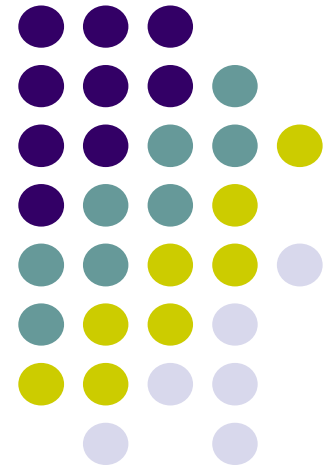
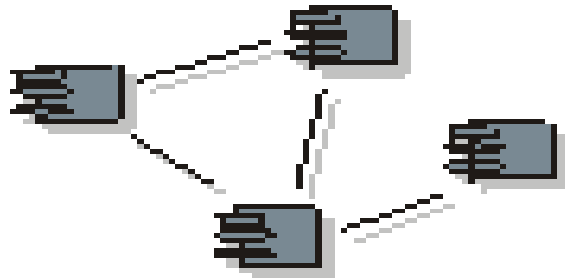
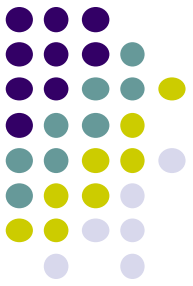


Components and Deployment Diagrams

Components and Component Packages
Dependencies
Processors and Devices
Connections
Examples



The Implementation Model in the Component View



An **implementation model** is a collection of components, and the implementation subsystems which contain them.

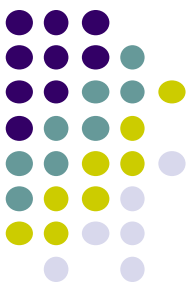
A **component** diagram has a higher level of abstraction than a Class Diagram - usually a **component** is implemented by one or more classes (or objects at runtime).



Components

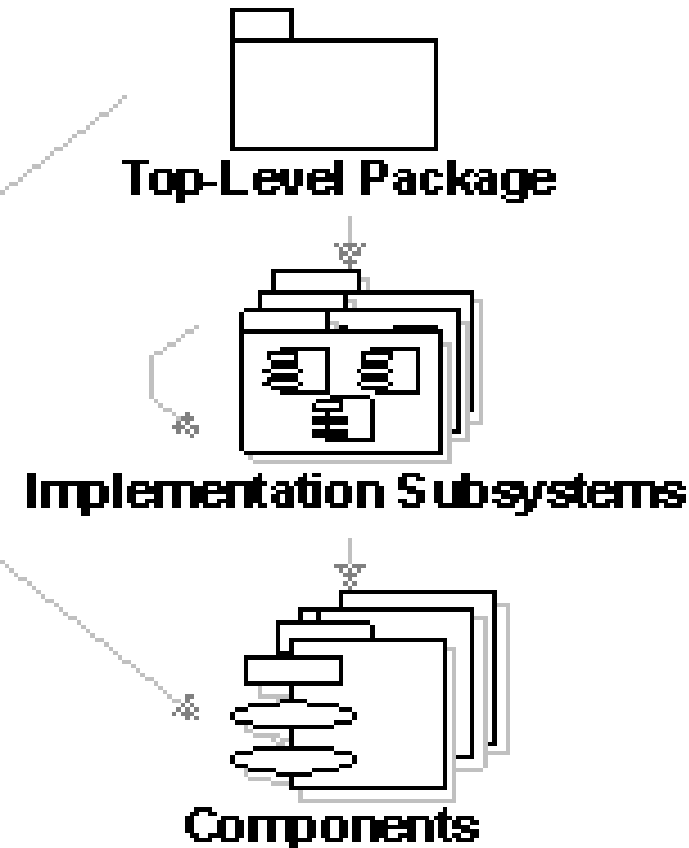
- Components are building blocks so a component can eventually encompass a large portion of a system.
- Components include both deliverable components, such as executables, and components from which the deliverables are produced, such as source code files.

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.

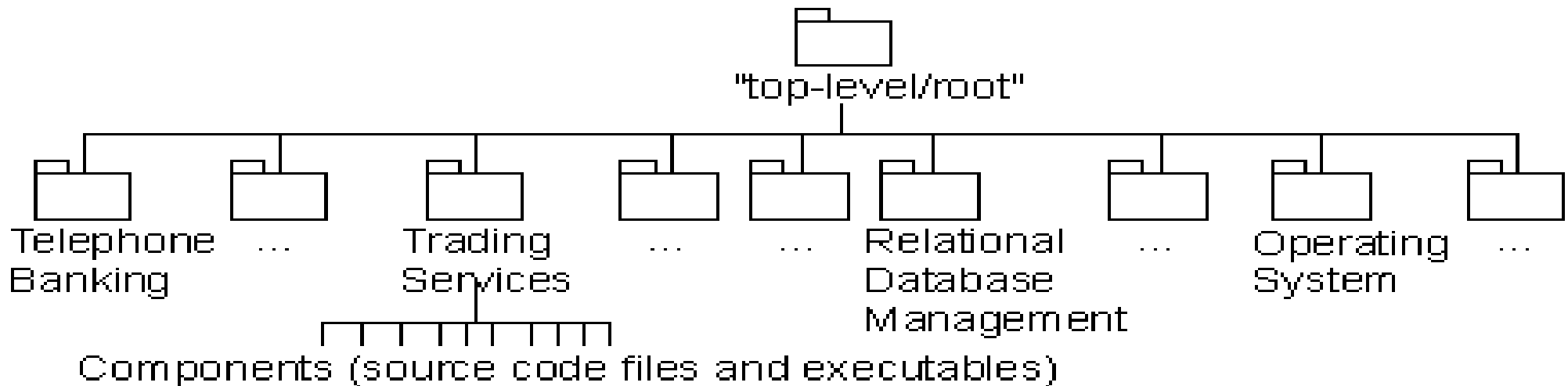
The implementation model can be divided into components that are ***deliverables***, such as executables that are delivered to customers; and those ***components from which the deliverables are produced***, such as source code.





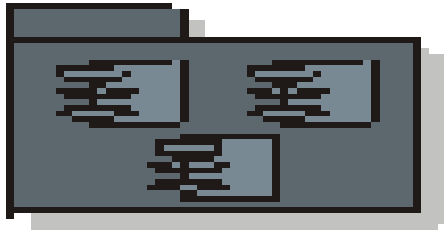
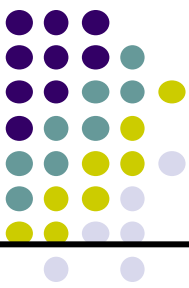
Implementation Model - Example

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.

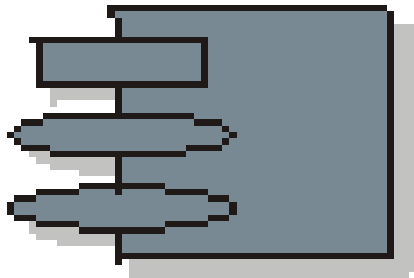
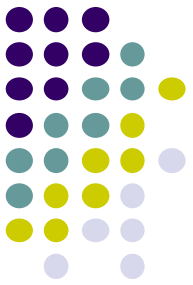


An **implementation subsystem** is a collection of components and other implementation subsystems that are used to structure the implementation model by dividing it into smaller parts.

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

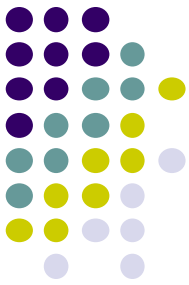


A **component** represents a piece of software code (source, binary or executable, relational schema), or a file containing information.

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

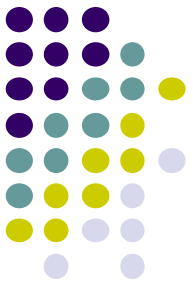
Source code files	.h, .cpp and .hpp files for C++, CORBA IDL, or .java for Java
Binary files	.o files that are linked into executables



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



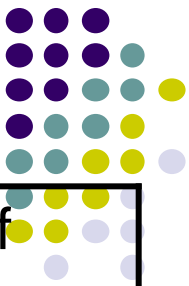
An interface circle attached to the component icon means that the component supports that particular interface. There is no explicit relationship arrow between a component and its interfaces.

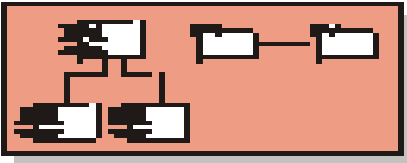



Component Specification contains tabs such as:

- 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

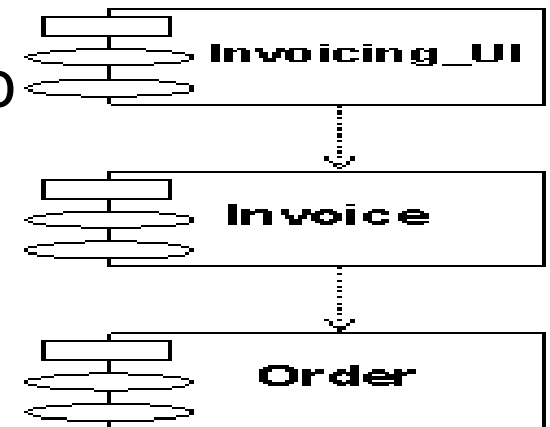
Component Diagrams. Dependencies

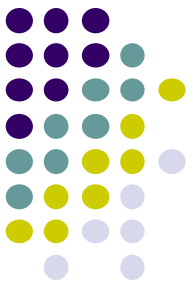


	A component diagram shows a collection of declarative (static) model elements, such as components, and implementation subsystems, and their relationships.
	A dependency from a component A to a component B indicates component A has a <u>compilation</u> dependency, or a <u>run-time</u> dependency to B.

