Android Application Design

Software System Design

Zhu Hongjun

Session 2: Architectural Design

- Architectural Design
- Software Design Principles
- Package Diagram and UML
- Android Architecture
- Java Programming Language
- Kotlin Programming Language
- Conclusions

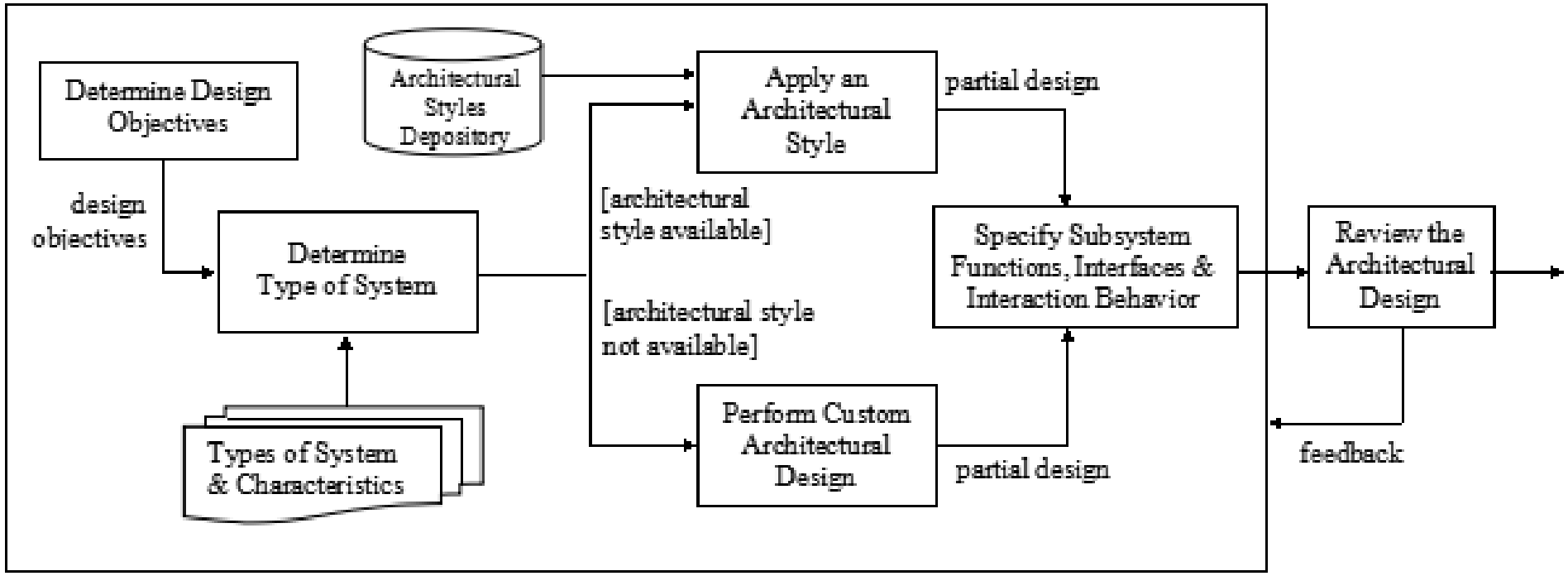
Architectural Design

- The Software architecture of a system refers to the style of design of the structure of the system including the interfacing and interaction among its components
- Software architectural design is a decision-making process to determine the software architecture for the system under development

Architectural Design

- Architectural design process
  - determine design objectives
  - determine type of system
  - apply an architectual style or perform custom architectural design
  - specify subsystem functions, interfaces, and interaction behavior
  - review the architectural design

Architectural design process



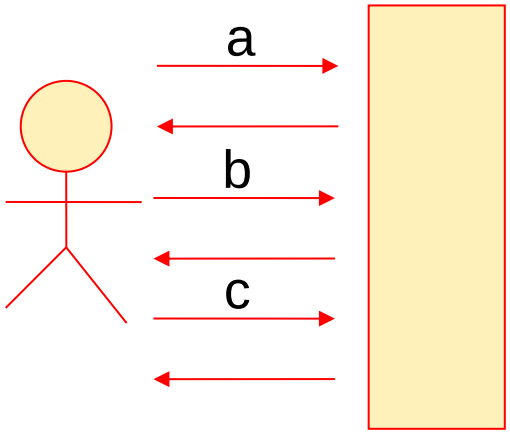
Architectural Design

- Determine design objectives
  - ease of change and maintenance
  - use of commercial off-the-shelf(COTS) parts
  - system performance
  - reliability
  - security
  - software fault tolerance
  - recovery

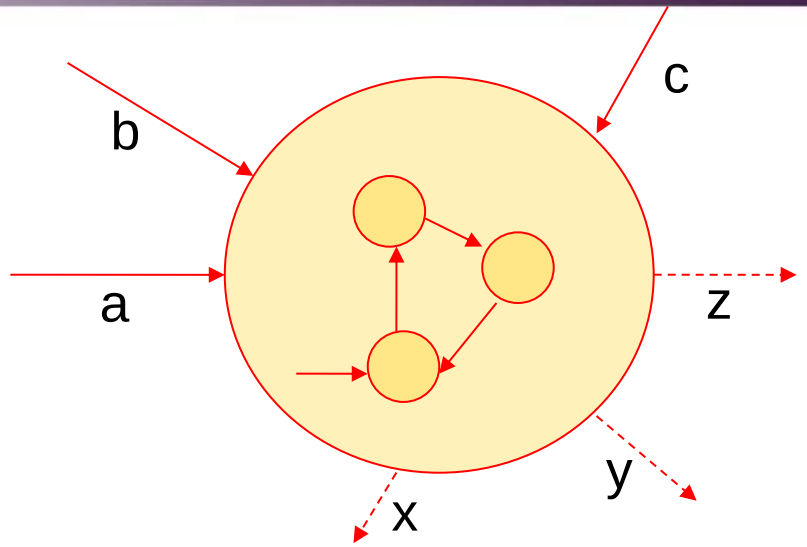
Architectural Design

- Determine system type
  - four types of systems
    - interactive subsystems
      - the interaction between system and actor
    - event-driven subsystems
      - state-dependent, commonly seen in embedded system
    - transformational subsystems
      - stateless, transforming input into output
    - object-persistence subsytems
      - storing and retrieving data from database or file system

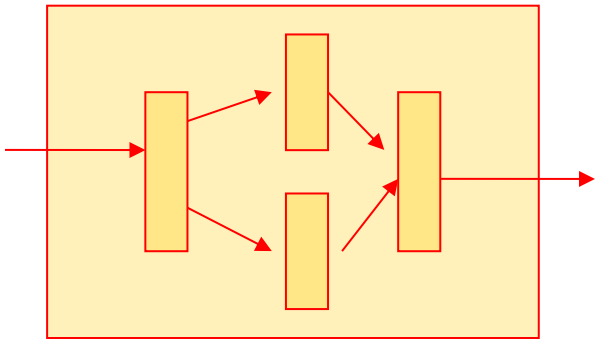
Four types of subsystem



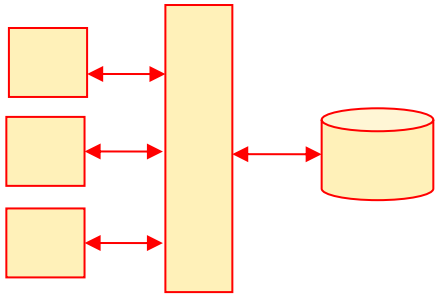
interactive subsystem



event-driven subsystem



transformational subsystem

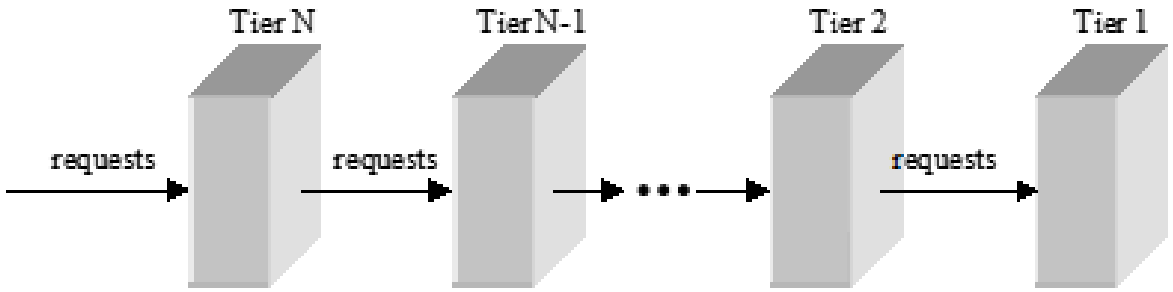


database subsystem

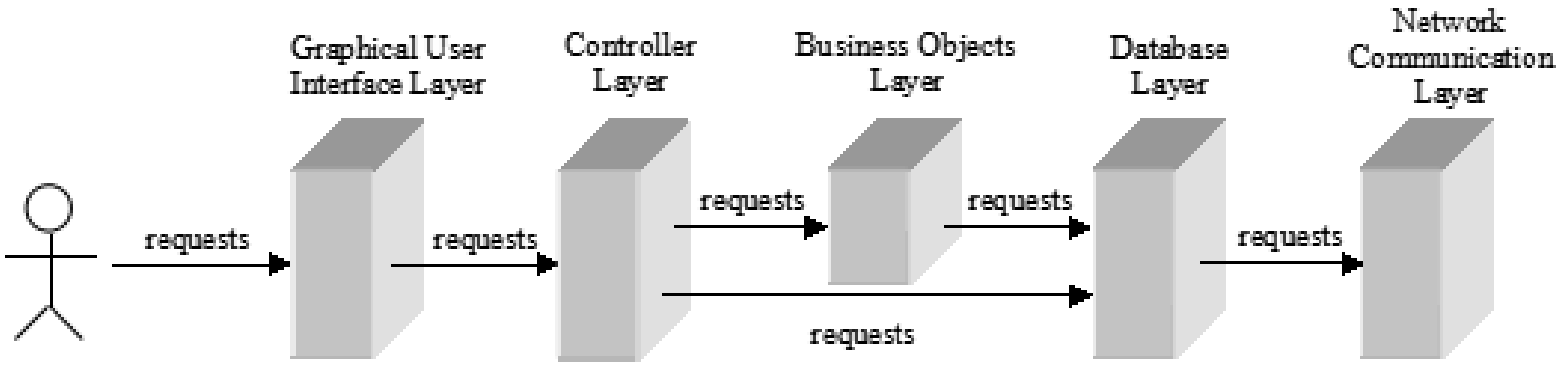
Architectural Design

- Applying architectural styles
  - architectural styles
    - N-tier architecture
    - Client-Server architecture
    - Main program and subroutine architecture
    - Event-driven system architecture
    - Pesistence framework architecture

N-tier architecture

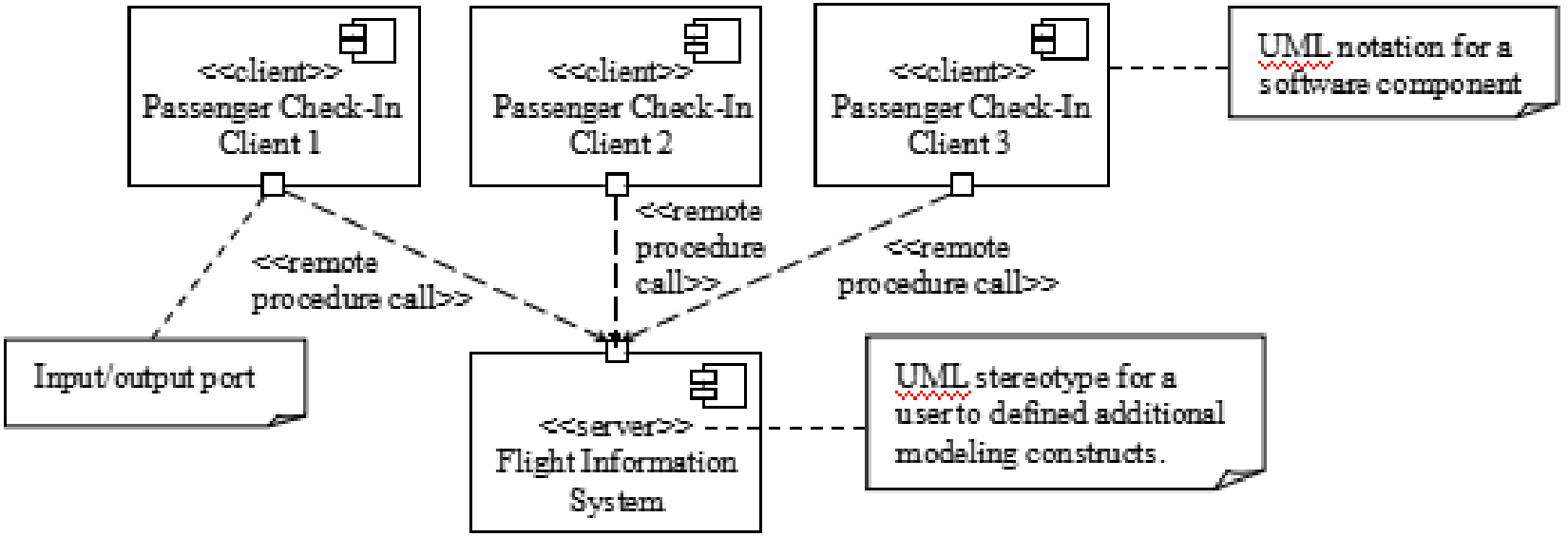


(a) N-tier architectural style

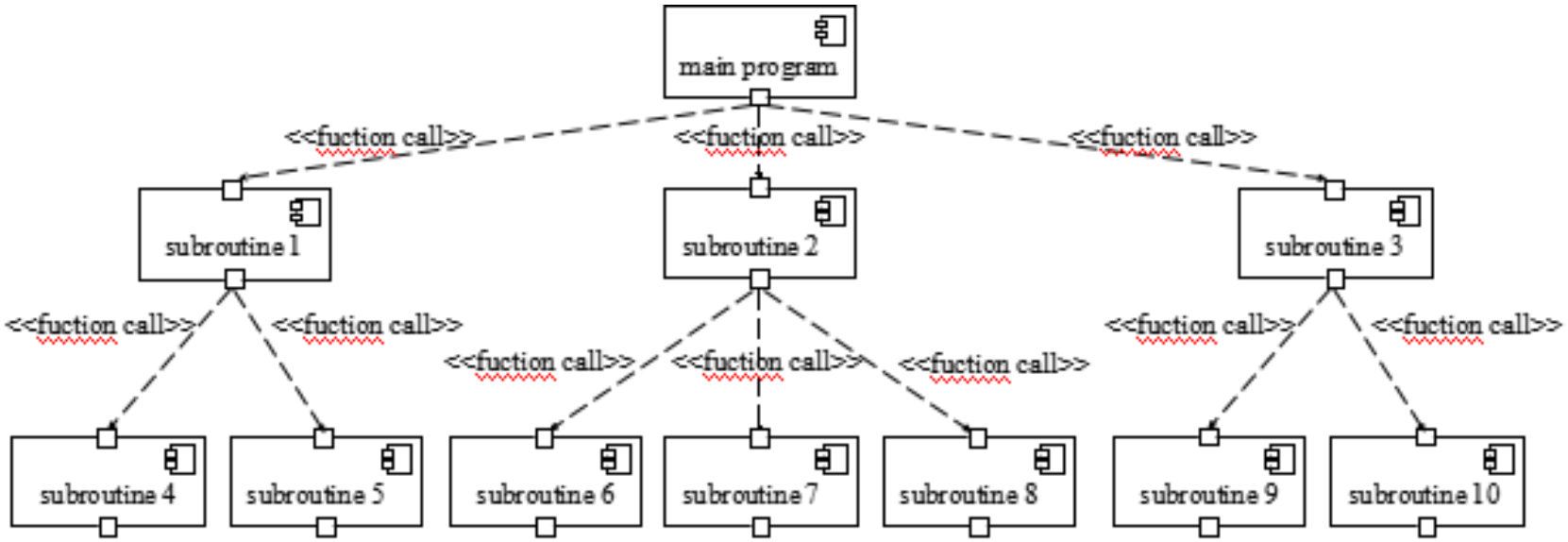


(b) An example N-tier architecture

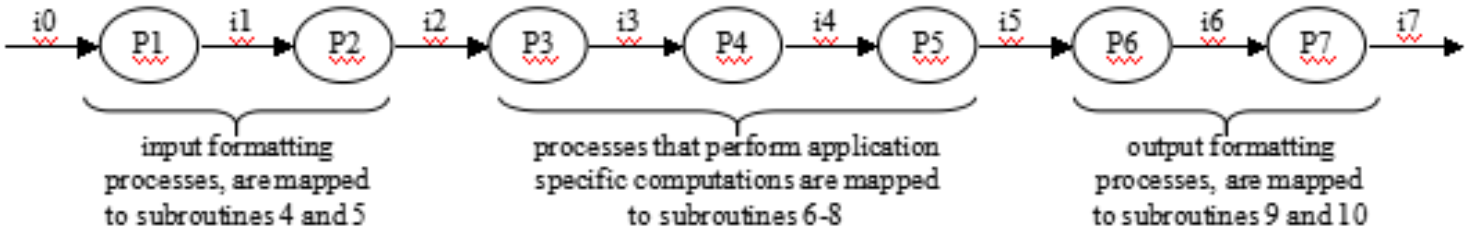
Applying client-server architectural style



