Architectural Modelling

Components

Components

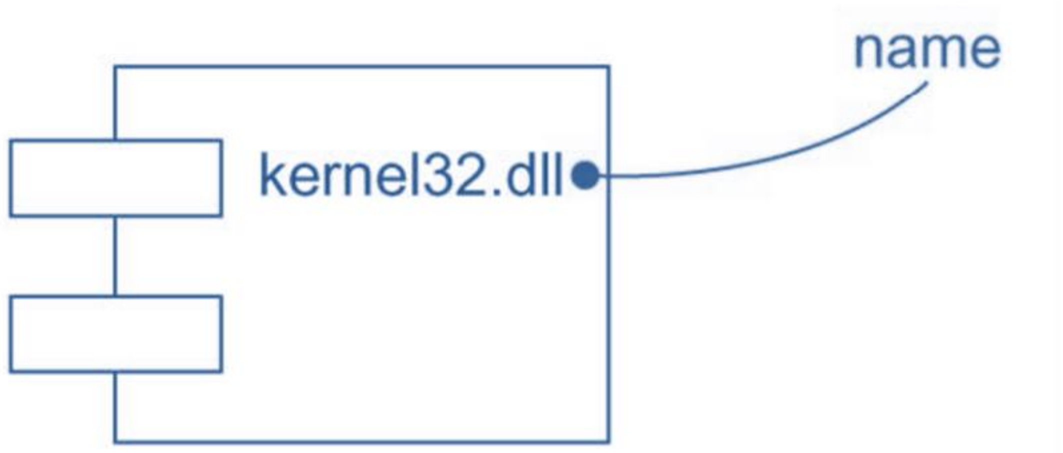
- A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.
- You use components to model the physical things that may reside on a node, such as executables, libraries, tables, files, and documents.
- . A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations.

Components (cont.)

- In software, many operating systems and programming languages directly support the concept of a component.
- . Object libraries, executables, COM+ components, and Enterprise Java Beans are all examples of components that may be represented directly in the UML by using components.
- Not only can components be used to model these kinds of things, they can also be used to represent other things that participate in an executing system, such as tables, files, and documents

Component notation

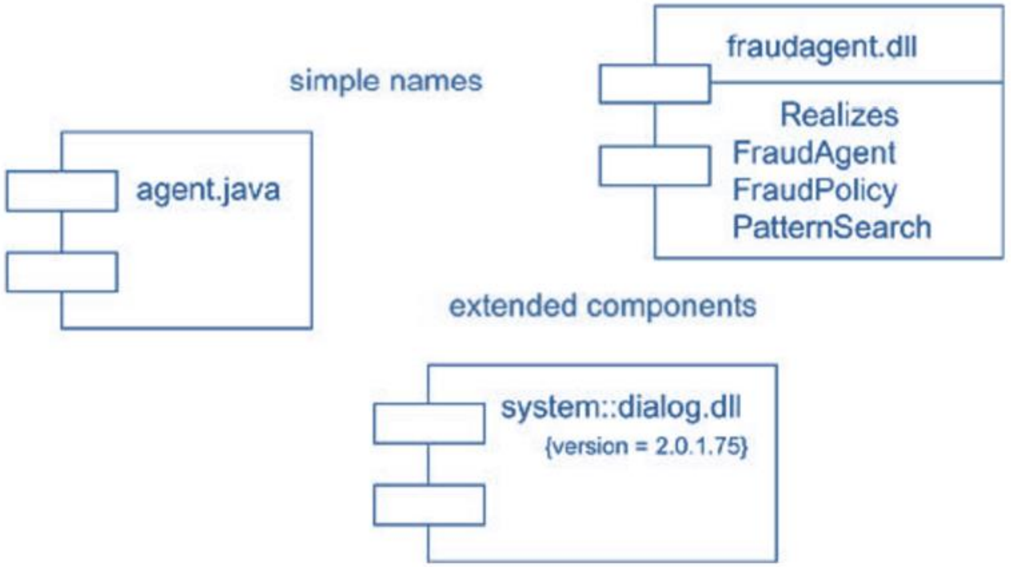
- Graphically, a component is rendered as a rectangle with tabs.



Component Name

- A component name must be unique within its enclosing package. A component name may be text consisting of any number of letters, numbers, and certain punctuation marks
- Every component must have a name that distinguishes it from other components.
- A name is a textual string. That name alone is known as a simple name; a path name is the component name prefixed by the name of the package in which that component lives.
- A component is typically drawn showing only its name.
- Like classes, you may draw components adorned with tagged values or with additional compartments to expose their details.

Simple and Extended Components



Components and Classes

In many ways, components are like classes

- Both have names
- Both may realize a set of interfaces
- Both may participate in dependency, generalization, and association relationships
- Both may be nested
- Both may have instances
- Both may be participants in interactions.

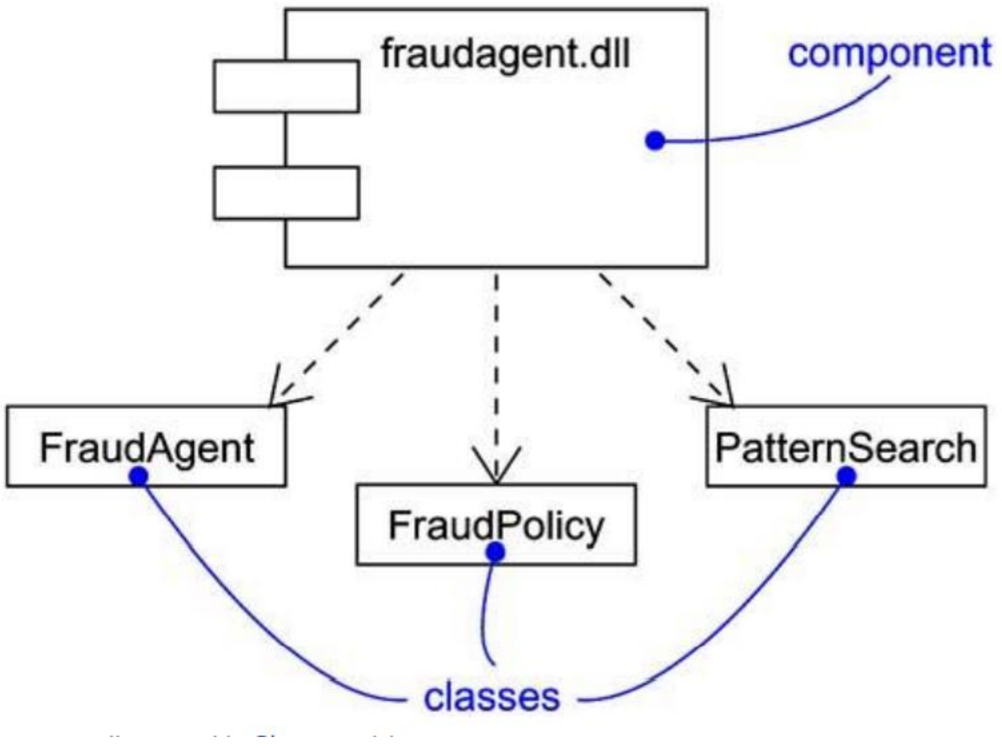
Components and Classes (cont.)