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).

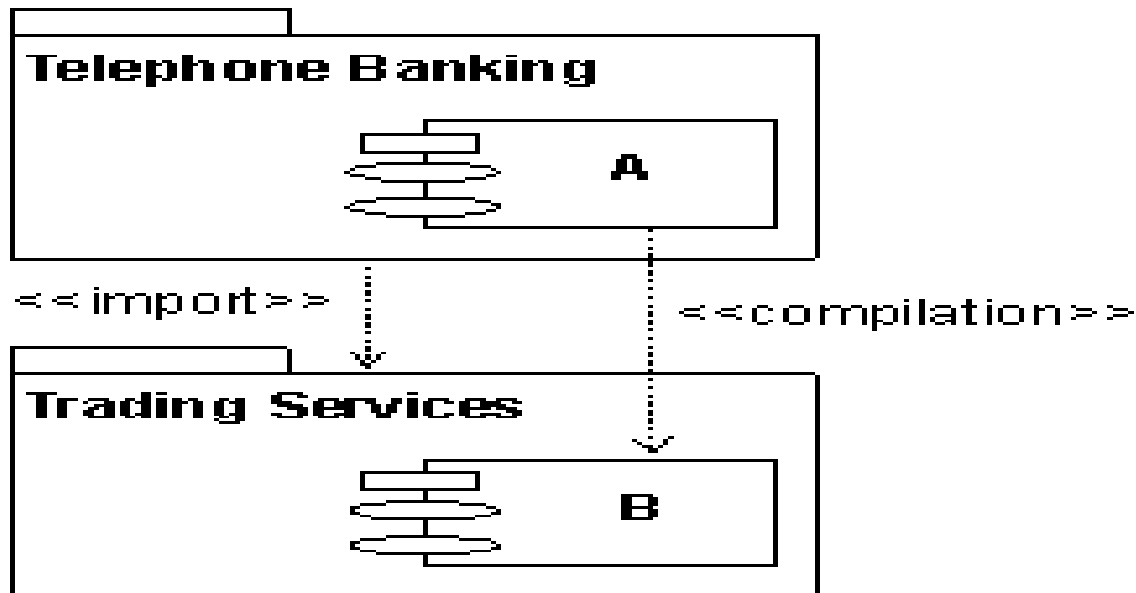
Example: Invoicing_UI (the top), requires Invoice, which requires Order to compile.



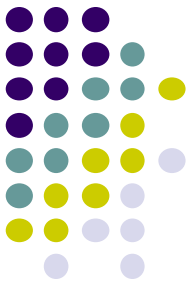


Import Dependency Among Packages

- 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.

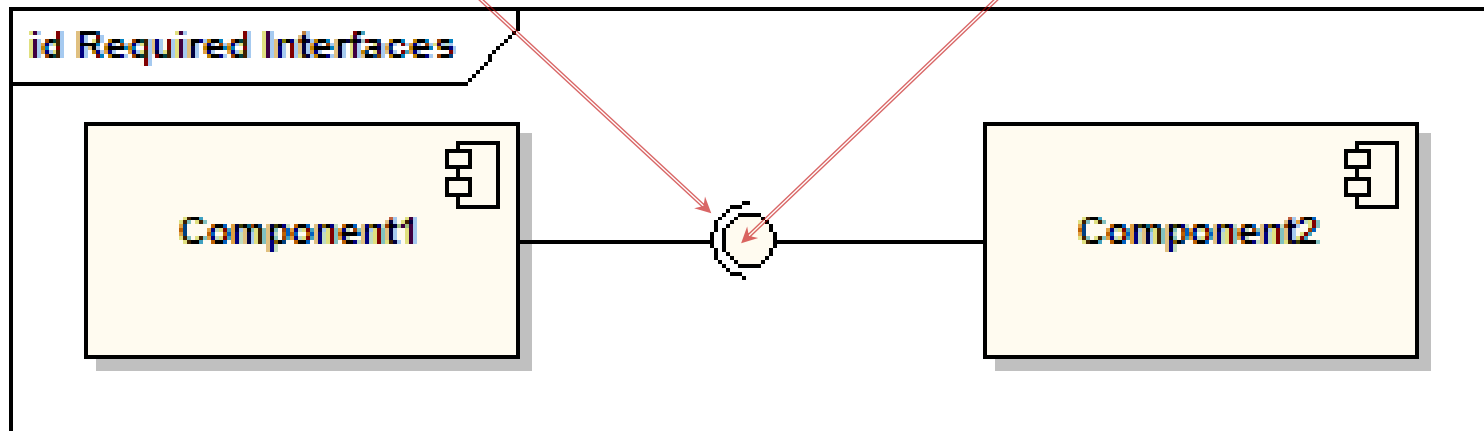


The subsystem Telephone Banking has an import dependency to the subsystem Trading Services, allowing components in Telephone Banking to compile against public (visible) components in Trading Services.



Assembly connectors (UML 2.*)

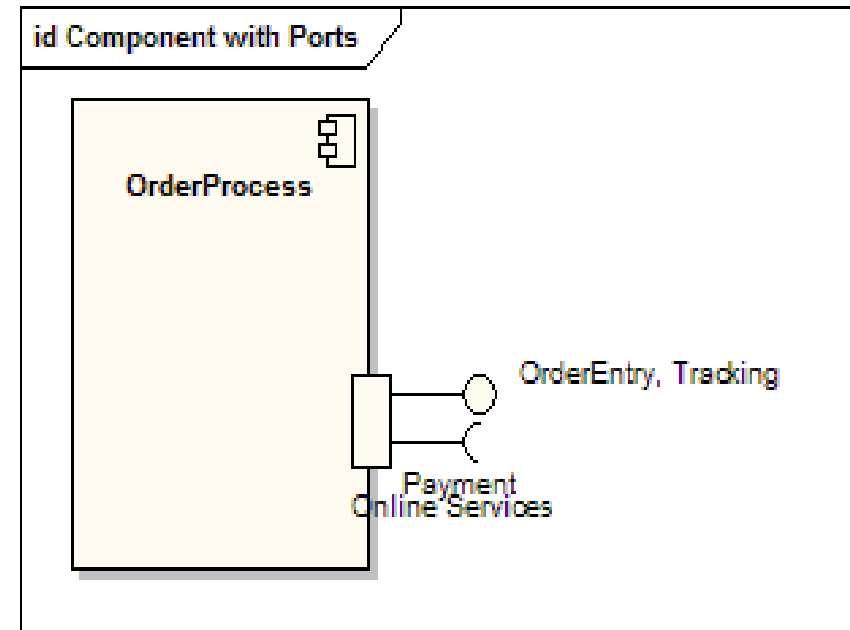
- The assembly connector bridges a component's required interface (Component1) with the provided interface of another component (Component2);
- The assembly connector allows one component to provide the services (the boll) that another component requires (the socket).



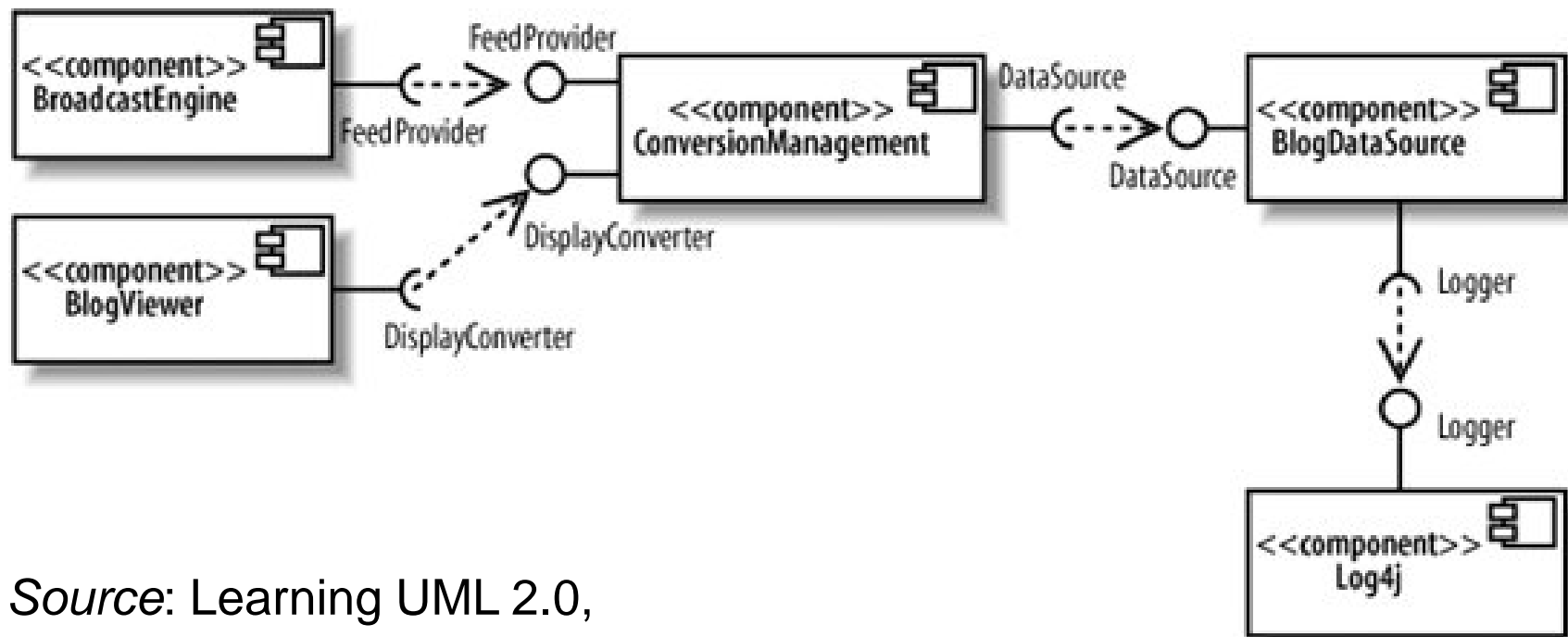
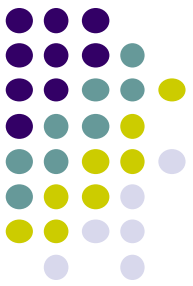


Components with *ports* (UML 2.*)

- **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*.