Main program and subroutines architecture

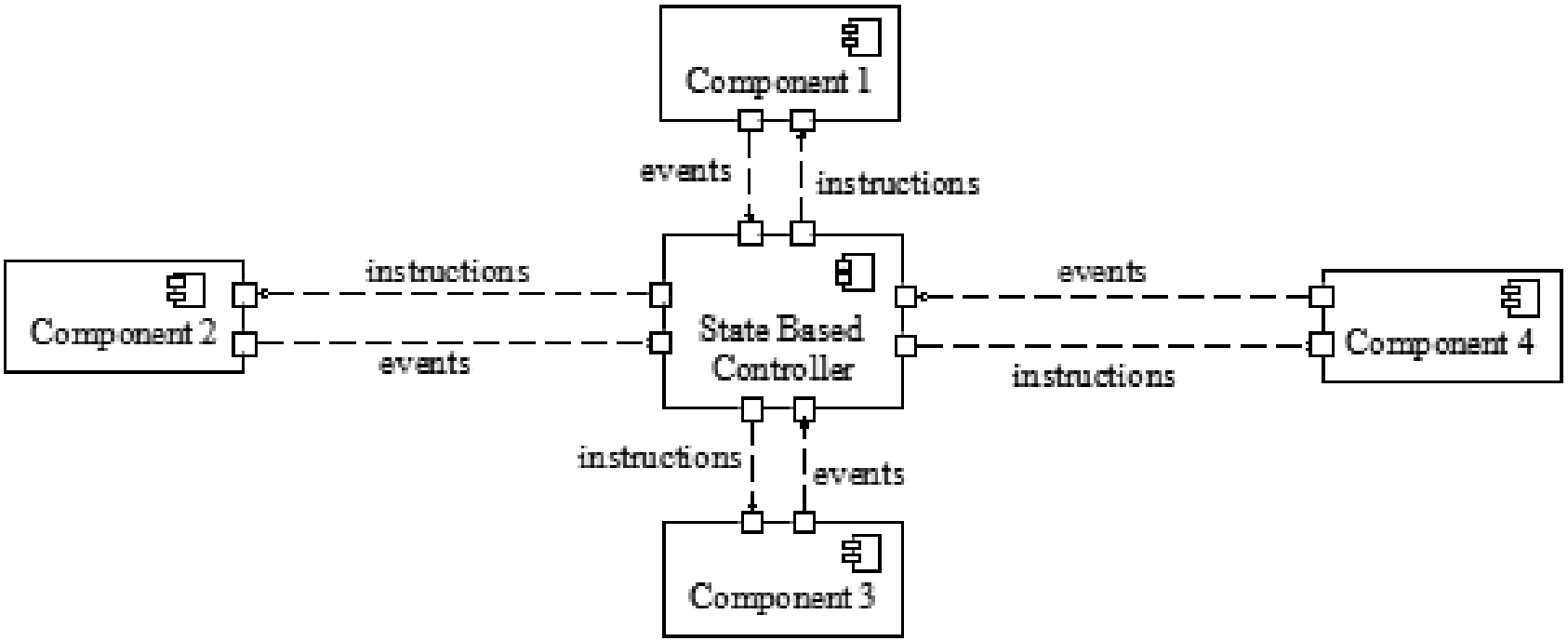


(a) A main program and subroutines architecture

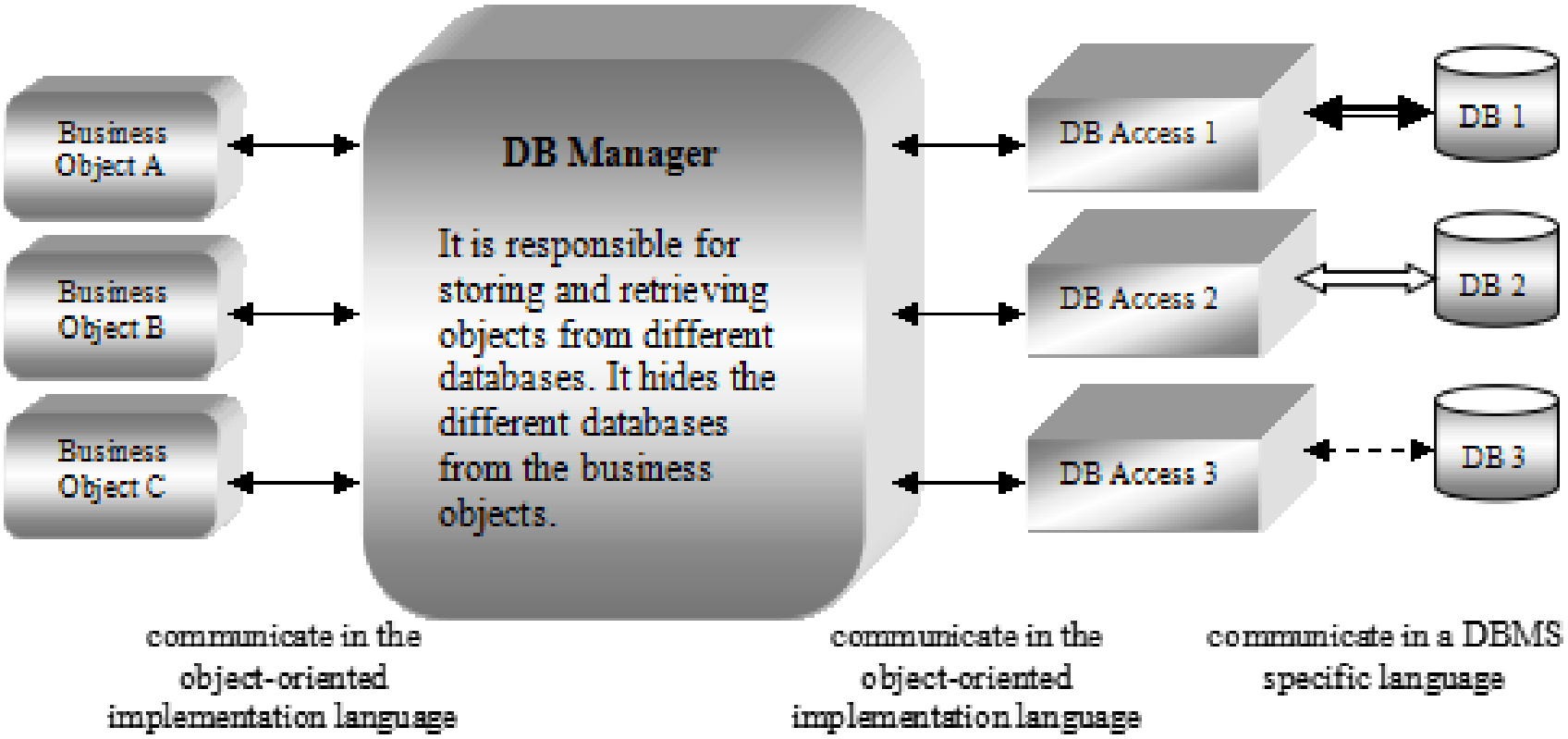


(b) A data flow model and its mapping to the architecture design

Event-driven system architecture



Persistence framework architectural style



Architectural Design

- Perform custom architectural design
  - design patterns are useful
- Specify subsystem functions and interfaces
  - define the input and output of each subsystem
  - specify the interaction behavior between the subsystems

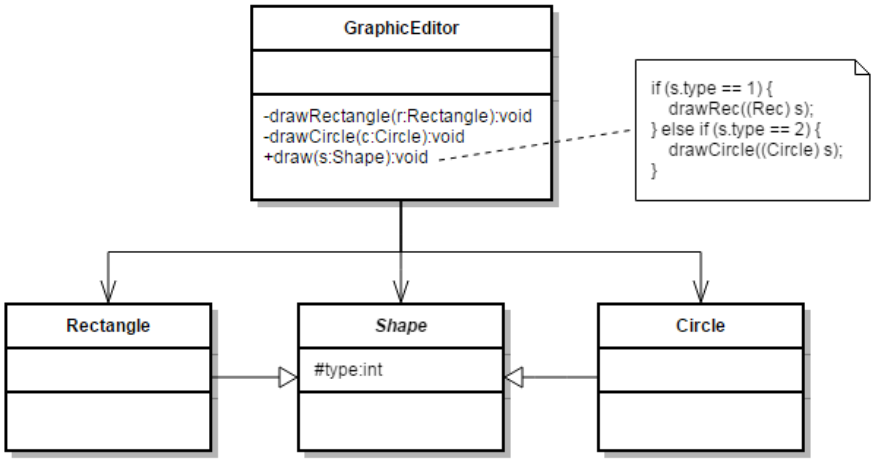
Architectural Design

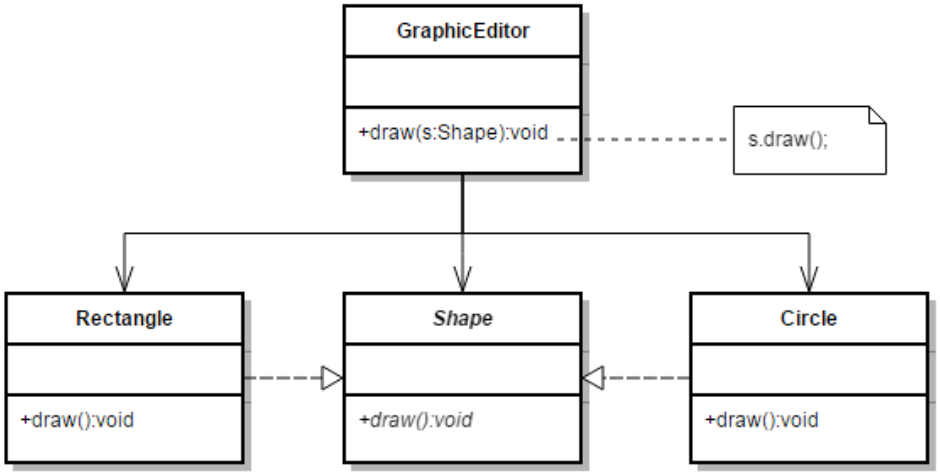
- Review the architectural design
  - the review verifies that if the design satifies software requirements and design objects
  - and if the design follows the software design principles
    - design for change
    - separation of concerns
    - etc.

Software Design Principles

- Design for change
  - a good design should yield a system that can adapt to change
- Open close principle
  - the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code

OCP Demo


GraphicEditor
-drawRectangle(r:Rectangle):void
-drawCircle(c:Circle):void
+draw(s:Shape):void
if (s.type == 1) {
drawRec((Rec) s);
} else if (s.type == 2) {
drawCircle((Circle) s);
}
Rectangle
Shape
#type:int
Circle


GraphicEditor
+draw(s:Shape):void
s.draw();
Rectangle
+draw():void
Shape
+draw():void
Circle
+draw():void

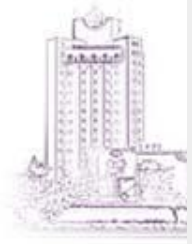
Software Design Principles

- Separation of concerns
  - a good design means than the responsibility of different concerns should be assigned to different subsystem or objects
- Interface segregation principle
  - clients should not be forced to implement interfaces they don't use

ISP Demo

```
// interface segregation principle - bad example
interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ...... eating in launch break
        }
}

class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

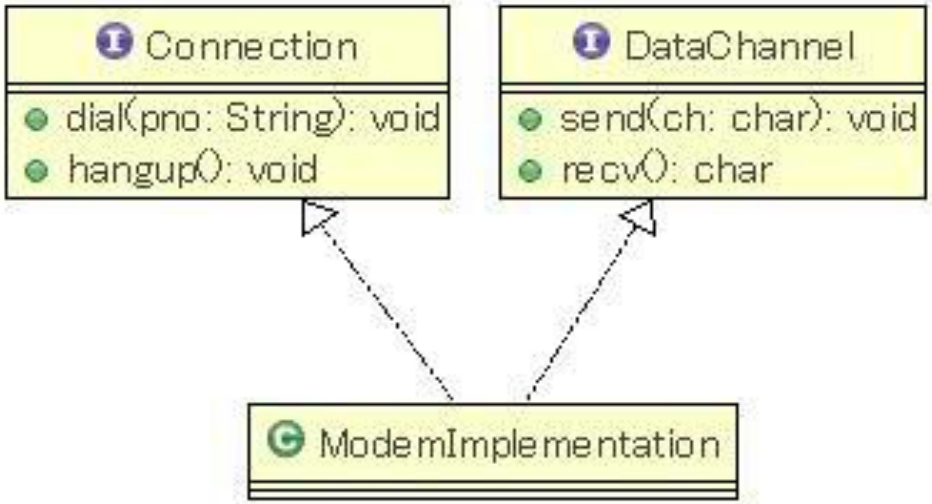
```
// interface segregation principle - good example
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
        public void work();
}

interface IFeedable{
        public void eat();
}

class Worker implements IWorkable, IFeedable{
        public void work() {
                // ....working
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Robot implements IWorkable{
        public void work() {
                // ....working
        }
}

class SuperWorker implements IWorkable, IFeedable{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}

class Manager {
        Workable worker;

        public void setWorker(Workable w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

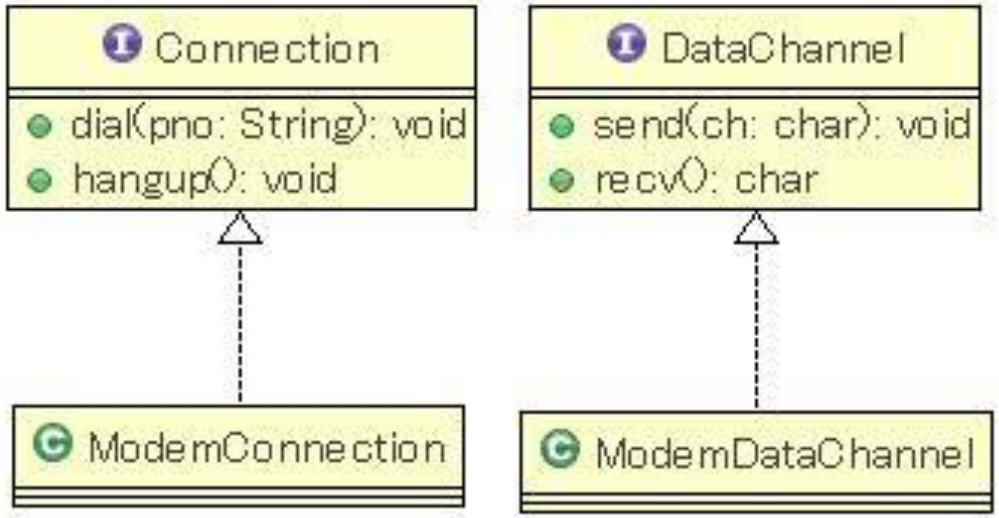


Software Design Principles

- Information hiding
  - a good design should hide implementation detail from the rest of the system
- Single responsibility principle
  - a class should have only one reason to change

