Apache Felix on Androids

Marcel Offermans
Christian van Spaandonk

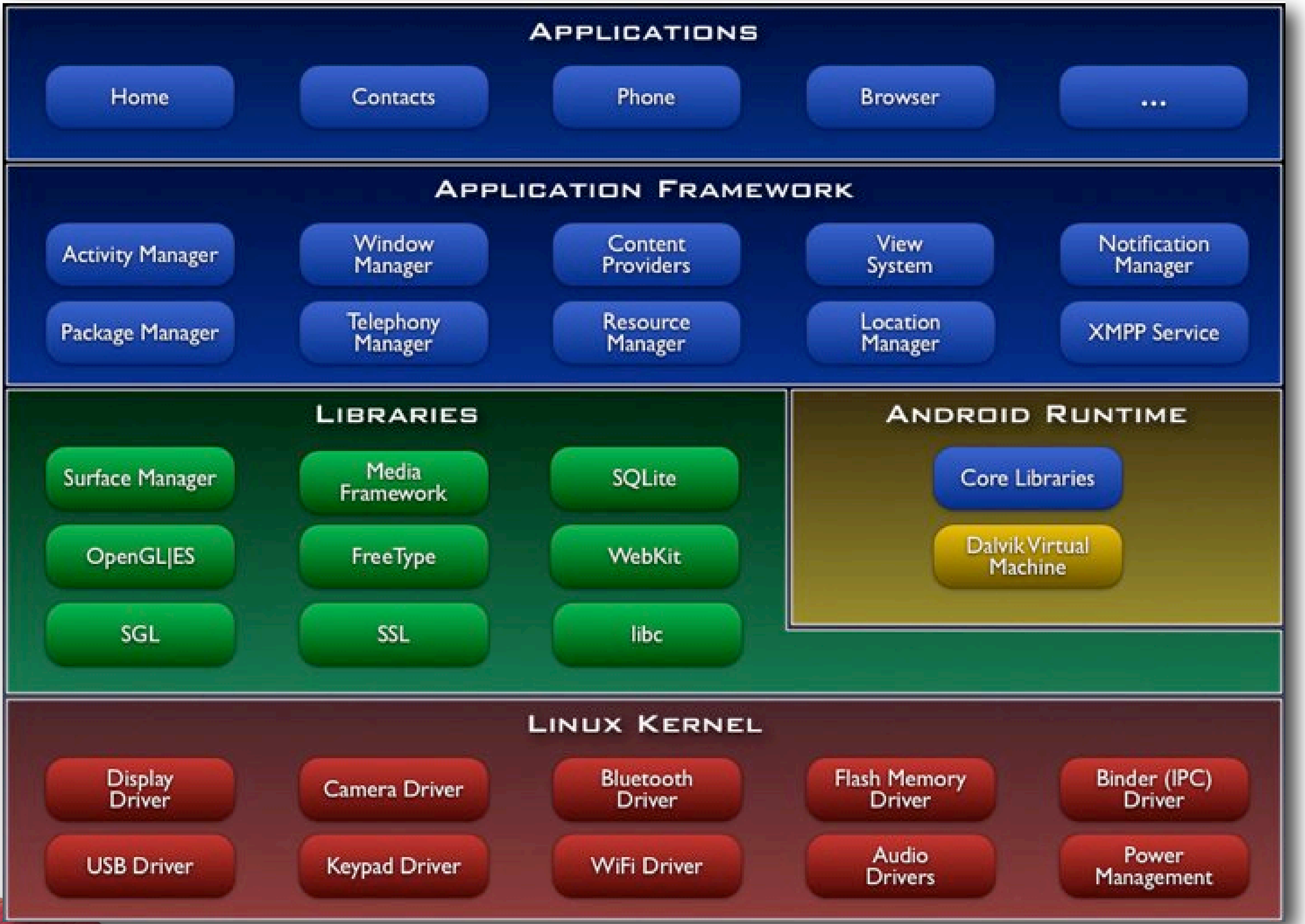
Agenda

- Introduction to Google Android
- Demo: Hello world
- The OSGi framework
- Combining Android and OSGi
- Getting Felix to run
- Application design and deployment
- Demo: modular desktop application
- Demo: context awareness

Android

- Device Architecture
- Dalvik Virtual Machine
- From Source to Deployment
- Anatomy of an Application
- Application life cycle

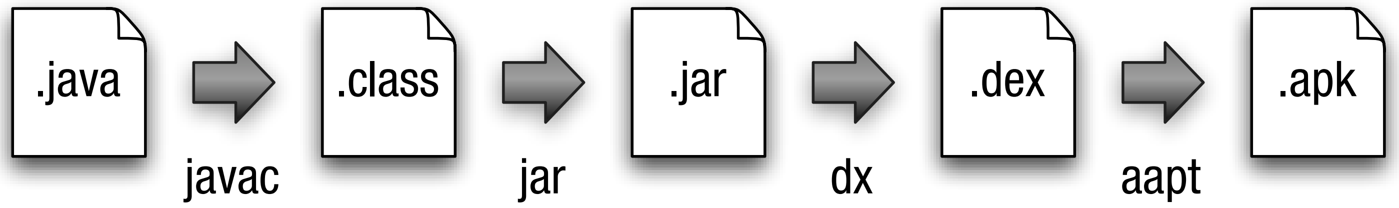
Architecture



Dalvik Virtual Machine

- interpreter only, register based
- optimized to run multiple instances
- executes files in .dex format
- runs on posix-compliant OS
- looks, feels and smells like Java ;)

