

# Android Services & Local IPC: The Command Processor Pattern (Part 1)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)



Professor of Computer Science

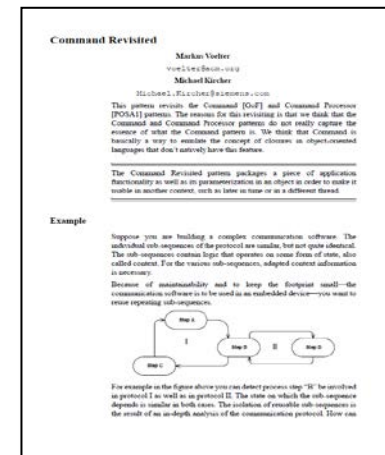
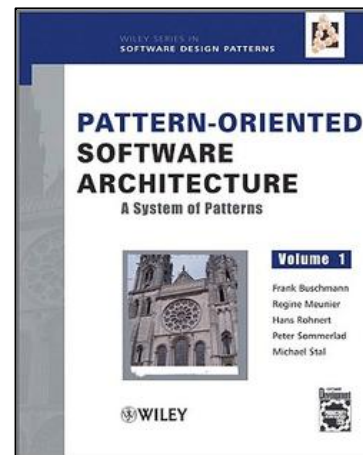
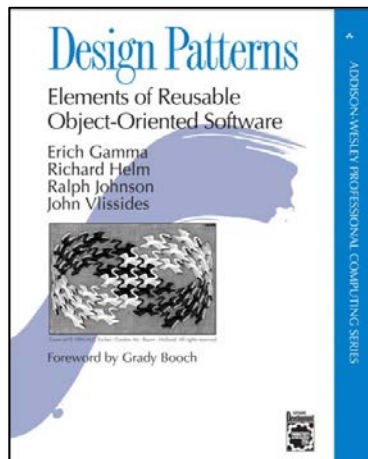
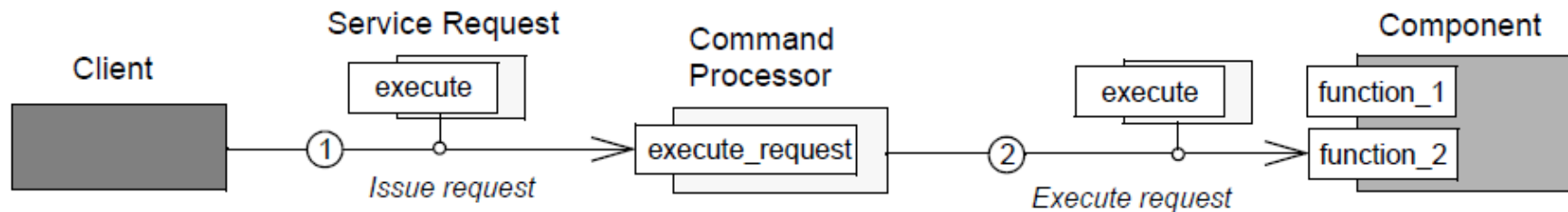
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Module

- Understand the *Command Processor* pattern



# Challenge: Processing a Long-Running Action

## Context

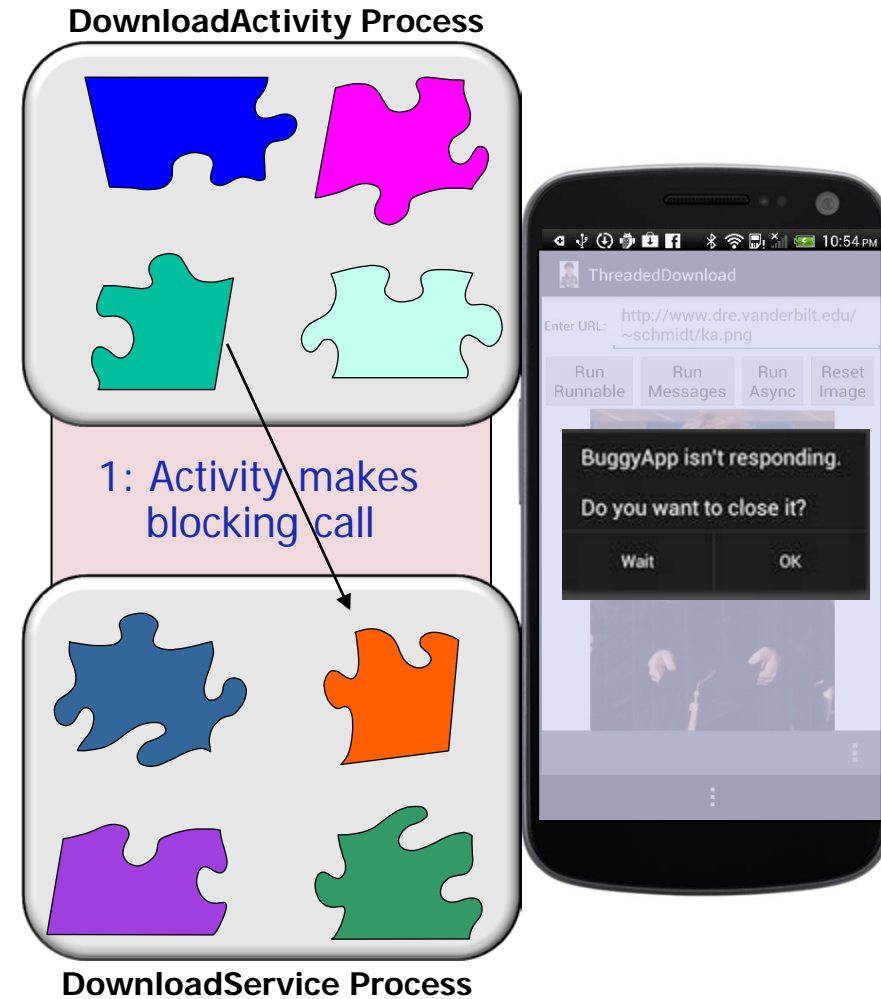
- Synchronous method calls in an Activity can block client for extended periods
- e.g., the `downloadImage()` call will block the `DownloadActivity` while the `DownloadService` fetches the image



# Challenge: Processing a Long-Running Action

## Problems

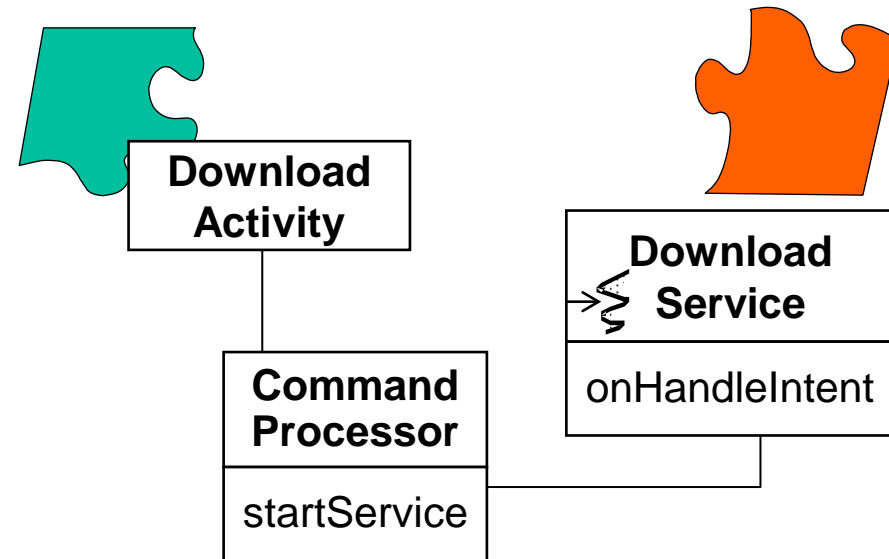
- Android generates an “Application Not Responding” (ANR) dialog if an app doesn’t respond to user input within a short time (~3 seconds)
- Calling a potentially lengthy operation like `downloadImage()` in the main thread can therefore be problematic