SRP demo



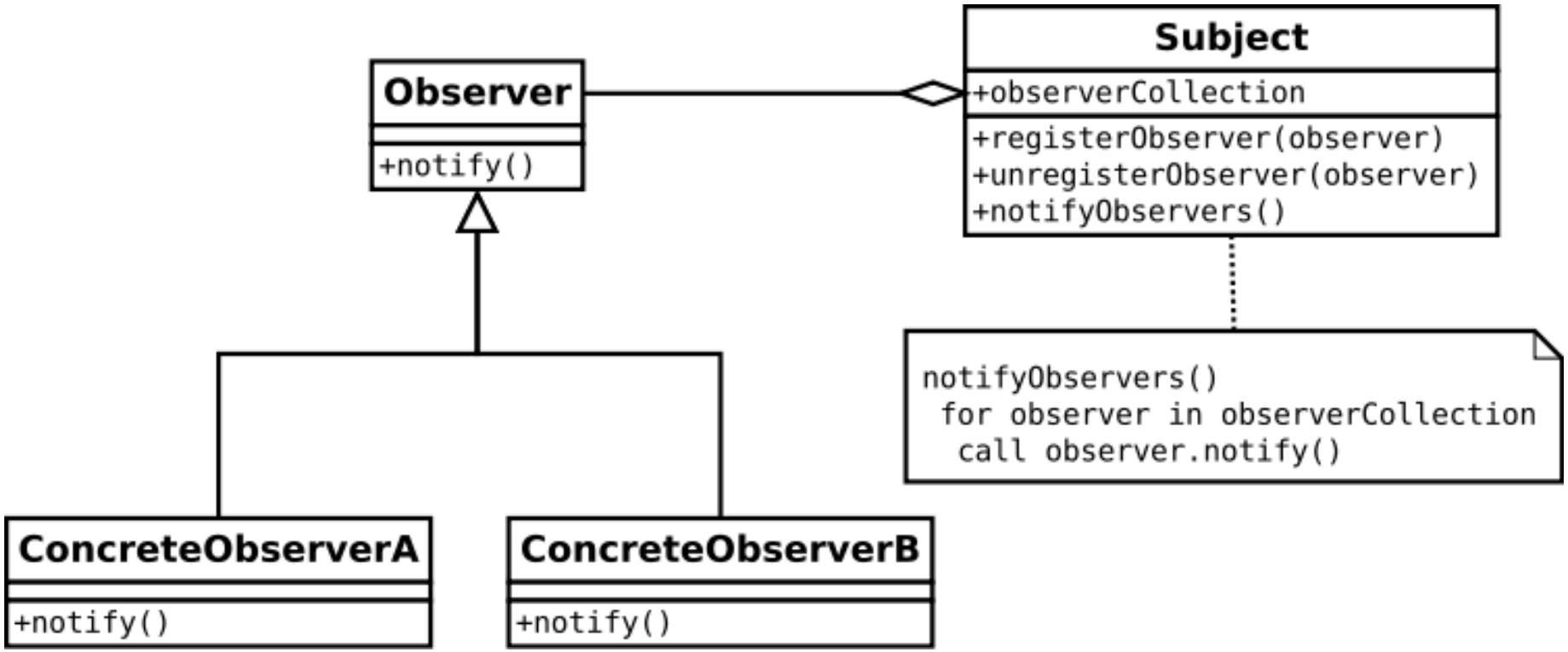


cn/~waterzhj

Software Design Principles

- Keep it simple and stupid
  - a good design should avoid unnecessary complexity
  - less is more
- Hollywood principle
  - don't call us, we will call u
  - IoC, DI
  - Observer pattern

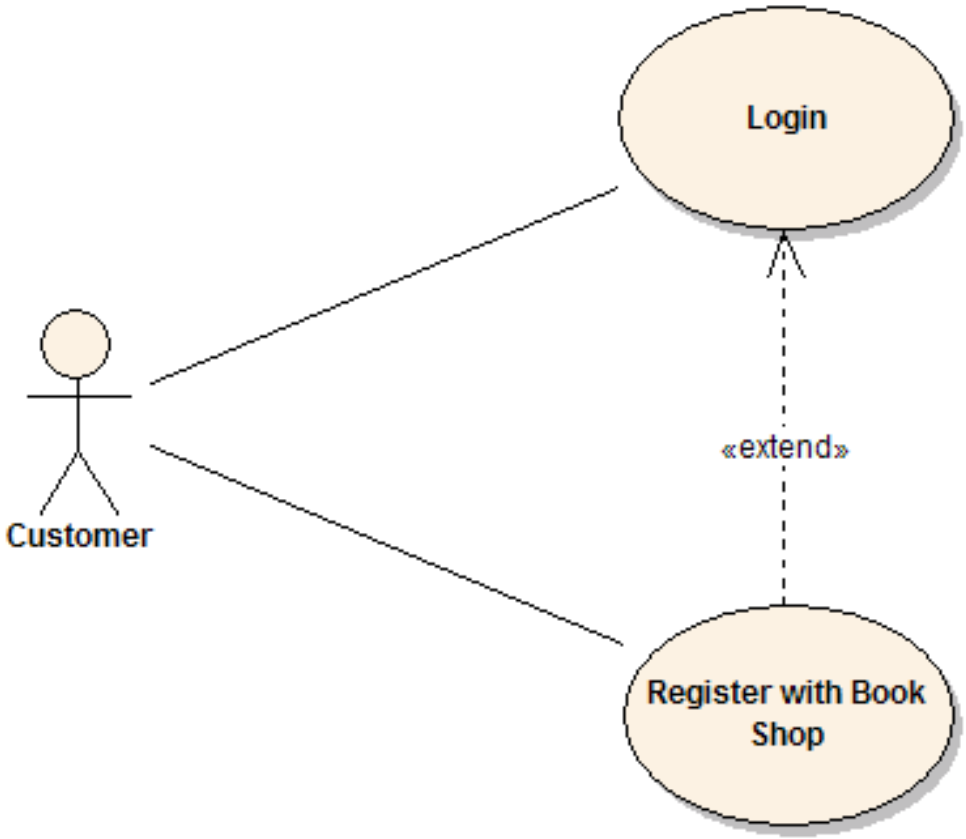
Hollywood principle demo



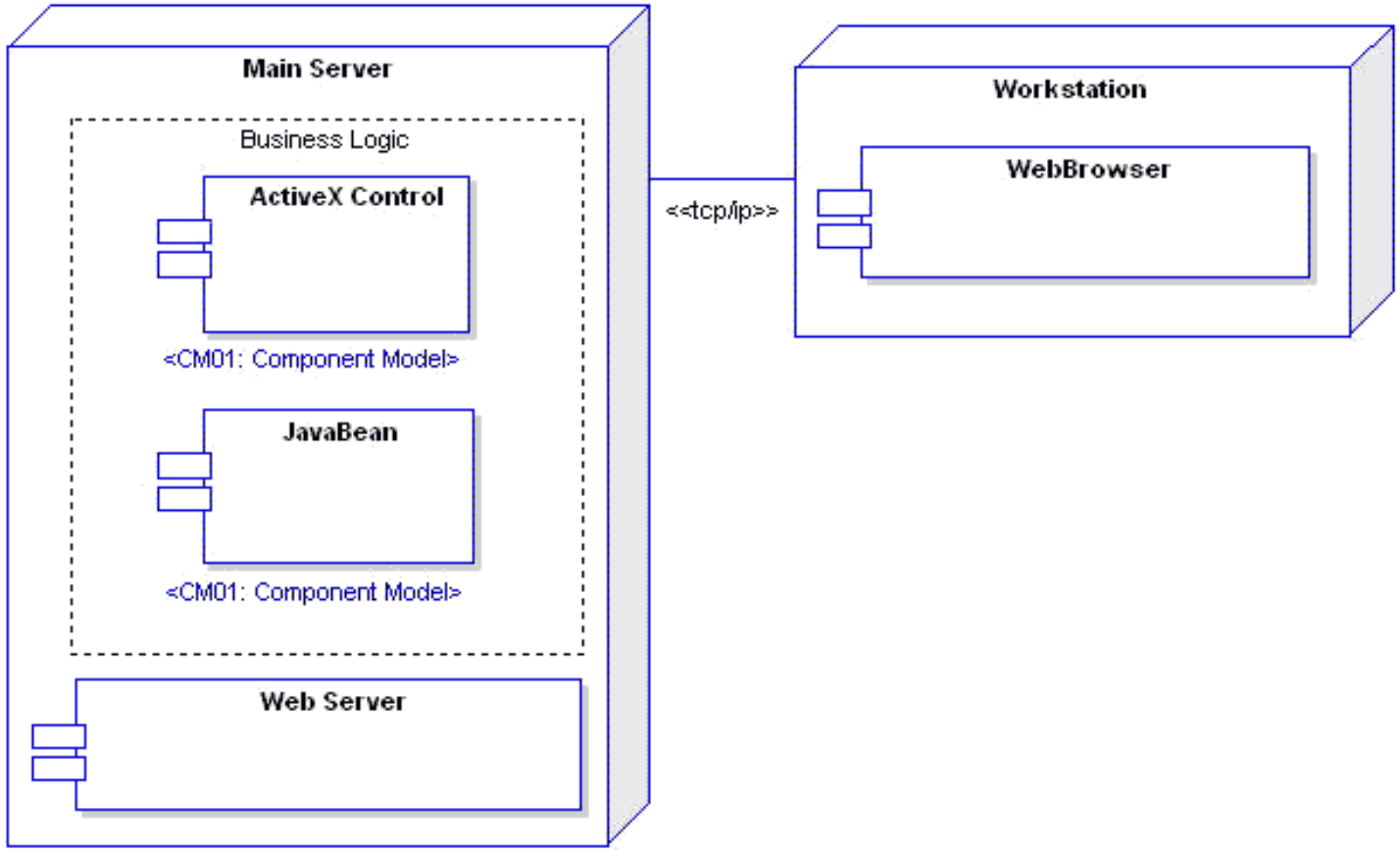
Package Diagram and UML

- UML
  - Unified Modeling Language
  - UML 2.0 defines thirteen types of diagrams, three represent general types of behavior
    - Structure Diagrams
      - Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram
    - Behavior Diagrams
      - Use Case Diagram, Activity Diagram, State Machine Diagram
    - Interaction Diagrams
      - Sequence Diagram, Communication Diagram, Timing Diagram, Interaction Overview Diagram

UC diagram demo

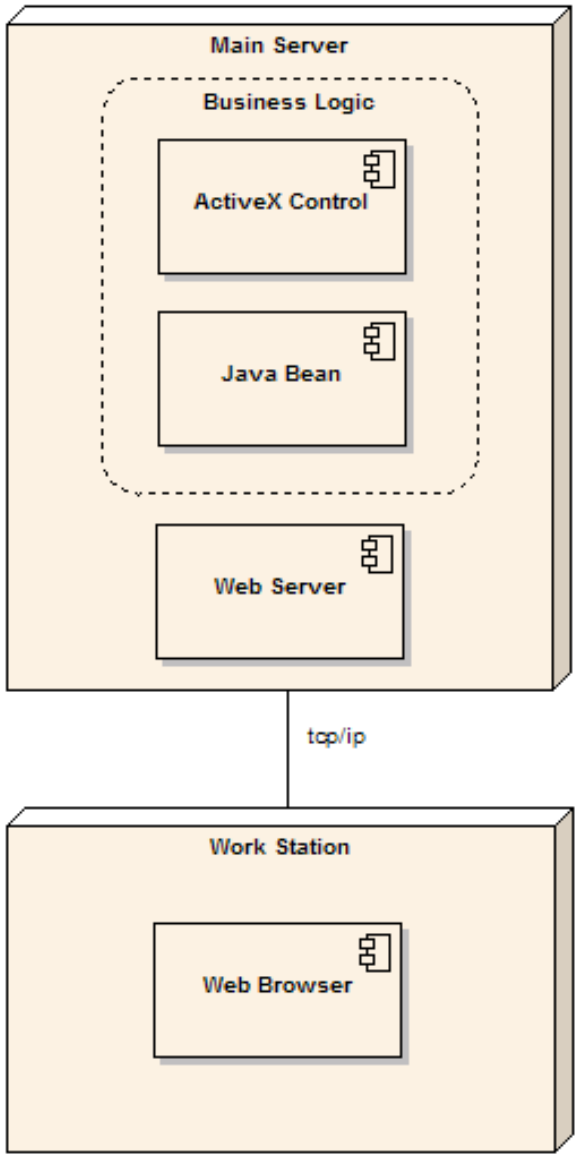


Component diagram demo

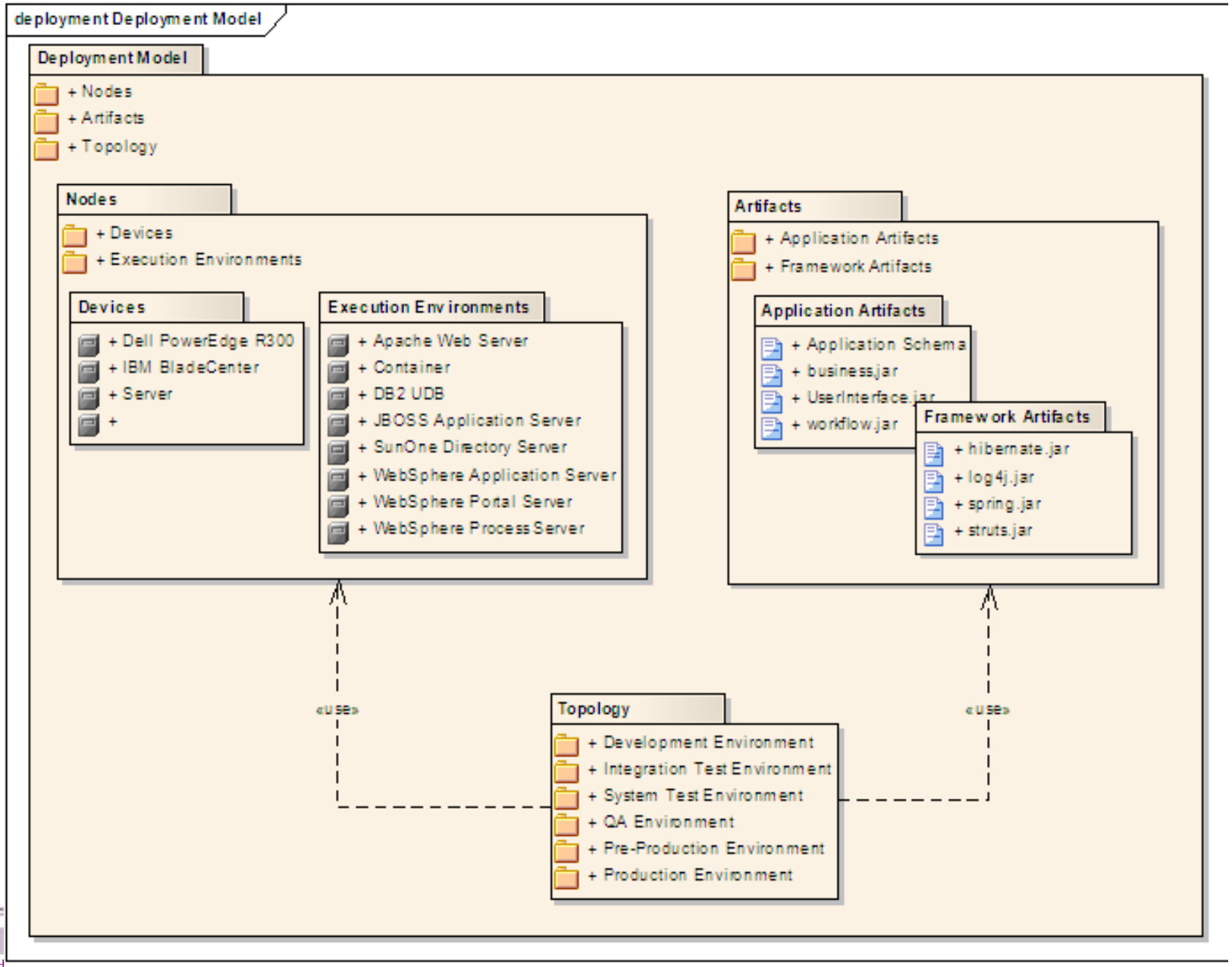


Deployment diagram demo

The physical model shows where and how system components will be deployed. It is a specific map of the physical layout of the system.



