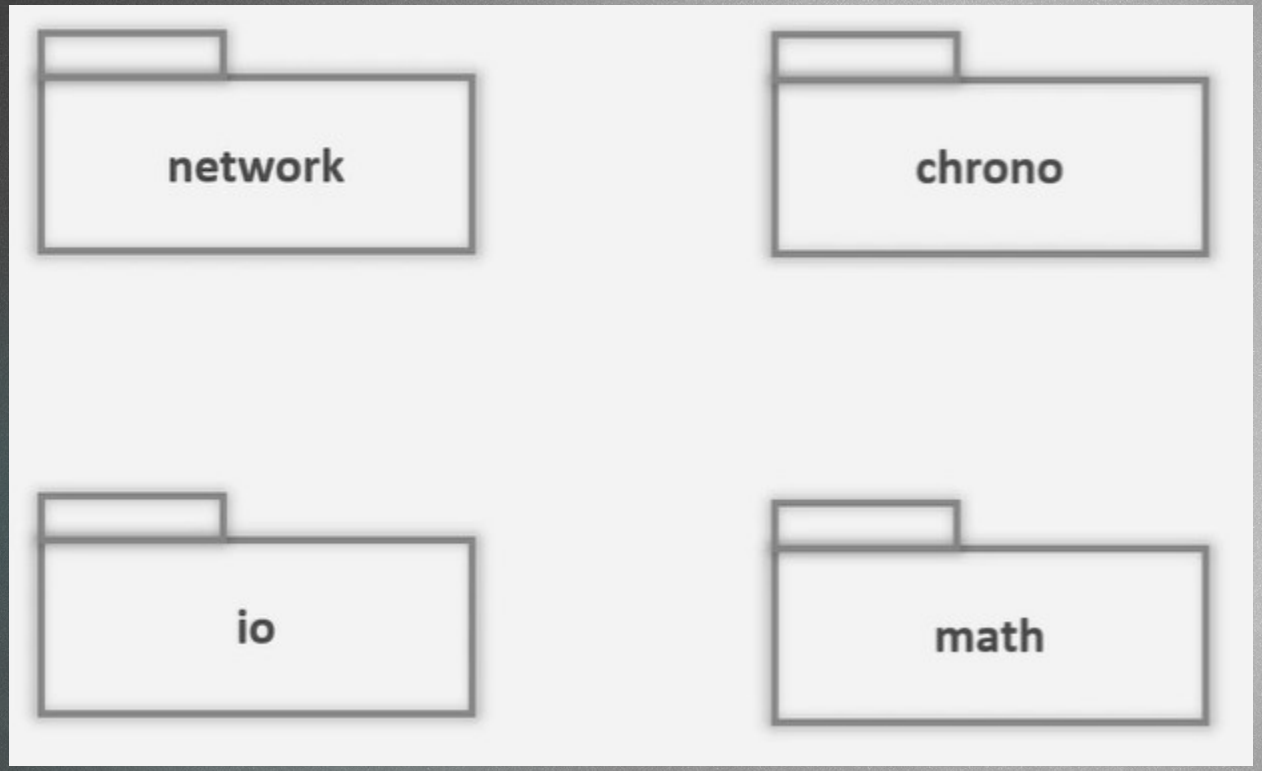
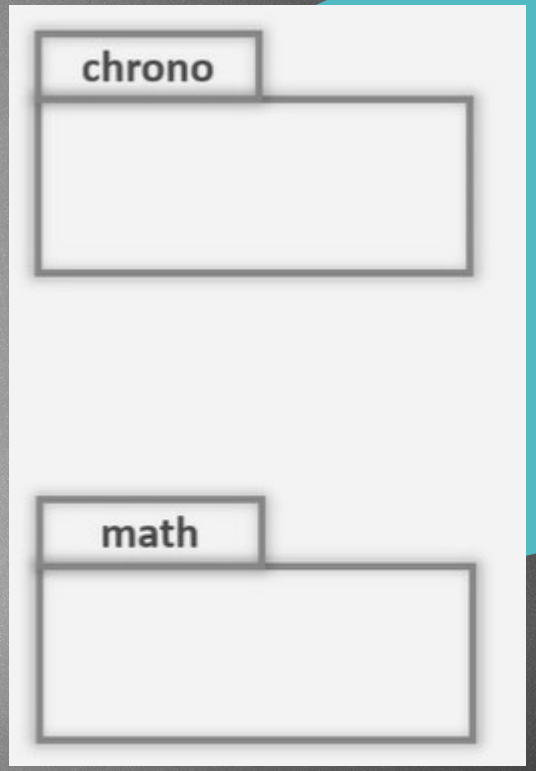
Diagrams
Package and Deployment Diagram

Package Diagram

- Packages/namespaces are used to organize classes in a large application.
- This reduces the complexity pf understanding the architecture of the application
- UML provides packages to model o group of classes.
- The packages are places in a package diagram along with their dependencies
- The are part of the development view