From Source to Deployment

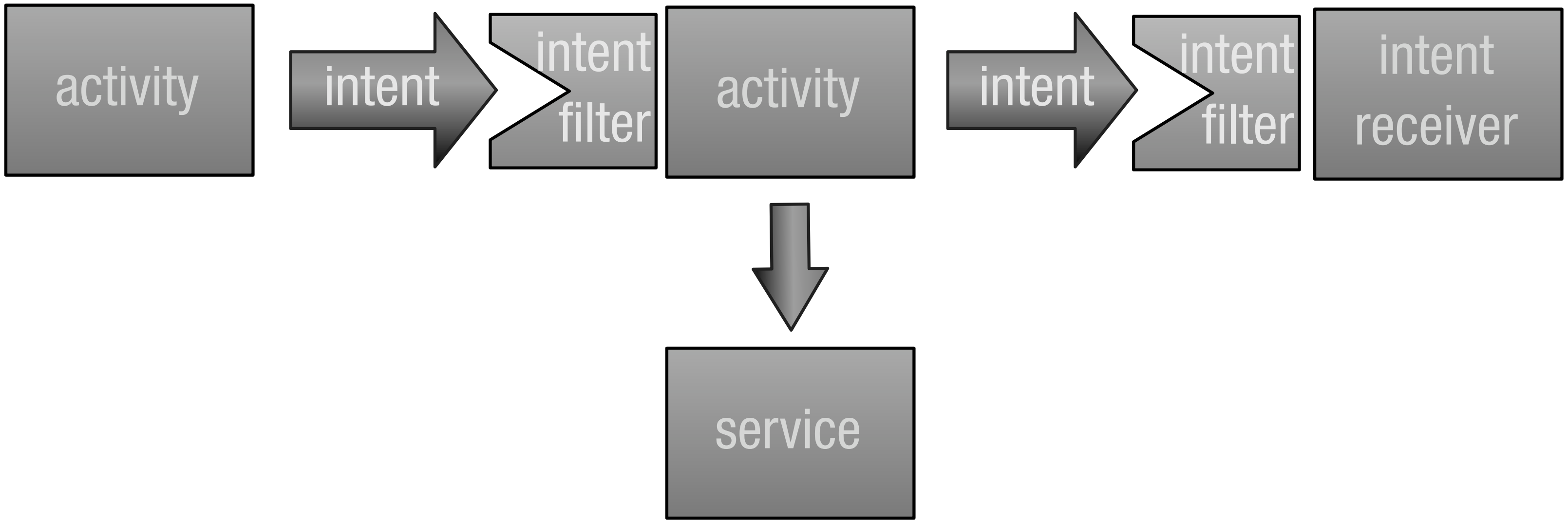


- Eclipse Plugin: Android Dev Tools
  - compiles and packages automatically
  - launch and debug in emulator or phone
- Command line: activityCreator.py
  - generates project structure
  - Ant build.xml file
  - IntelliJ project files

Anatomy

- **activity**, a single screen
- **intent**, describes what you want done
- **intent filter**, describes intents that can be handled
- **intent receiver**, UI that reacts to intent
- **service**, background process with API
- **content provider**, for shared data access