Package diagram demo



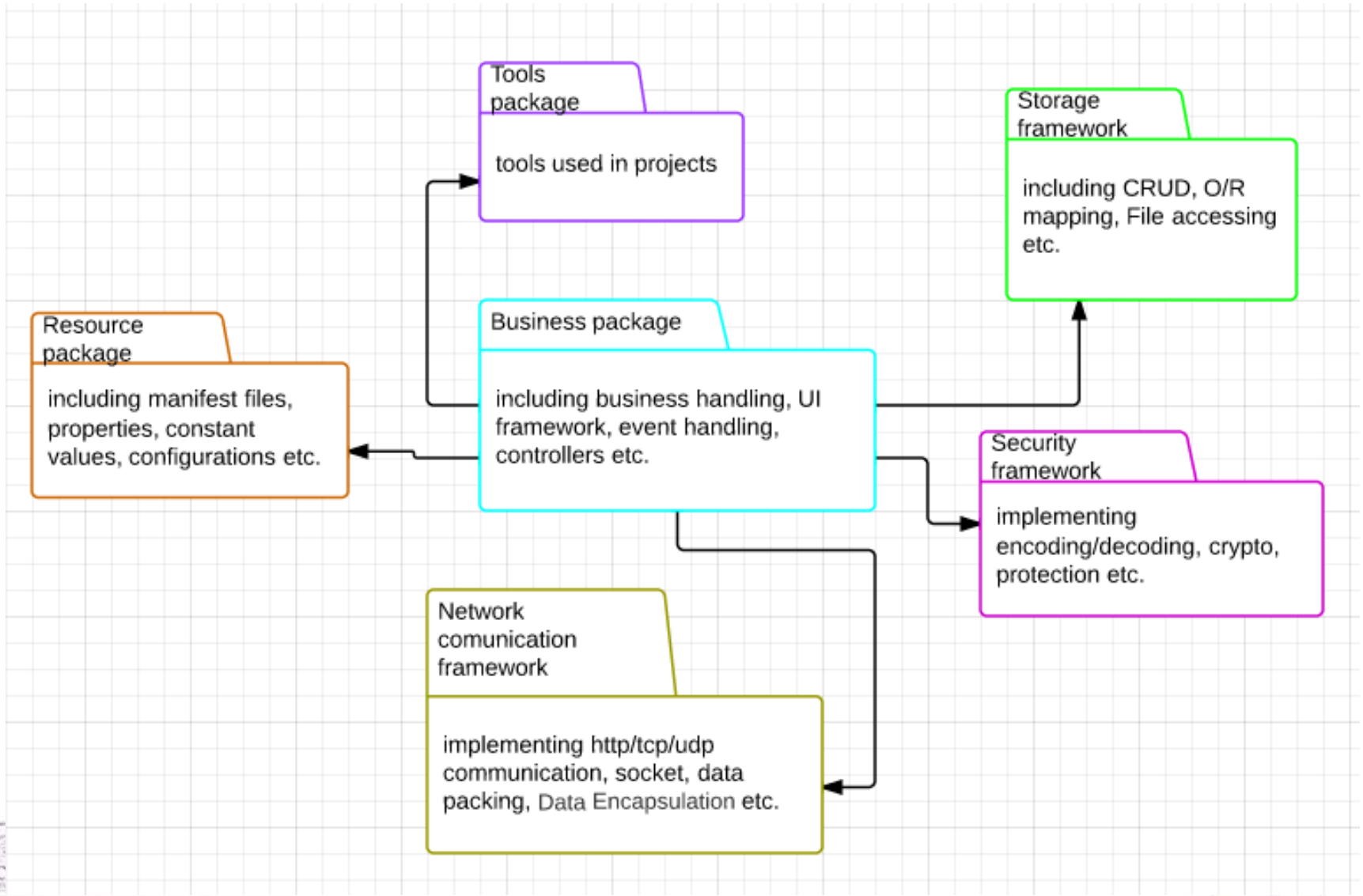
Package Diagram and UML

- Package diagram
  - package diagram shows the arrangement and organization of model elements in middle to large scale project
  - package diagram can show both structure and dependencies between sub-systems or modules

Package Diagram and UML

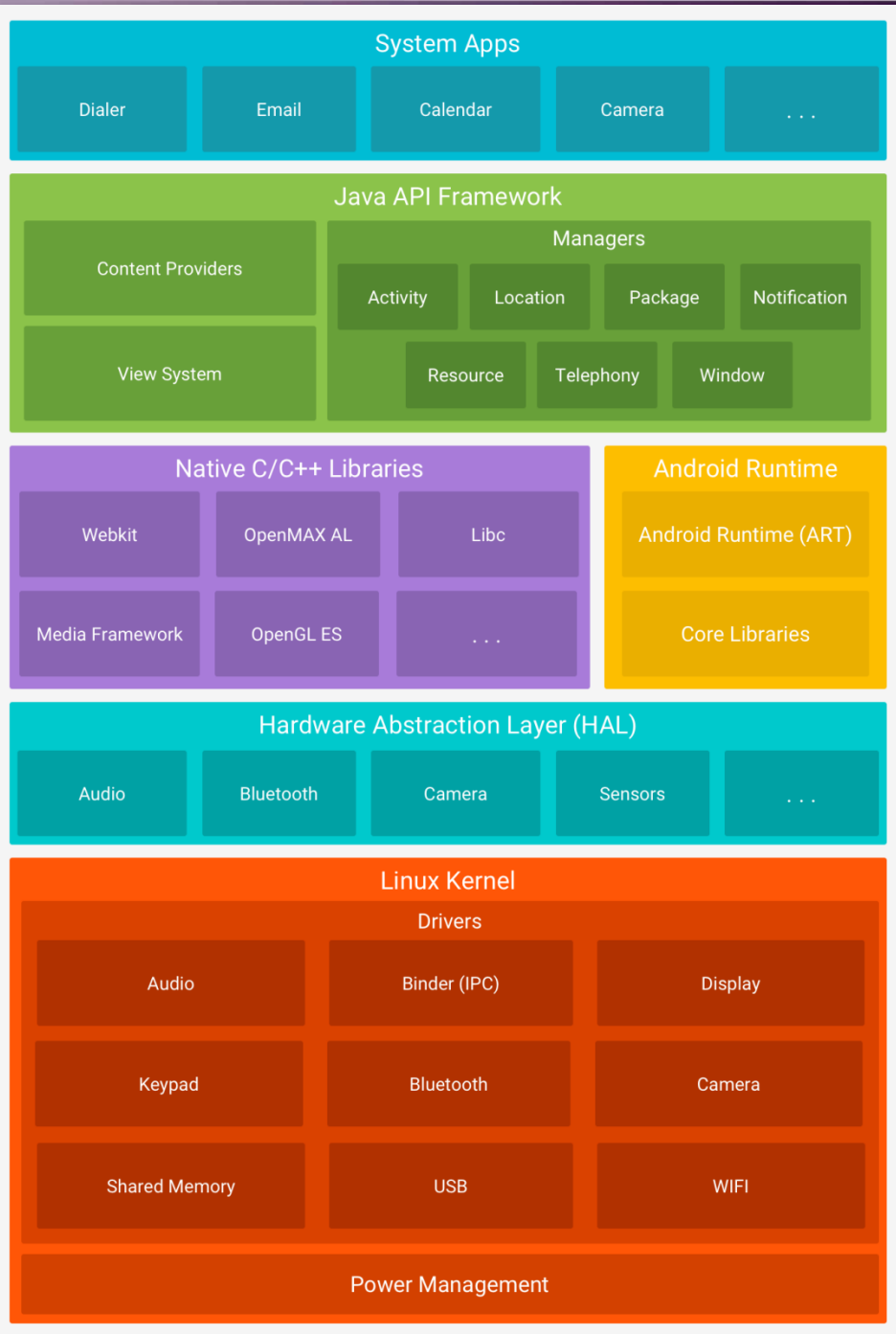
- Package diagram (cont.)
  - notations

Common packages in projects

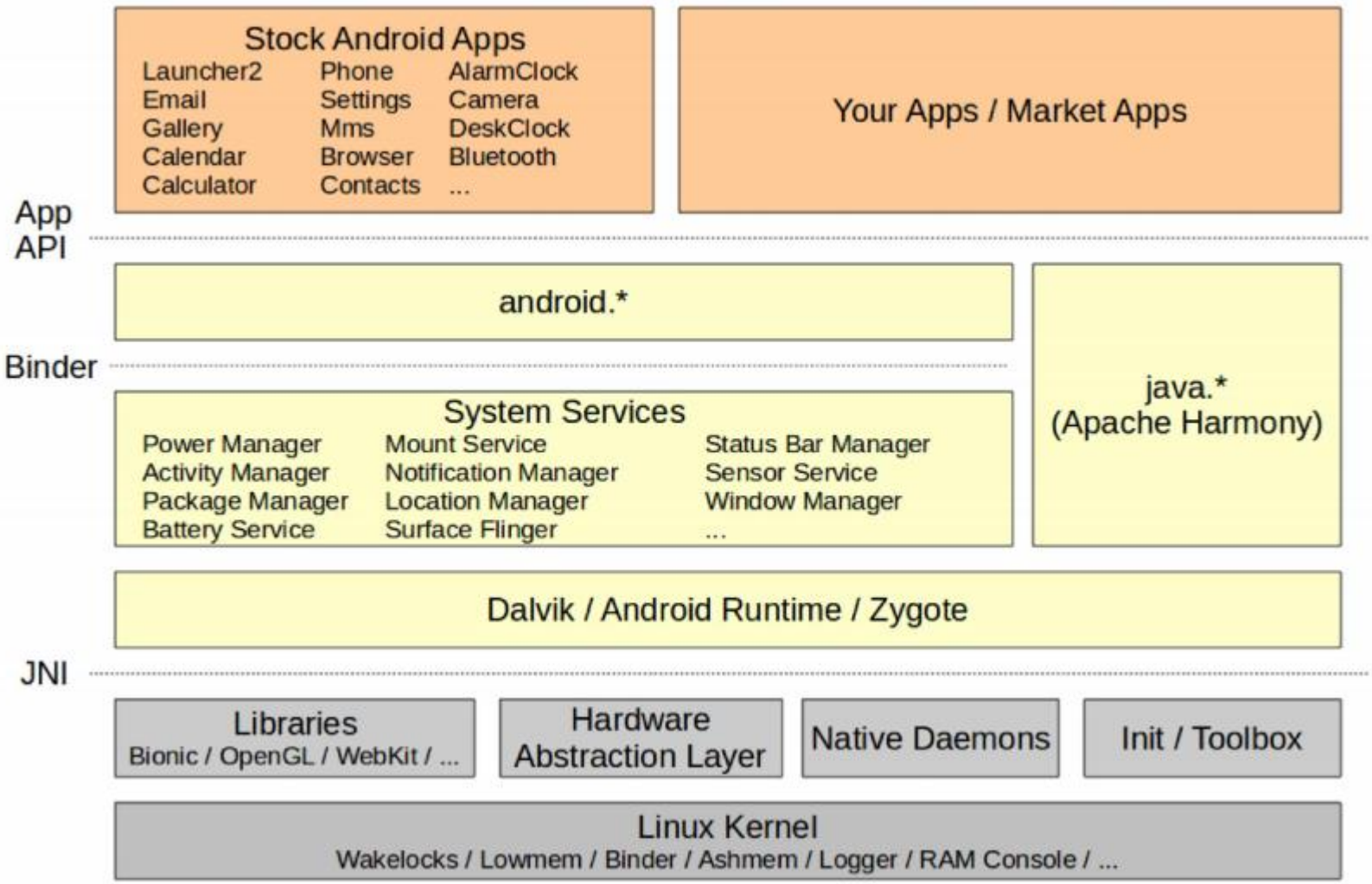


Android Architecture

Android应用软件设计　朱洪军　ht



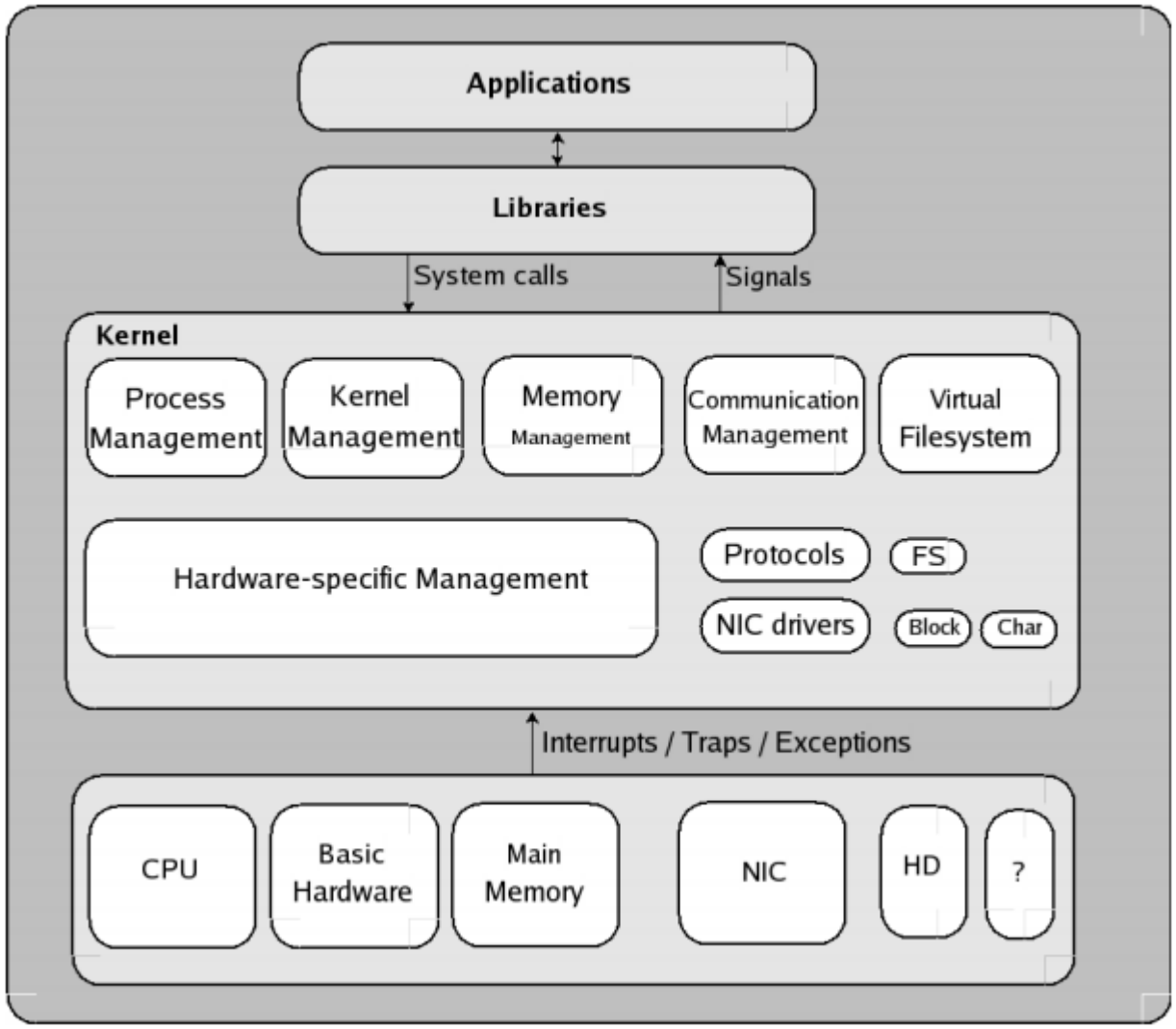
Android Architecture



Android Architecture

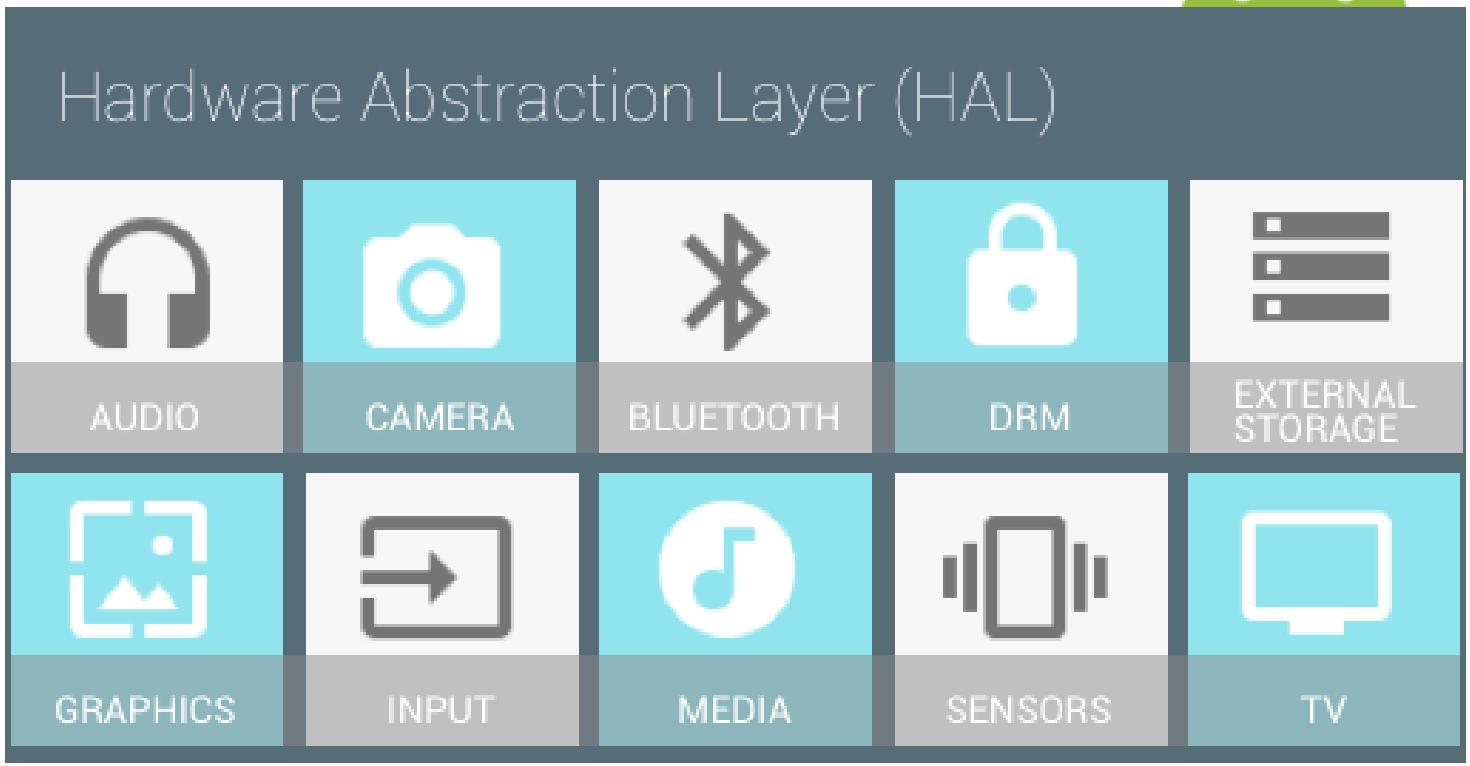
- Linux Kernel（Version 2.6）
  - Core services (including hardware drivers, process and memory management, security, network, and power management) are handled by a Linux 2.6 kernel
  - The kernel also provides an abstraction layer between the hardware and the remainder of the stack