There are some significant differences between components and classes. ·

- Classes represent logical abstractions; components represent physical things that live in the world of bits.
- In short, components may live on nodes, classes may not.·
- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
- · Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.

Components and Classes (cont.)

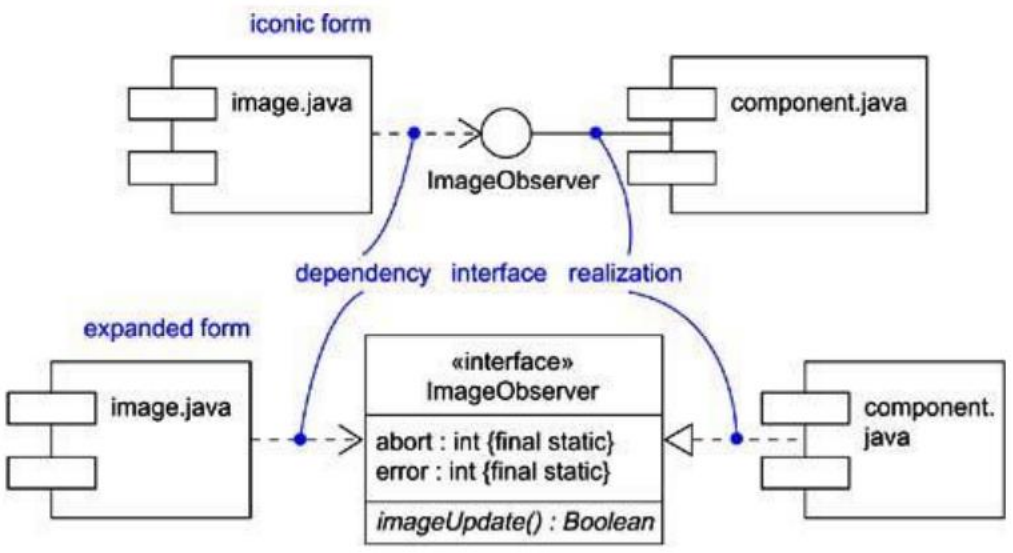
- In particular, a component is the physical implementation of a set of other logical elements (classes)



Components and interfaces

- An interface is a collection of operations that are used to specify a service of a class or a component.
- The relationship between a component and its interfaces in one of two ways. The first (and most common) style renders the interface in its elided, iconic form.
- The component that realizes the interface is connected to the interface using an elided realization relationship. The second style renders the interface in its expanded form, perhaps revealing its operations.
- The component that realizes the interface is connected to the interface using a full realization relationship.
- In both cases, the component that accesses the services of the other component through the interface is connected to the interface using a dependency relationship.

Components and interfaces (cont.)



Kinds of Components

Deployment components:

These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs).

Work product components:

These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created.

Execution components:

These components are created as a consequence of an executing system, such as a COM+ object, which is instantiated from a DLL.

Organizing Components

Packages:

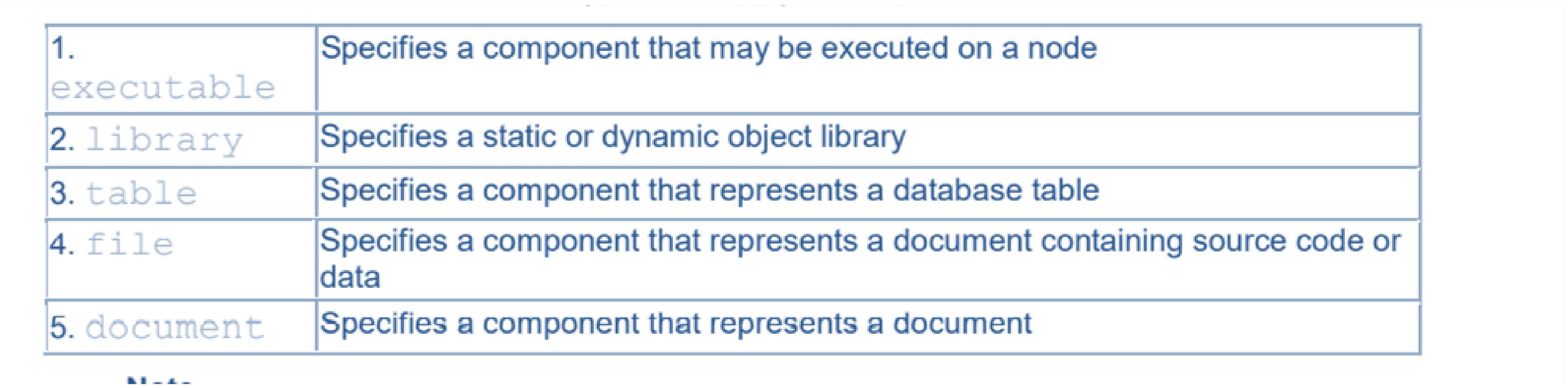
You can organize components by grouping them in packages in the same manner in which you organize classes.

Relationships:

You can also organize components by specifying dependency, generalization, association (including aggregation), and realization relationships among them.

Stereotypes

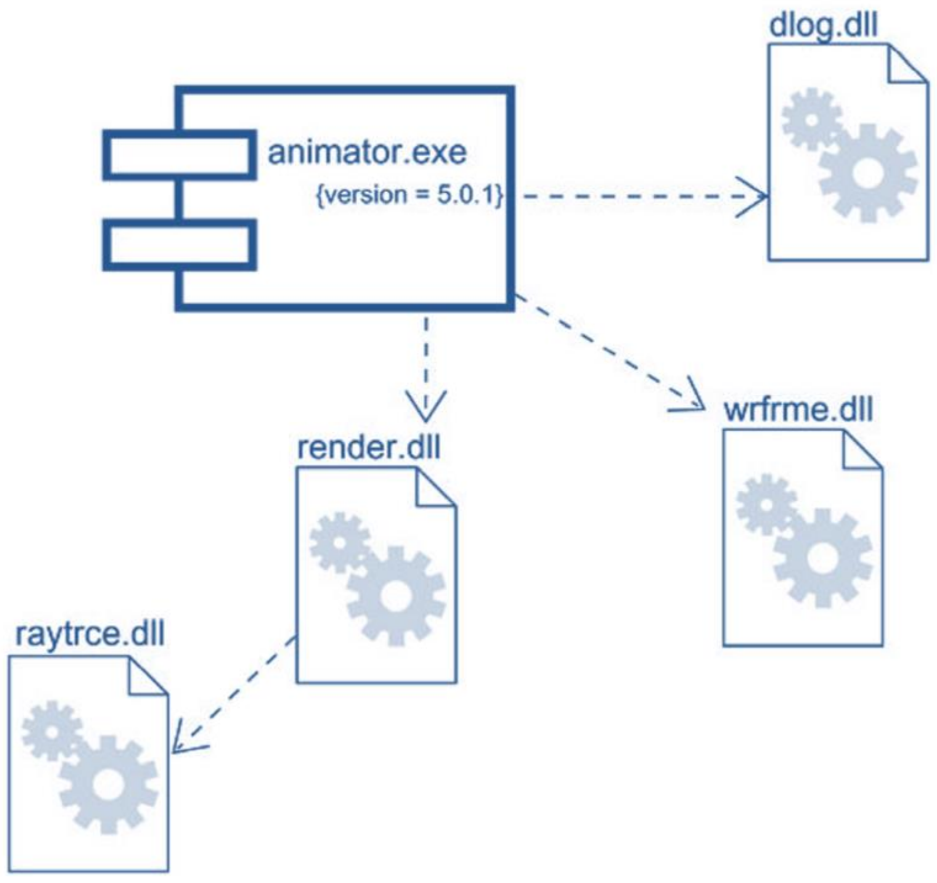
The UML defines five standard stereotypes that apply to components:



1. executable	Specifies a component that may be executed on a node
2. library	Specifies a static or dynamic object library
3. table	Specifies a component that represents a database table
4. file	Specifies a component that represents a document containing source code or data
5. document	Specifies a component that represents a document

Modeling Executables and Libraries

- Model any executables and libraries as components, using the appropriate standard elements. If your implementation introduces new kinds of components, introduce a new appropriate stereotype. ·
- If it's important for you to manage the seams in your system, model the significant interfaces that some components use and others realize. ·
- As necessary to communicate your intent, model the relationships among these executables, libraries, and interfaces. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.



This figure includes one executable (animator.exe, with a tagged value noting its version number) and four libraries (dlog.dll, wrfrme.dll, render.dll, and raytrce.dll)

Modelling an API

- · Identify the programmatic seams in your system and model each seam as an interface, collecting the attributes and operations that form this edge. ·
- Expose only those properties of the interface that are important to visualize in the given context; otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary. ·
- Model the realization of each API only insofar as it is important to show the configuration of a specific implementation.

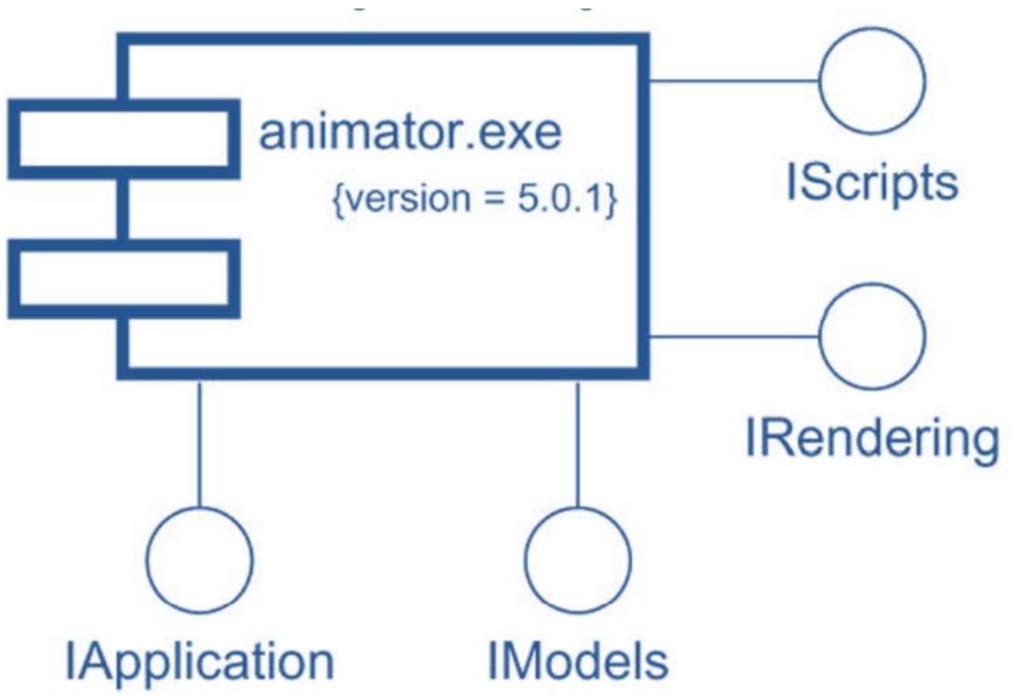


Figure exposes four interfaces that form the API of the executable: IApplication, IModels, IRendering, and IScripts.

Modelling Tables, Files, and Documents

To model tables, files, and documents

- ·Identify the ancillary components that are part of the physical implementation of your system. ·
- Model these things as components. If your implementation introduces new kinds of artifacts, introduce a new appropriate stereotype. ·
- As necessary to communicate your intent, model the relationships among these ancillary components and the other executables, libraries, and interfaces in your system. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

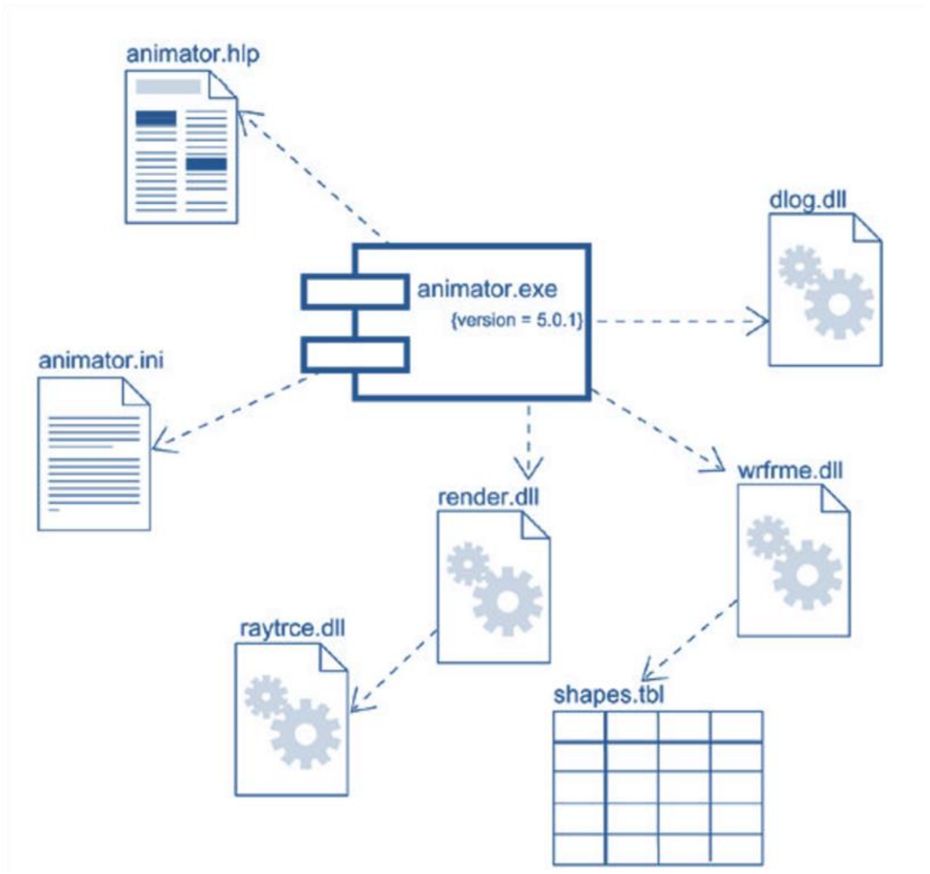


Figure shows the tables, files, and documents that are part of the deployed system surrounding the executable animator.exe. This figure includes one document (animator.hlp), one simple file (animator.ini), and one database table (shapes.tbl), all of which use the UML's standard elements for documents, files, and tables, respectively.