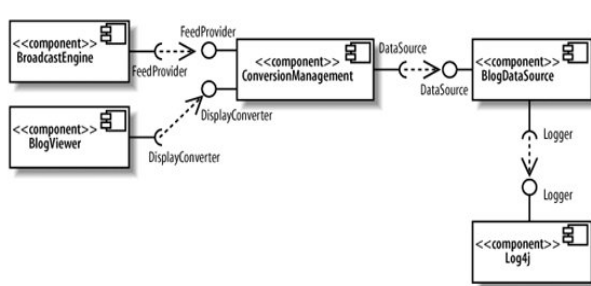
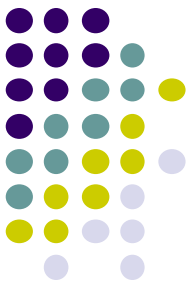


Focusing on the key components and interfaces

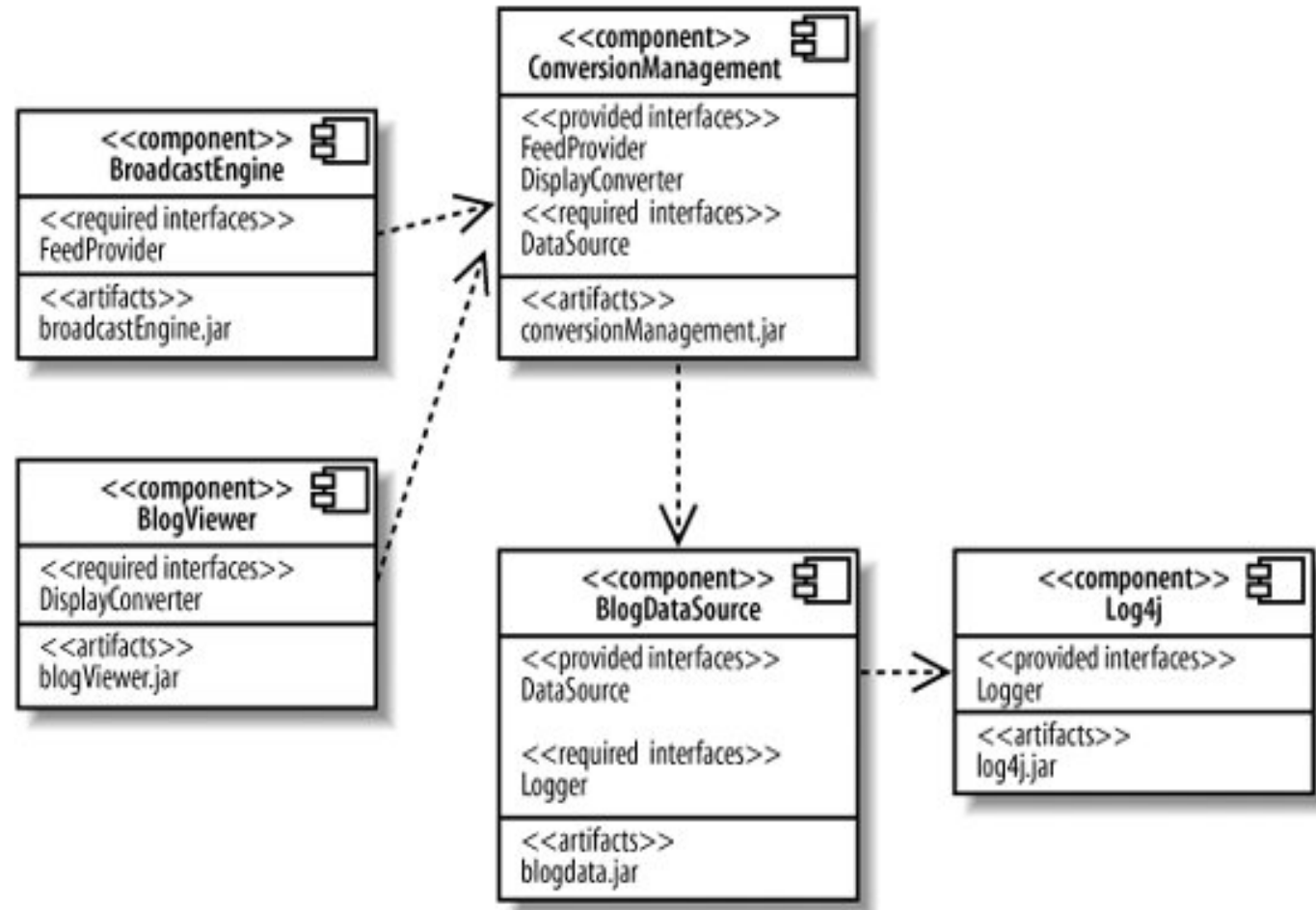


Source: Learning UML 2.0,
by Kim Hamilton and Russell Miles,
O'Reilly 2006.

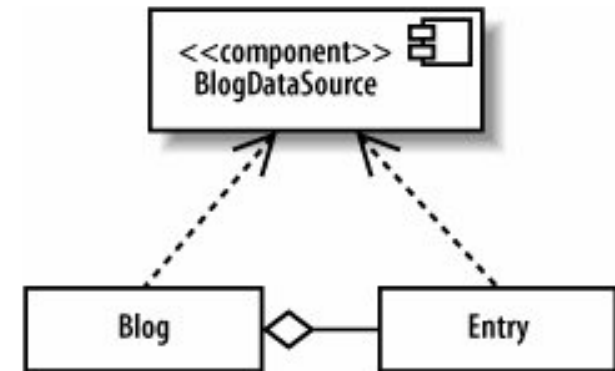
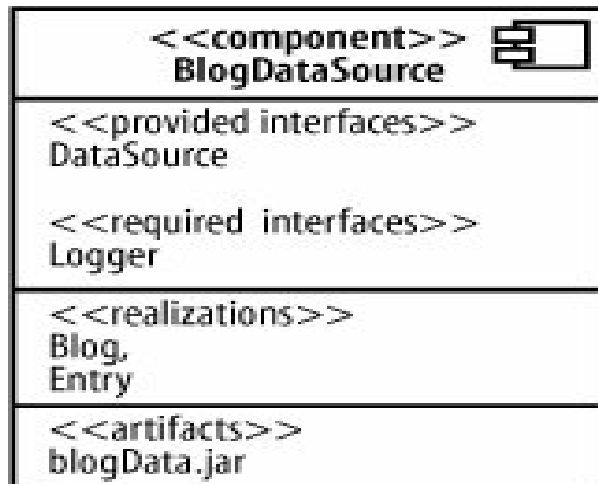
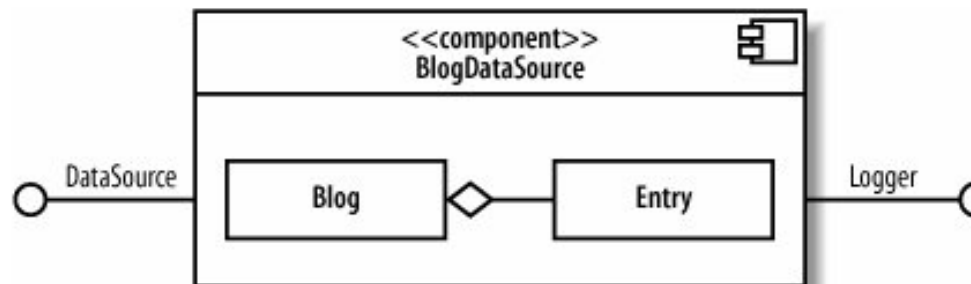
Focusing on component dependencies and manifesting interfaces