Linux kernel



Android Architecture

- HAL
  - defines a standard interface for hardware vendors to implement and allows Android to be agnostic about lower-level driver implementations
  - HAL implementations are packaged into modules (.so) file and loaded by the Android system at the appropriate time



HAL components

Android Architecture

- Runtime and Libraries
  - Runtime
    - Dalvik Virtual Machine (before 5.0)
      - Execute .dex files
    - Android Runtime (ART, 5.0 and higher)
      - Execute .oat files
    - Core Libraries
      - Compile Java code to .dex file

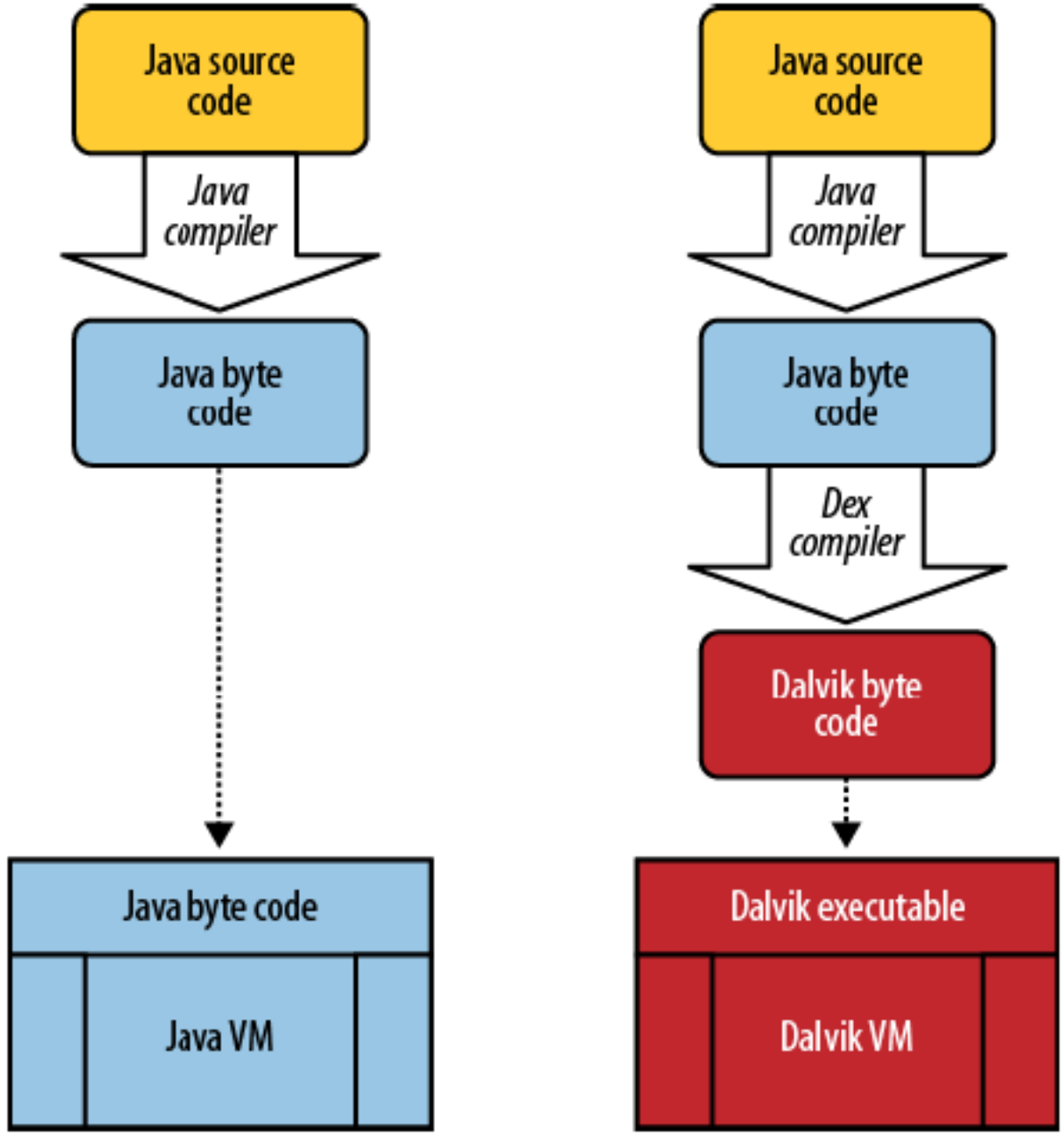
Android Architecture

- Libraries（C/C++）
  - Running on top of the kernel, Android includes various C/C++ core libraries such as libc and SSL
    - A media library for playback of audio and video media
    - A surface manager to provide display management
    - Graphics libraries that include SGL and OpenGL for 2D and 3D graphics, etc.

Android Architecture

- Dalvik VM, Based on QEMU (before 5.0)
  - Depending on linux kernel， execute .dex file
    - .class file is object file, which needs to be linked as .dex file for running in Dalvik VM
    - Dx tool is used to linke .class file to .dex file
    - Dex is a compressed file format, being used in limited memory system
  - Multiple Dalvik instances can be running at the same time

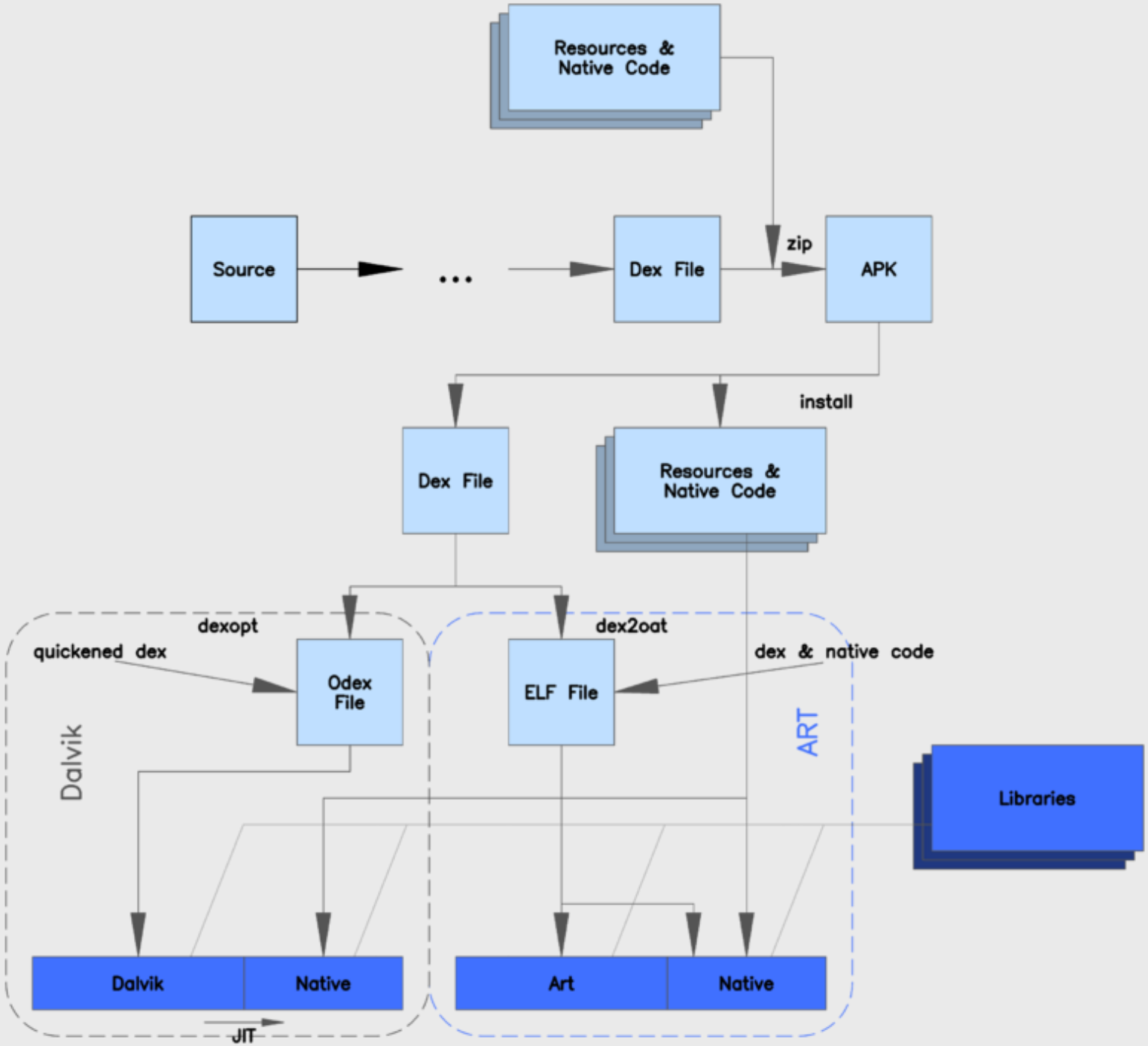
Java versus Dalvik



Android Architecture

- ART (5.0 and higher)
  - execute .oat file
    - a private elf file owned by android
  - introduces the use of ahead-of-time (AOT) compilation by compiling entire applications into native machine code upon their installation
    - improves the overall execution efficiency and reduces power consumption

ART versus Dalvik



Android Architecture

- Once installed on a device, each Android application lives in its own security sandbox
- The android system implements the principle of least privilege (POLP)
  - The Android operating system is a multi-user Linux system in which each application is a different user
  - By default, the system assigns each application a unique Linux user ID
  - Each process has its own virtual machine (VM), so an application's code runs in isolation from other applications
  - By default, every application runs in its own Linux process

Android Architecture

- Application Framework
  - An array of managers that provide services for
    - Activity、Views、Content Providers
    - Resources、Windows, etc.

Android Architecture

- Application Framework (cont.)

FRAMEWORK SERVICE	DESCRIPTION
Activity Manager	Manages Intent resolution/destinations, app/activity launch, and so on
View System	Manages views (UI compositions that a user sees) in activities
Package Manager	Manages information and tasks about packages currently and previously queued to be installed on the system
Telephony Manager	Manages information and tasks related to telephony services, radio state(s), and network and subscriber information
Resource Manager	Provides access to non-code app resources such as graphics, UI layouts, string data, and so on
Location Manager	Provides an interface for setting and retrieving (GPS, cell, WiFi) location information, such as location fix/coordinates
Notification Manager	Manages various event notifications, such as playing sounds, vibrating, flashing LEDs, and displaying icons in the status bar

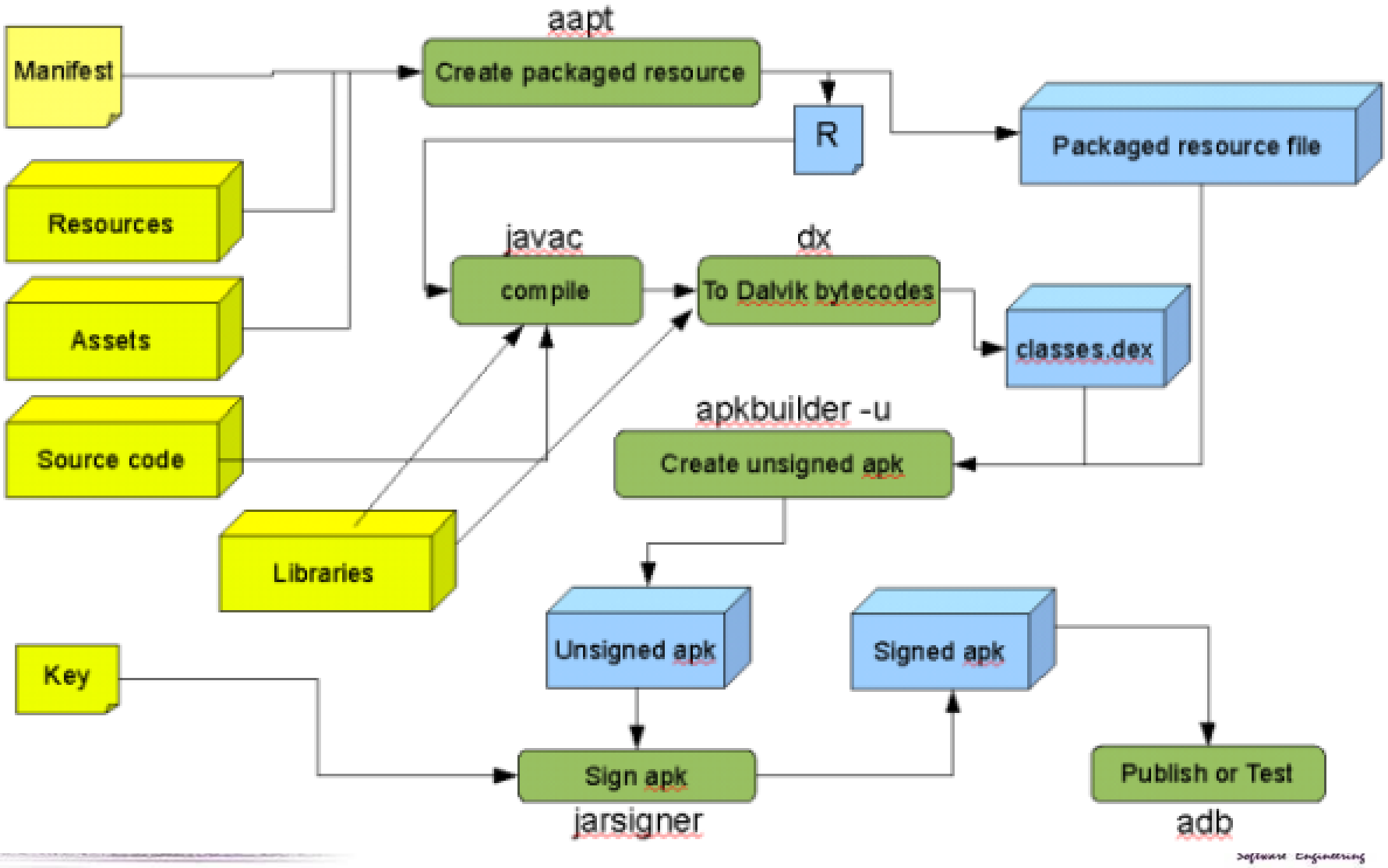
Android Architecture

- Applications
  - Different user applications written in Java
  - Such as
    - Contacts
    - Browser
    - Etc.