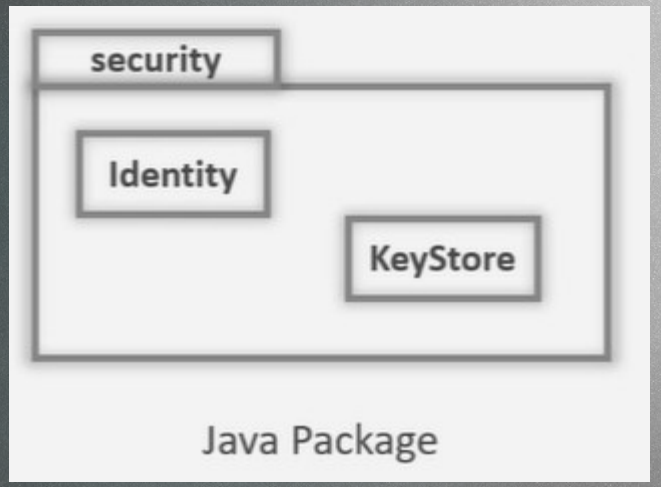
Package

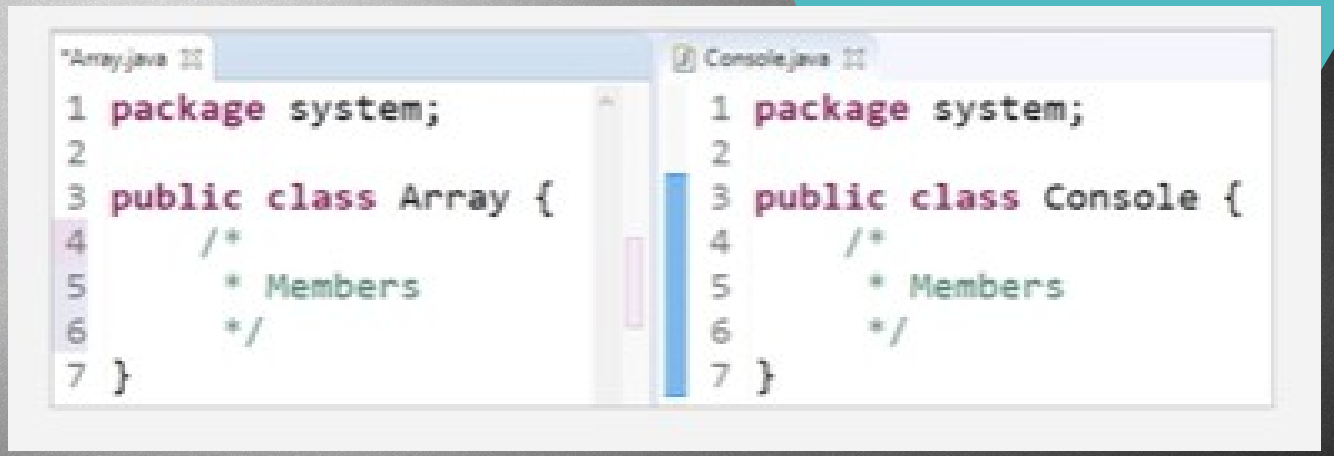




Package content

- A package organizes UML elements
- These elements can be drawn inside the package





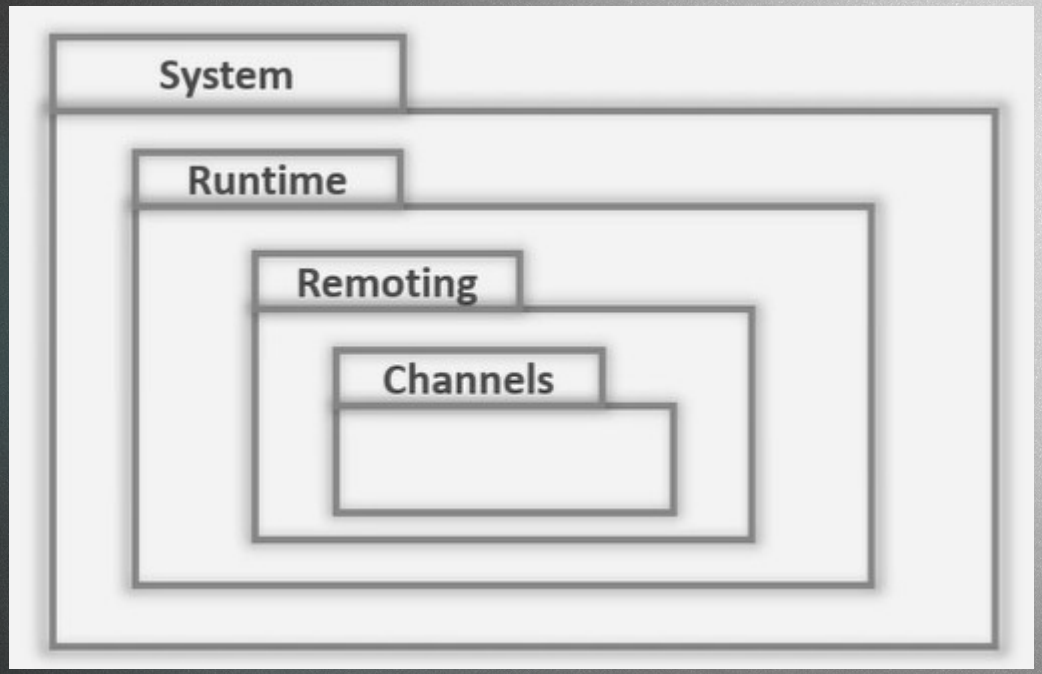
```java
package system;

public class Array {
    /*
     * Members
     */
}
```

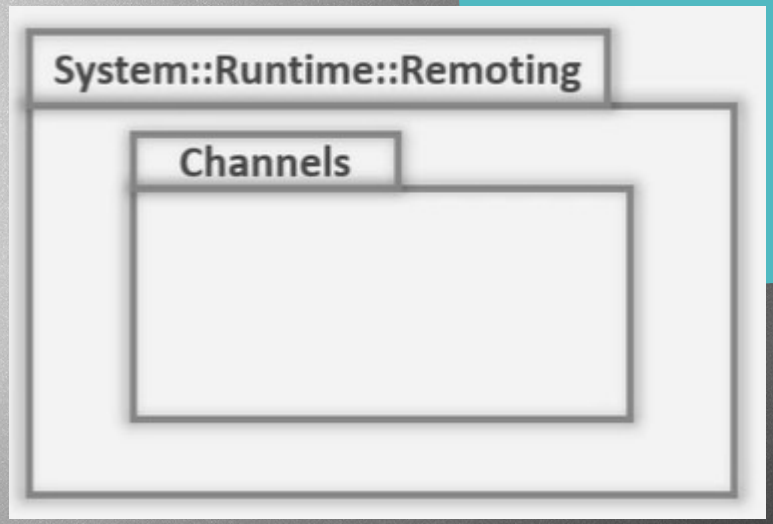
```java
package system;

public class Console {
    /*
     * Members
     */
}
```