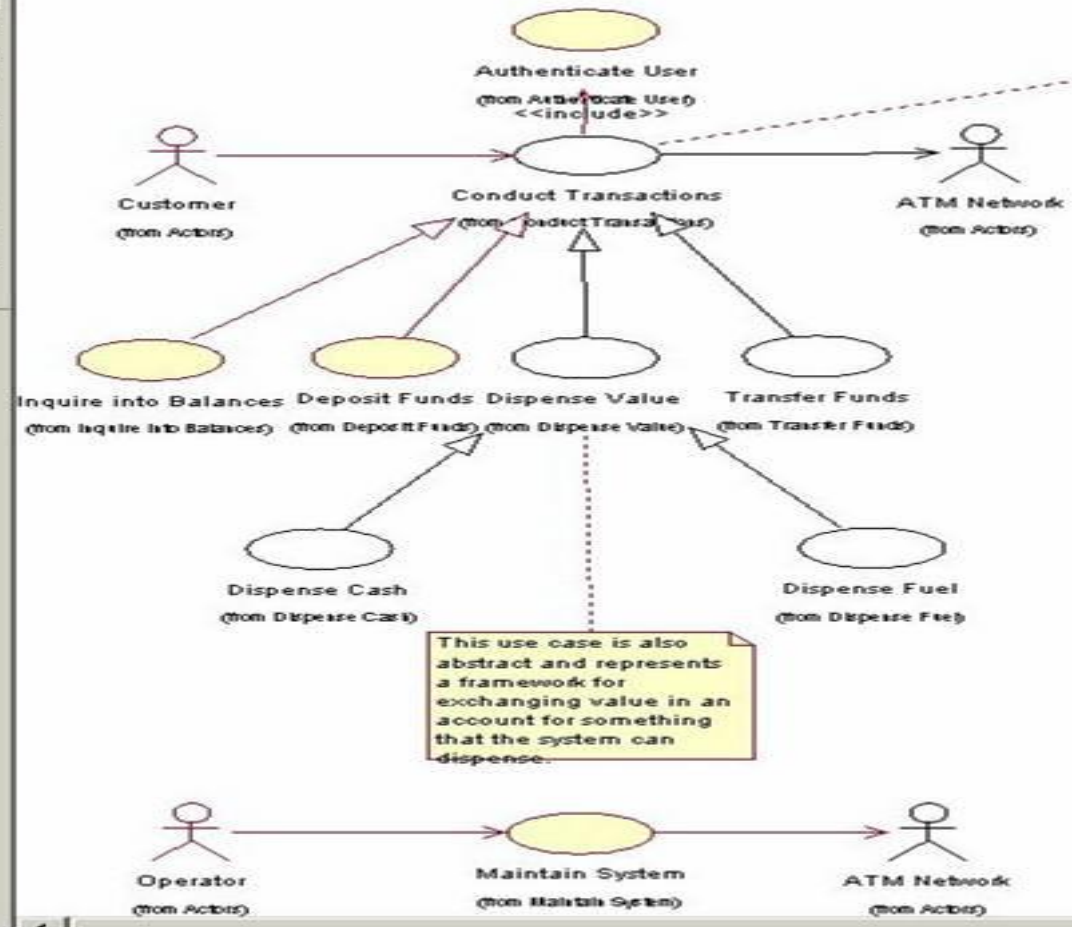
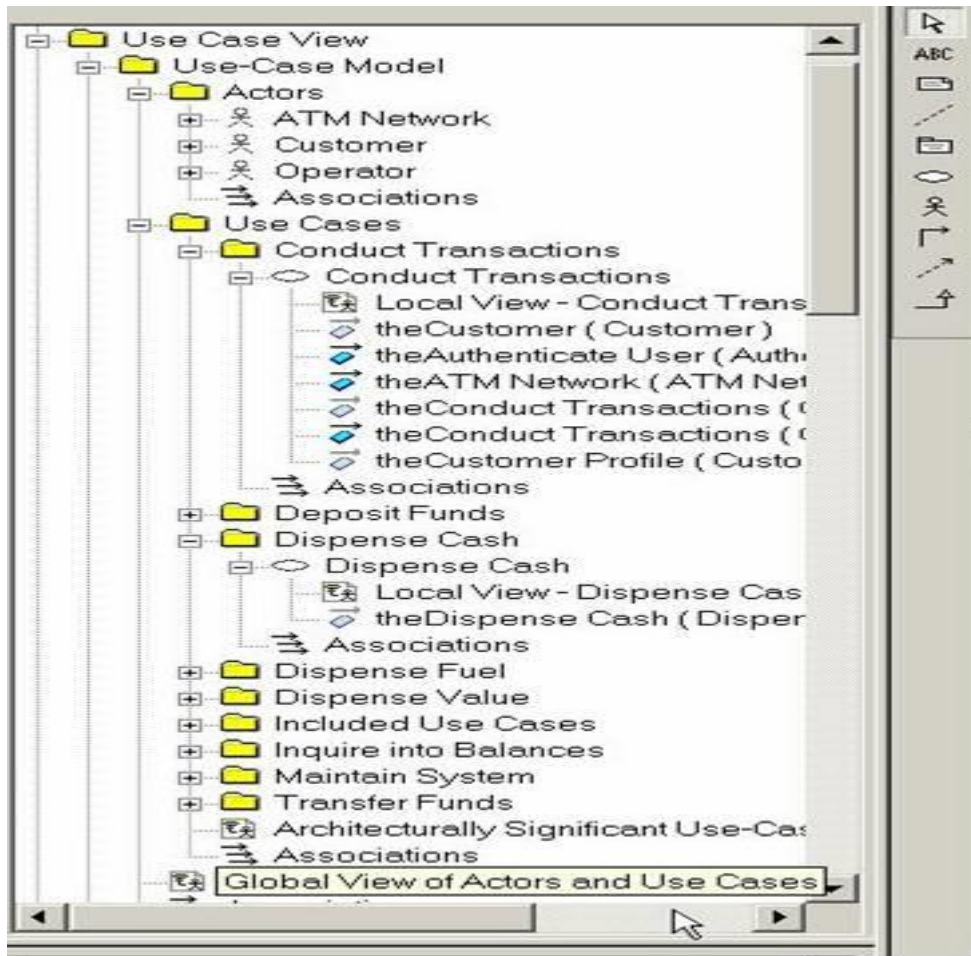
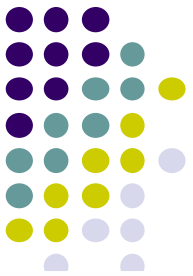
Source: Learning UML 2.0,
by Kim Hamilton and
Russell Miles,
O'Reilly 2006.



Classes realizing a component – alternative views

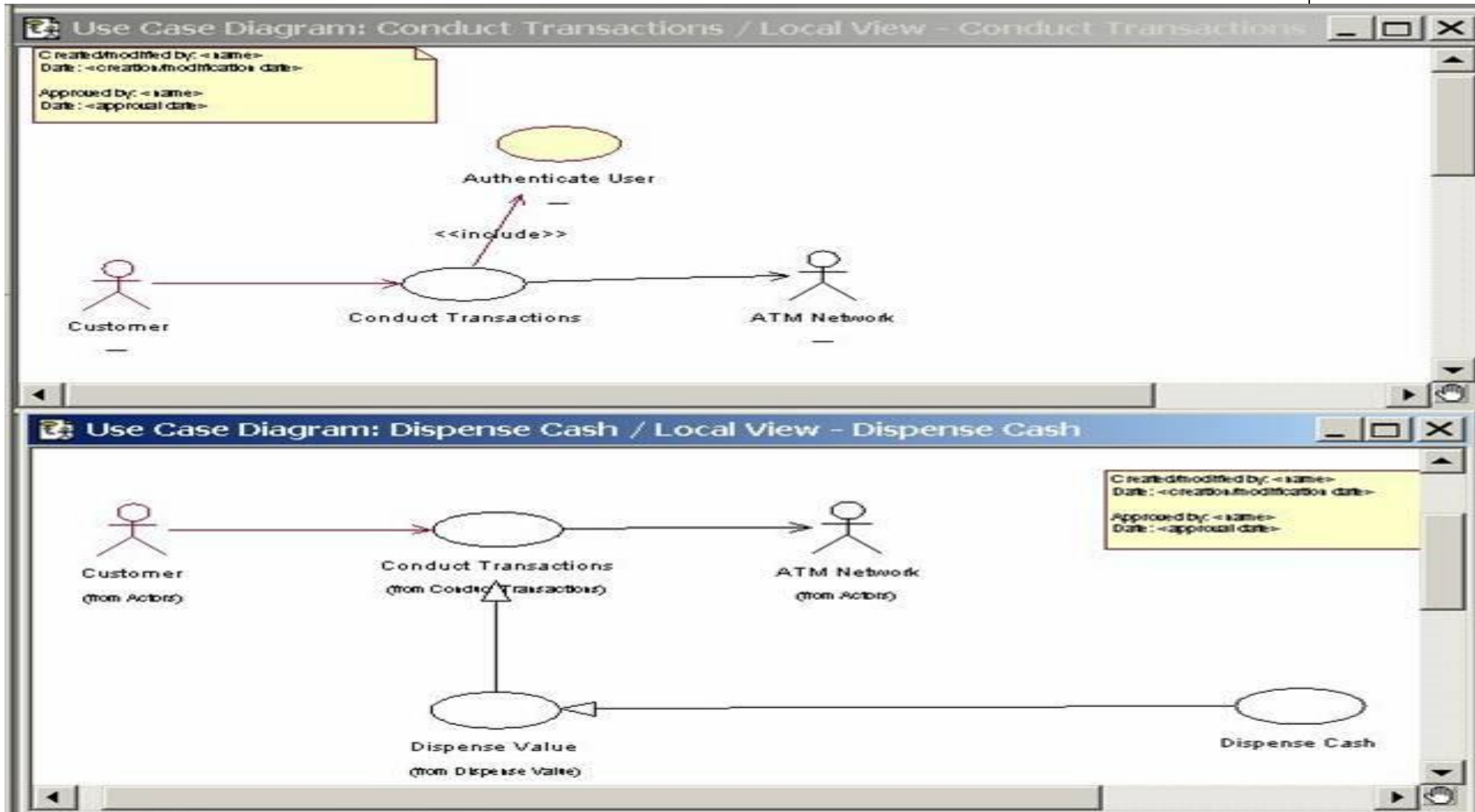
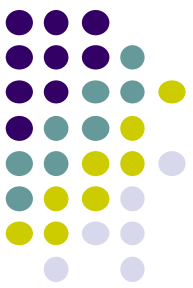