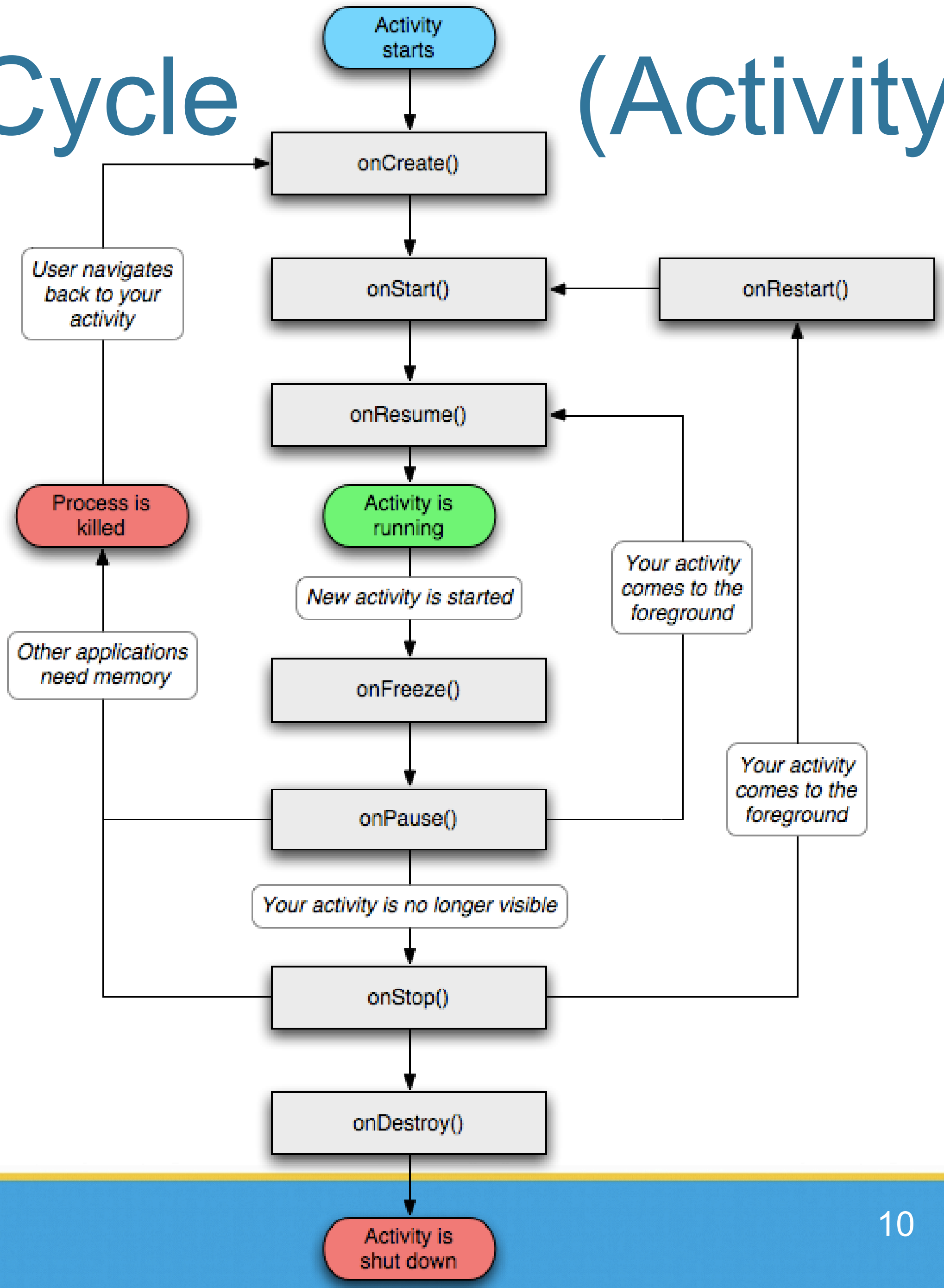
Anatomy Example



Life Cycle

- is not controlled by the application
- android maintains “importance hierarchy” based on:
  - foreground process
  - visible proces
  - service proces
  - background proces
  - empty proces

Life Cycle (Activity)

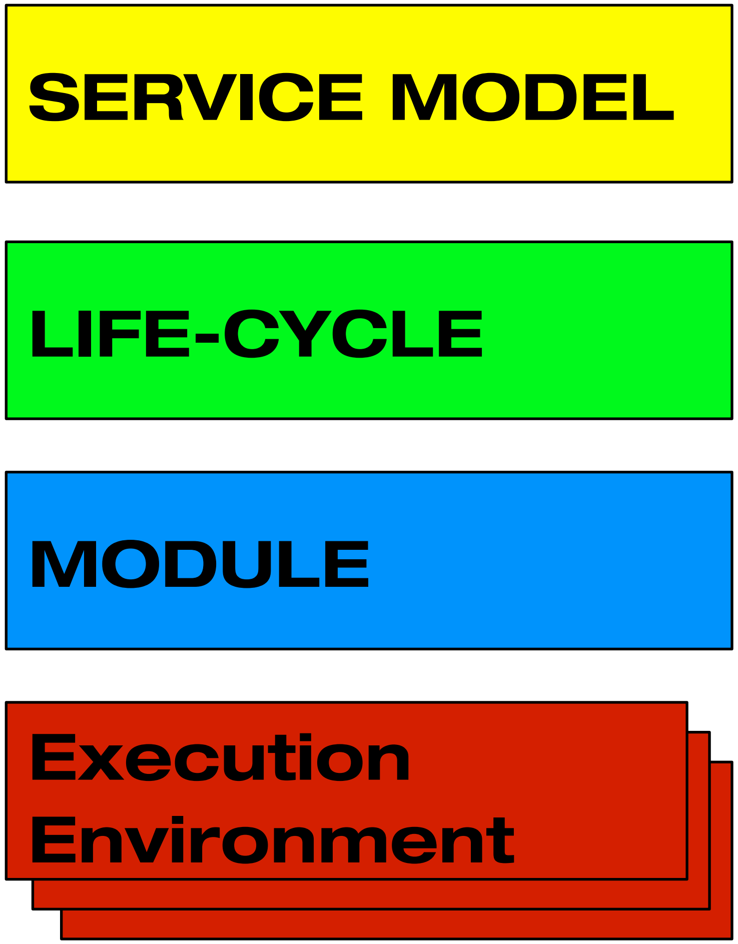


Activity starts
onCreate()
User navigates back to your activity
onStart()
onRestart()
onResume()
Process is killed
Activity is running
Your activity comes to the foreground
New activity is started
Other applications need memory
onFreeze()
onPause()
Your activity comes to the foreground
Your activity is no longer visible
onStop()
onDestroy()
Activity is shut down

Demo: hello world

- Create an application with an Activity in Eclipse
- Set the “hello world” text
- Create a breakpoint
- Deploy and debug the application