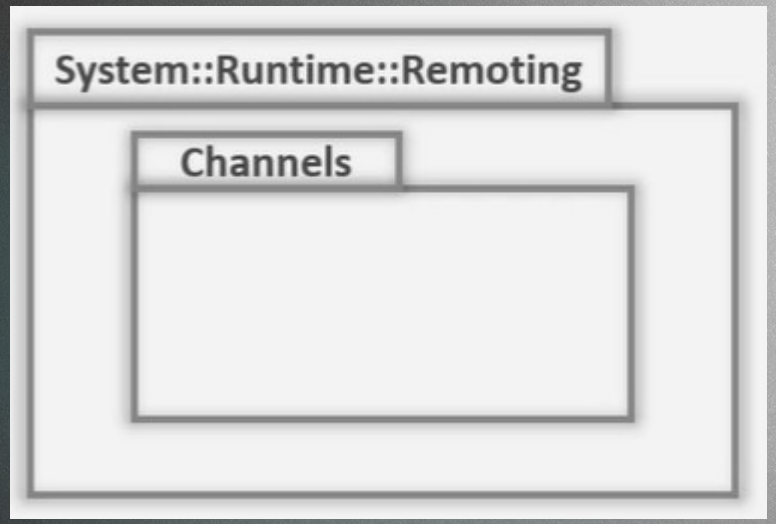
Nested Packages

- Packages can be nested
- This allows us to model the deep nesting of the packages in large applications

Nested Packages





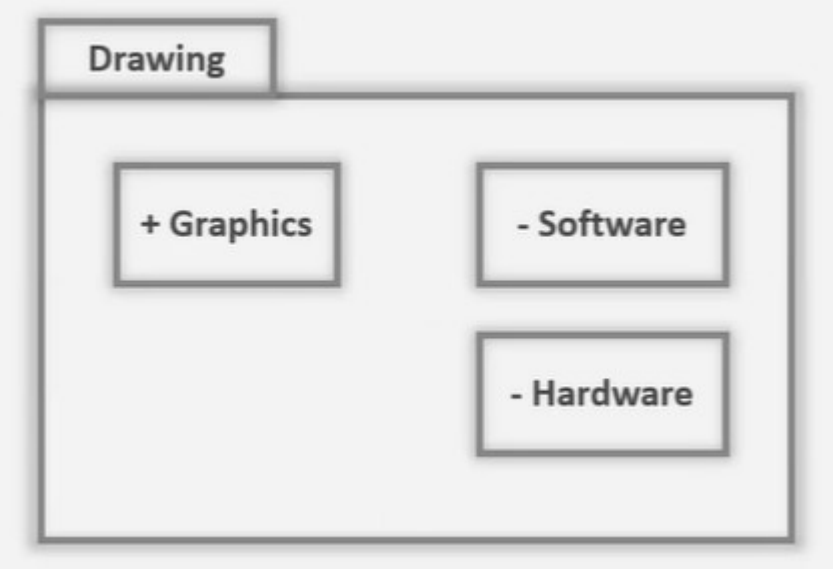



```
package system.runtime.remoting.channels;

public class Test {

}
```

Element Visibility

- The elements inside the packages can have two of visibility
- Private:
  - Elements with public visibility can be used outside the package
- Public:
  - Elements with the private visibility can be used only by elements of the same package
- Public visibility uses the + sign & private visibility use - sign

Example



Graphics.java

```
1 package drawing;
2
3 public class Graphics {
4
5 }
6
```

Software.java

```
1 package drawing;
2
3 class Software {
4
5 }
6
```

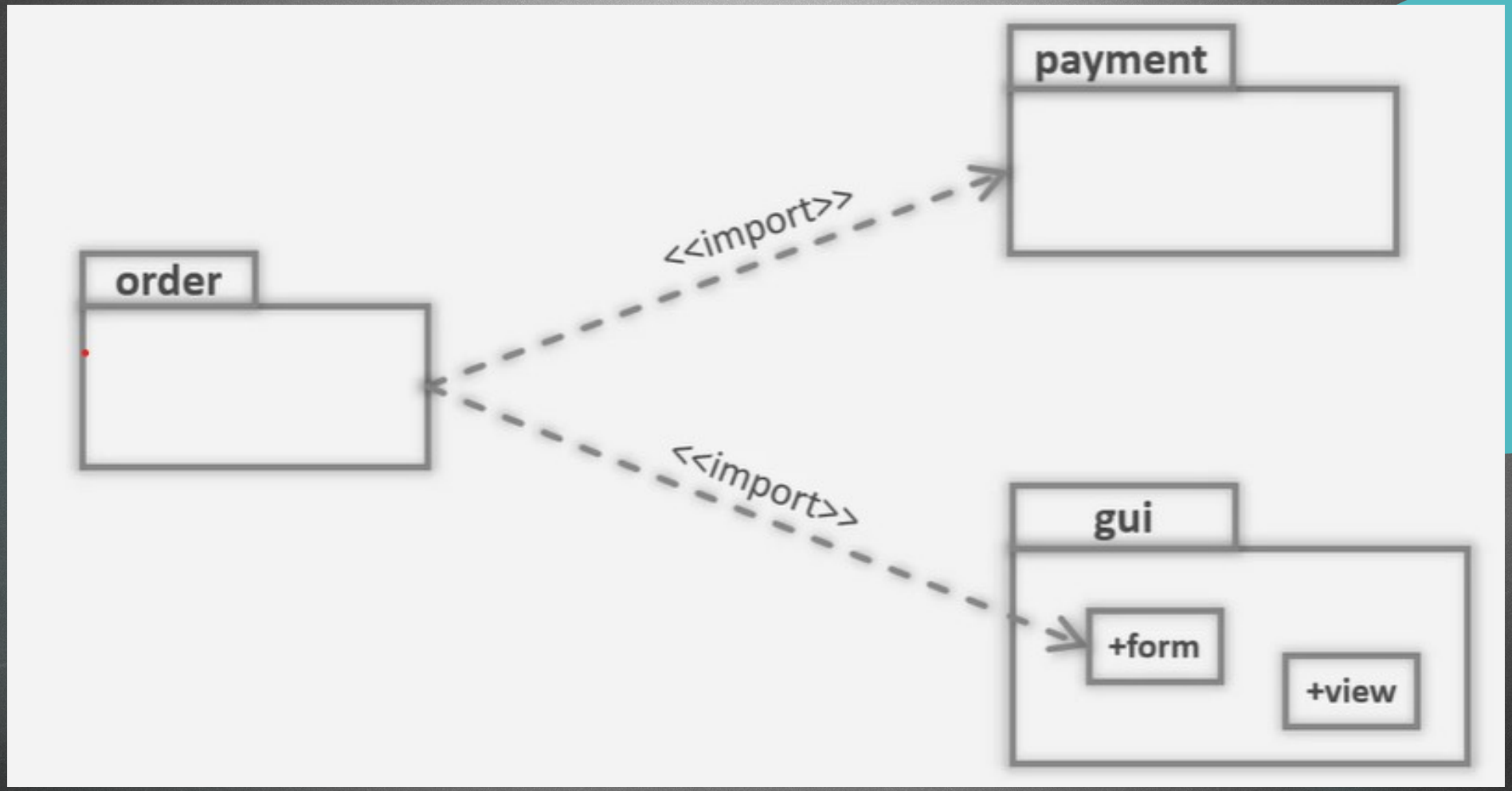
Hardware.java

```
1 package drawing;
2
3 class Hardware {
4
5 }
6
```