Modelling Source Code

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies. ·
- If it's important for you to bolt these models to your configuration management and version control tools, you'll want to include tagged values, such as version, author, and check in/check out information, for each file that's under configuration management.
- · As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.

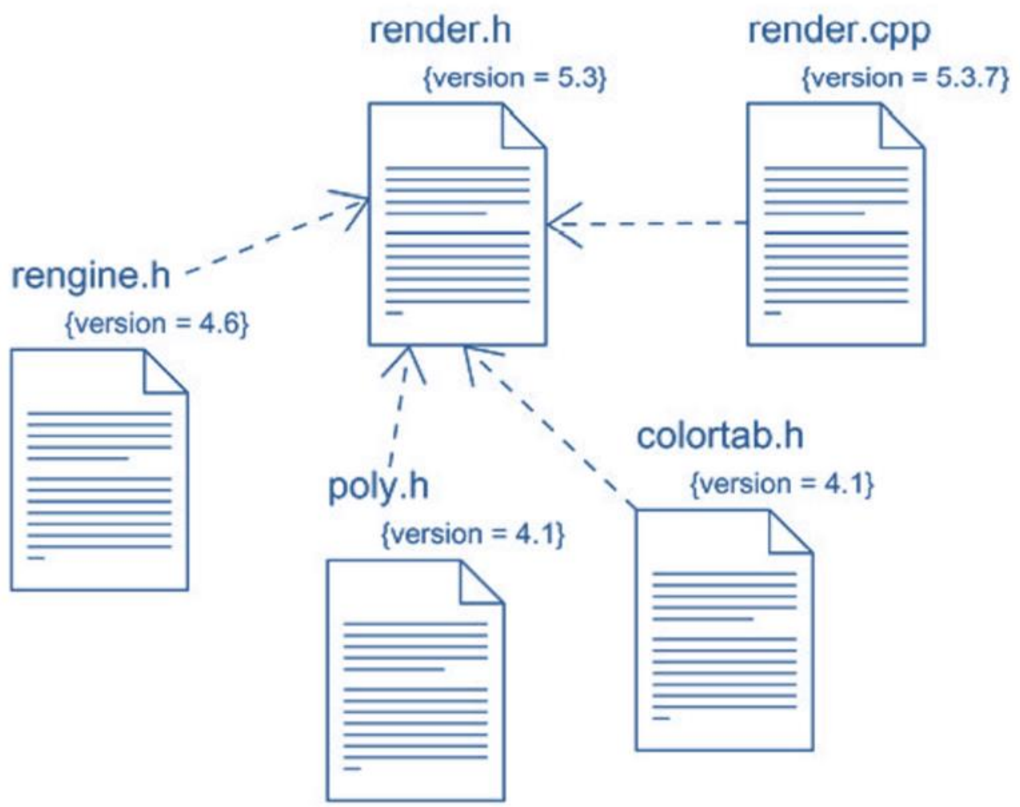


Figure shows some source code files that are used to build the library render.dll from the previous examples. This figure includes four header files (render.h, rengine.h, poly.h, and colortab.h) that represent the source code for the specification of certain classes. There is also one implementation file (render.cpp) that represents the implementation of one of these headers.

Architectural Modelling

Component Diagrams

Component Diagrams

- **Introduction:**
- Component diagrams are **important in modeling the physical aspects** of object-oriented systems.
- You use component diagrams to **model the static implementation view of a system**.
- This involves modeling the physical things that reside on a node, such as
  - executables,
  - libraries,
  - tables, files, and documents.

Definition

- A component diagram shows a set of components and their relationships. Graphically, a component diagram is a collection of vertices and arcs.

Contents

- Component diagrams commonly contain
  - Components
  - Interfaces
  - Relationships – dependency, generalization, and realization.
  - Notes
  - Constraints.
  - Packages or subsystems