Case Study: ATM example (IBM Rose XDE)

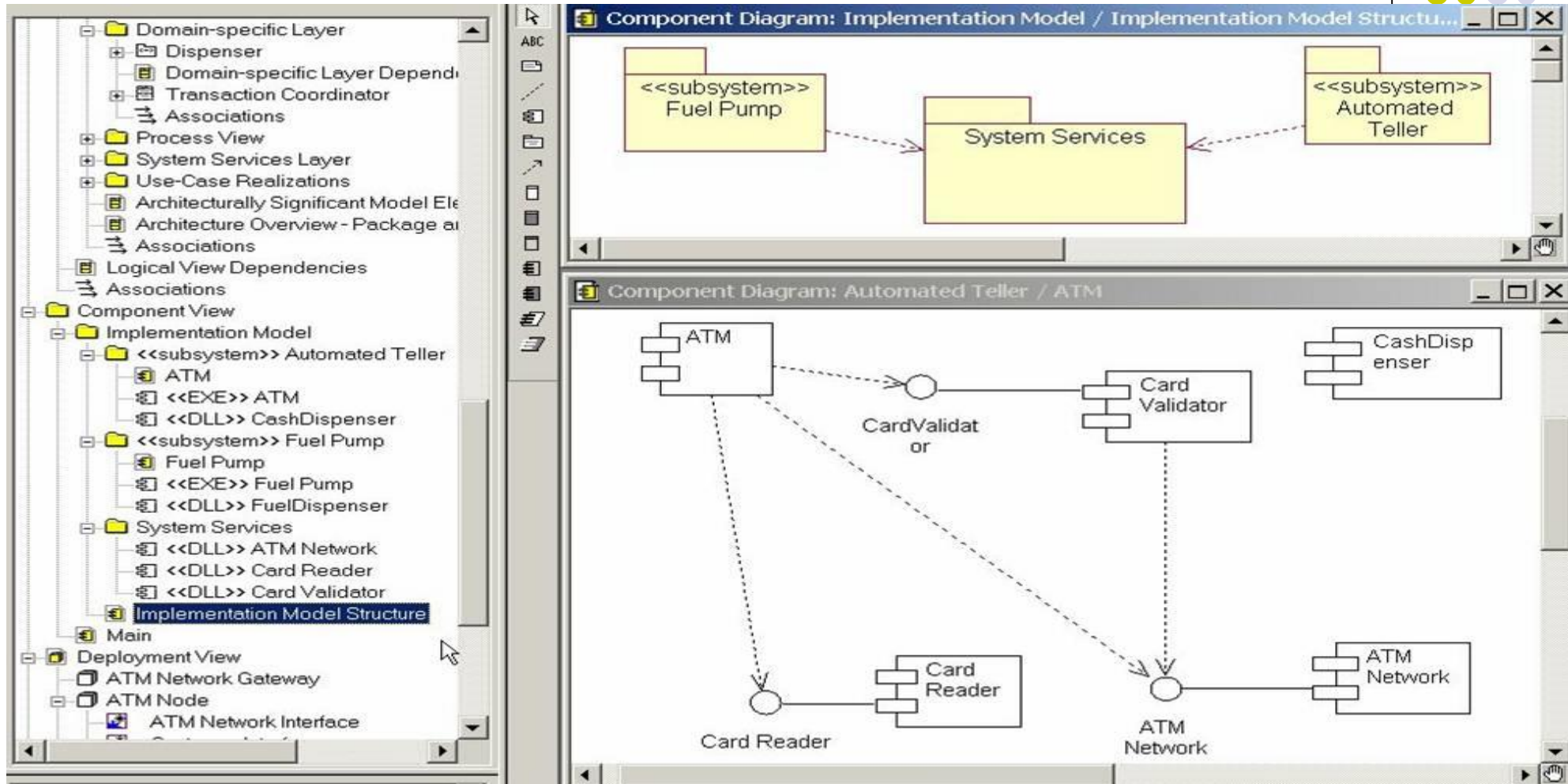


ATM Use Cases and Actors – Global View

ATM Example – Use Cases Local Views

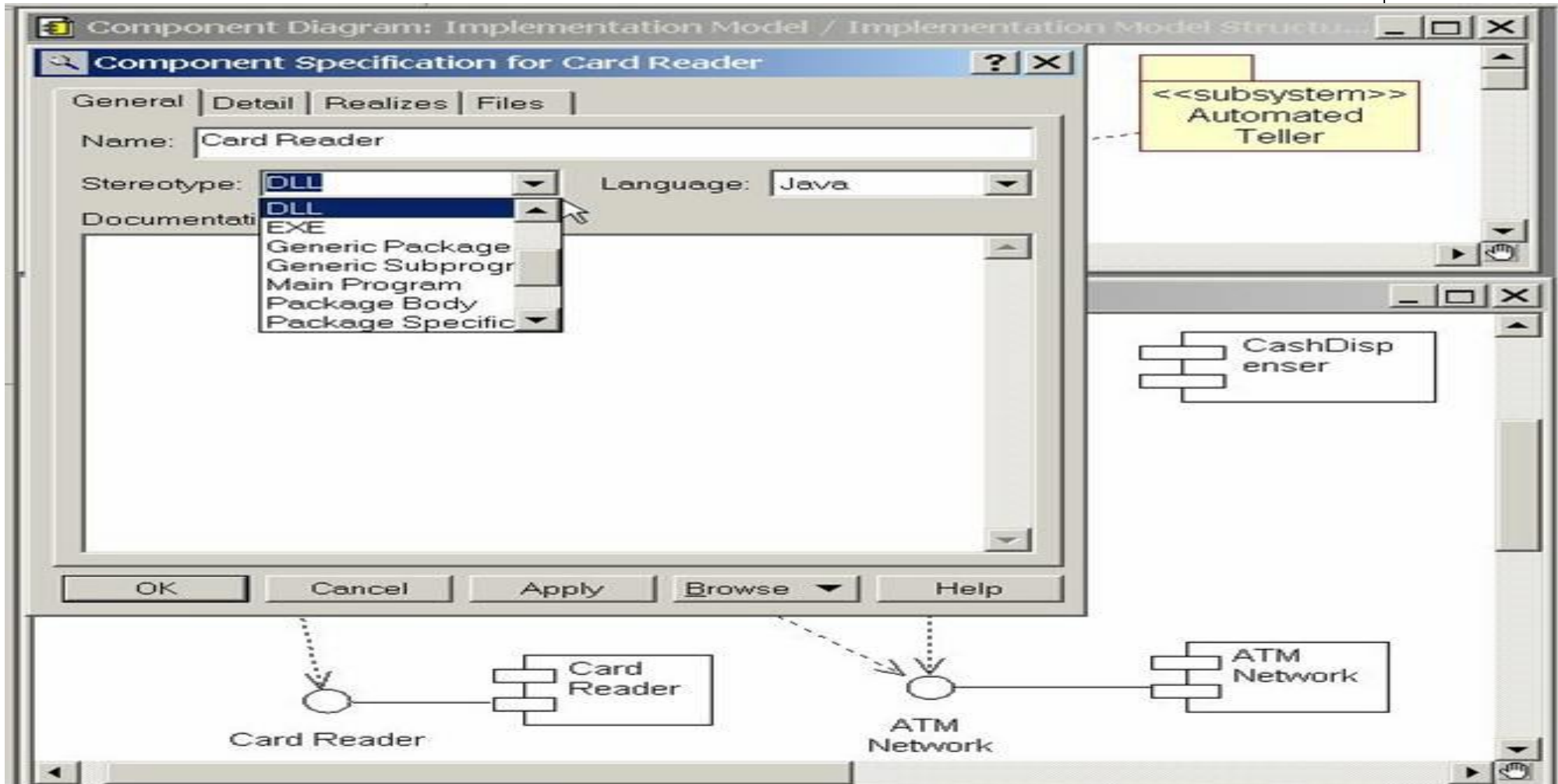
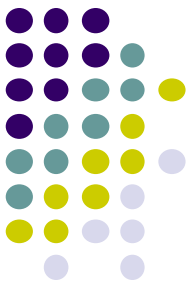