Importing Packages

- To access an element in another package, you have to use full scoped names
- Alternatively, a package can be imported that causes all the public elements to be available without using full scoped names
- The imported packages is called target packages
- The import creates a dependency & is shown with a dashed arrow
- The arrow points to the target packages & uses <<import>> stereotype
- A package can import a specific element of the whole
- Note that element should be public

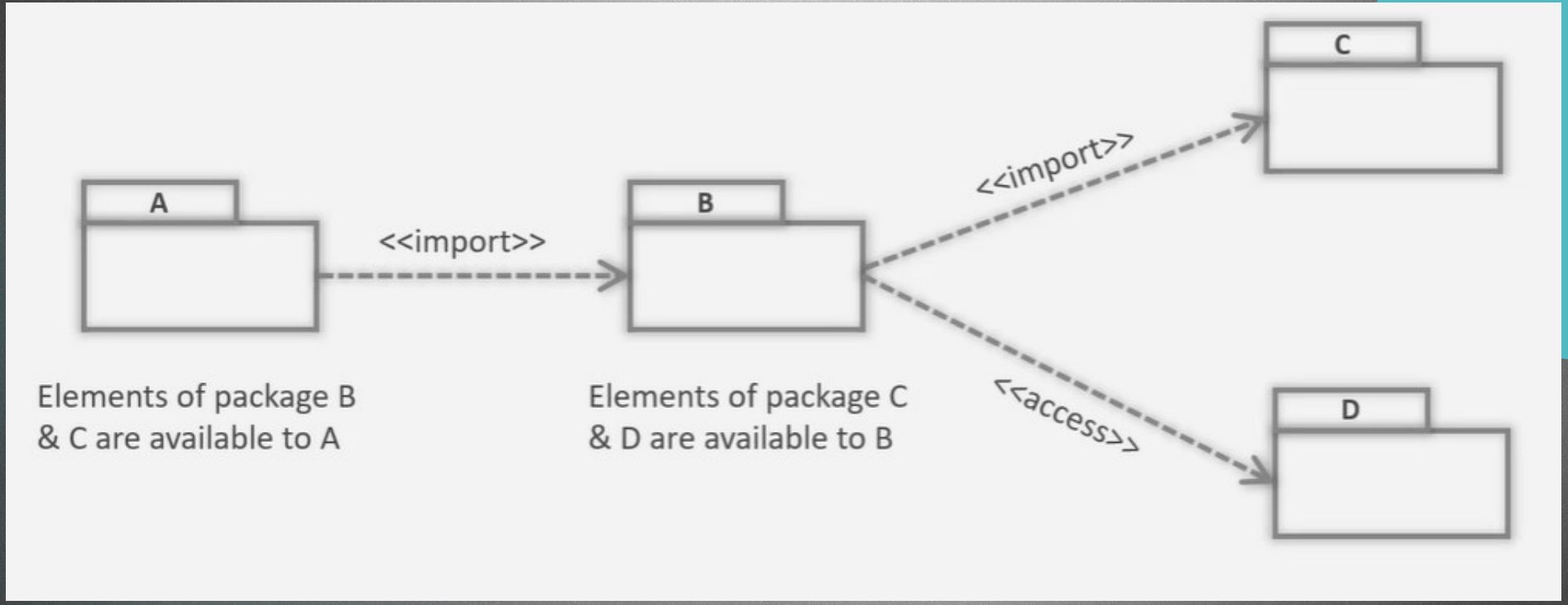
Import



Import

- The import relation itself has visibility
- An import can be public import or private import
- In public import, elements have public visibility inside the importin pancakes
- To avoid exposing the imported packages elements, you can use private import by specifying <<access>> stereotype

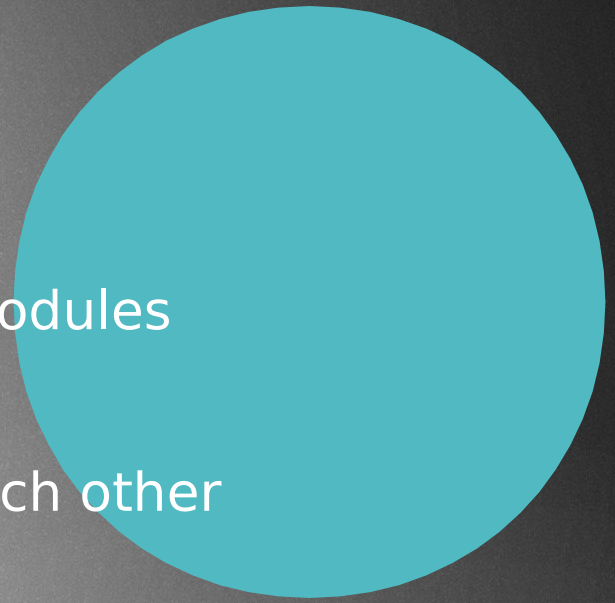
Example


A
<<import>>
B
<<import>>
C
<<access>>
D
Elements of package B
& C are available to A
Elements of package C
& D are available to B

```java
package com.poash.oop;

import drawing.* ;
import drawing.Graphics;

public class Program {
    private void Draw() {
        Graphics gph = new Graphics() ;
        //Full qualified name
        drawing.Graphics g = new drawing.Graphics() ;
```

When to use?