# Challenge: Processing a Long-Running Action

## Solution

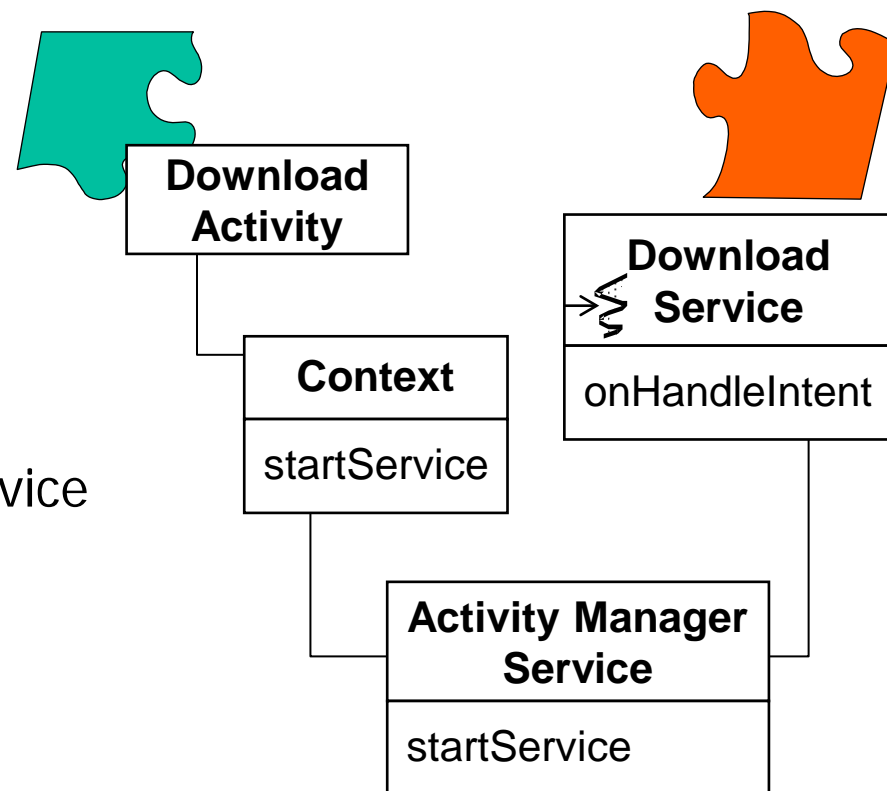
- Create a *command processor* that encapsulates a download request as an object that can be passed to a Service to execute the request



# Challenge: Processing a Long-Running Action

## Solution

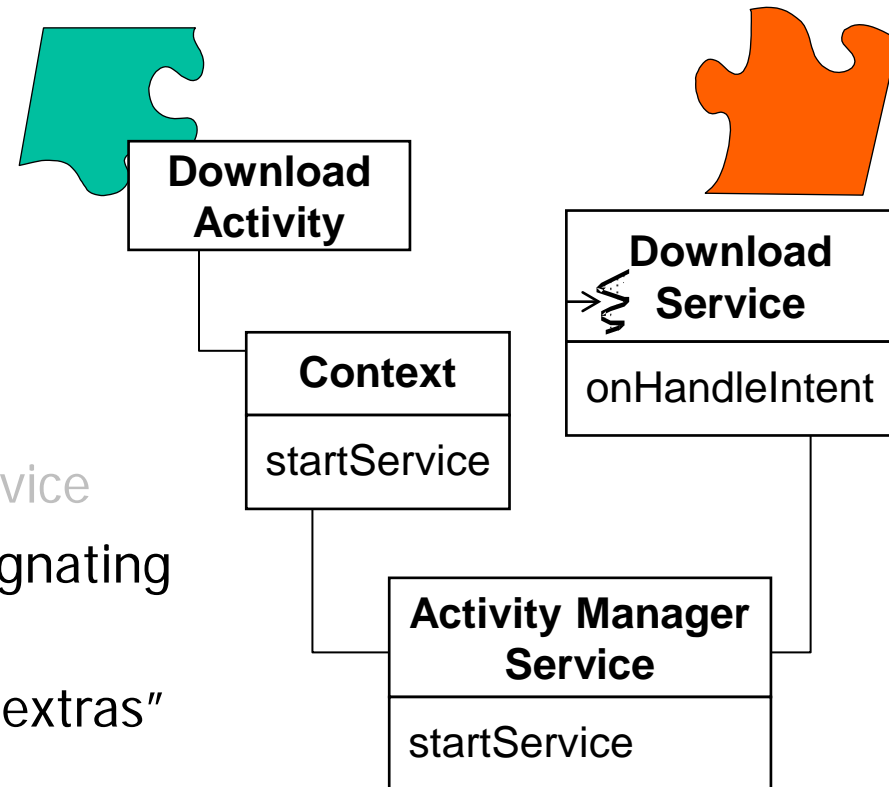
- Create a *command processor* that encapsulates a download request as an object that can be passed to a Service to execute the request
- This process works as follows:
  - Implement a DownloadService that inherits from Android's IntentService



# Challenge: Processing a Long-Running Action

## Solution

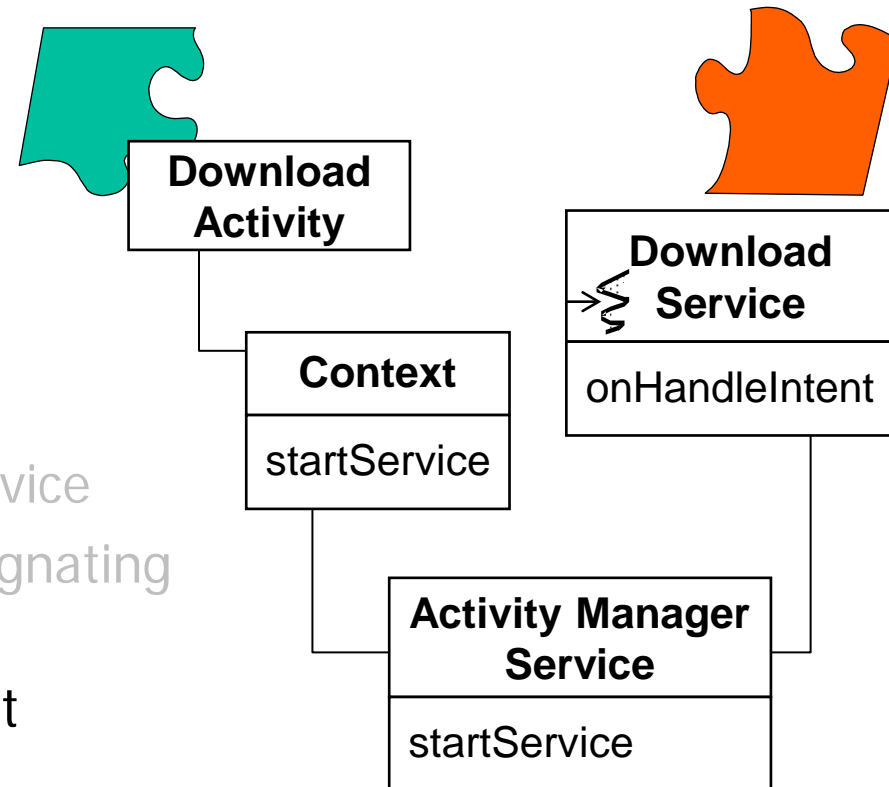
- Create a *command processor* that encapsulates a download request as an object that can be passed to a Service to execute the request
- This process works as follows:
  - Implement a `DownloadService` that inherits from Android's `IntentService`
  - Activity creates Intent command designating `DownloadService` as target
    - Add URL & callback Messenger as "extras"



# Challenge: Processing a Long-Running Action

## Solution

- Create a *command processor* that encapsulates a download request as an object that can be passed to a Service to execute the request
- This process works as follows:
  - Implement a DownloadService that inherits from Android's IntentService
  - Activity creates Intent command designating DownloadService as target
  - Activity calls startService() with Intent