ATM Example – Component View

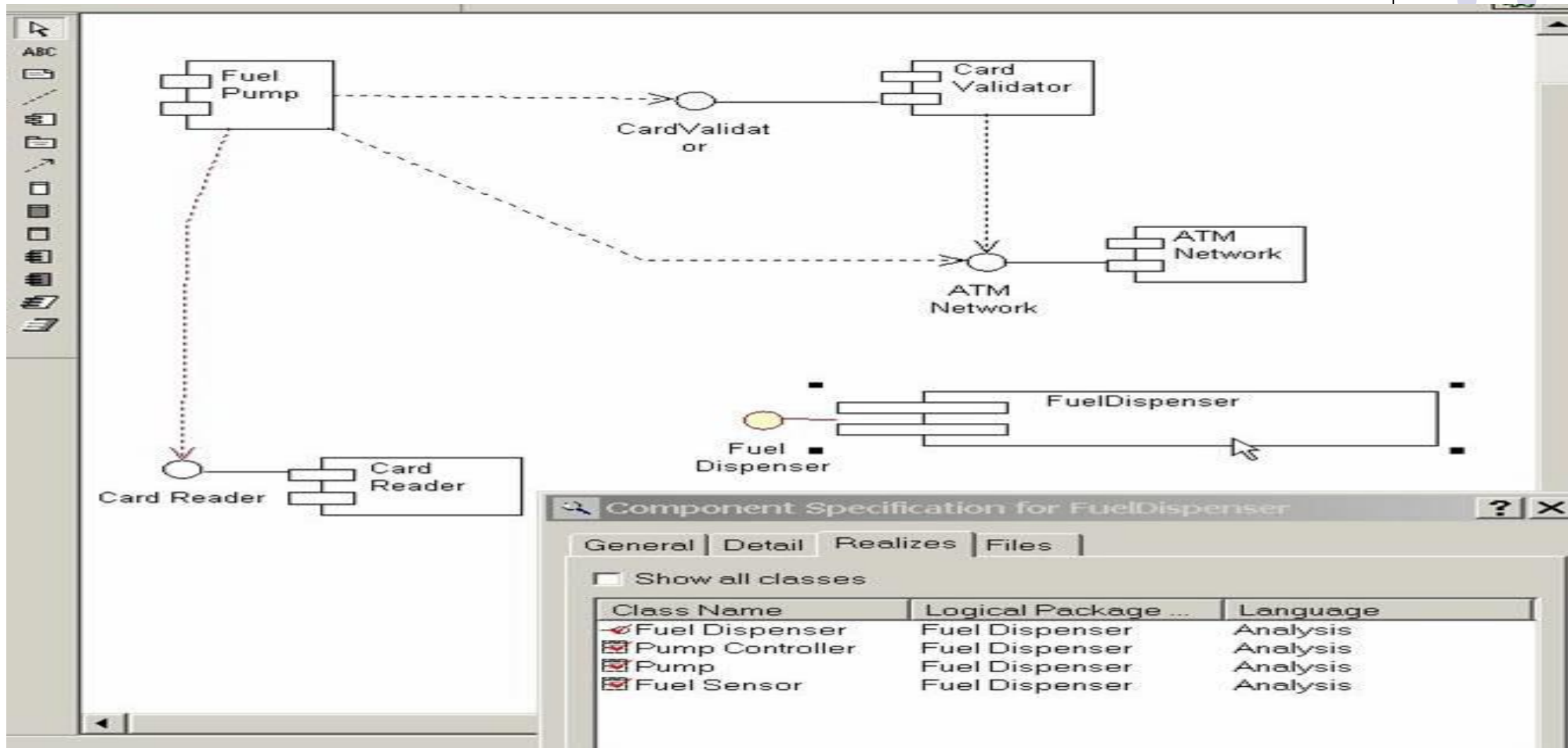
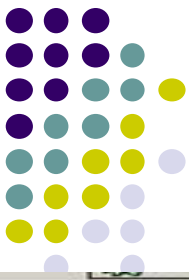


The implementation model is built by three subsystems

ATM Example: Component Specification

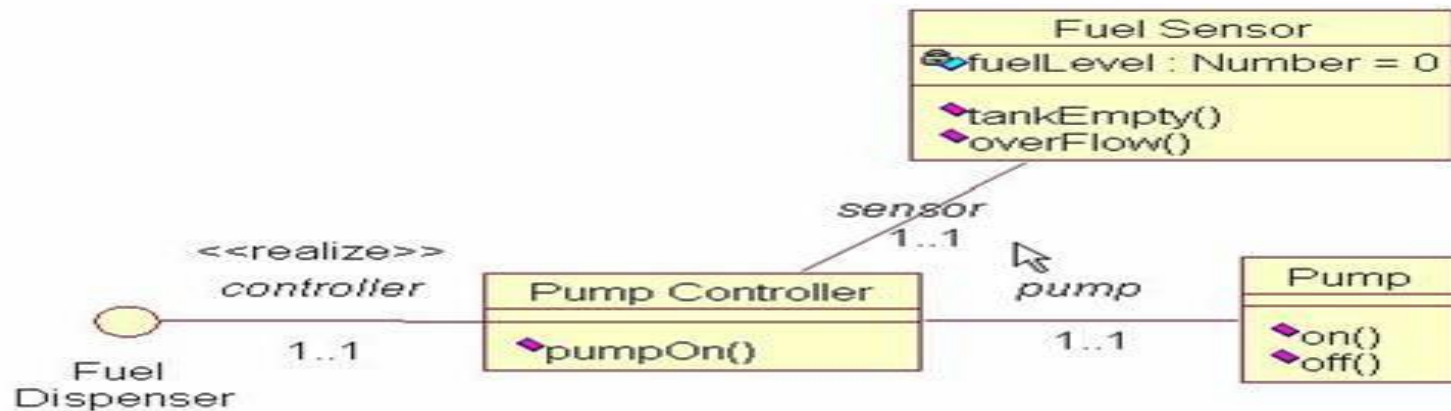


ATM Example: Component Specification - 2



The Fuel Dispenser component realizes one interface and three classes

ATM Example: Class Specification shows Built Components

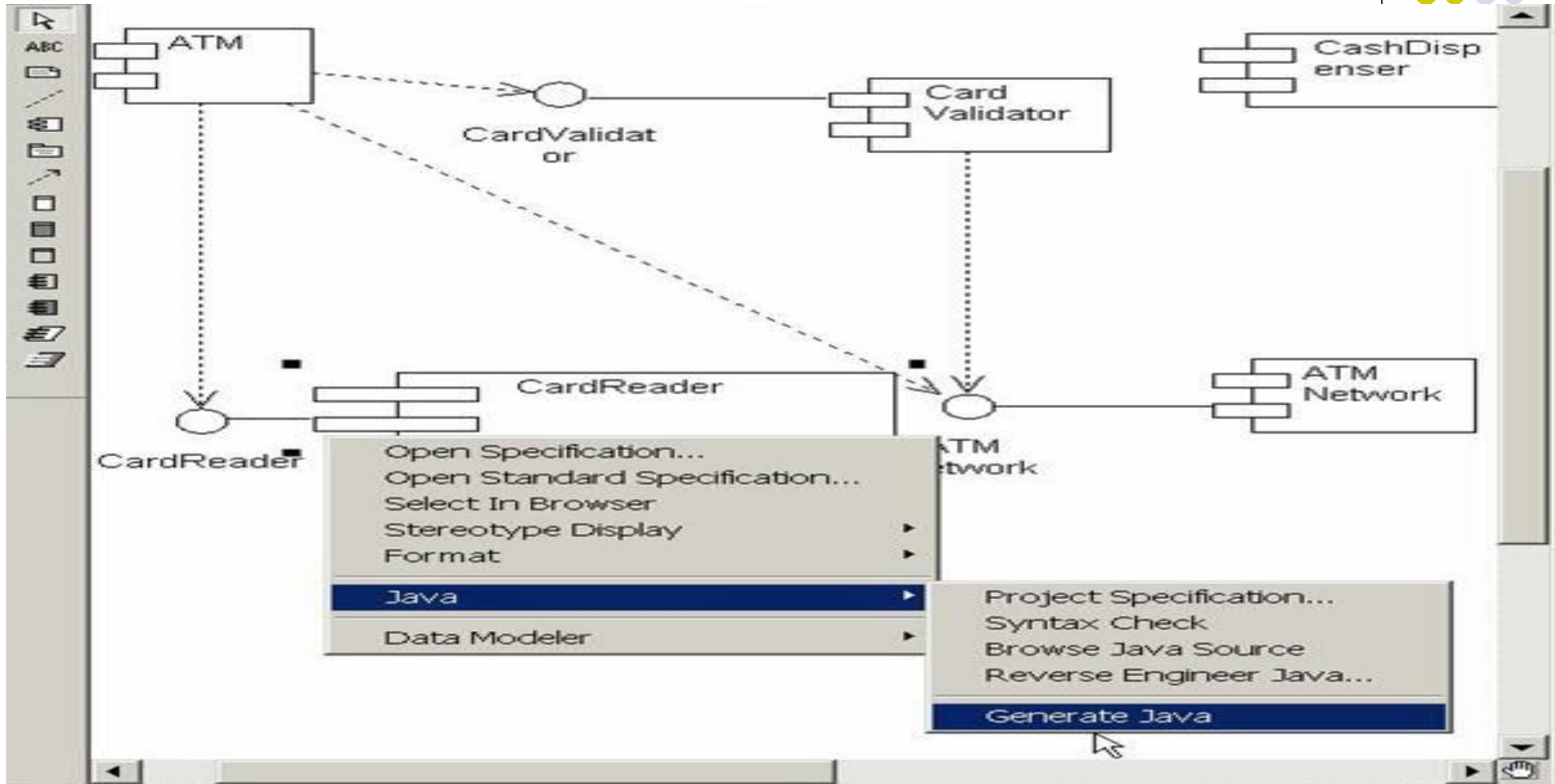


Class Specification for Fuel Sensor

Component Name	Package Name	Language
<input checked="" type="checkbox"/> FuelDispenser	Fuel Pump	Analysis
<input checked="" type="checkbox"/> ATM	Automated Teller	Analysis
<input checked="" type="checkbox"/> CashDispenser	Automated Teller	Analysis
<input checked="" type="checkbox"/> Fuel Pump	Fuel Pump	Analysis
<input checked="" type="checkbox"/> Card Validator	System Services	Analysis
<input checked="" type="checkbox"/> ATM Network	System Services	Analysis
<input checked="" type="checkbox"/> 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