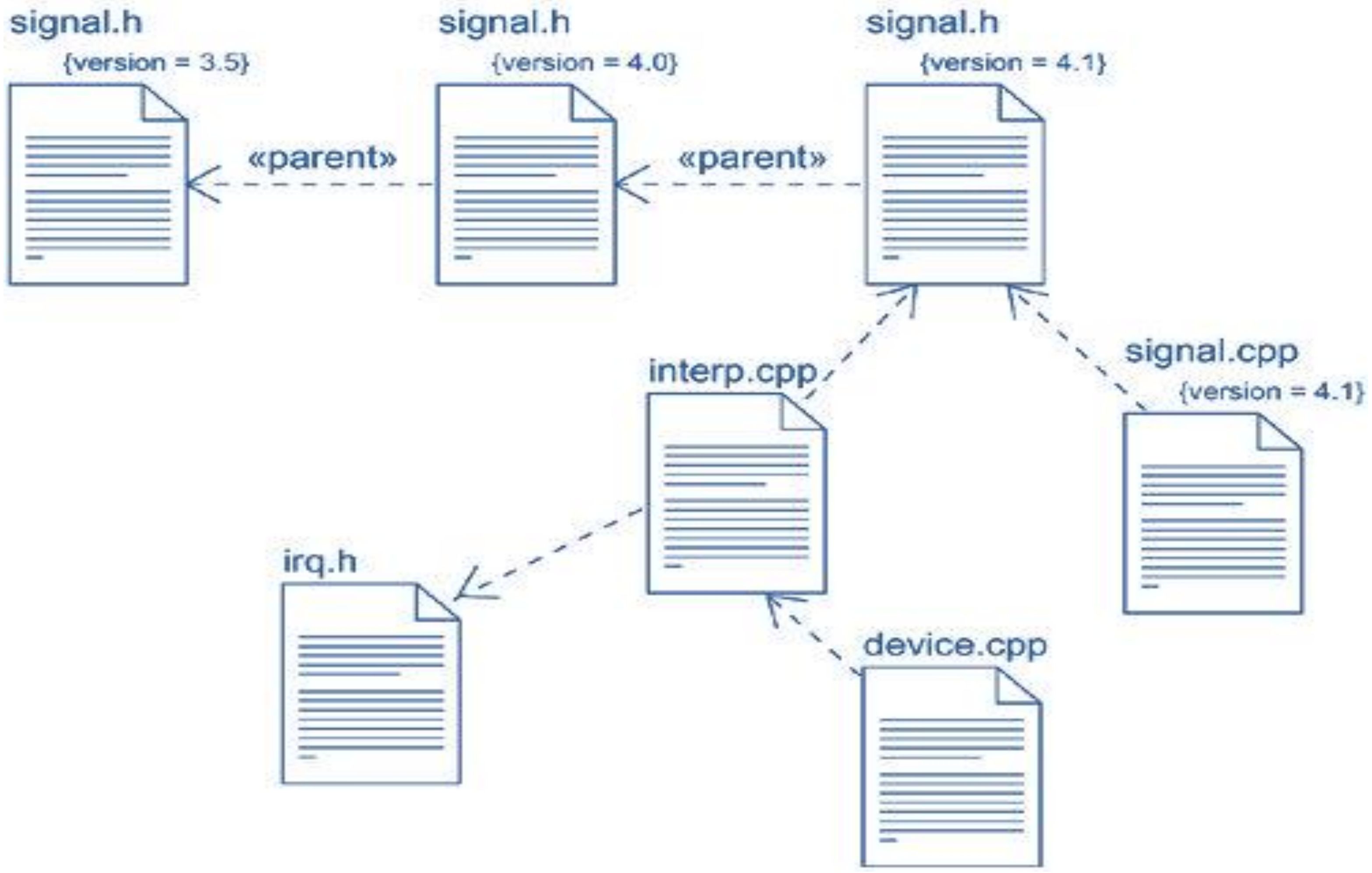
Common Uses

- Primarily supports the configuration management of a system’s parts, made up of components. You will typically use component diagrams in one of four ways.
  - To model source code.
  - To model executable release.
  - To model physical databases.
  - To model adaptable systems.

To model a system’s source code

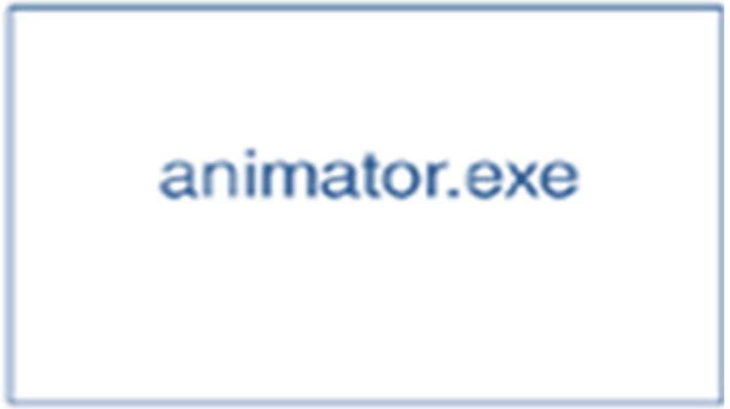
- By forward engineering identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Model the compilation dependencies among these files using dependencies.

Modeling Source code



To model an executable release

- Identify the set of components you would like to model.
- Consider the stereotypes of each component in this set.
- For each component in this set, consider its relationship to its neighbors.



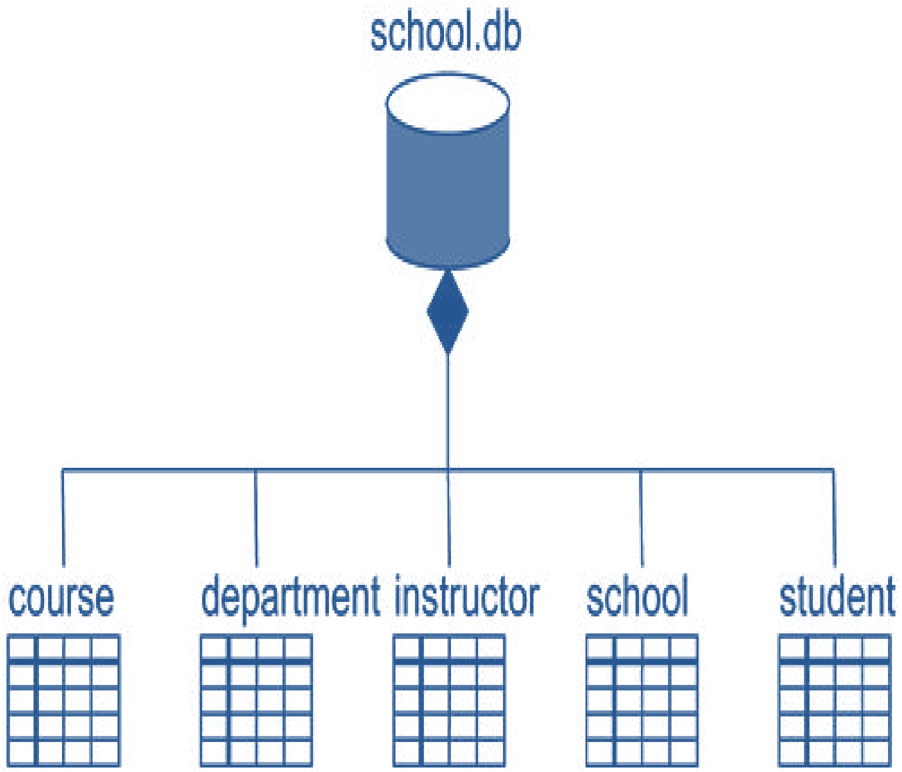
To model a physical database

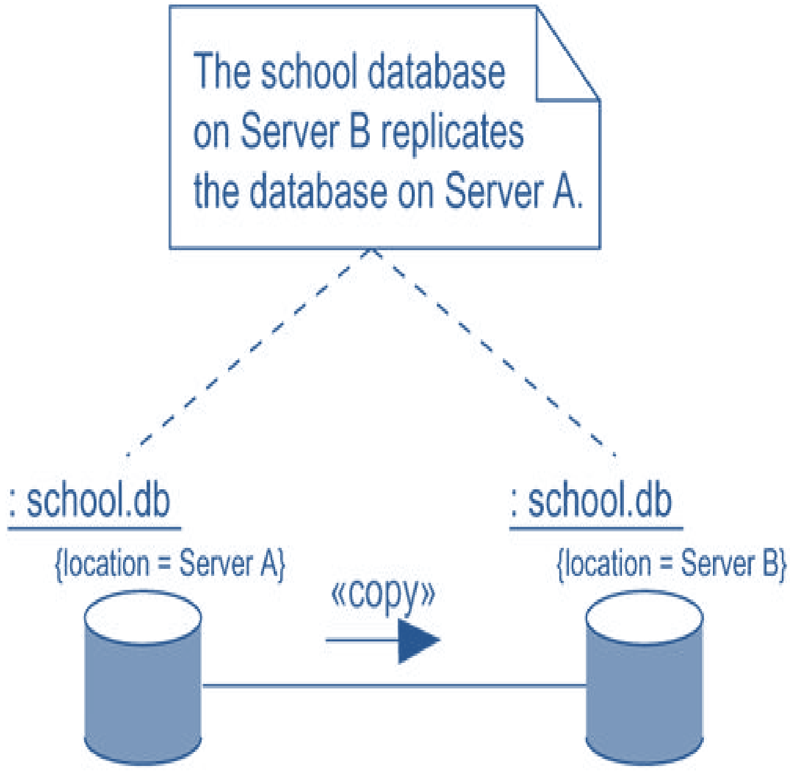
- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. create a component diagram
- To construct your modeling, that contains components stereotyped as tables.