OSGi Framework Layering



L3 - Provides a publish/find/bind service model to decouple bundles

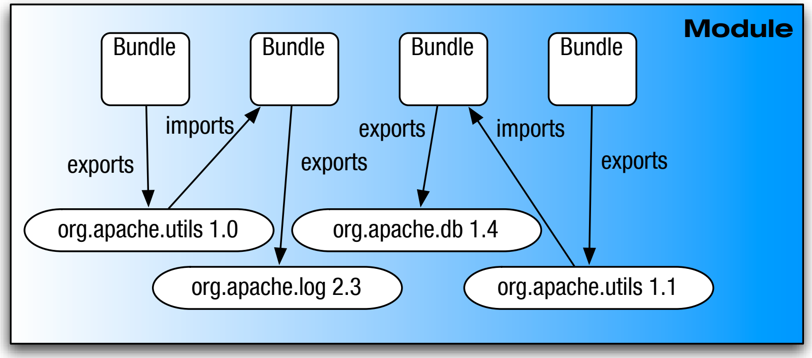
L2 - Manages the life cycle of a bundle in a framework without requiring the vm to be restarted

L1 - Creates the concept of a module (aka. bundle) that both isolate and share classes from each other in a controlled way

L0 - well defined profiles that define the environment in which bundles can work, ie:
* CDC/Foundation
* JavaSE-6
* Android-1.0

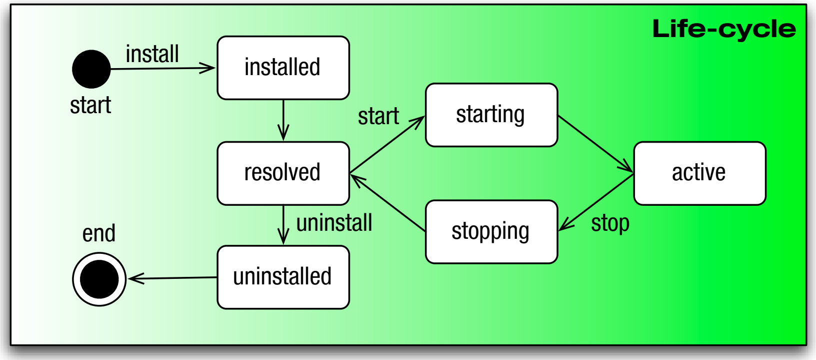
Module Layer

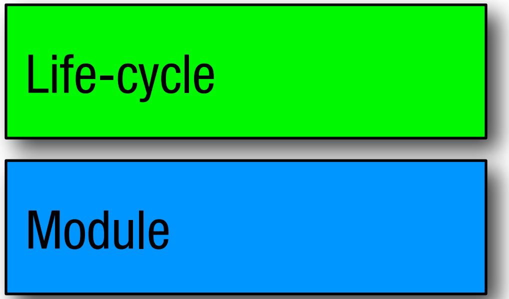
- Unit of deployment is the bundle
- Separate class loader per bundle
- Class sharing at the package level
- Packages are versioned, multiple versions concurrently supported
- Framework handles the consistency



Life-cycle Layer

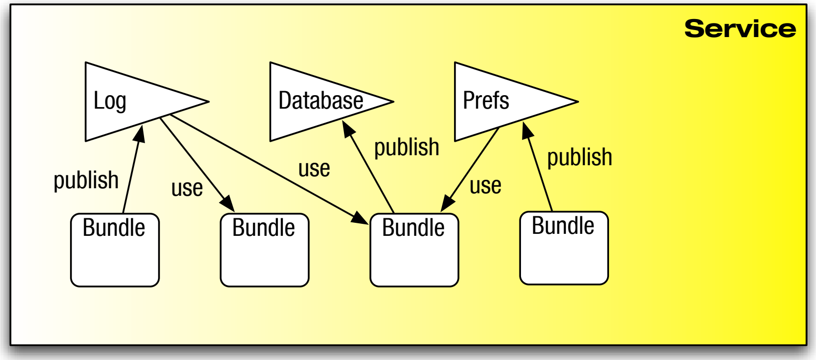
- Managed life cycle for each bundle
- Bundles can be:
  - added,
  - updated and
  - removed