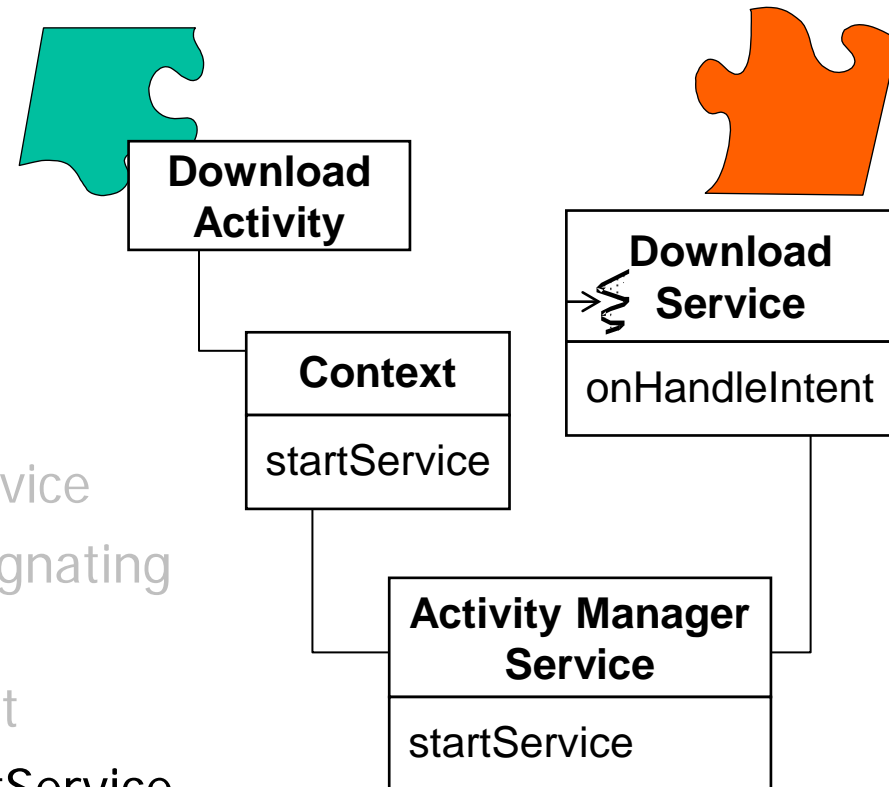




# Challenge: Processing a Long-Running Action

## Solution

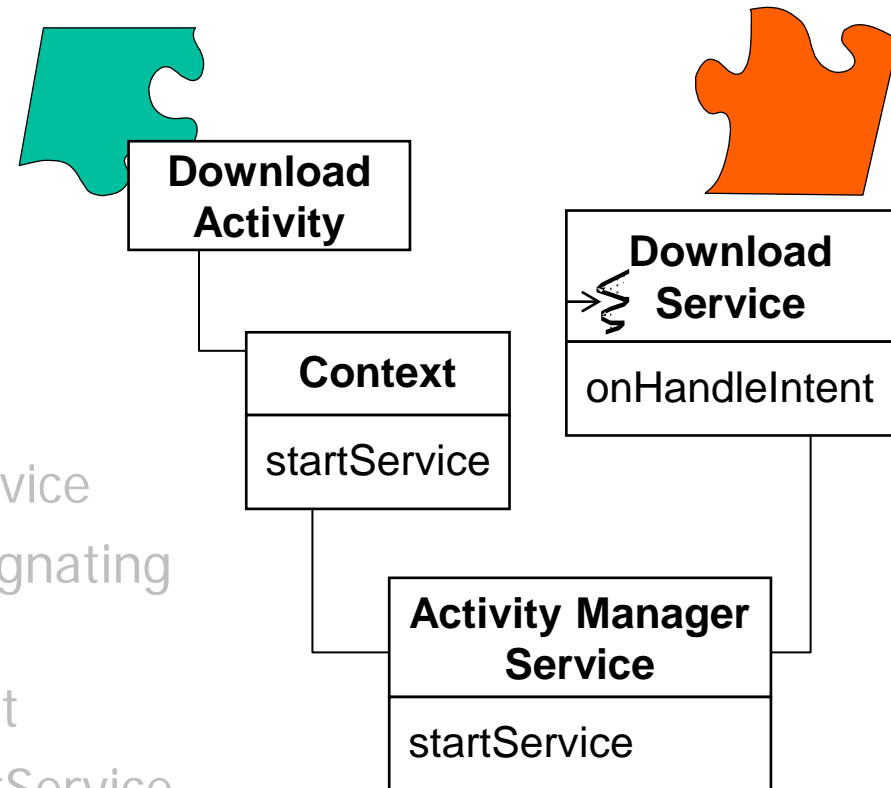
- Create a *command processor* that encapsulates a download request as an object that can be passed to a Service to execute the request
- This process works as follows:
  - Implement a DownloadService that inherits from Android's IntentService
  - Activity creates Intent command designating DownloadService as target
  - Activity calls startService() with Intent
  - Activity Manager Service starts IntentService, which spawns internal thread



# Challenge: Processing a Long-Running Action

## Solution

- Create a *command processor* that encapsulates a download request as an object that can be passed to a Service to execute the request
- This process works as follows:
  - Implement a DownloadService that inherits from Android's IntentService
  - Activity creates Intent command designating DownloadService as target
  - Activity calls startService() with Intent
  - Activity Manager Service starts IntentService, which spawns internal thread
  - IntentService calls onHandleIntent() to download image in separate thread

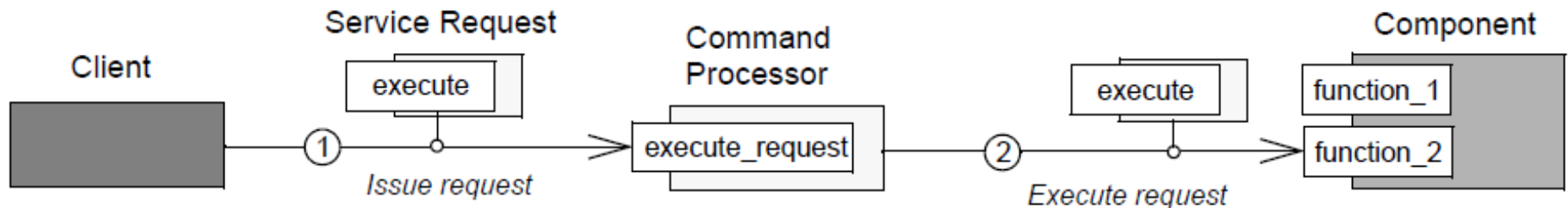


# Command Processor POA1 Design Pattern

## Intent

GoF book contains description of similar *Command* pattern

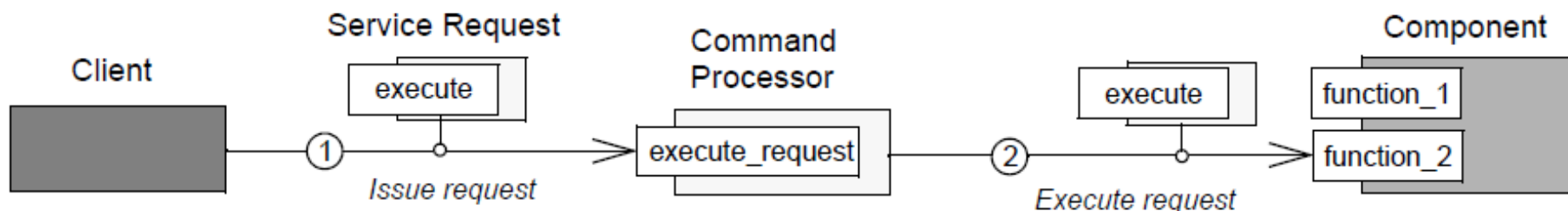
- Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context, such as later in time or in a different thread



# Command Processor POA1 Design Pattern

## Applicability

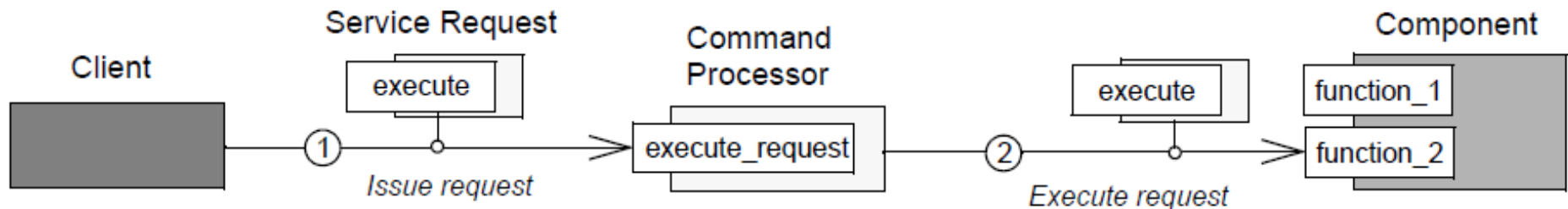
- When there's a need to decouple the decision of what piece of code should be executed from the decision of when this should happen
  - e.g., specify, queue, & execute service requests at different times