To model an adaptable system

- Consider the physical distribution of the components that may migrate from node to node.
- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances.

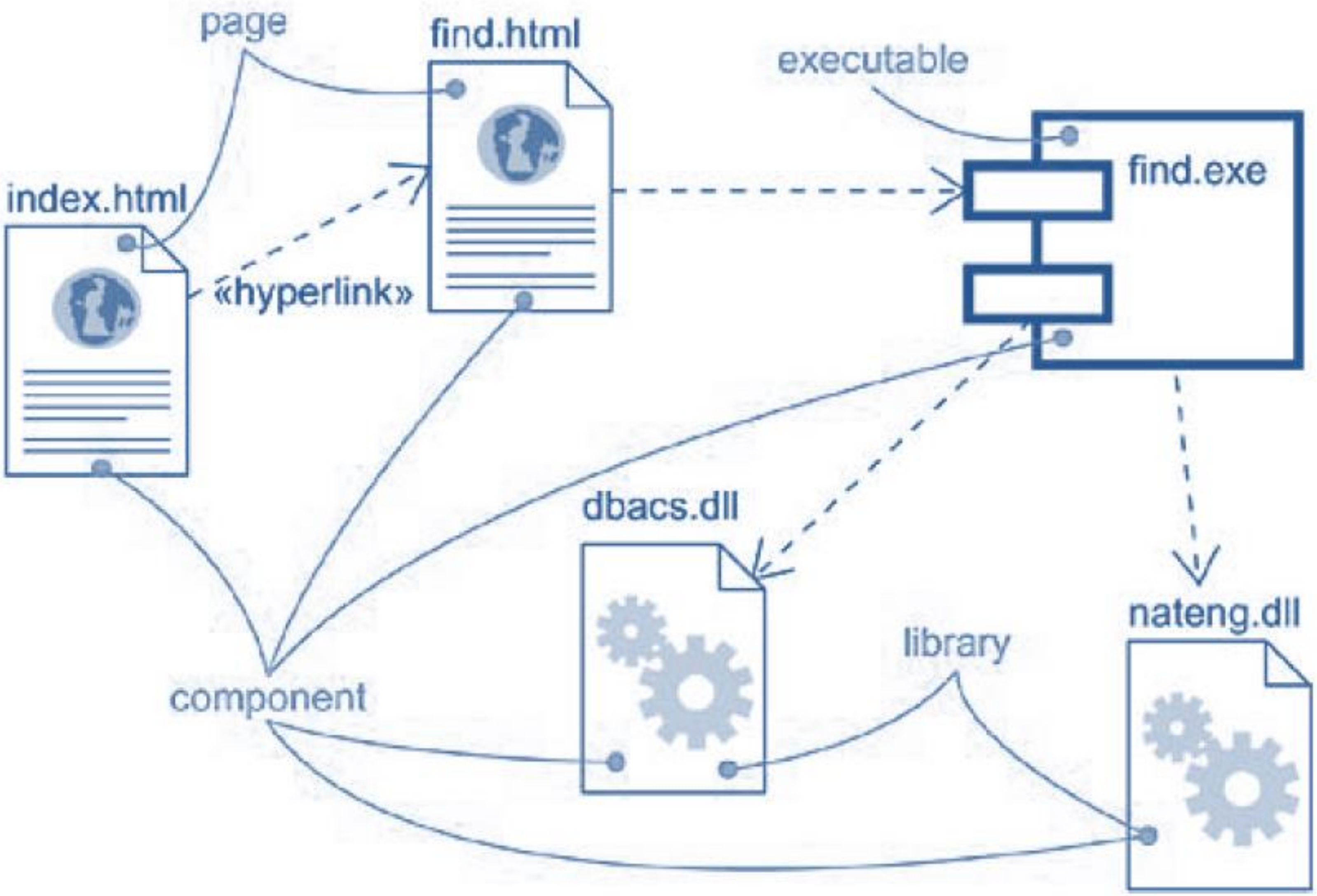
Physical database and Adaptable Systems

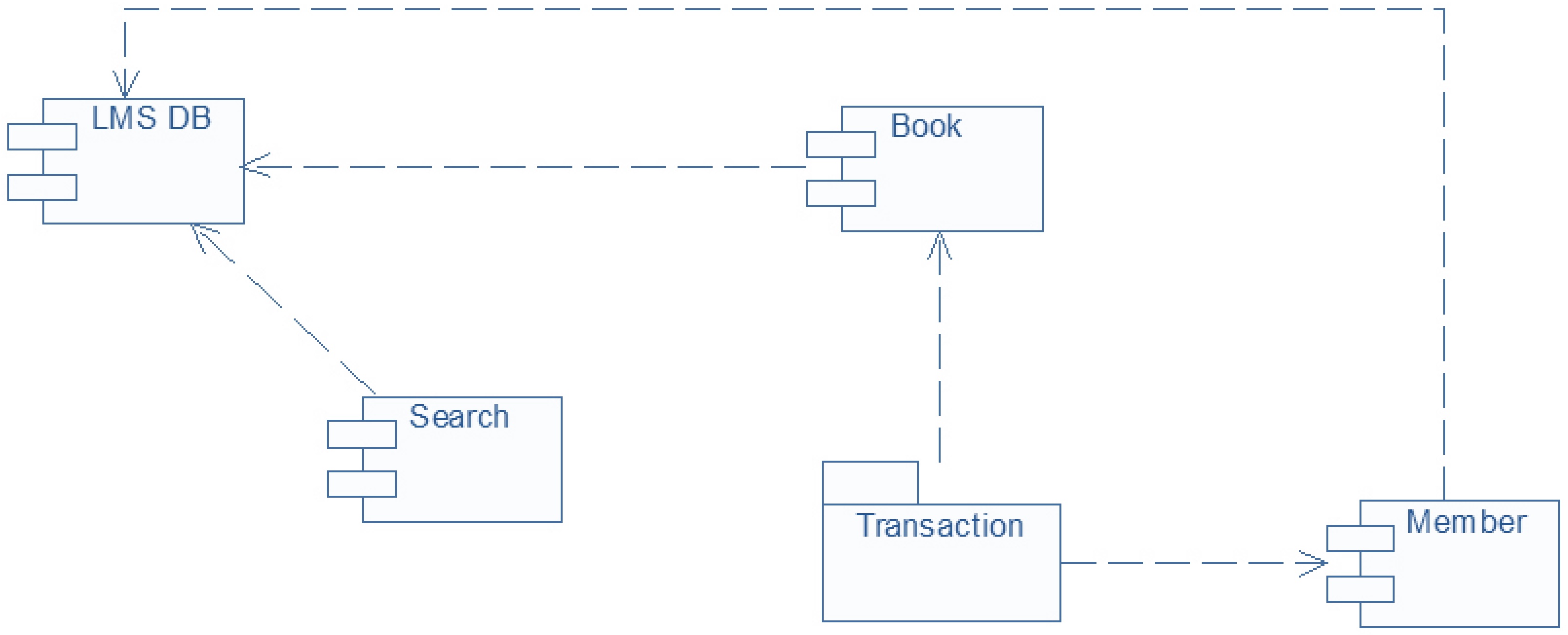




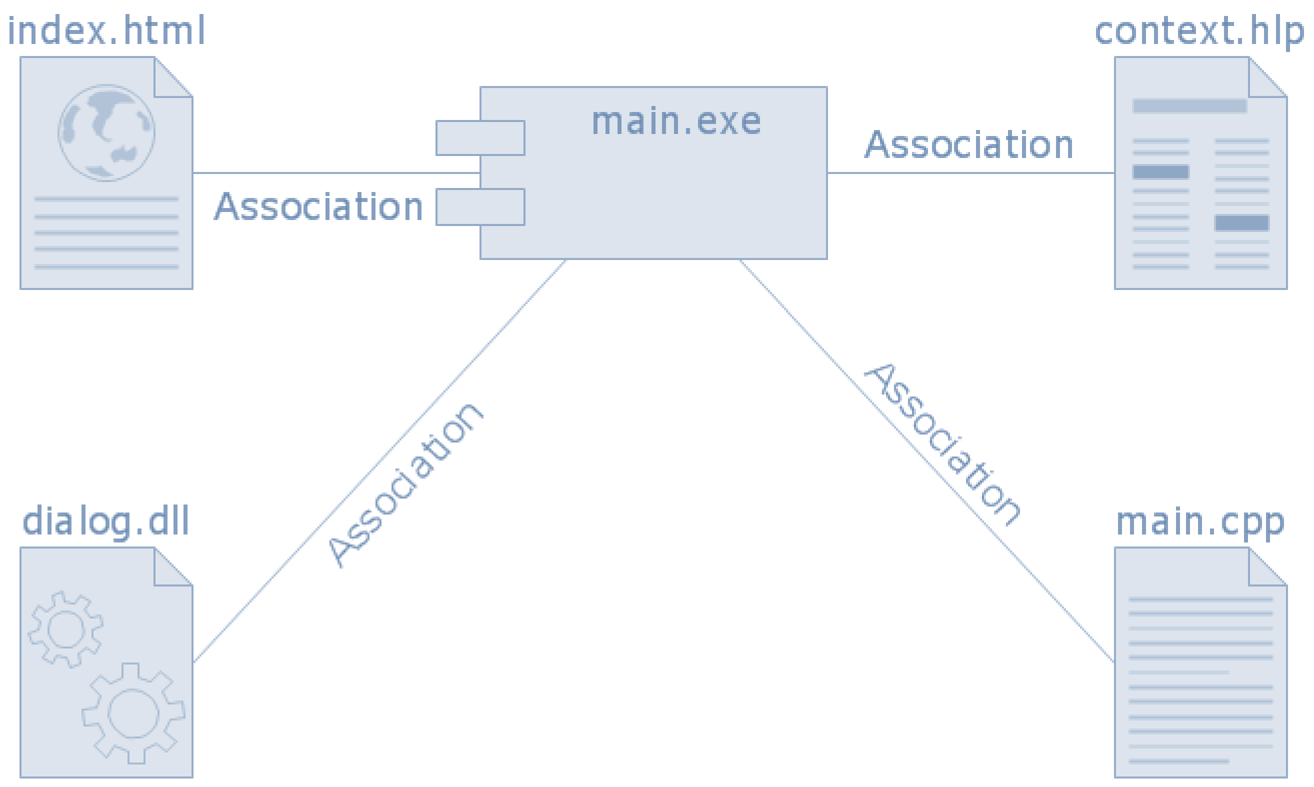
Forward and Reverse Engineering