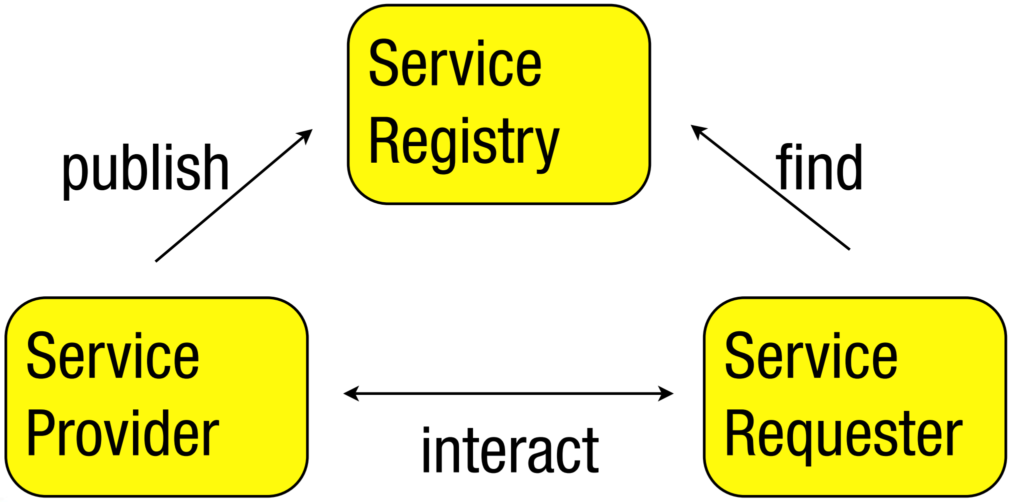


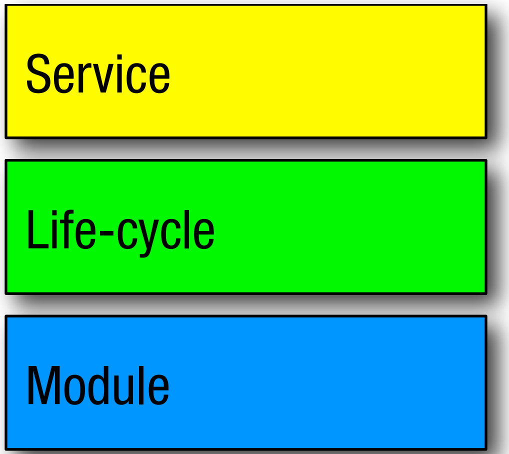


Service Layer

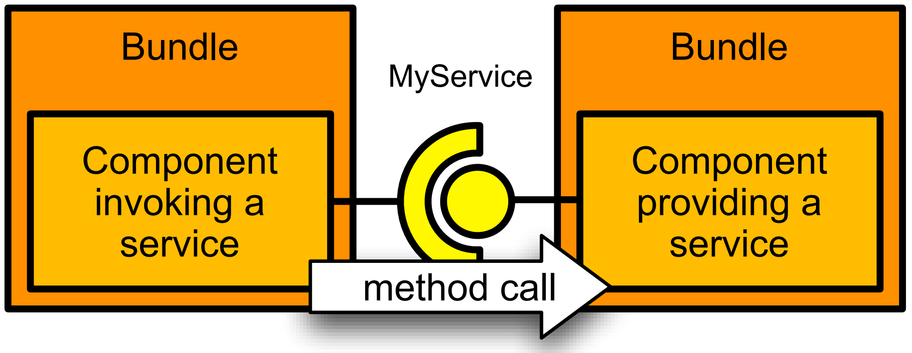
- Preferred way for bundles to interact
- Service registry can even be distributed in OSGi R4.2