# Command Processor POA1 Design Pattern

## Applicability

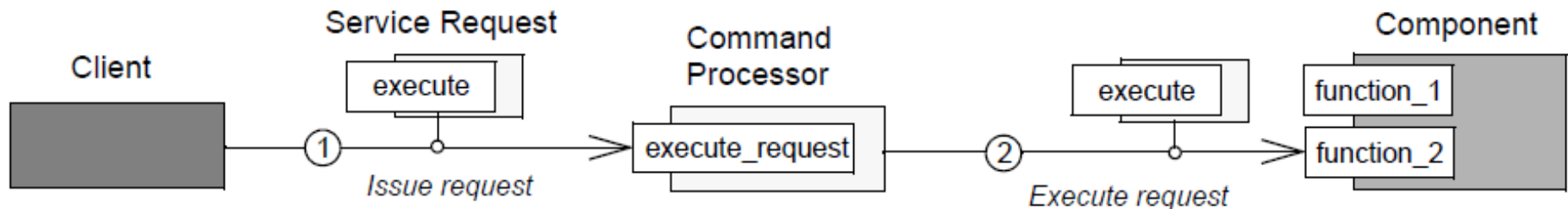
- When there's a need to decouple the decision of what piece of code should be executed from the decision of when this should happen
- When there's a need to ensure service enhancements don't break existing code



# Command Processor POA1 Design Pattern

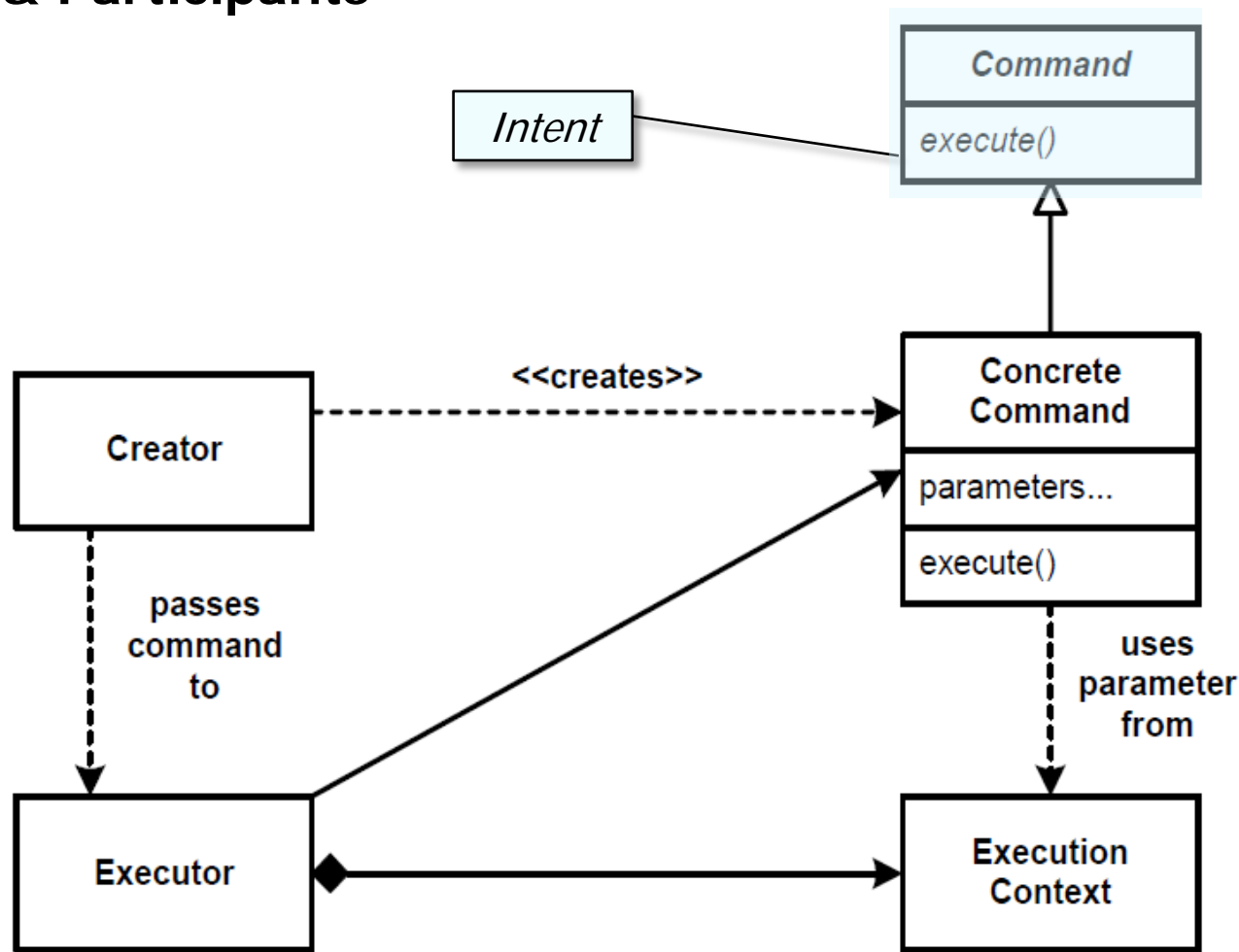
## Applicability

- When there's a need to decouple the decision of what piece of code should be executed from the decision of when this should happen
- When there's a need to ensure service enhancements don't break existing code
- When additional capabilities (such as undo/redo & persistence) must be implemented consistently for all requests to a service