- To show high-level view of the system
- The view could be requirements or design
- Organize the complexity by dividing the system into modules
- Keep track of dependencies
- Understand how major parts of the system relate to each other

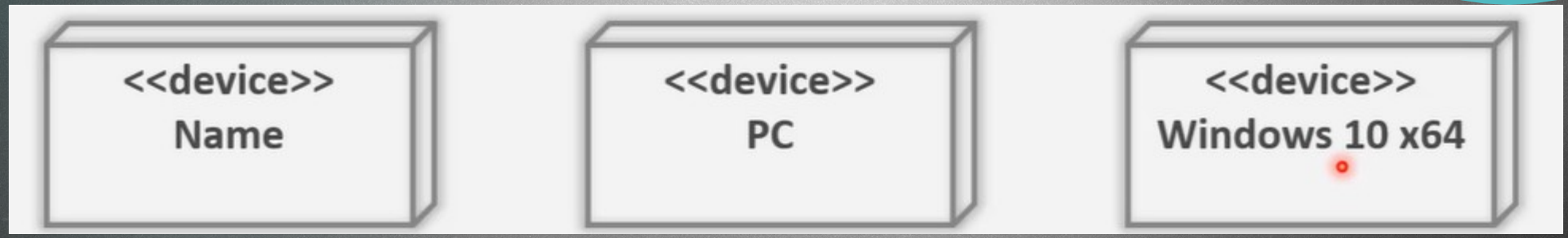


Deployment Diagram

- Represents the physical view in Kruchen’s 4+1 model
- Concerned with the physical elements of the system
- These elements are executable files and the hardware they run on
- Describes the elements, where they are located on the hardware and how the communicate with each other

Node

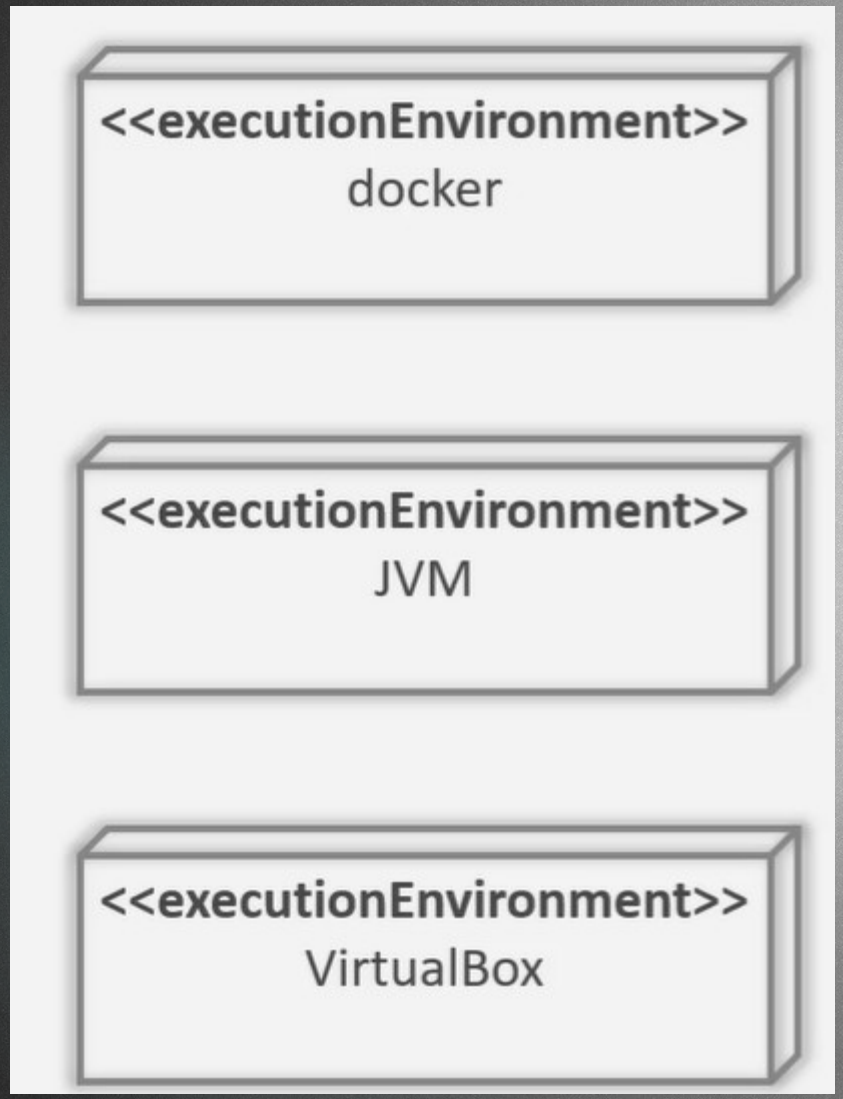
- The core part in a deployment diagram is a node
- It shows the computer hardware and is shown in a 3D box with <<device>> stereotype and a name
- The name can be general or specific name i.e. PC is general name, but windows 10 x64 is a specific name