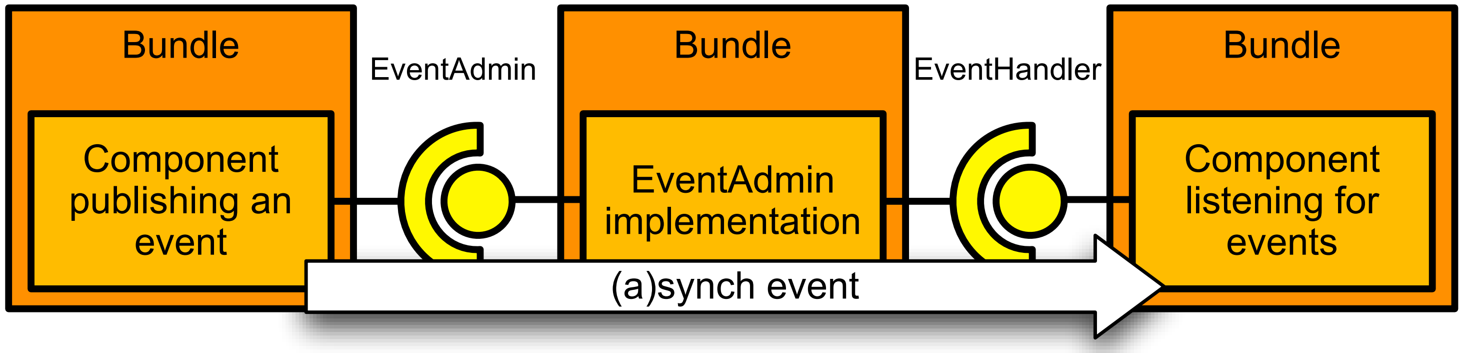






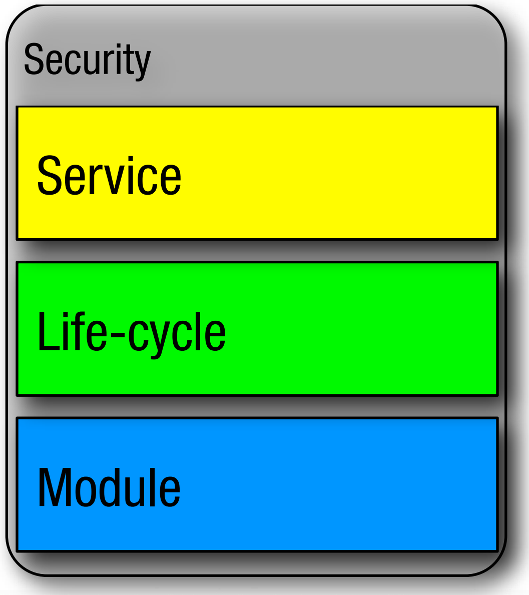
Side step: interaction styles



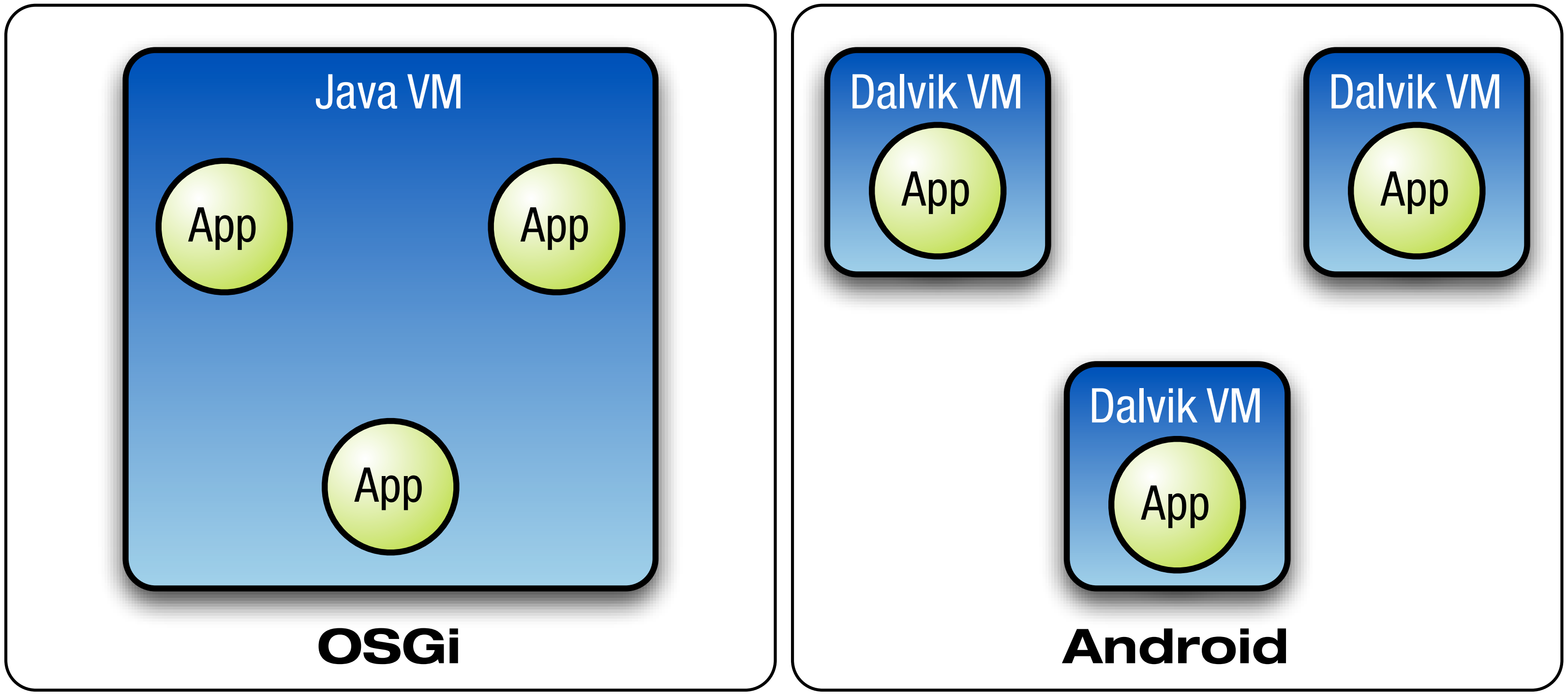

Bundle
Component publishing an event
EventAdmin
Bundle
EventAdmin implementation
EventHandler
Bundle
Component listening for events
(a)synch event

Security Concepts Overview

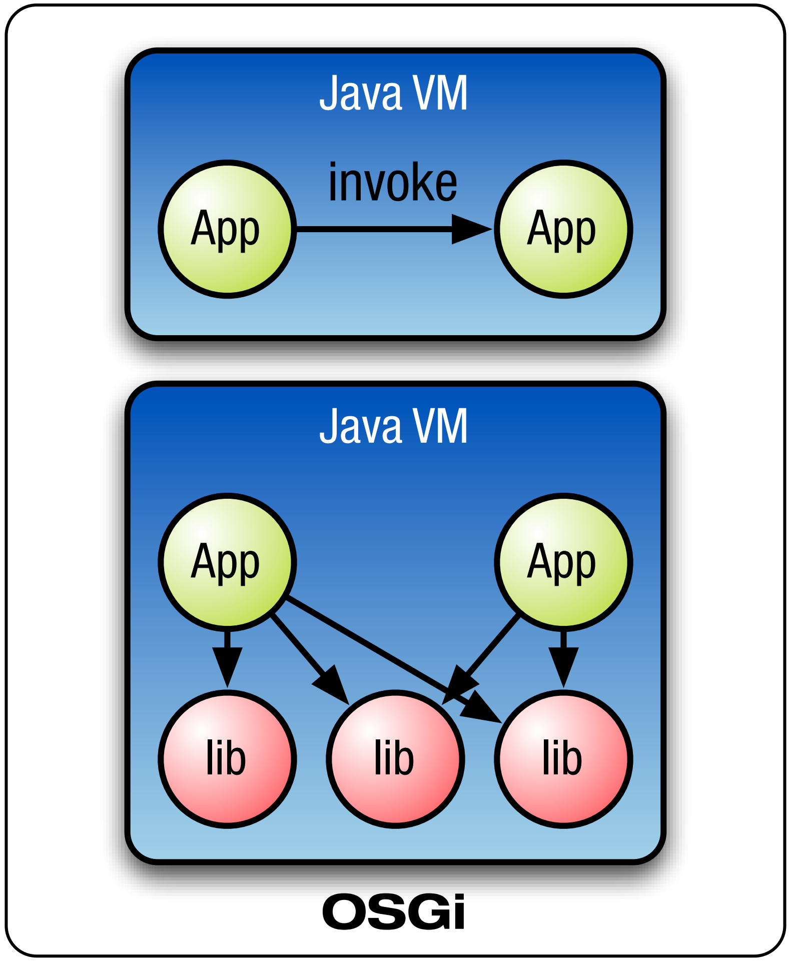
- Codebased security of the Java Security Model
  - uses Protection Domains
  - stack walk based Permission Check
  - signed bundles
- PA and CPA provide management infrastructure
- IF all conditions match THEN apply permissions