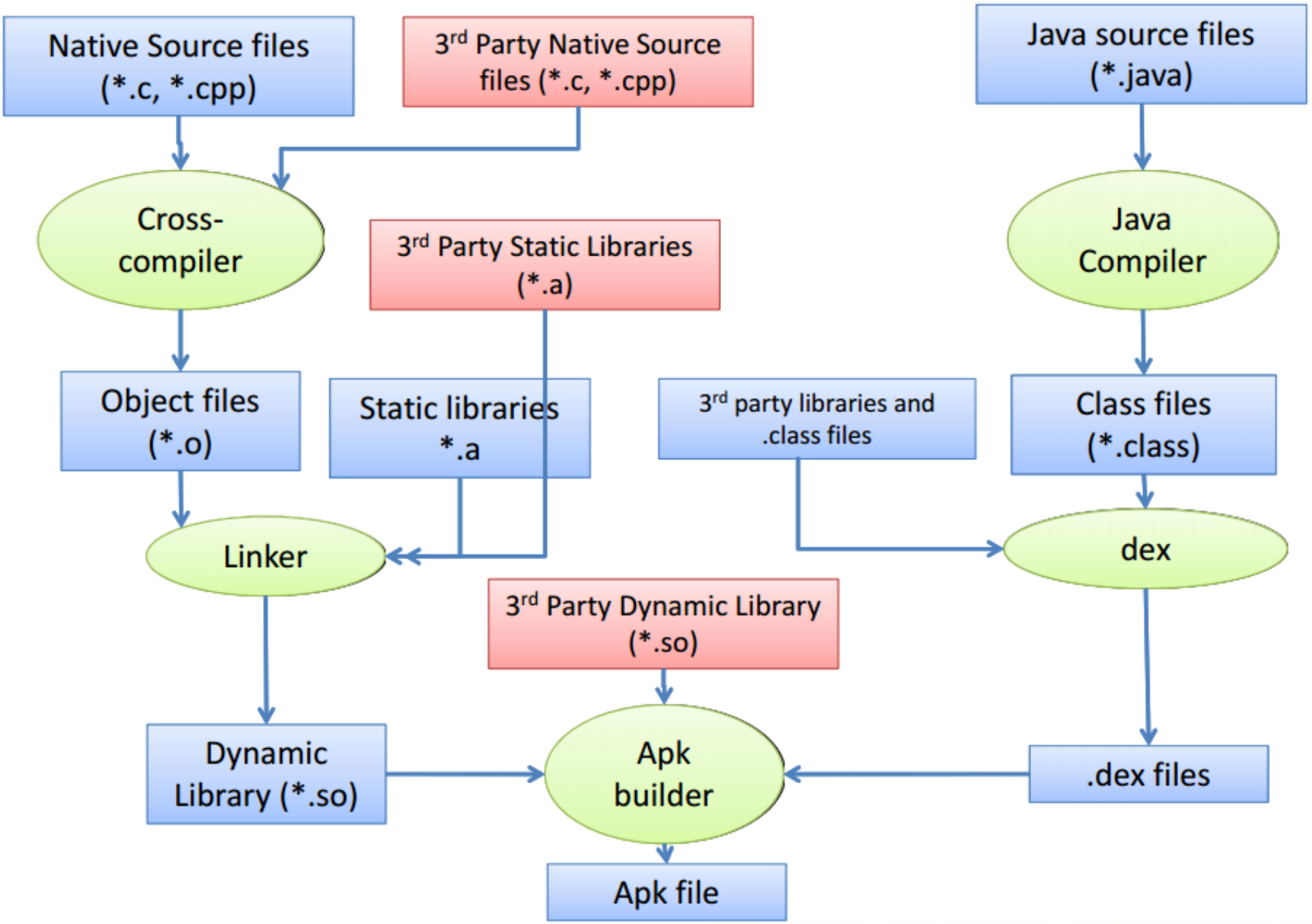
Android Architecture

- Applications (cont.)
  - APK
    - *Android package*, Android SDK tools compile the code—along with any data and resource files—into an *Android package*, an archive file with an .apk suffix
    - All the code in a single .apk file is considered to be one application
    - .apk is the file that Android-powered devices use to install the application

APK build process



APK build tool chain diagram

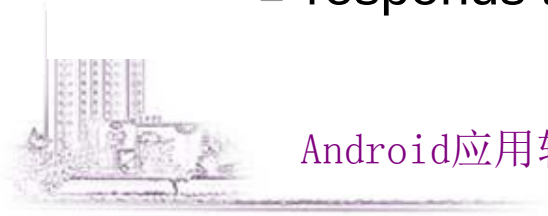


Android Architecture

- Application Components
  - 1 android app = N components
  - apps can use components of other applications
  - app processes are automagically started whenever any part is needed
    - one app has N entry points, !1, and !main()
  - four types of components
    - activity, service, content provider, broadcast receiver

Android Architecture

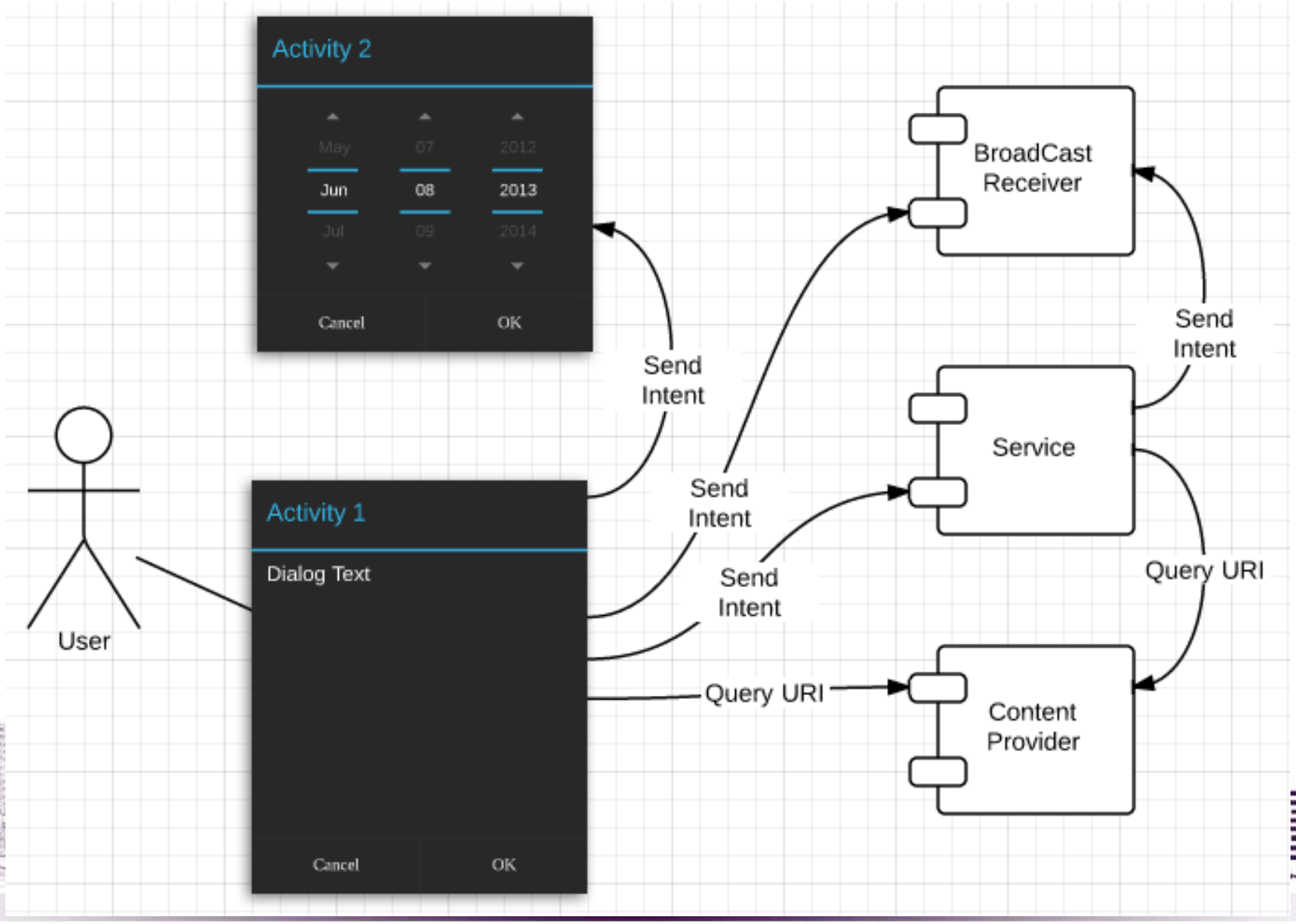
- Application Components (cont.)
  - Activities
    - represents a single screen with a user interface
  - Services
    - runs in the background to perform long-running operations or to perform work for remote processes and has no a user interface
  - Content Providers
    - store and retrieve data and make it accessible to all applications
  - Broadcast Receivers
    - responds to system-wide broadcast announcements



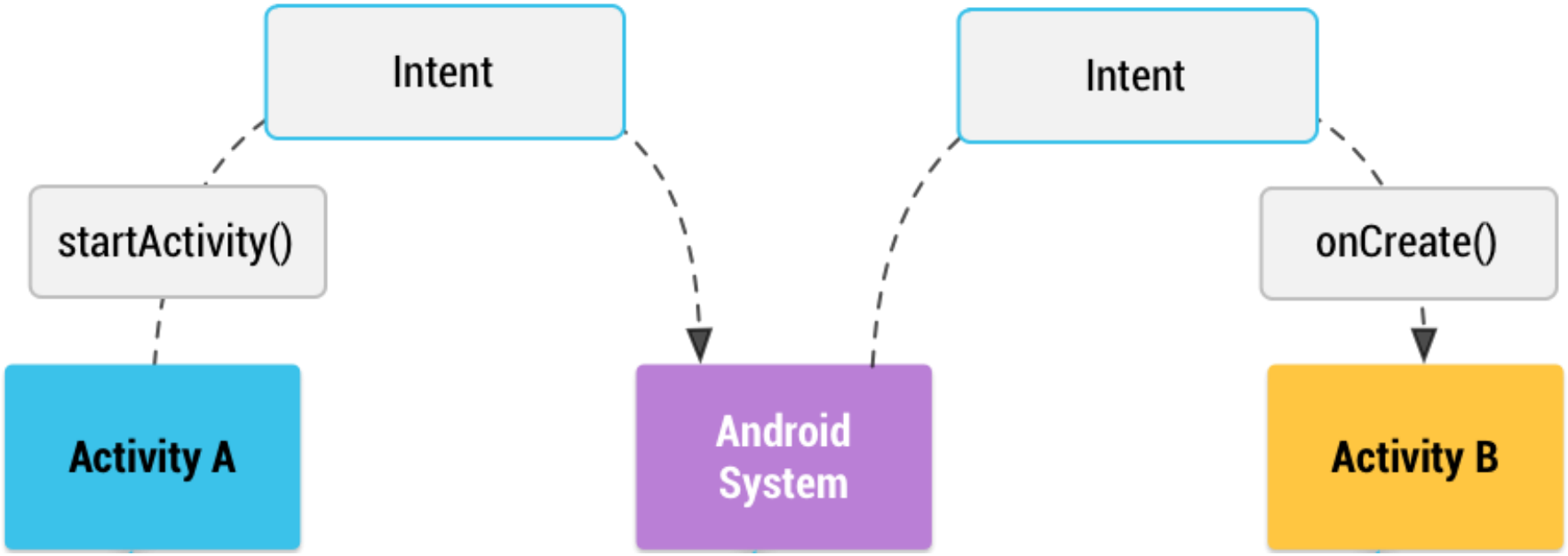
Android Architecture

- Intent
  - Three of the core components of an application — activities, services, and broadcast receivers — are activated through messages, called *intents*
  - Intent messaging is a facility for late run-time binding between components in the same or different applications

Activate android components



Intent delivery



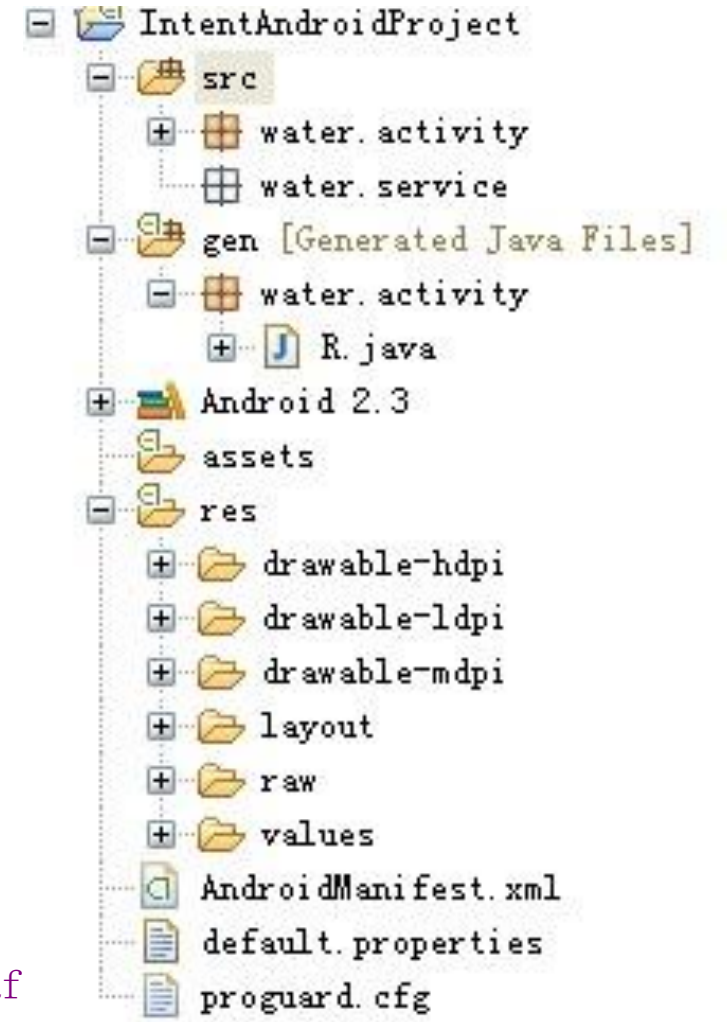
Android Architecture

- Intent (cont.)
  - intent is a passive data structure holding an abstract description of an operation to be performed
  - it contains
    - Component name
    - Action
    - Data
    - Category
    - Extras
    - Flags

```
Intent intent = new Intent();
intent.putExtra("sms_body","要发送短信的内容！");
intent.setAction(Intent.ACTION_SENTTO);
intent.setData(Uri.parse("smsto:15895899925"));
startActivity(intent);
```

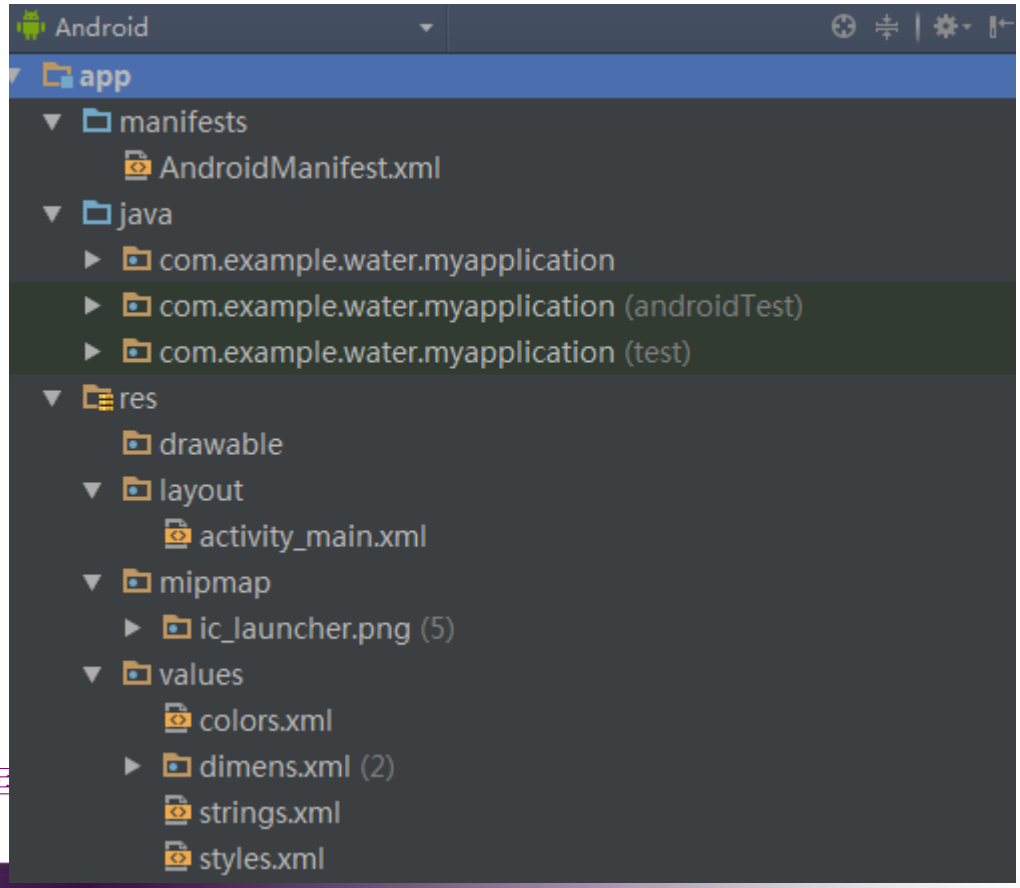
Android Architecture

- Android project structure--Eclipse
  - src package
  - gen package
  - Library
  - assets folder
  - res folder
    - drawable
    - Layout
    - …
  - project Menifest file