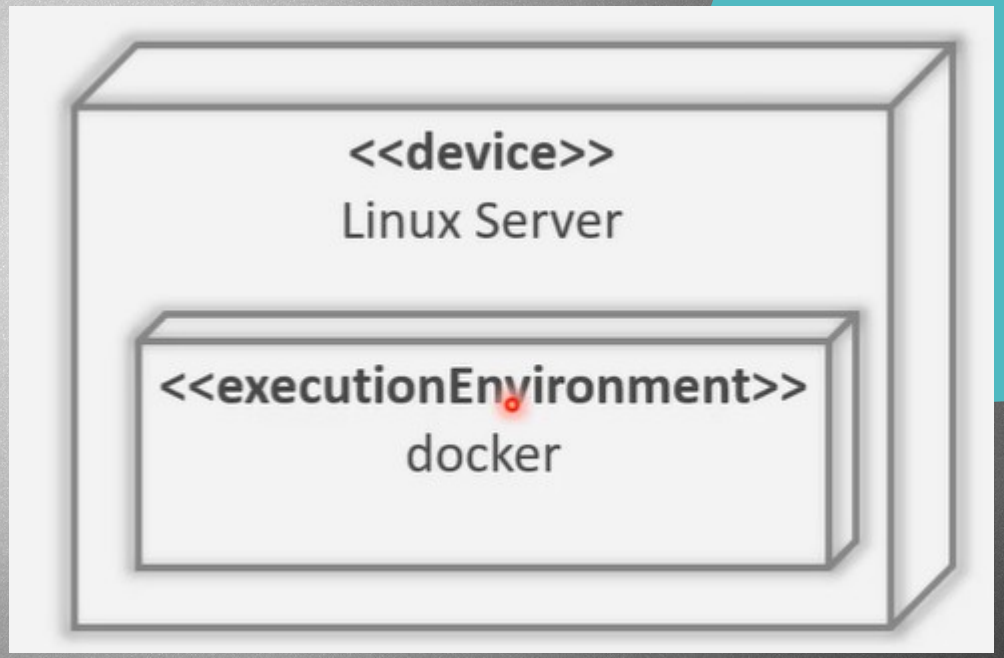


Node

- Nodes don’t need to represent a hardware
- Some type of the software provide an execution environment in which other software components can be executed
- So, a node can be hardware or software resources that can host software or related files
- A software node can be an application context
  - Generally not part of the software that is developed
  - Provide by a third-[arty that provides services to our software
- Examples of such execution environments are OS, Web server, Docker, App server, etc.

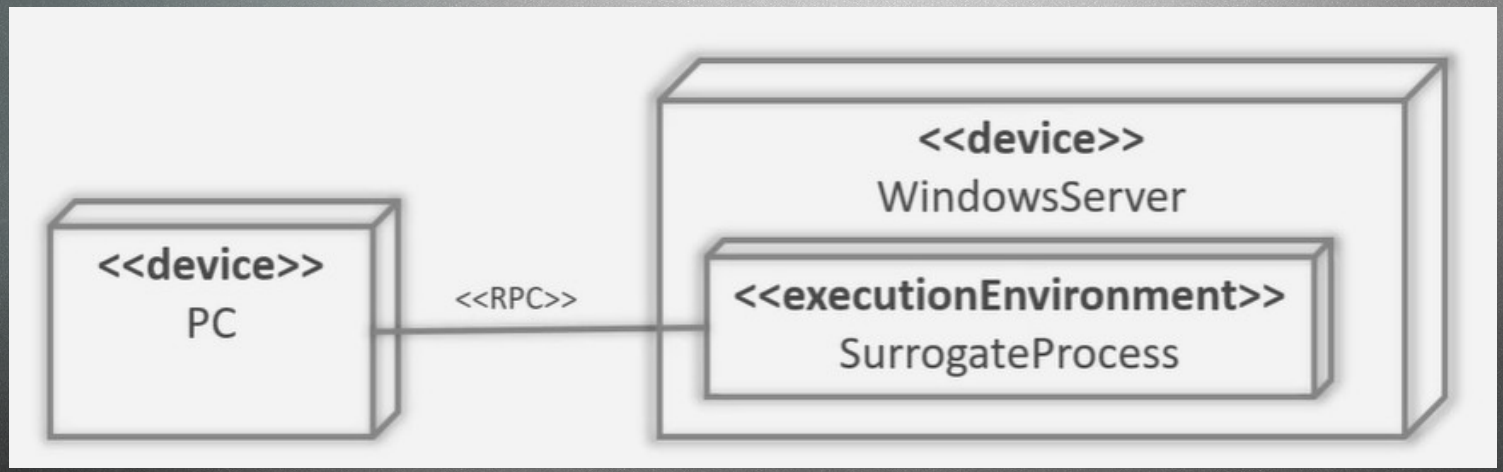
Example


<<executionEnvironment>>
docker
<<executionEnvironment>>
JVM
<<executionEnvironment>>
VirtualBox


<<device>>
Linux Server
<<executionEnvironment>>
docker

Node Communication

- A node may have to communicate with other nodes to fulfil its tasks
- This is shown through communication oaths; a solid line that connects two nodes
- The type of communication is shown through a stereotype



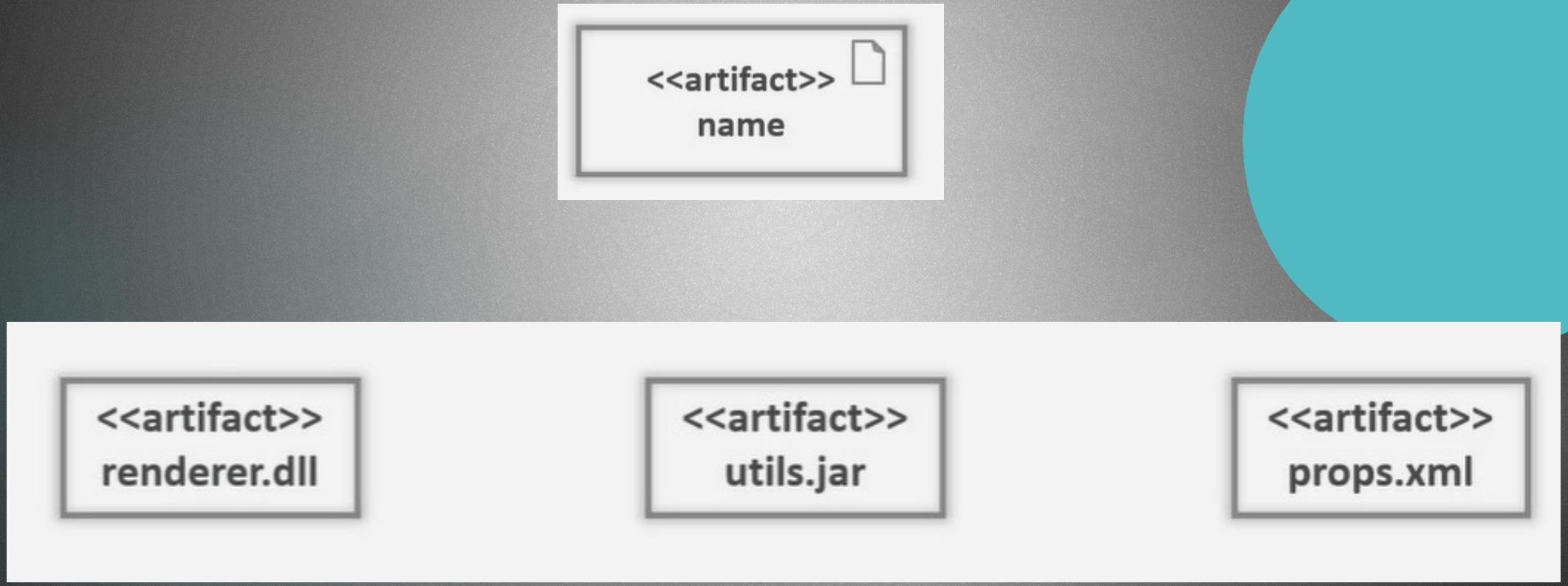
Artifacts

- An artifact is a physical file that is generated pout of software development process
- An artifact can be of the following type
  - Executable, such as .exe or .jar file
  - Libraries, such as .lib, .dll
  - Source files, such as .cpp, .java
  - Configuration files used by software att runtime, such as .xml or .ini