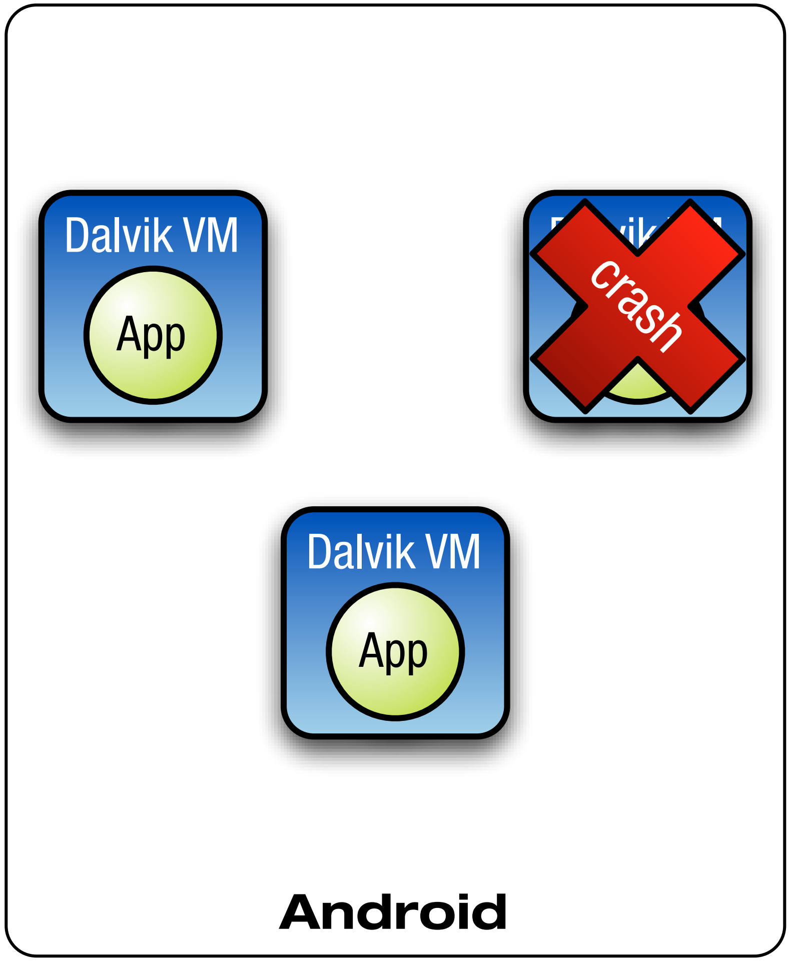


OSGi and Android



Benefits of each model





Why combine them?

- Using and enforcing a modular design
- Build applications faster through re-use of existing OSGi components
- Applications tailored for the user, only give him what he wants/needs
- Dynamic loading and unloading, you do not always need all application components

Android Services

- declared in AndroidManifest.xml
- can be started and stopped: Context.startService(), .stopService()
- you can bind to it to use it
- services run in remote processes, IDL compiler generates stubs
  - primitives, collections, Parcelable's by value
  - other AIDL interfaces by reference

Getting Felix to run

- Initial efforts by Karl Pauls and me
- Felix is portable, so we just dex'ed it
- since 1.0.3 we are Android aware
  - found a way to dynamically load classes
  - relies on an undocumented class
- Google, we need an API for:
  - dynamic class loading
  - dynamic security policies

ApacheCon

Deploying on a dev phone

- Some manual preparation is necessary
- Phone is configured so apps cannot dynamically load classes
- Fixed by:
  - becoming root
  - chmod 777 /data/dalvik-cache

Side step: other frameworks

- EclipseCon 2008, Santa Clara:
  - Neil Bartlett and BJ Hargrave ported both Equinox and Concierge to Android
- ProSyst:
  - ported their embedded server
- Knopflerfish:
  - no plans as far as we know

Application design

- Basis of the application is an Activity, exposed through ActivityService

```
public interface ActivityService {
    public Activity getActivity();
    public Object lookupSystemService(String name);
}
```

- Felix looks for a ViewFactory to create its main view

```
public interface ViewFactory {
    public View create(Context context);
}
```

