<<DLL>> Card Re
<<DLL>> Card Va
Implementation Mode
Main
Deployment View
ATM Network Gateway
ATM Node
ATM Network Interfa
Customer Interface
Device Controller
ATM Main
Fuel Pump
Card Reader
Cash Dispenser
Display
Keypad
Log Device
Network Interface
Pump
Receipt Printer
Model Properties

ABC

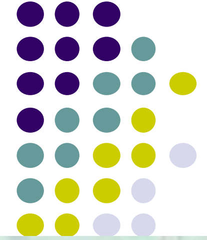
Log Device
Cash Dispenser
Display
Receipt Printer
ATM Node
Keypad
Processor ATM Node
Card Reader
Network Interface
preemptive
ATM Main
Customer Interface
ATM Network Interface
Device Controller

Processor:
* 200 Mhz Pentium
Memory:
* 64 Mb

T-1 network connection

ATM Network Gateway
preemptive
Network Interface

UML deployment diagram [Bruegge & Dutoit]



myMac:Mac

:Safari

:UnixHost

:WebServer

aPC:PC

:IExplorer

:UnixHost

:Database

Figure 7-2 A UML deployment diagram representing the allocation of components to different nodes and the dependencies among components. Web browsers on PCs and Macs can access a WebServer that provides information from a Database.

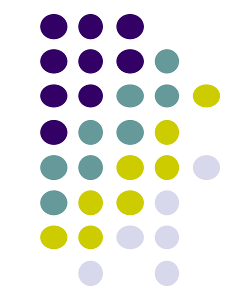
The refined diagram [Bruegge & Dutoit]

The deployment diagram in Figure 7-2 focuses on the allocation of components to nodes and provides a high-level view of each component. Components can be refined to include information about the interfaces they provide and the classes they contain. Figure 7-3 illustrates the GET and POST interfaces of the WebServer component and its containing classes.

WebServer
URI
DBQuery
HttpRequest
DBResult
File
GET
POST

Figure 7-3 Refined view of the WebServer component (UML deployment diagram). WebServer provides two interfaces: a browser can either GET the content of a file referred by a URL or POST a form.

The MyTrip example [Bruegge & Dutoit]

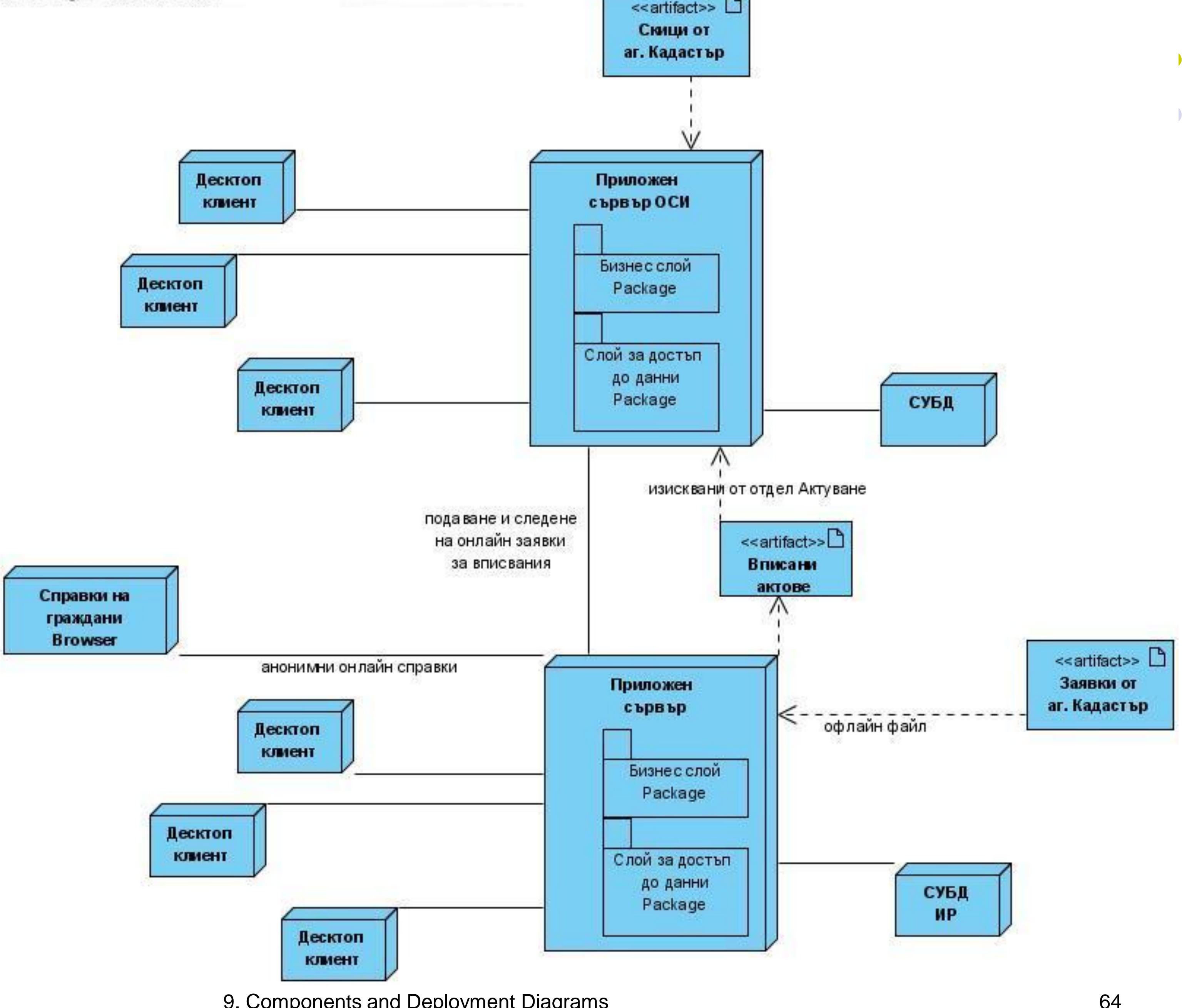


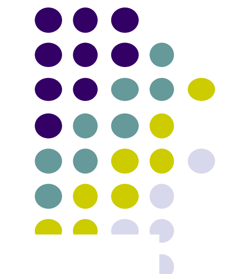
In MyTrip, we deduce from the requirements that PlanningSubsystem and RoutingSubsystem run on two different nodes: the former is a Web-based service on an Internet host, the latter runs on the onboard computer. Figure 7-4 illustrates the hardware allocation for MyTrip with two nodes called :OnBoardComputer and :WebServer.

:OnBoardComputer
RoutingSubsystem
:WebServer
PlanningSubsystem

Figure 7-4 Allocation of MyTrip subsystems to hardware (UML deployment diagram). RoutingSubsystem runs on the OnBoardComputer; PlanningSubsystem runs on a WebServer.

A real example





UML 2.x Diagrams

Diagram

Behaviour Diagram

Structure Diagram

Activity Diagram

State Machine Diagram

Interaction Diagram

Use Case Diagram

Communication Diagram

Interaction Overview Diagram

Sequence Diagram

Timing Diagram

Class Diagram

Component Diagram

Object Diagram

Composite Structure Diagram

Deployment Diagram

Package Diagram

Profile Diagram