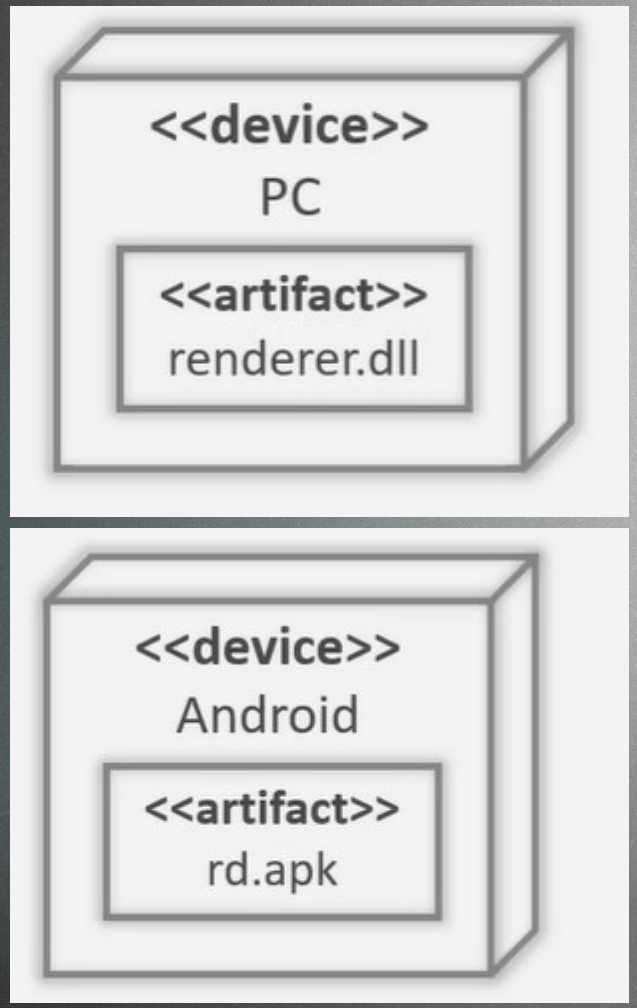
Artifacts

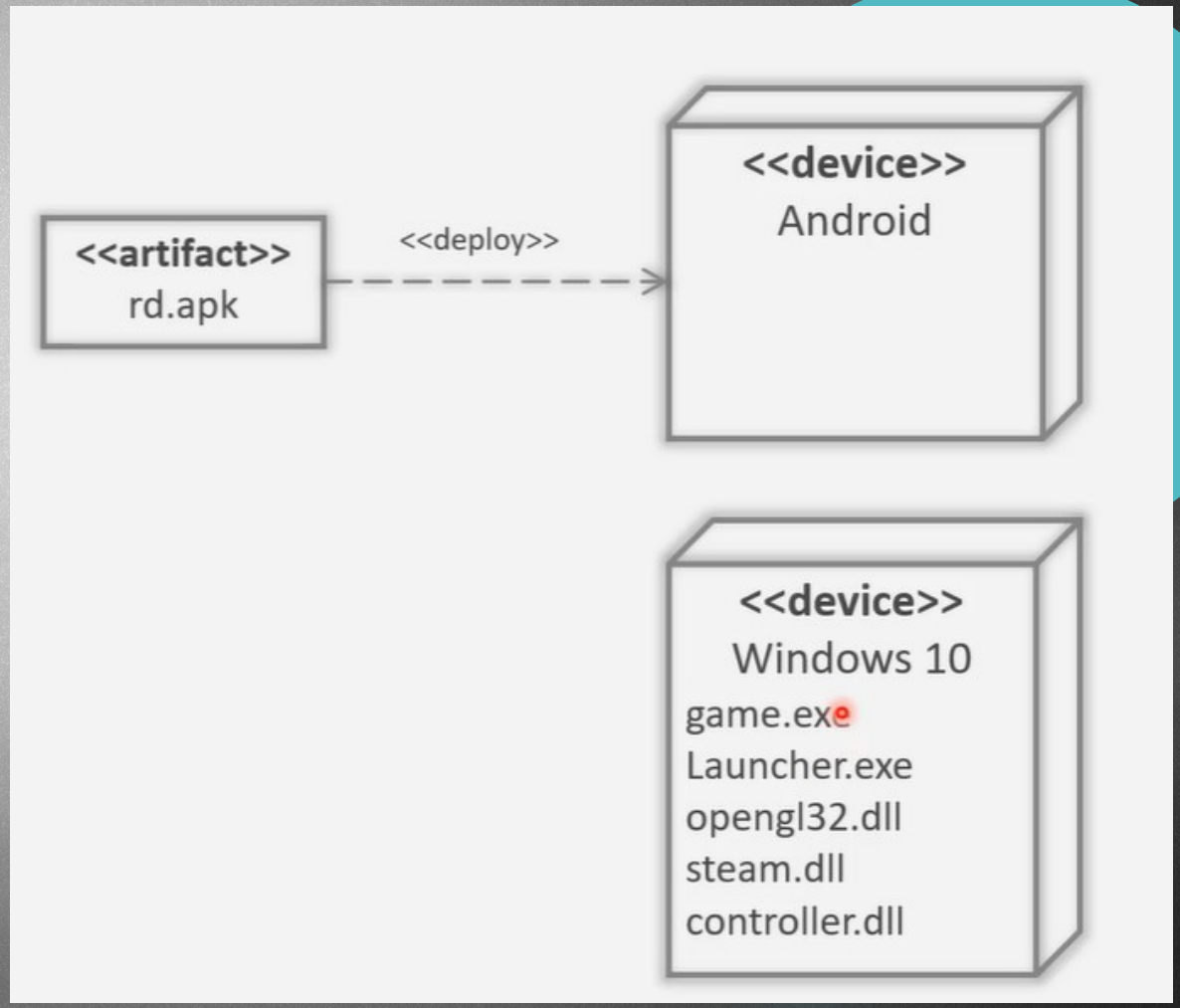
- UML provides multiple notations for represents artifacts
- Artifacts are represented in a rectangle
  - With the <<artifact>> stereotype
  - With a document icon in the top right corner
  - With a dependency arrow towards the node
- If there are too many artifacts, you can list them directly inside a node
- You can also show dependency between artifacts through a dependency arrow