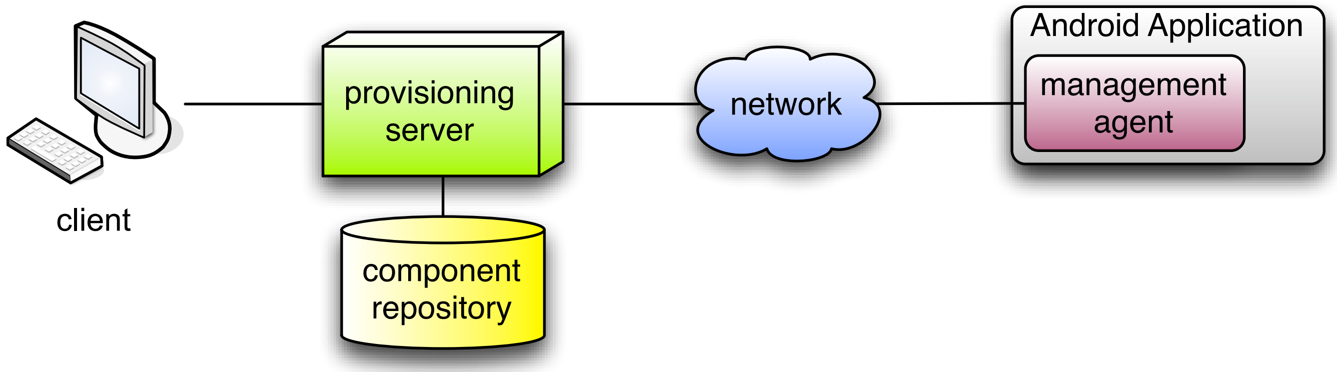
- Security is declared here

Management Agent

- Responsible for installation and update
- Communicates with a provisioning server
- Can be used to:
  - centrally manage and deploy components
  - allow a “store” like or context aware interface to select components client side
- We embed the management agent as part of the application

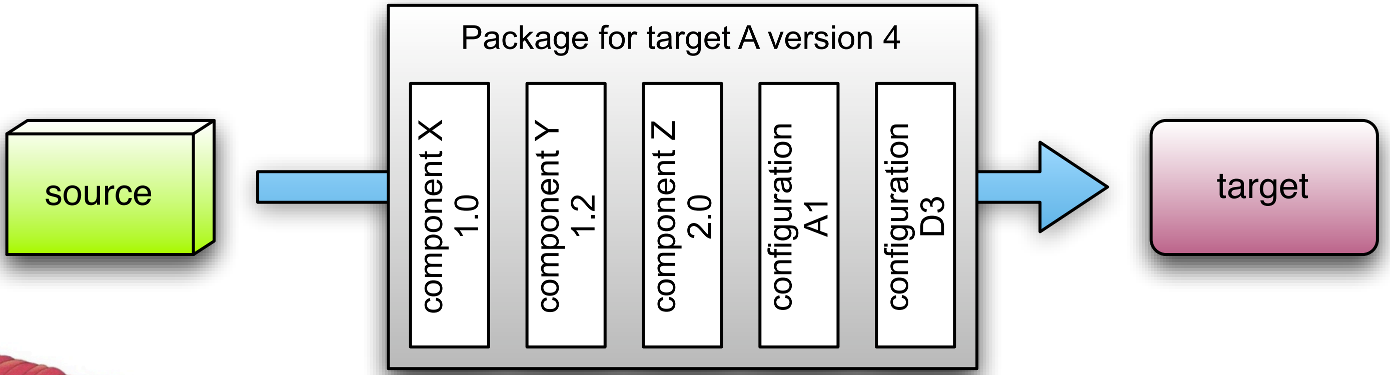
Topology

- Client: on the laptop
- Server: far, far away on the net
- Phone: using 3G/GPRS



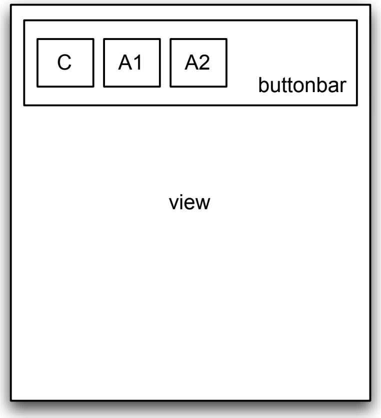
Side step: Deployment Admin

- streams deployment packages
- packages get installed transactionally
- supports fix packages with deltas
- can install arbitrary file types
- types handled by resource processors



Desktop Application

- Desktop component (ViewFactory) shows a button bar at the top



- Applications plug in, show their UI below the button bar, register interface:

```
public interface DesktopApplication {
    public static final String NAME = "name";
    public ImageView getImageView(Context context);
    public View getView(Context context);
}
```

Demo: dynamic deployment

- Bundles for:
  - desktop, button bar and plugin mechanism
  - weather, a simple weather application
  - maps, a mockup mapping application
- Deploy and use applications
- Undeploy applications

Context Aware Extension

- Combines the centrally managed model with a local one
- Phone can enable/disable certain components based on certain logic by talking to the ArtifactHandler service

```
public interface ArtifactHandler {
    public List<Artifact> listComponents() throws IOException;
    public void add(String name) throws IOException;
    public void remove(String name) throws IOException;
}
```

Demo: context awareness

- Same application as before
- Weather bundle is context aware:
  - only gets installed when your home WiFi network can be found
- Show deployment and undeployment without user intervention

Wrapping it up

- learned how to deploy and debug Android application
- seen how we can use OSGi and a management agent to deploy stuff
- seen some live demos

Links

- Apache Felix
  - http://felix.apache.org/
- Google Android
  - http://developer.android.com/
- Sample code
  - https://opensource.luminis.net/confluence/display/SITE/Apache+Felix+on+Androids
-

ApacheCon

Q & A

? ? !

!

? !