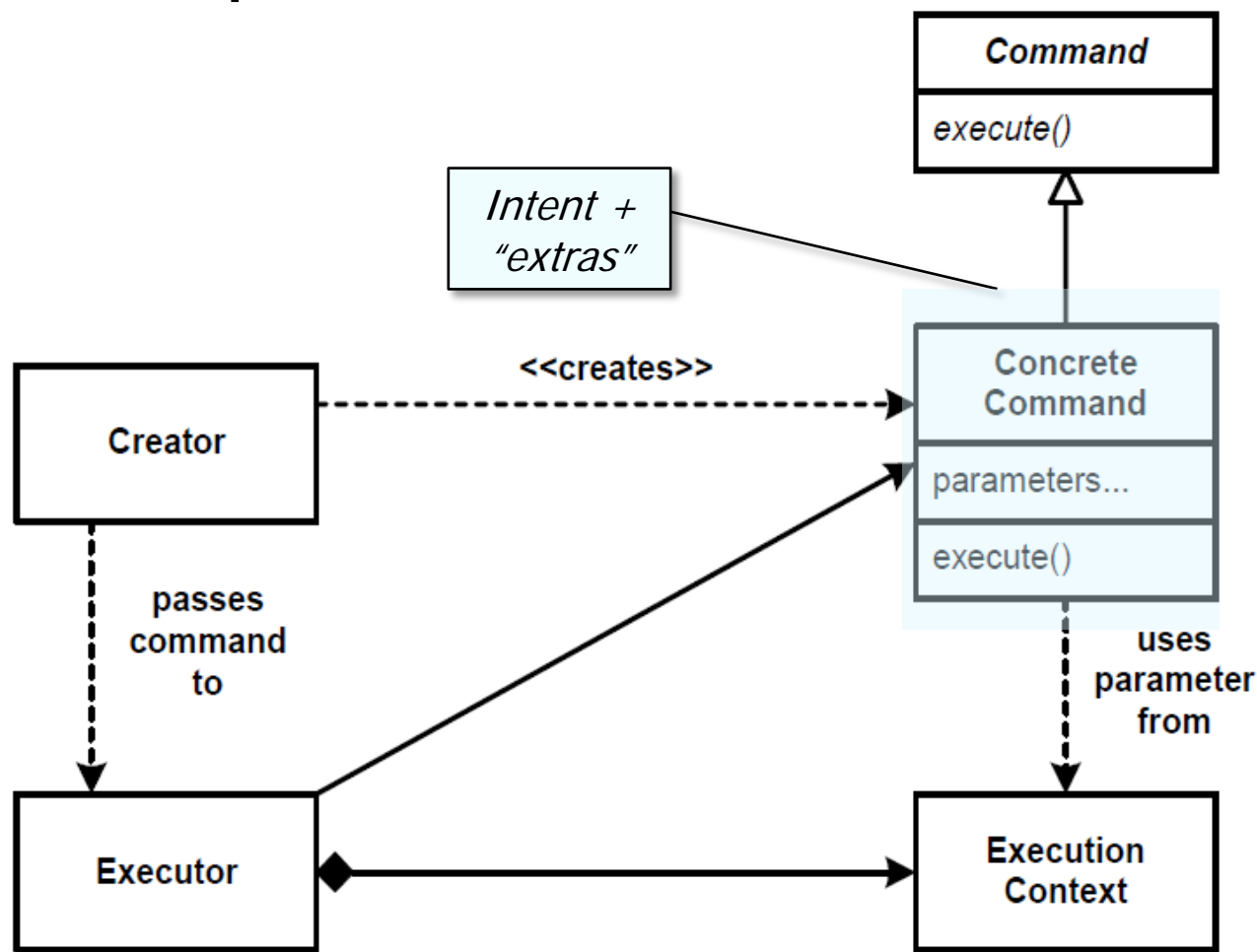
# Command Processor POA1 Design Pattern

## Structure & Participants



# Command Processor POA1 Design Pattern

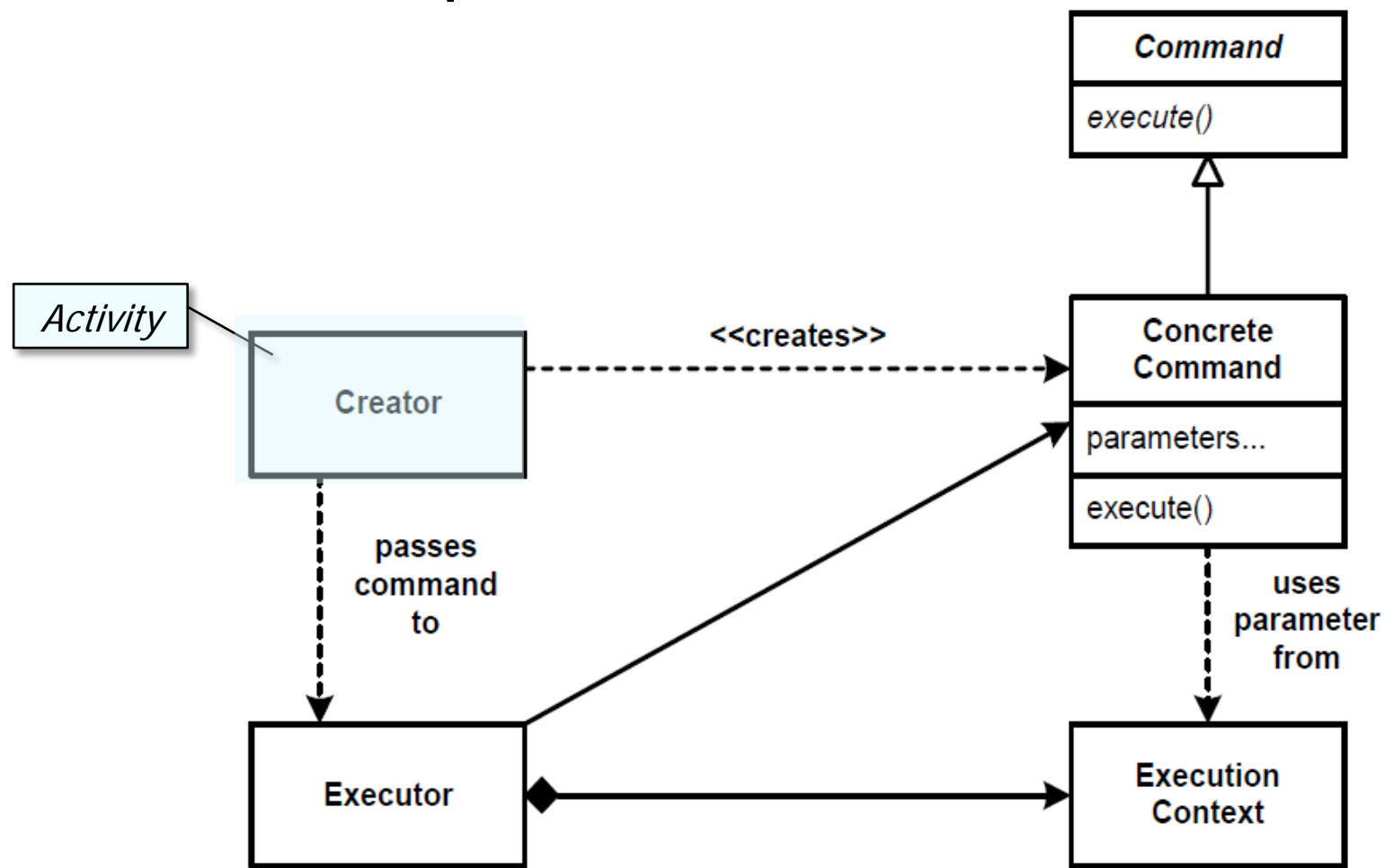
## Structure & Participants





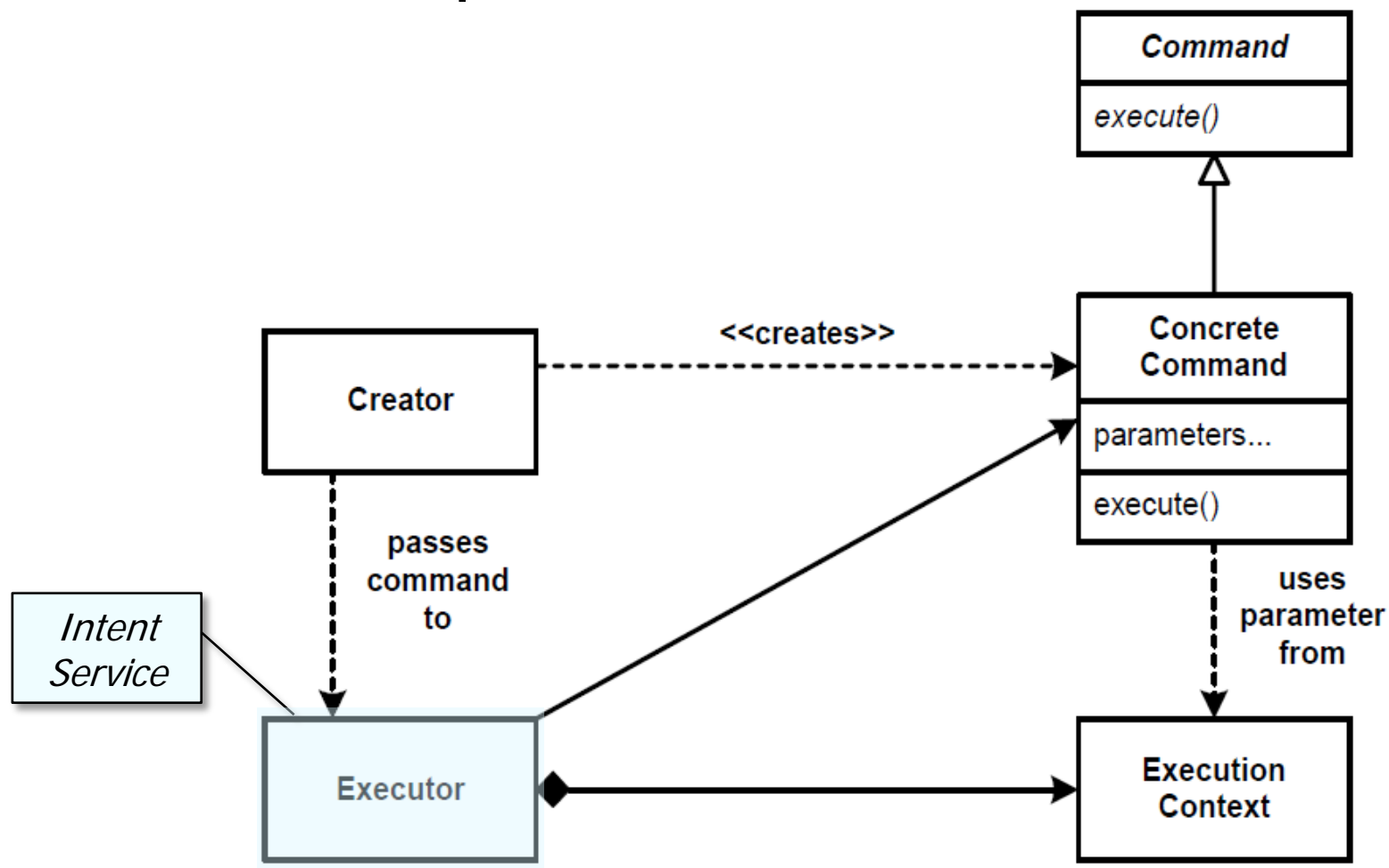
# Command Processor POA1 Design Pattern

## Structure & Participants



# Command Processor POA1 Design Pattern

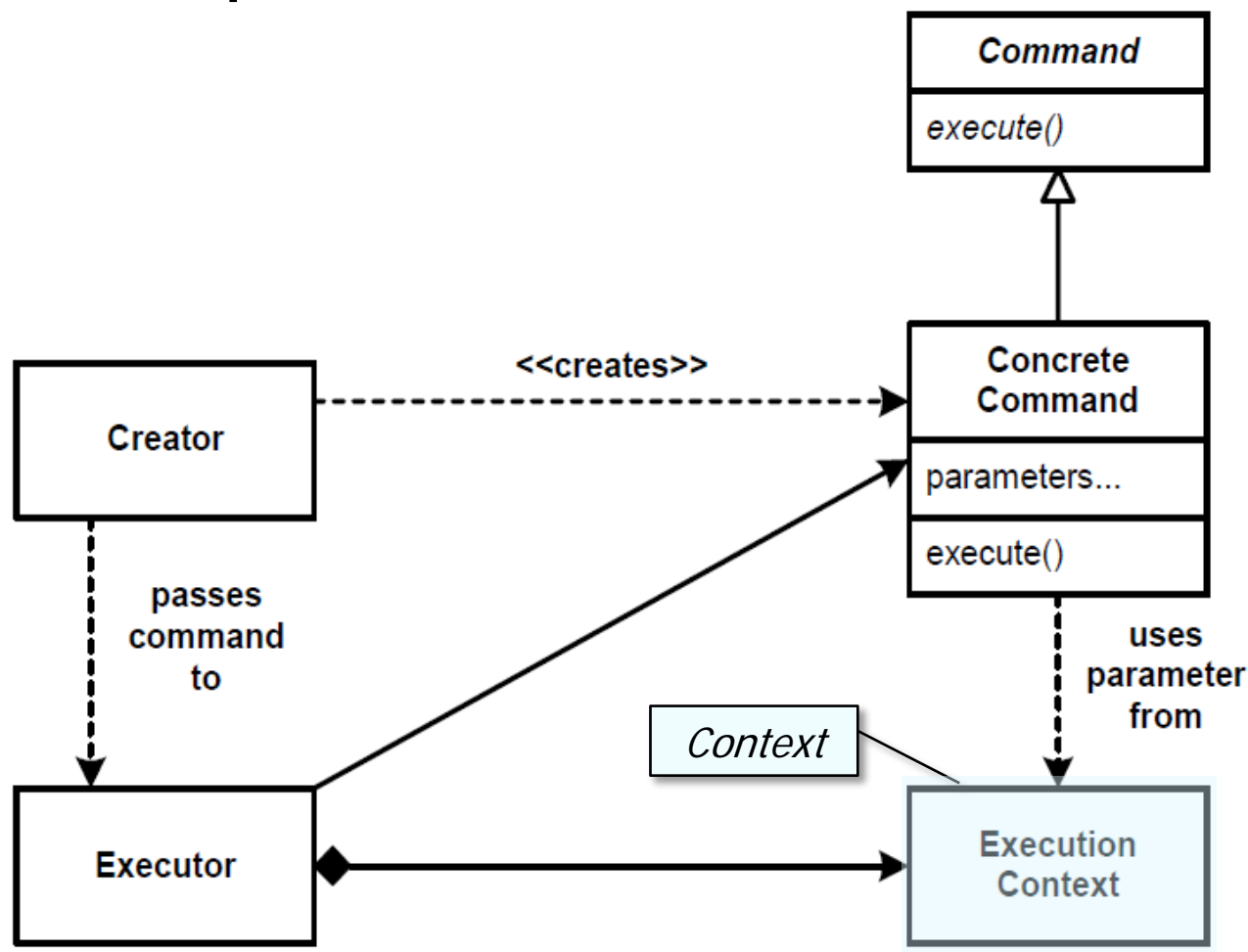
## Structure & Participants



# Command Processor

# POSA1 Design Pattern

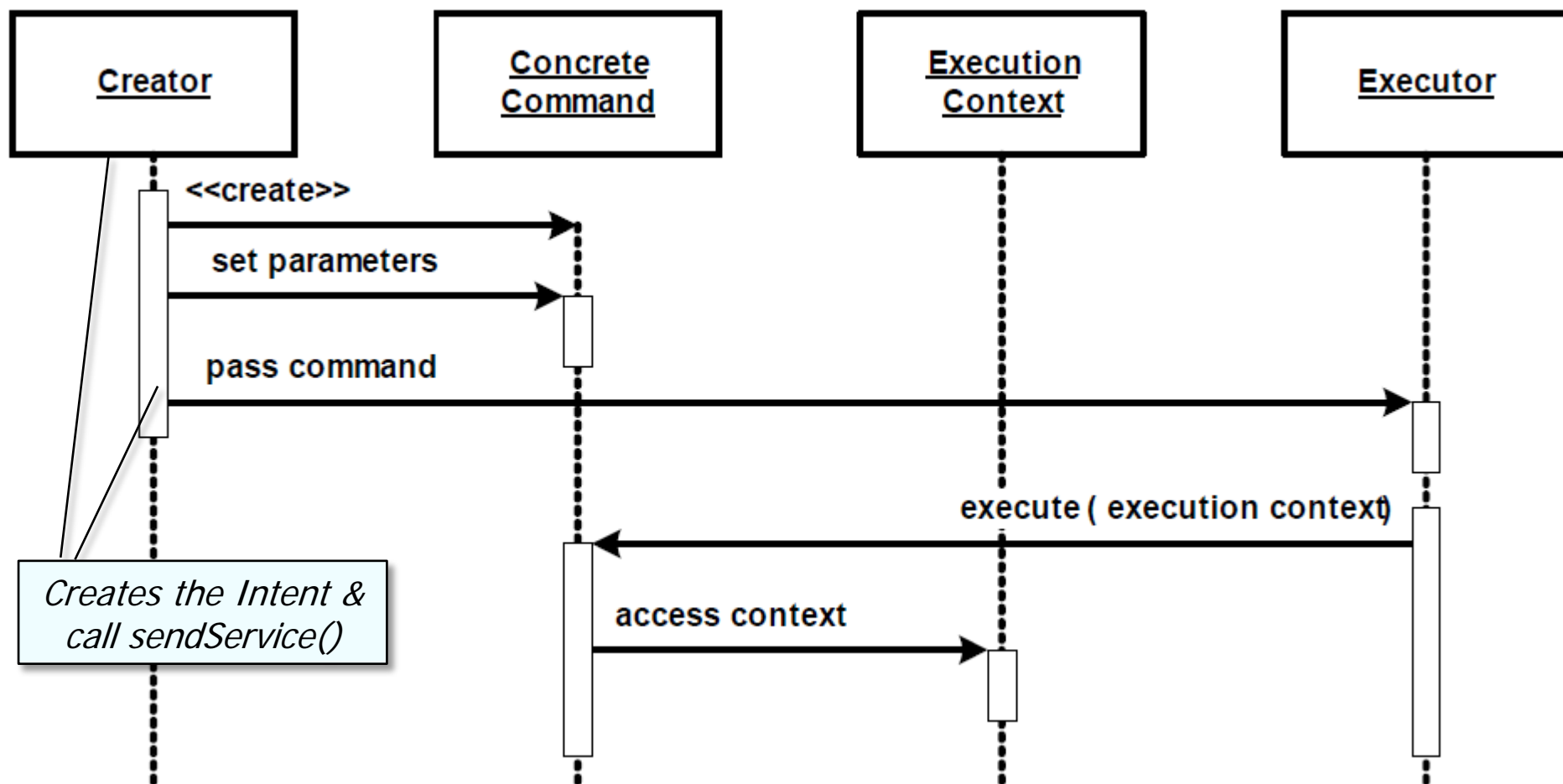
## Structure & Participants



# Command Processor

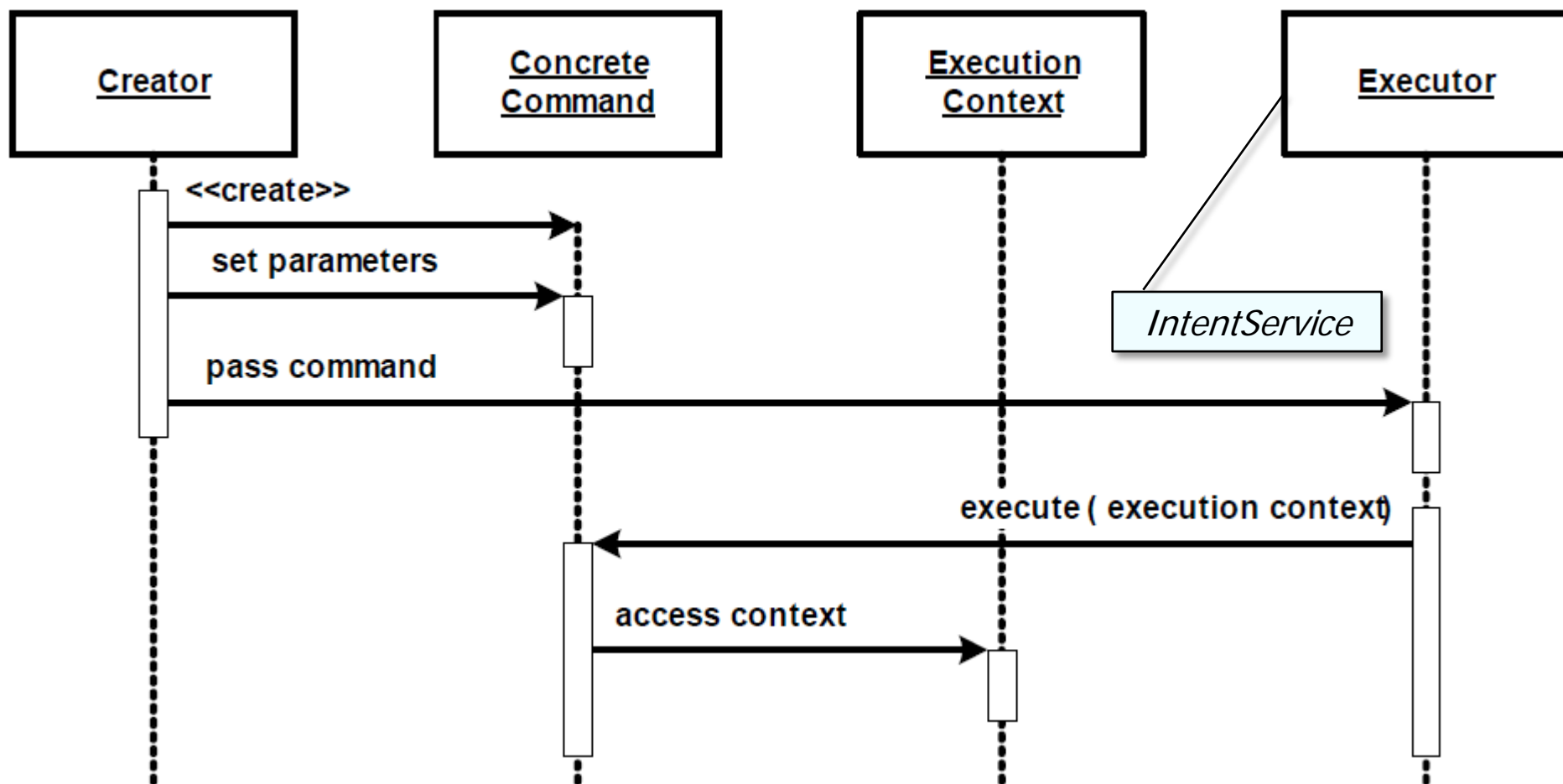