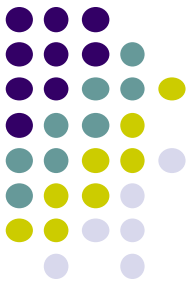


The **deployment architectural view** shows the configuration of run-time processing elements and the software processes living in them. **Deployment diagrams** are created to show the different nodes along with their connections in the system. They represent system topology and mapping executable subsystems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

Issues concerned:

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

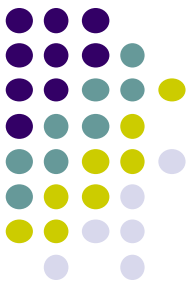
A deployment diagram shows processors, devices, and connections. Each model contains a single deployment diagram which shows the connections between its processors and devices, and the allocation of its processes to processors.



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



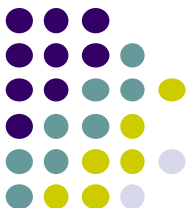
Artifacts within nodes

- 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.

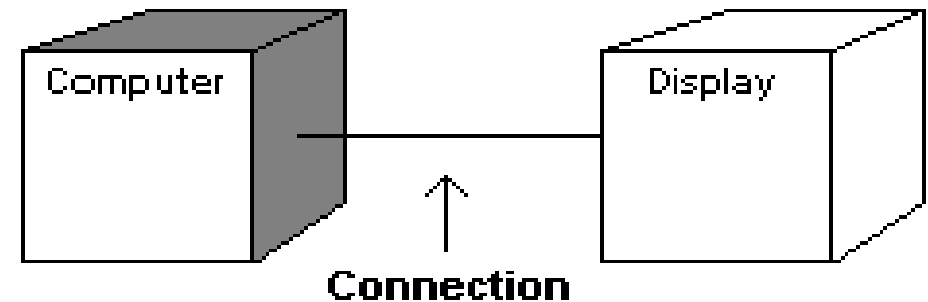
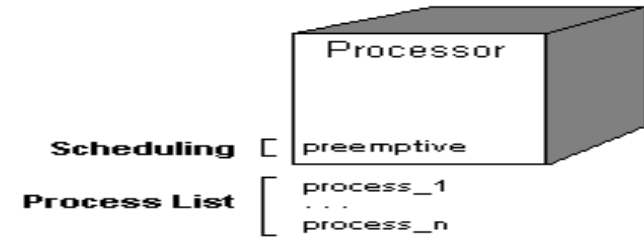
Processors, Devices and Connections



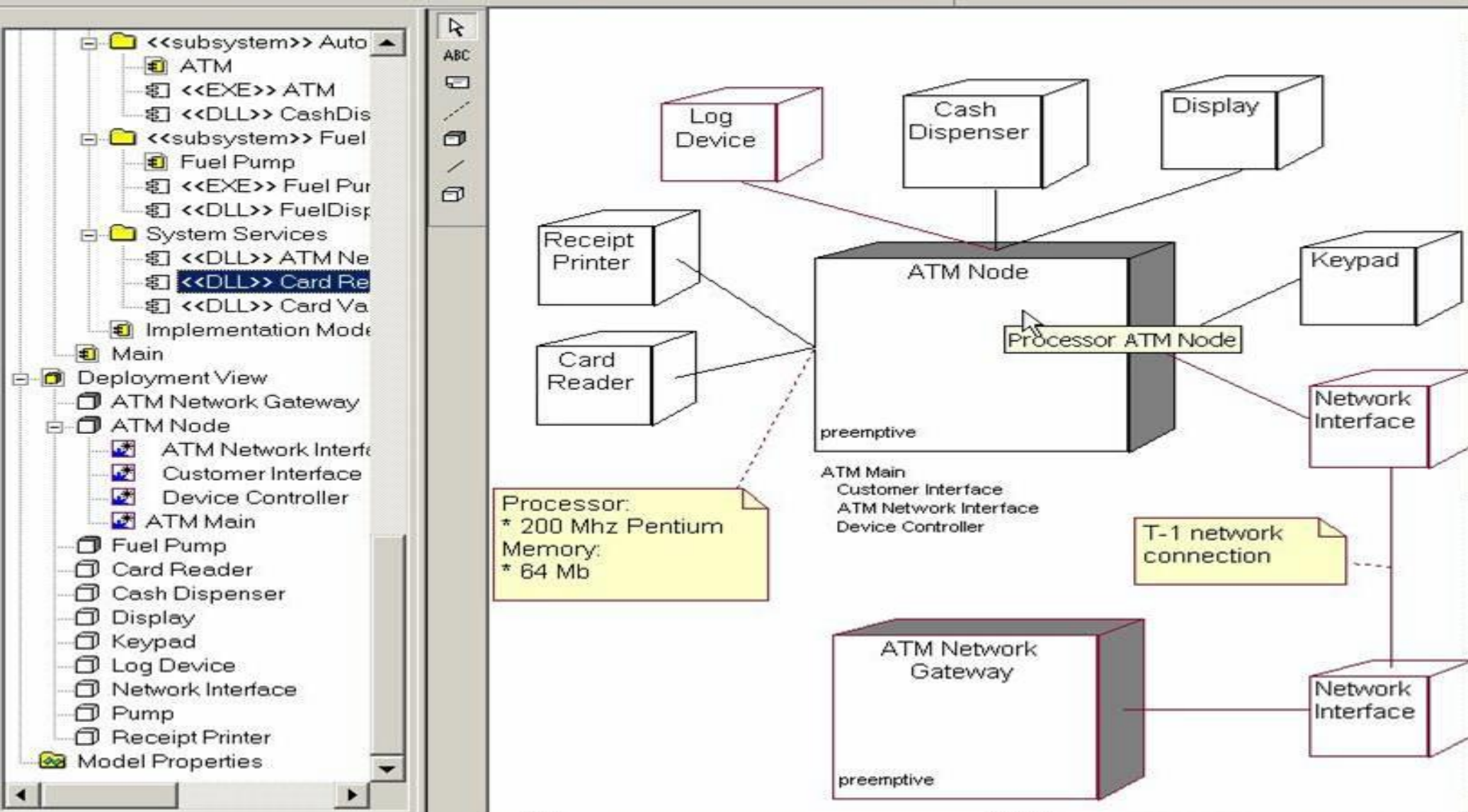
Processor - identify its processes and specify the type of process scheduling (preemptive, non-preemptive, cyclic, executive, manual).

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





UML deployment diagram [Bruegge & Dutoit]

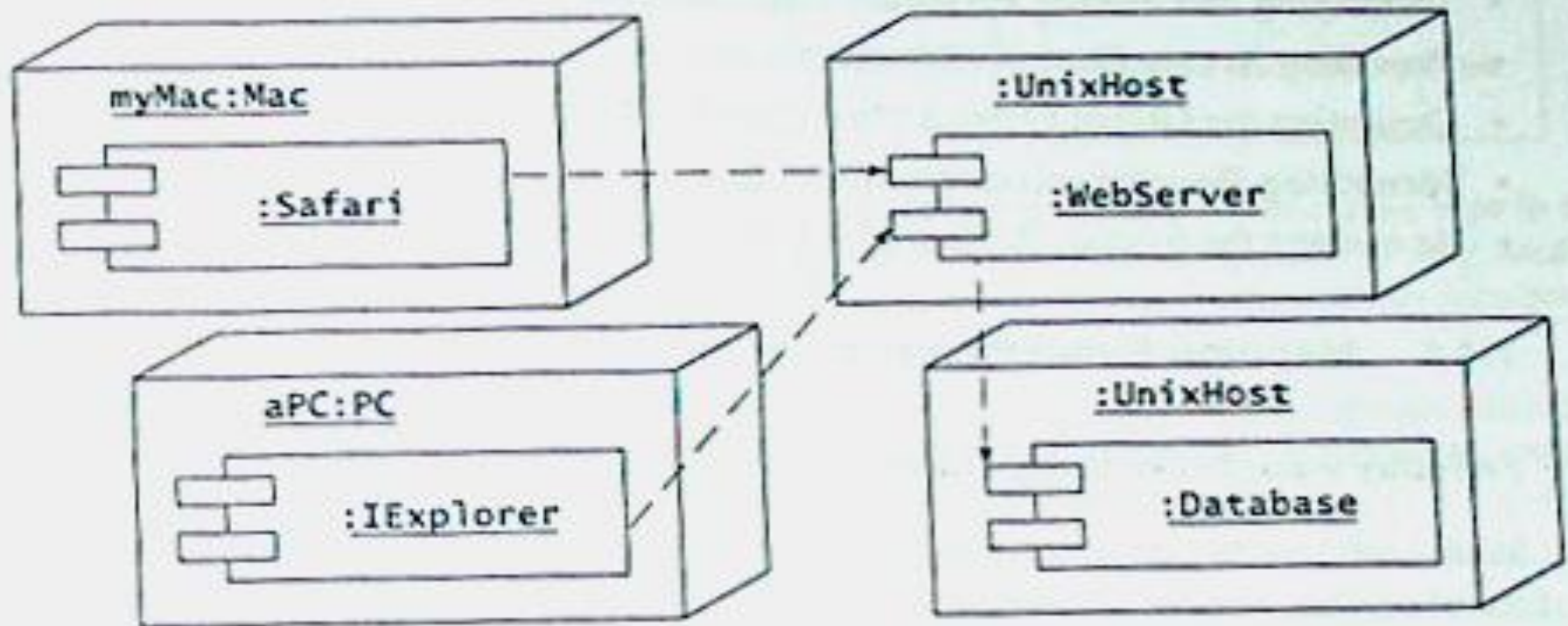


Figure 7-2 A UML deployment diagram representing the allocation of components to different nodes and the dependencies among components. Web browsers on PCs and Macs can access a WebServer that provides information from a Database.