Example


<<artifact>>
name
<<artifact>>
renderer.dll
<<artifact>>
utils.jar
<<artifact>>
props.xml

Example

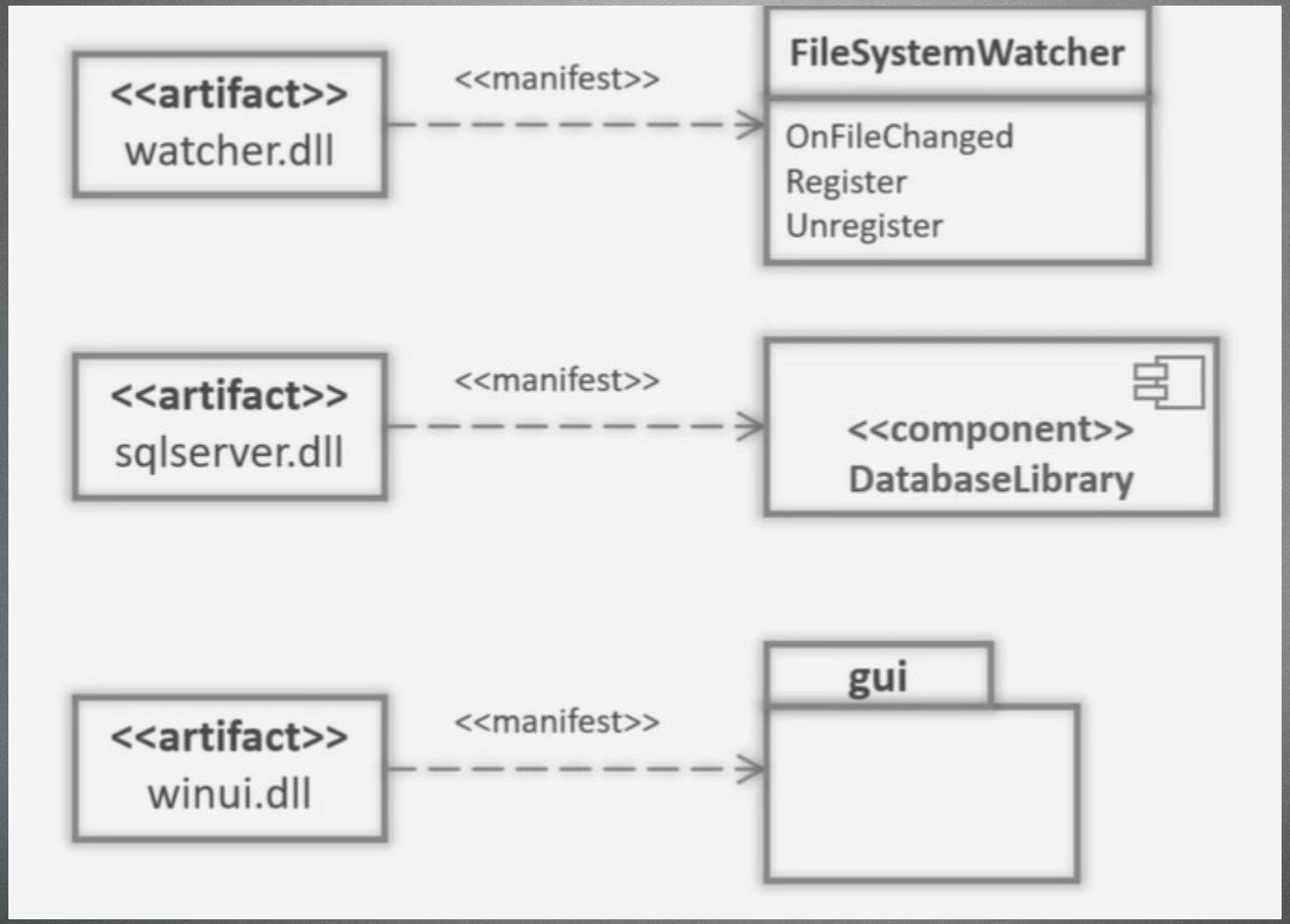

<<device>>
PC
<<artifact>>
renderer.dll
<<device>>
Android
<<artifact>>
rd.apk


<<artifact>>
rd.apk
<<deploy>>
<<device>>
Android
<<device>>
Windows 10
game.exe
Launcher.exe
opengl32.dll
steam.dll
controller.dll

Artifacts

- Software can be modularized through packages or components
- These are ultimately complied into one or more files or artifacts
- If an artifact is physical representation of a component, then the artifact *manifest* that component
- Note that artifact can manifest a class, package or a component
- This relationship is shown through a dependency arrow with <<manifest>> stereotype

Example


<<artifact>>
watcher.dll
<<manifest>>
FileSystemWatcher
OnFileChanged
Register
Unregister
<<artifact>>
sqlserver.dll
<<manifest>>
<<component>>
DatabaseLibrary
<<artifact>>
winui.dll
<<manifest>>
gui

Advantages

- Useful in all stages of the design process
- Especially, in the earlier stages, you can get an overview of the architecture of the system without dealing with details
- A rough sketch can be v=created that can be refined in the later stages of the software development
- The refinement can help express the current view of the physical system with other stakeholders

When to use?

- Create rough sketch of the physical layout of the system
- Discover issued involved in deployment process
- Identify & explore dependencies between your software & its environment
- Visualize the physical topology of the system's deployment