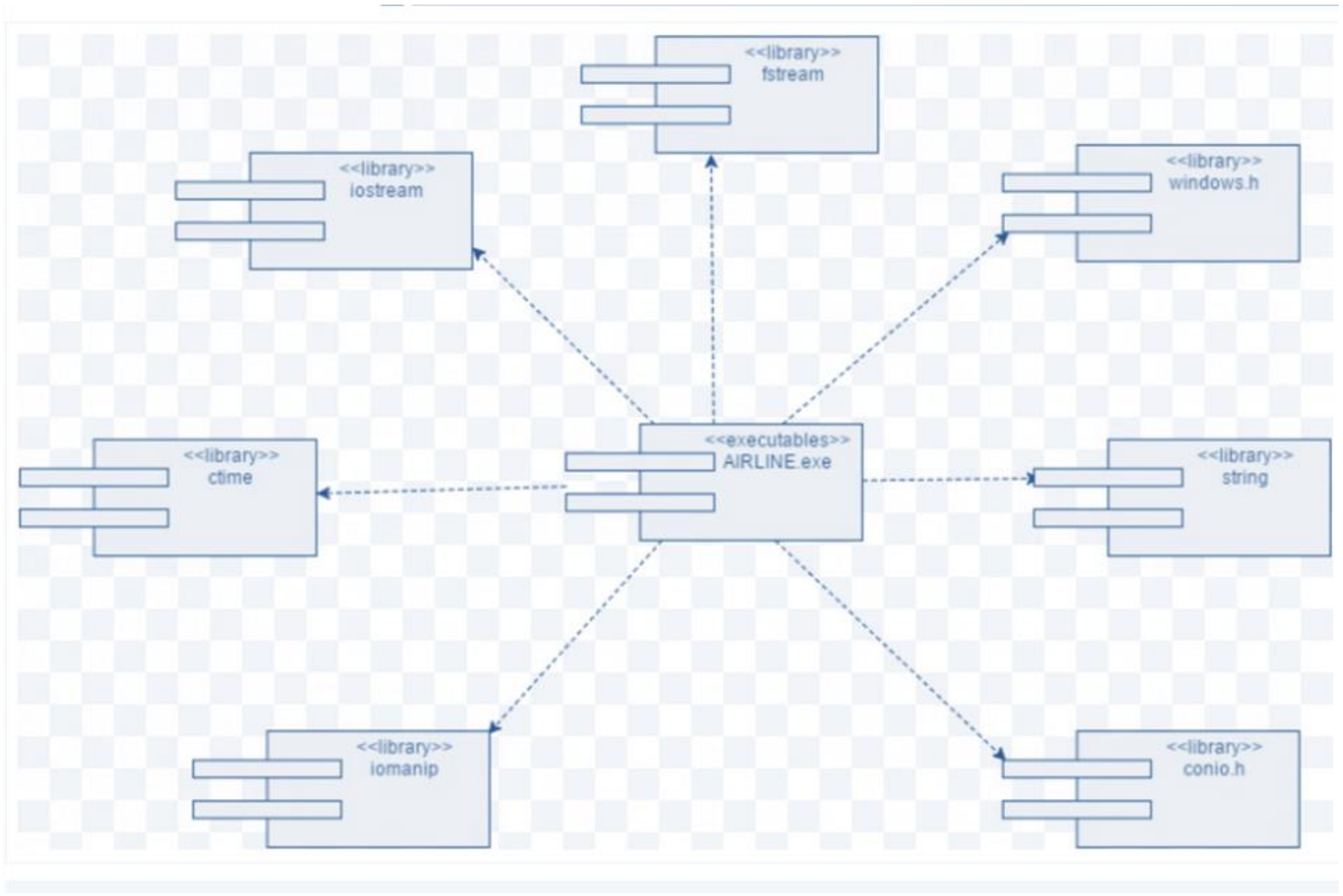
- **Forward Engineering:**
  - Forward engineering components are direct, because components are themselves physical things (executables, libraries, tables, and documents)..
- **Reverse Engineering:**
  - Reverse engineering a component diagram is not a perfect process because there is always a loss of information.