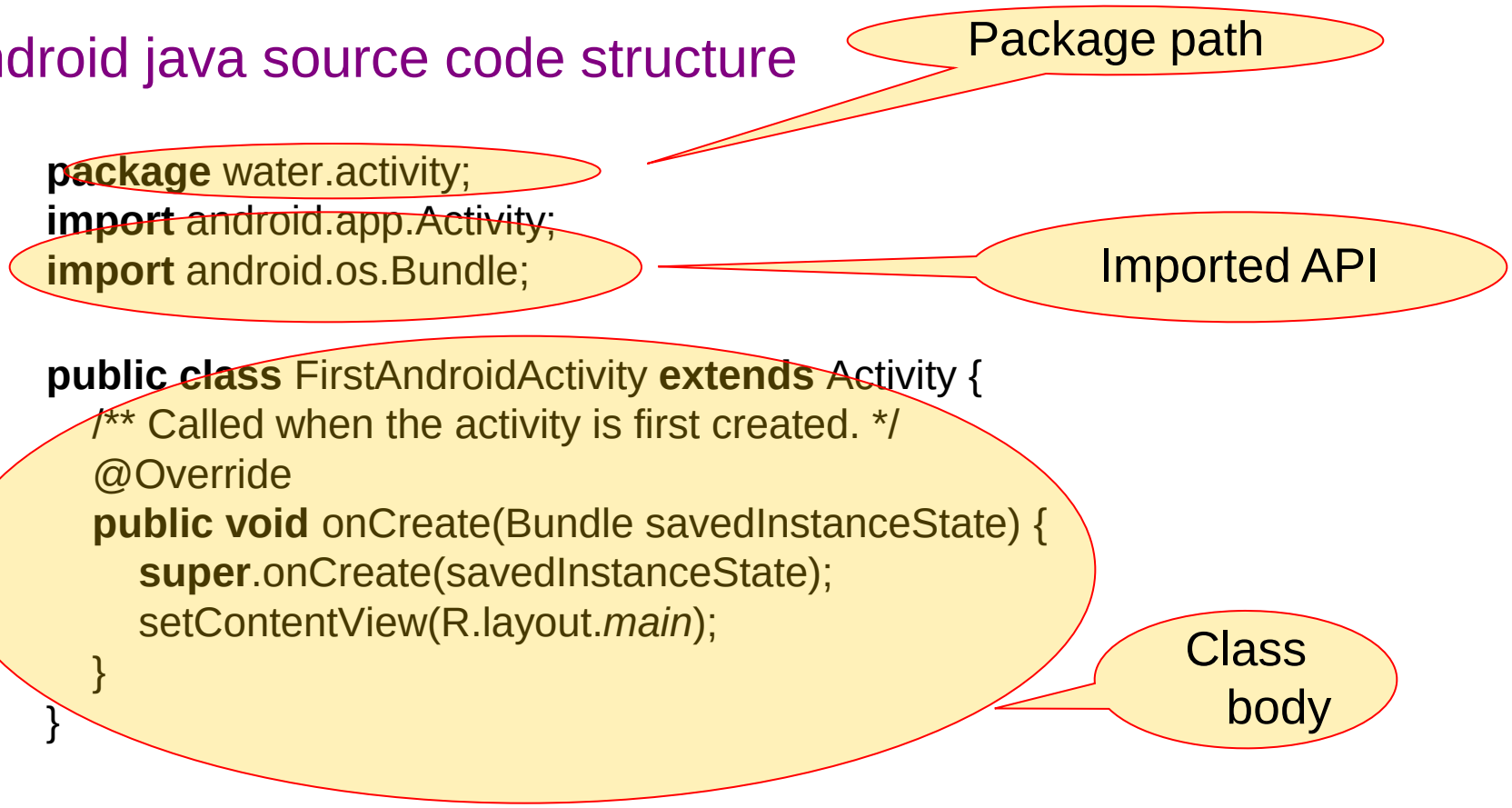
Android Architecture

- Android project structure--Android Studio
  - java package
  - res package
  - manifests package



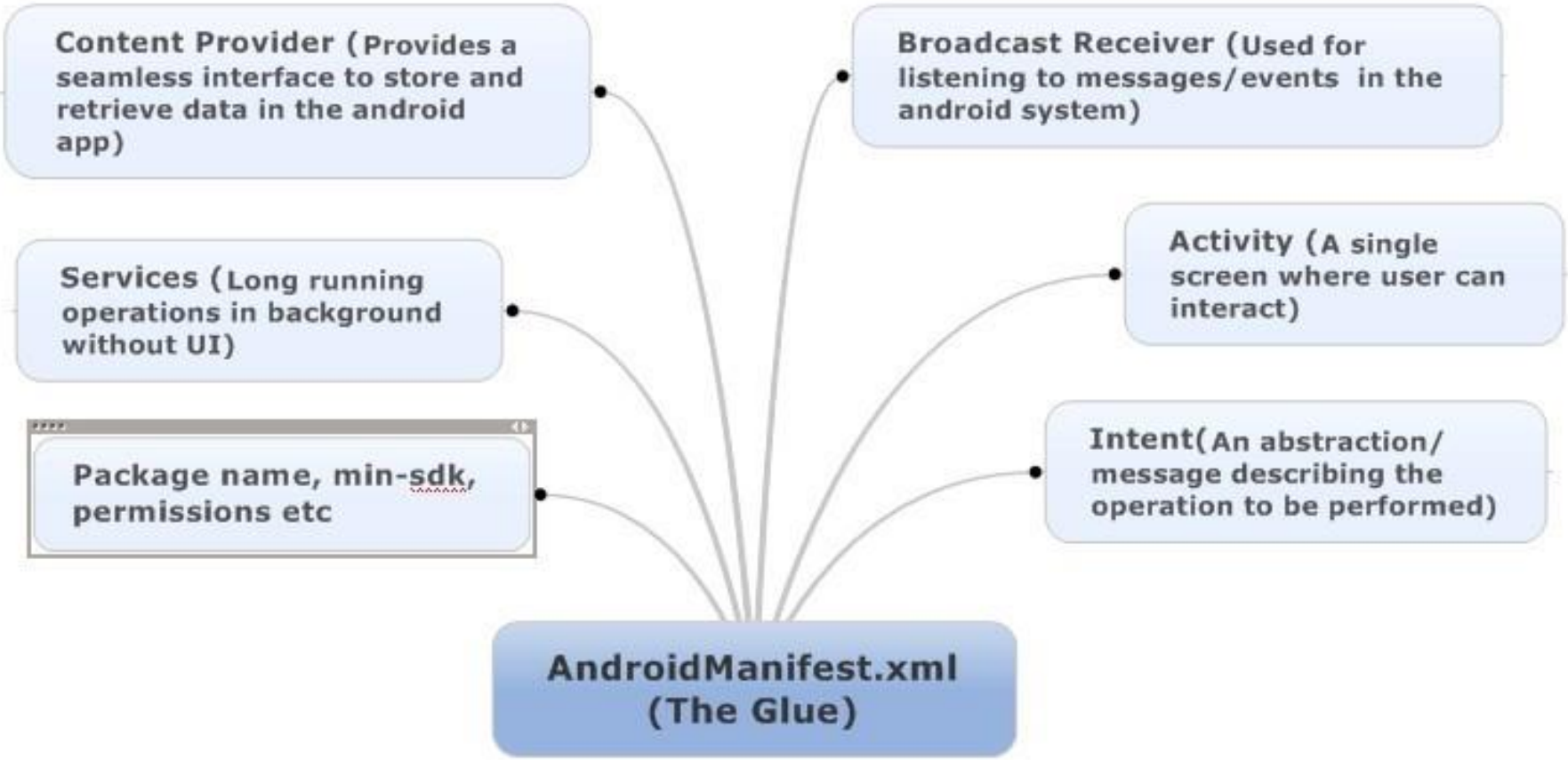
Android Architecture

- Android java source code structure



Android Architecture

- AndroidMenifest.xml file

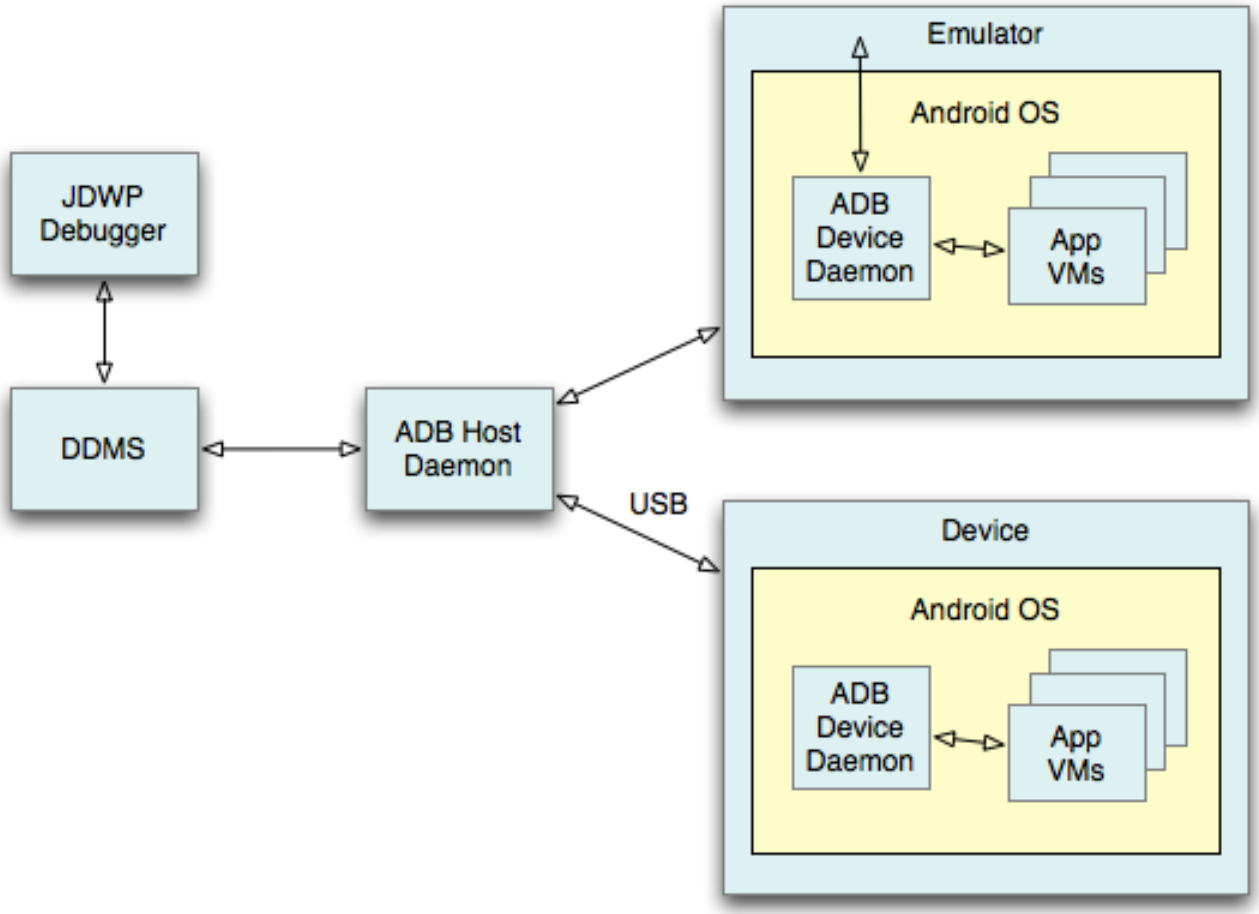


Android menifest file demo

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="water.activity" android:versionCode="1" android:versionName="1.0">
    <uses-sdk android:minSdkVersion="9" />
    <uses-permission android:name="android.permission.CALL_PHONE"></uses-permission>

    <permission android:protectionLevel="normal" android:name="water.permission.VIEW"/><!--添加自定义权限 -->

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".FirstAndroidActivity" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name="SecondAndroidActivity"
            android:permission="water.permission.VIEW">
            <intent-filter> <!--定义intent filter -->
                <action android:name="water.action.FIRSTVIEW" /><!--添加自定义action -->
                <action android:name="water.action.SECONDVIEW" /><!--添加自定义action -->
                <category android:name="water.category.FIRSTCATEGORY" /><!--添加自定义category -->
                <category android:name="android.intent.category.DEFAULT"></category><!--添加DEFAULT category -->
                <data android:mimeType="text/html" android:scheme="http"
                    android:host="www.testAndroid.com" android:port="8888" /><!--添加自定义data -->
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Android Architecture

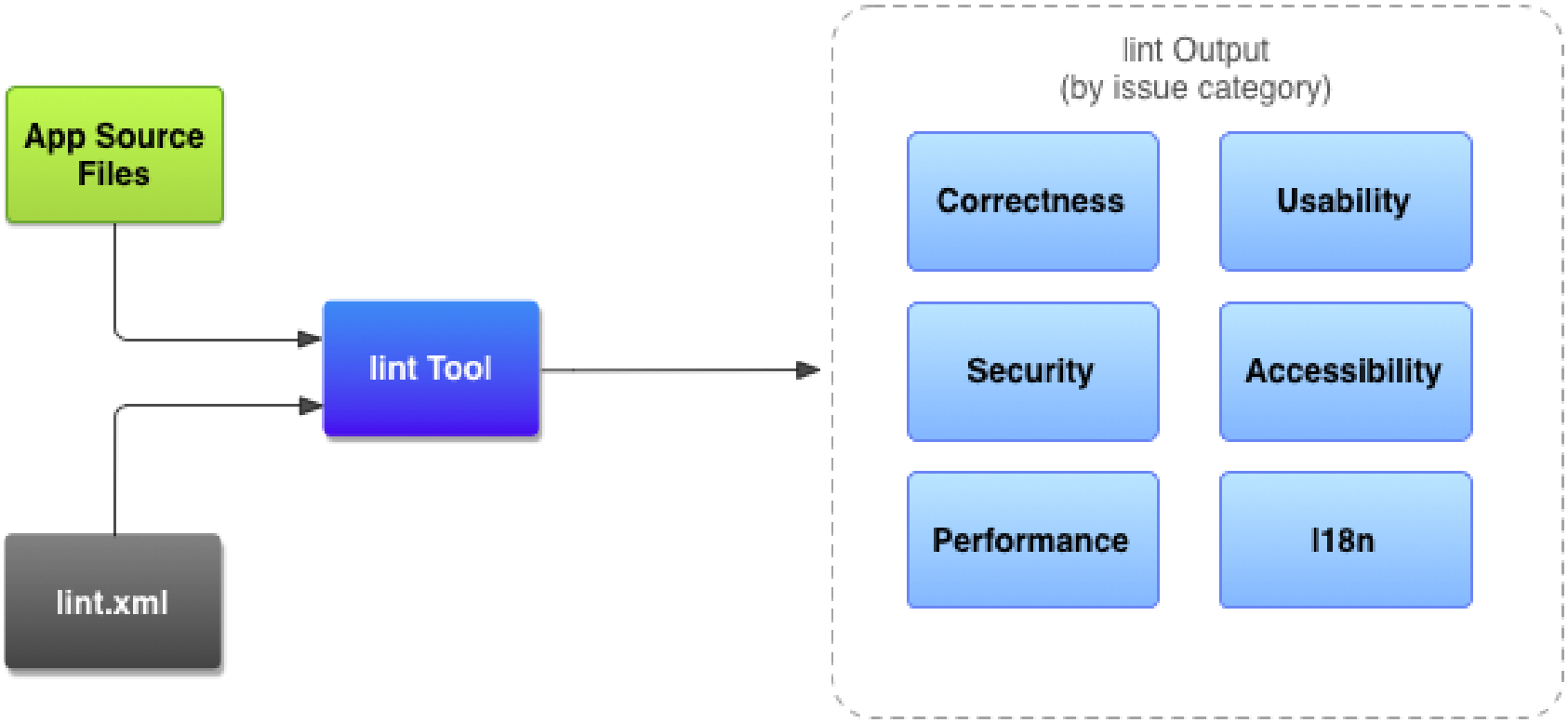
- Android debugging



Android Architecture

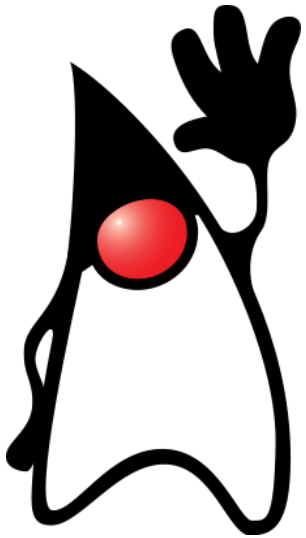
- Android debugging (cont.)
  - adb
  - logcat
  - ddms
  - lint
  - monkey/monkeyrunner
  - junit
  - hierarchy viewer
  - etc.