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.

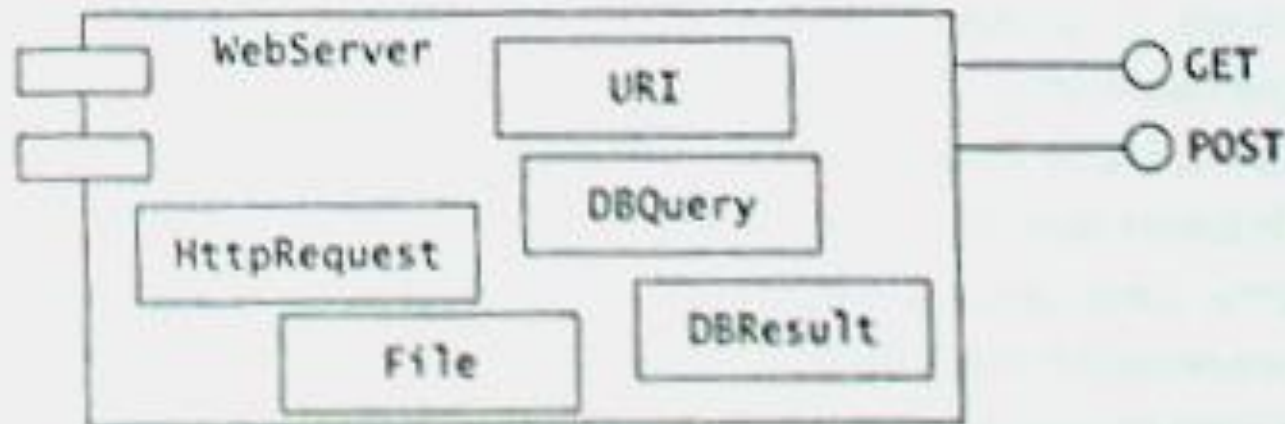
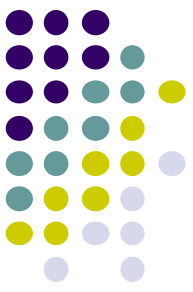


Figure 7-3 Refined view of the WebServer component (UML deployment diagram). WebServer provides two interfaces: a browser can either GET the content of a file referred by a URL or POST a form.



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.

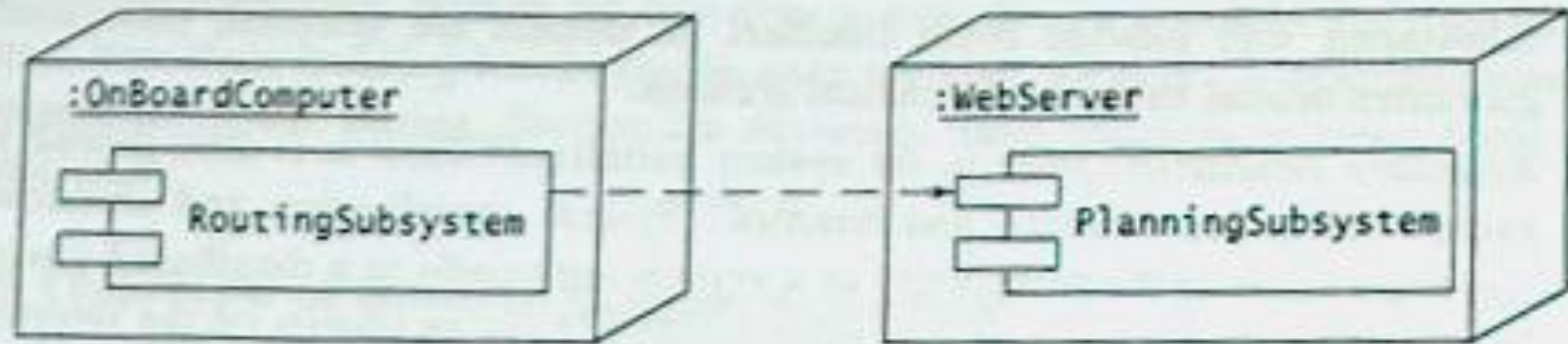
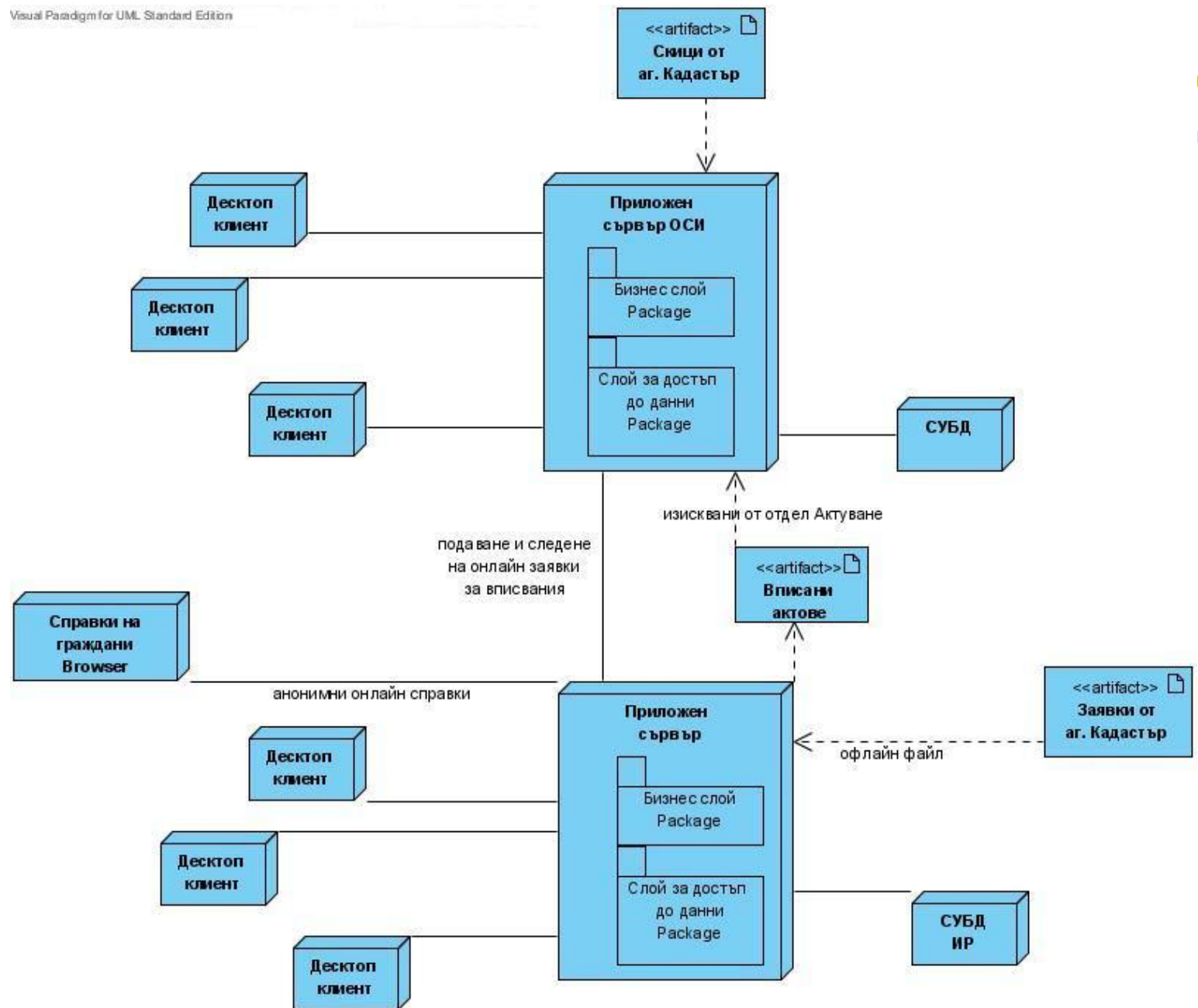


Figure 7-4 Allocation of MyTrip subsystems to hardware (UML deployment diagram). RoutingSubsystem runs on the OnBoardComputer; PlanningSubsystem runs on a WebServer.

A real example





UML 2.x Diagrams