page
find.html
executable
find.exe
index.html
«hyperlink»
dbacs.dll
nateng.dll
library
component

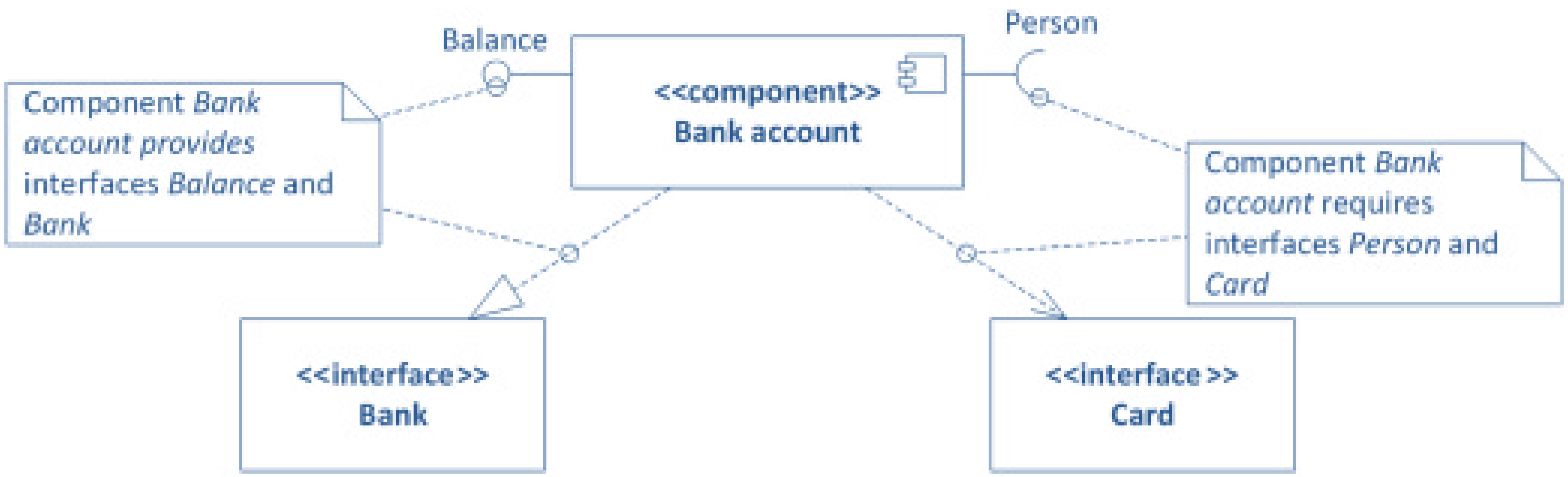

LMS DB
Book
Search
Transaction
Member

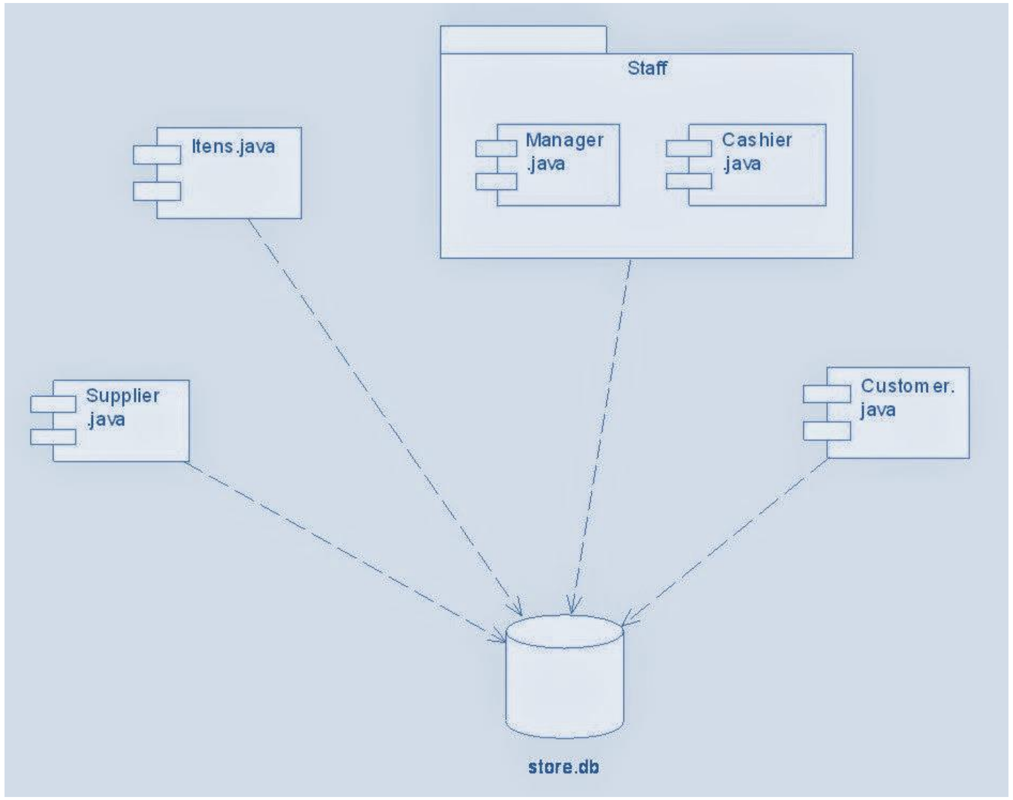
UML Component Diagram Template





<<library>>
fstream
<<library>>
iostream
<<library>>
windows.h
<<library>>
ctime
<<executables>>
AIRLINE.exe
<<library>>
string
<<library>>
iomanip
<<library>>
conio.h


Balance
Person
<<component>>
Bank account
Component *Bank account provides* interfaces *Balance* and *Bank*
Component *Bank account* requires interfaces *Person* and *Card*
<<interface>>
Bank
<<interface >>
Card



Reference:
1. Booch, Grady. The unified modeling language user guide. Pearson Education India, 2005.