Android lint workflow



Java Programming Language

- Java is an OO programming language
  - Java applications are typically compiled to bytecode (class file) that can run on any Java virtual machine (JVM) regardless of computer architecture
  - The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1991 and first released in 1995
  - Oracle acquired Sun in 2010

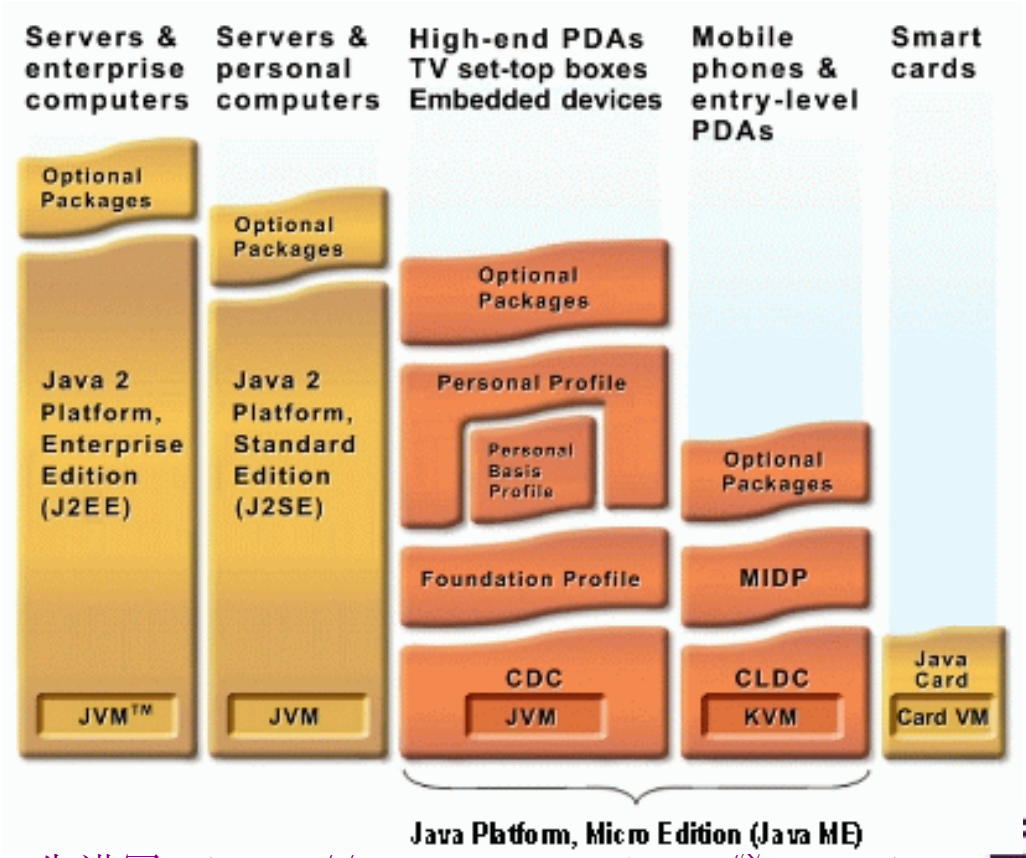


Java Programming Language

- Major release versions
  - JDK 1.0 (January 21, 1996)
  - JDK 1.1 (February 19, 1997)
  - J2SE 1.2 (December 8, 1998)
  - J2SE 1.3 (May 8, 2000)
  - J2SE 1.4 (February 6, 2002)
  - J2SE 5.0 (September 30, 2004)
  - Java SE 6 (December 11, 2006)
  - Java SE 7 (July 28, 2011)
  - Java SE 8 (March 18, 2014)

Java Programming Language

- Java editions
  - java card
  - java me
  - java se
  - java ee
  - javaFX



Java Programming Language

- Principles of OOP
  - encapsulation
  - inheritance
  - polymorphism
- Principles in java
  - keep classes/methods small
  - use comments
  - use a consistent style
  - use built-in logging

Java Programming Language

- Hello world in java

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, World");
   }
}
```

```
% javac HelloWorld.java
```

```
% java HelloWorld
```

```
Hello, World
```

```
/*************************************************************************
 *  Compilation:  javac HelloWorld.java
 *  Execution:    java HelloWorld
 *
 *  Prints "Hello, World". By tradition, this is everyone's first program.
 *
 *  % java HelloWorld
 *  Hello, World
 *
 *  These 17 lines of text are comments. They are not part of the program;
 *  they serve to remind us about its properties. The first two lines tell
 *  us what to type to compile and test the program. The next line describes
 *  the purpose of the program. The next few lines give a sample execution
 *  of the program and the resulting output. We will always include such
 *  lines in our programs and encourage you to do the same.
 *
 *************************************************************************/

public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }

}
```

Java Programming Language

- Struture of java class

```
package packageName;
import ClassNameToImport;
accessSpecifier class ClassName {
    accessSpecifier dataType variableName [= initialValue];
    accessSpecifier ClassName([argumentList]) {
        constructorStatement(s)
    }
    accessSpecifier returnType methodName([argumentList]) {
        methodStatement(s)
    }
     // This is a comment
     /* This is a comment too */
     /* This is a
          multiline
          comment */
}
```

Java Programming Language

- Primitive data types

Type	Size	Default value	Range of values
boolean	n/a	false	true or false
byte	8 bits	0	-128 to 127
char	16 bits	(unsigned)	\u0000' \u0000' to \uffff' or 0 to 65535
short	16 bits	0	-32768 to 32767
int	32 bits	0	-2147483648 to 2147483647
long	64 bits	0	-9223372036854775808 to 9223372036854775807
float	32 bits	0.0	1.17549435e-38 to 3.4028235e+38
double	64 bits	0.0	4.9e-324 to 1.7976931348623157e+308

Java Programming Language

- Arithmetic operators

Operator	Usage	Description
+	a + b	Adds a and b
+	+a	Promotes a to int if it's a byte, short, or char
-	a - b	Subtracts b from a
-	-a	Arithmetically negates a
*	a * b	Multiplies a and b
/	a / b	Divides a by b
%	a % b	Returns the remainder of dividing a by b (the modulus operator)
++	a++	Increments a by 1; computes the value of a before incrementing
++	++a	Increments a by 1; computes the value of a after incrementing
--	a--	Decrements a by 1; computes the value of a before decrementing
--	--a	Decrements a by 1; computes the value of a after decrementing
+=	a += b	Shorthand for a = a + b

Java Programming Language

- Relational and conditional operators

Operator	Usage	Returns true if ...
>	a > b	a is greater than b
>=	a >= b	a is greater than or equal to b
<	a < b	a is less than b
<=	a <= b	a is less than or equal to b
==	a == b	a is equal to b
!=	a != b	a is not equal to b
&&	a && b	a and b are both true, conditionally evaluates b (if a is false, b is not evaluated)
||	a || b	a or b is true, conditionally evaluates b (if a is true, b is not evaluated)
!	!a	a is false
&	a & b	a and b are both true, always evaluates b
|	a | b	a or b is true, always evaluates b
^	a ^ b	a and b are different

Java Programming Language

- Control flow statements
  - if then / if then else
  - switch case

```
if (futureMonths.isEmpty()) {
    System.out.println("Invalid month number");
} else {
    for (String monthName : futureMonths) {
        System.out.println(monthName);
    }
}
```

```
switch (month) {
    case 1:  futureMonths.add("January");
    case 2:  futureMonths.add("February");
    case 3:  futureMonths.add("March");
    case 4:  futureMonths.add("April");
    case 5:  futureMonths.add("May");
    case 6:  futureMonths.add("June");
    case 7:  futureMonths.add("July");
    case 8:  futureMonths.add("August");
    case 9:  futureMonths.add("September");
    case 10: futureMonths.add("October");
    case 11: futureMonths.add("November");
    case 12: futureMonths.add("December");
             break;
    default: break;
}
```

Java Programming Language

- Control flow statements (cont.)
  - while / while do
  - for / for each

```
while (count < 11) {
    System.out.println("Count is: " + count);
    count++;
}
```

```
for(int i=1; i<11; i++){
    System.out.println("Count is: " + i);
}
```

```
for (int item : numbers) {
    System.out.println("Count is: " + item);
}
```

```
do {
    System.out.println("Count is: " + count);
    count++;
} while (count < 11);
```

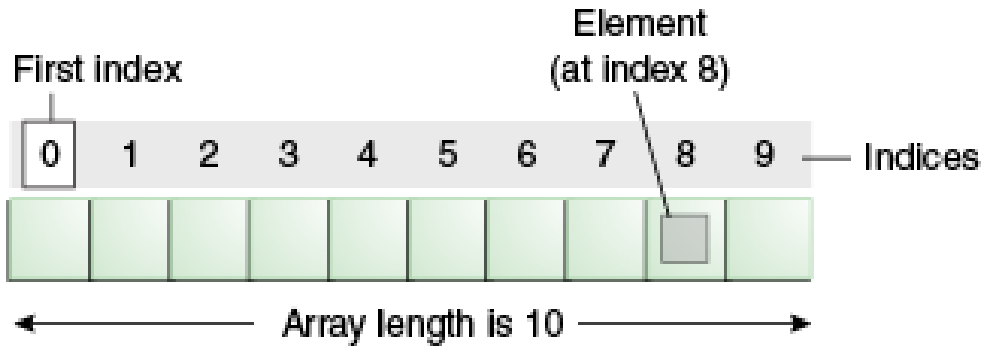
Java Programming Language

- Boxing and unboxing

Primitive	JDK counterpart
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double

Java Programming Language

- Java collections
  - Arrays
    - an array is a container object that holds a fixed number of values of a single type
    - the length of an array is established when the array is created. After creation, its length is fixed



Array

```
<terminated> MainEntra
a
b
c
d
```

```
package water.servlet;

import java.util.Arrays;

public class MainEntrance {

    public static void main(String[] args) {

        String[] arr = new String[] { "a", "d", "c", "b" };
        Arrays.sort(arr);

        for (String a : arr) {
            System.out.println(a);
        }

    }

}
```

Java Programming Language

- Java collections (cont.)
  - List
    - a List is a collection construct that is by definition an ordered collection, also known as a sequence
    - a Java List collection can only hold objects
    - List is an interface, so you can't instantiate it directly. You'll work with its most commonly used implementation, ArrayList
    - Unlike sets, lists typically allow duplicate elements

List

```
class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

}
```

```
package water.servlet;

import java.util.ArrayList;
import java.util.List;

public class MainEntrance {

    public static void main(String[] args) {

        List<Person> pers = new ArrayList<Person>();

        Person li = new Person();
        li.setName("Li Hua");
        li.setAge(19);

        pers.add(li);

        Person xi = new Person();
        xi.setName("Xi Hui");
        xi.setAge(20);

        pers.add(xi);

        for (int i = 0; i < pers.size(); i++) {
            System.out.println(pers.get(i).getName());
        }

    }

}
```

Java Programming Language

- Java collections (cont.)
  - Set
    - a Set is a collections construct that by definition contains unique elements — that is, no duplicates
    - sets contain at most one null element
    - because Set is an interface, you can't instantiate it directly

```
public interface Set<E>
extends Collection<E>
```

Set

```
Value :a
Value :b
Value :c
Value :d
```

```java
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

public class SetExample {

    public static void main(String[] args) {

        // Set example with implement TreeSet
        Set<String> s=new TreeSet<String>();

        s.add("b");
        s.add("a");
        s.add("d");
        s.add("c");

        Iterator it=s.iterator();

        while(it.hasNext())
        {
          String value=(String)it.next();

          System.out.println("Value :"+value);
        }
    }
}
```

Java Programming Language

- Java collections (cont.)
  - Map
    - a Map is a handy collection construct because it lets you associate one object (the key) with another (the value)
    - the key to the Map must be unique, and it's used to retrieve the value at a later time
    - a Java Map collection can only hold objects
    - and you can't instantiate it directly

Map

```
Key :1 Value :One
Key :2 Value :Two
Key :3 Value :Three
Key :4 Value :Four
```

```java
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;

public class MapExample {

    public static void main(String[] args) {

        Map<Object,String> mp=new HashMap<Object, String>();


        // adding or set elements in Map by put method key and value pair
        mp.put(new Integer(2), "Two");
        mp.put(new Integer(1), "One");
        mp.put(new Integer(3), "Three");
        mp.put(new Integer(4), "Four");

        //Get Map in Set interface to get key and value
        Set s=mp.entrySet();

        //Move next key and value of Map by iterator
        Iterator it=s.iterator();

        while(it.hasNext())
        {
            // key=value separator this by Map.Entry to get key and value
            Map.Entry m =(Map.Entry)it.next();

            // getKey is used to get key of Map
            int key=(Integer)m.getKey();

            // getValue is used to get value of key in Map
            String value=(String)m.getValue();

            System.out.println("Key :"+key+"  Value :"+value);
        }
    }
}
```

Kotlin Programming Language

- Is based on JVM, and published by JetBrains: http://kotlinlang.org/
- Was announced as an official language on Android platform (2017.05.17) by Google
- Can be compiled into several types of code
  - Java class bytecode
  - Javascript
  - Native code
  - Dalvik/oat bytecode
  - etc.

Kotlin Programming Language

- A statically typed programming language
  - Null safety
  - Object-Oriented programming
  - Functional programming
  - Java introp
  - Javascript interaction
  - Etc.

Kotlin Programming Language

- Kotlin “hello world”

```
fun main(args: Array<String>) {

    val kotlin: Kotlin = Kotlin()

    kotlin.sayHi("hello world")

}
```

```
class Kotlin {

    fun sayHi(hi: String) {
        print(hi)
    }

}
```

Kotlin Programming Language

- Basic types
  - Numbers
    - —Double, Float, Long, Int, ……
  - Characters
    - —Char
  - Booleans
    - —true，false
  - Arrays
    - —Array<T>, ByteArray, ShortArray, ……
  - Strings
    - —String

Type	Bit width
Double	64
Float	32
Long	64
Int	32
Short	16
Byte	8

Array & Char

```
fun main(args: Array<String>) {

    val a: Char = 'a'
    check(a)

}

fun check(c: Char) {
    print(c.toInt() == 97)
}
```

```
true
```

```
val a = Array(5, { i -> "kotlin:" + i.toString() })

for (b in a) {
    println(b)
}
```

```
kotlin:0
kotlin:1
kotlin:2
kotlin:3
kotlin:4
```

String & String template

```
val hi="hello kotlin! "

for(c in hi){
    print(c)
}

println(hi)

print("$hi.length is ${hi.length}")
```

```
hello kotlin! hello kotlin!
hello kotlin! .length is 14
```

Kotlin Programming Language

- Control Flow
  - Options
    - If (condition)… else expression
    - when (condition) expression
  - Loops
    - for (in-expression)
    - while (condition) / do… while (condition)
  - break, continue, return
  - try…catch…finally

when

```
fun main(args: Array<String>) {

    val ch = 'A'

    hi(ch.toInt())

}

fun hi(value: Int) {

    when (value) {
        96 -> print('a')
        97 -> print('b')
        else -> print("others")
    }

}
```

```
others
```

Kotlin Programming Language

- Class & Object
  - Classes can contain
    - Constructor & initializer blocks
    - Functions
    - Properties
    - Nested and inner classes
    - Object declarations
  - Class modifier
    - abstract, final, enum, open, annotation

Kotlin Programming Language

- Class & Object
  - Object expressions
    - To create an object of an anonymous class that inherits from some type (or types)
  - Object declarations
    - To declare singletons
  - Companion Objects
    - Declared inside a class and marked with the companion keyword

Class and instance

```
fun main(args: Array<String>) {

    var p: Person = Person("tom", 30)
    print(p.login())

}
```

```
class Person(name: String, age: Int) {

    var nameProperty: String = name
        get() = field
        set(value) {
            if (value.isNullOrEmpty()) {
                field = ""
            } else {
                field = value
            }
        }

    private var passwordProperty: String = ""
        set(value) {
            field = value
        }

    var ageProperty: Int = age

    public fun login(): Boolean {

        if (!nameProperty.isNullOrEmpty()) {
            return verifiedAccount()
        } else {
            return false
        }

    }

    private fun verifiedAccount(): Boolean {

        return true
    }

}
```

Object expresion

```
fun main(args: Array<String>) {

    hi(object:Person("Catt",19){

        override fun login():Boolean{
            return false
        }

    })

}

fun hi(p:Person){

    print(p.login())

}
```


```
open class Person(name: String, age: Int) {

    var nameProperty: String = name
        get() = field
        set(value) {
            if (value.isNullOrEmpty()) {
                field = ""
            } else {
                field = value
            }
        }

    private var passwordProperty: String = ""
        set(value) {
            field = value
        }

    var ageProperty: Int = age

    open public fun login(): Boolean {

        if (!nameProperty.isNullOrEmpty()) {
            return verifiedAccount()
        } else {
            return false
        }

    }

    private fun verifiedAccount(): Boolean {

        return true
    }

}
```

Kotlin Programming Language

- Functions & Lambdas
  - Function scope
    - Local function
    - Member function
  - Generic function
  - Inline function
  - Extension function
  - Higher order function & Lambda
  - Tail recursive function

Function & lambda

```
fun main(args: Array<String>) {

    print(facSumDouble(2, 3, ::fac))

}

fun fac(value: Int, factor: Int = 2): Int {

    return value * factor
}

fun facSumDouble(x: Int, y: Int, f: (Int, Int) -> Int): Int {

    val factor = 4

    val lambdaExpression = { m: Int, n: Int -> (m + n) * 2 }

    fun sumDouble(a: Int, b: Int): Int {

        return lambdaExpression(f(a, factor), f(b, factor))

    }

    return sumDouble(x, y)
}
```

Kotlin Programming Language

- Interaction with other programming languages
  - Calling java from kotlin
  - Calling kotlin from java
  - Calling javascript from kotlin
  - Calling kotlin from javascript

Kotlin interacts with java

```kotlin
fun main(args: Array<String>) {

    val hi = "hi!"
    javaSays(hi);

}

fun javaSays(hi: String) {

    var s = Student(hi)
    s.sayHi()
    s.sayFromKotlin(hi)

}
```

```
java says:hi!
kotlin says:hi!
```

```java
public class Student {

    private String hi;

    public Student(String str) {
        hi = str;
    }

    public void sayHi() {
        System.out.println("java says:" + hi);
    }

    public void sayFromKotlin(String str) {
        new Kotlin().sayHi(str);
    }

}
```

```kotlin
class Kotlin {

    fun sayHi(hi: String) {
        print("kotlin says:" + hi)
    }

}
```

Conclusions

- Architectural Design
- Software Design Principles
- Package Diagram and UML
- Android Architecture
- Java Programming Language
- Kotlin Programming Language
- Conclusions