# POSA1 Design Pattern

## Dynamics



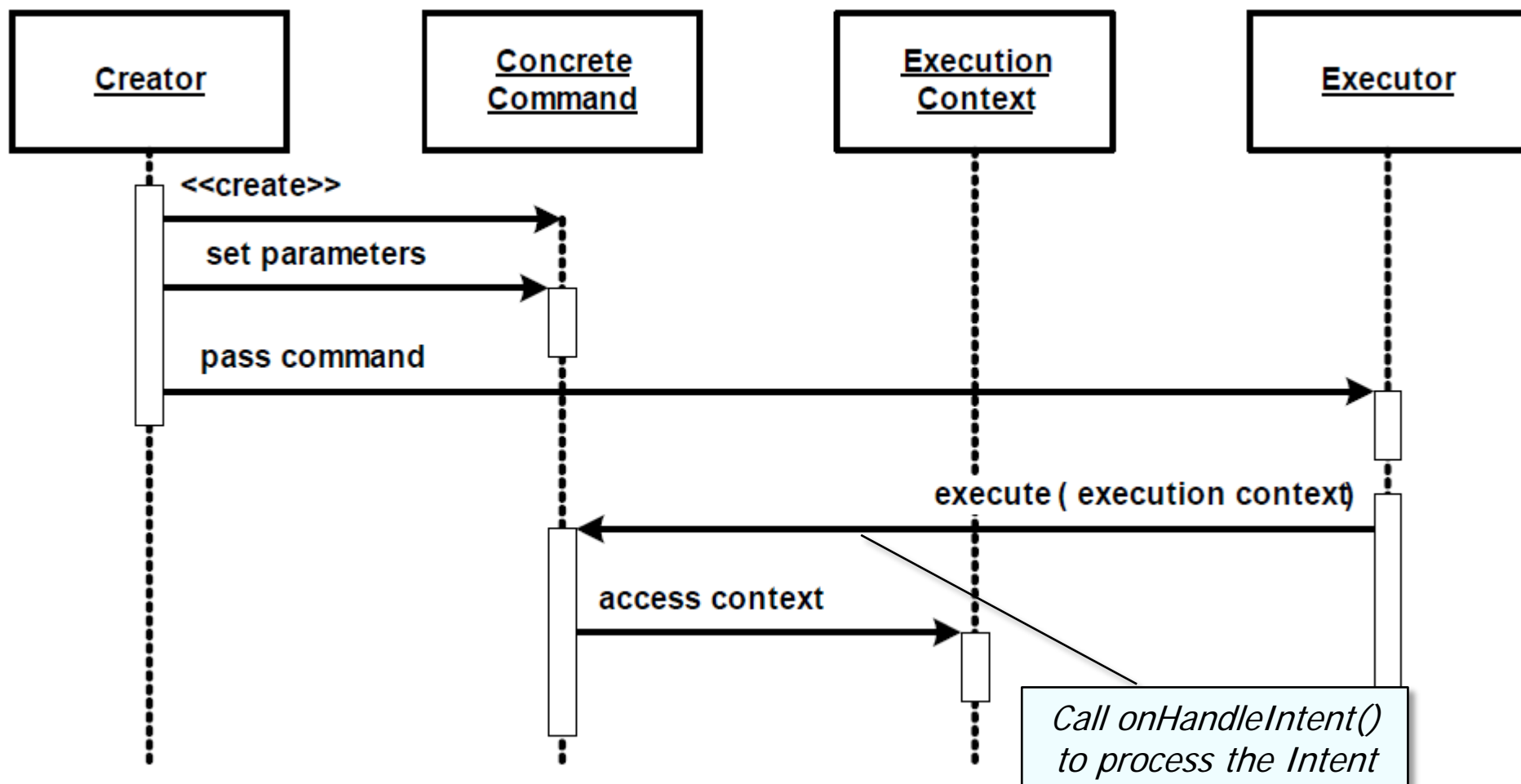
# Command Processor      POSA1 Design Pattern

## Dynamics



# Command Processor POA1 Design Pattern

## Dynamics



# Command Processor

# POSA1 Design Pattern

## Consequences

- + Client isn't blocked for duration of command processing

```
public void runMessengerDownload(View view) {  
    String url = editText.getText().toString();  
  
    Intent intent = new Intent(this, DownloadService.class);  
    intent.setData(Uri.parse (url));  
    Messenger messenger = new Messenger(handler);  
    intent.putExtra("MESSENGER", messenger);  
  
    startService(intent);  
}
```



Caller doesn't block

# Command Processor      POA1 Design Pattern

## Consequences

- + Client isn't blocked for duration of command processing
- + Allow different users to work with service in different ways via commands

```
public void onHandleIntent(Intent intent) {  
    Bundle extras = intent.getExtras();  
    if (extras != null && extras.get("MESSENGER") != null)  
        messengerDownload (intent);  
    else  
        broadcastDownload (intent);  
}
```





# Command Processor      POA1 Design Pattern

## Consequences

- Additional programming to handle info passed with commands (*cf.* Broker)

```
public void onHandleIntent(Intent intent) {  
    Bundle extras = intent.getExtras();  
    if (extras != null && extras.get("MESSENGER") != null)  
        messengerDownload (intent);  
    else  
        broadcastDownload (intent);  
}
```



# Command Processor      POA1 Design Pattern

## Consequences

- Additional programming to handle info passed with commands (*cf.* Broker)
- Supporting two-way operations requires additional patterns & IPC mechanisms

```
void sendPath (String outputPath, Messenger messenger) {  
    Message msg = Message.obtain();  
    msg.arg1 = result;  
    Bundle bundle = new Bundle();  
    bundle.putString(RESULT_PATH, outputPath);  
    msg.setData(bundle);  
    ...  
    messenger.send(msg);  
}
```



# Command Processor      POA1 Design Pattern

## Known Uses

- Android IntentService
- Many UI toolkits
  - InterViews, ET++, MacApp, Swing, AWT, etc.
- Interpreters for command-line shells
- Java **Runnable** interface

public abstract class      Summary: Inherited Constants | Ctors | Methods | Protected  
**IntentService**      Methods | Inherited Methods | [Expand All]  
extends Service      Added in API level 3

java.lang.Object  
↳ android.content.Context  
↳ android.content.ContextWrapper  
↳ android.app.Service  
↳ android.app.IntentService

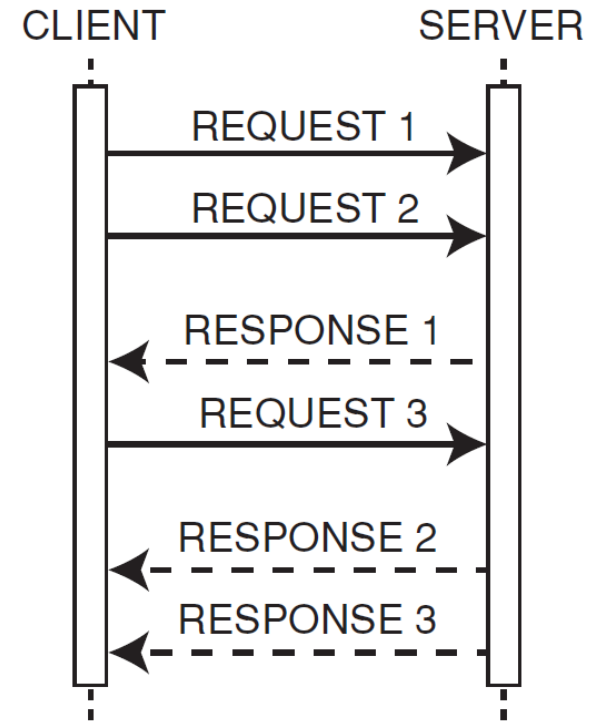
## Class Overview

IntentService is a base class for [Services](#) that handle asynchronous requests (expressed as [Intents](#)) on demand. Clients send requests through [startService\(Intent\)](#) calls; the service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work.

This "work queue processor" pattern is commonly used to offload tasks from an application's main thread. The IntentService class exists to simplify this pattern and take care of the mechanics. To use it, extend IntentService and implement [onHandleIntent\(Intent\)](#). IntentService will receive the Intents, launch a worker thread, and stop the service as appropriate.

## Summary

- *Command Processor* provides a relatively straightforward means for passing commands asynchronously between threads and/or processes in concurrent & networked software



# Android Services & Local IPC: The Command Processor Pattern (Part 2)

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)



Professor of Computer Science

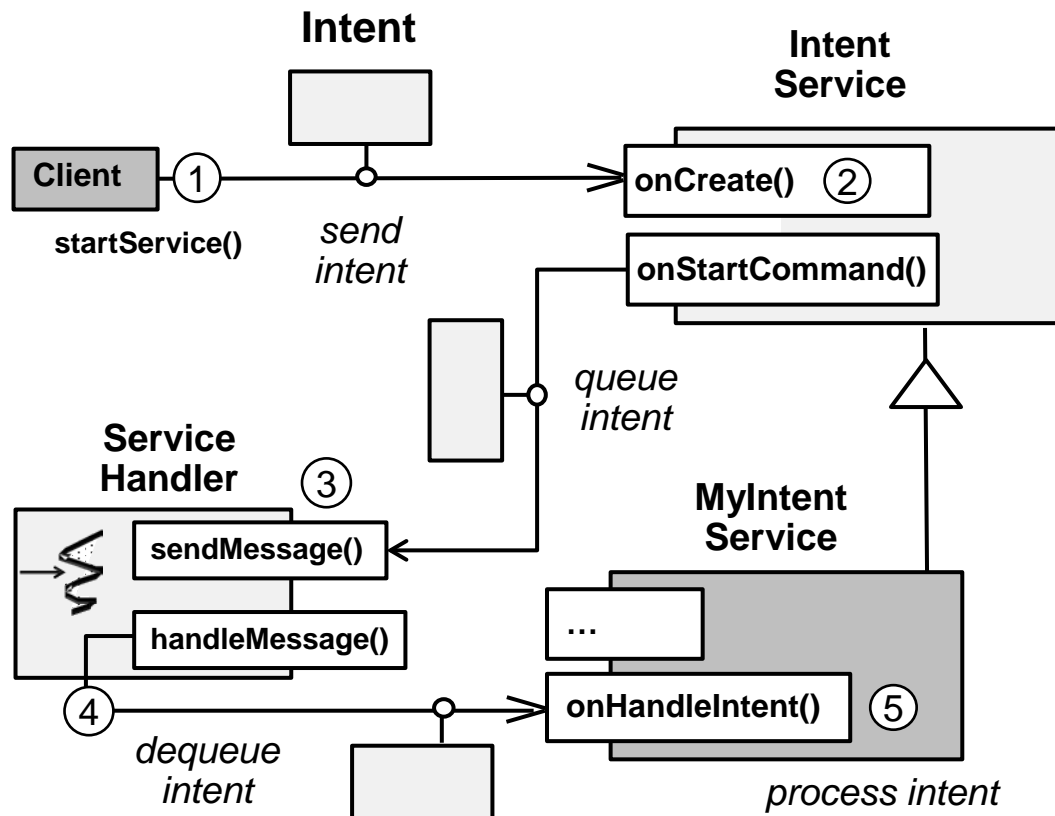
Institute for Software  
Integrated Systems

Vanderbilt University  
Nashville, Tennessee, USA



# Learning Objectives in this Part of the Module

- Understand how *Command Processor* is applied in Android



# Command Processor

# POSA1 Design Pattern

## Implementation

- Define an abstract class for command execution that will be used by the executor
- Typically define an execute() operation

```
public class Intent implements  
    Parcelable, Cloneable {  
    ...  
}
```

# Command Processor

# POSA1 Design Pattern

## Implementation

- Define an abstract class for command execution that will be used by the executor
- Add state which concrete commands need during their execution to the context
- Make the context available to the concrete command

```
public class Intent implements
    Parcelable, Cloneable {

    ...
    public Intent setData(Uri data)
    { /* ... */ }

    public Uri getData()
    { /* ... */ }

    public Intent putExtra
        (String name, Bundle value)
    { /* ... */ }

    public Object getExtra
        (String name)
    { /* ... */ }
}
```



# Command Processor

# POSA1 Design Pattern

## Implementation

- Define an abstract class for command execution that will be used by the executor
- Add state which concrete commands need during their execution to the context
- Define & implement the creator
  - e.g., using patterns like Abstract Factory or Factory Method

```
public class DownloadActivity
    extends Activity {
    ...

    public void onClick(View v) {
        Intent intent = new
            Intent(DownloadActivity.this,
                DownloadService.class);
        ...
        intent.setData
            (Uri.parse(userInput));
        ...
        startService(intent);
    }
    ...
}
```

# Command Processor

# POSA1 Design Pattern

## Implementation

- Define an abstract class for command execution that will be used by the executor
- Add state which concrete commands need during their execution to the context
- Define & implement the creator
- Define the context
  - If necessary allow it to keep references to command objects, but be aware of lifecycle issues

```
public abstract class Context {  
    public abstract void  
        sendBroadcast(Intent intent);  
  
    public abstract Intent  
        registerReceiver  
            (BroadcastReceiver receiver,  
             IntentFilter filter);  
}
```

# Command Processor

# POSA1 Design Pattern

## Implementation

- Define an abstract class for command execution that will be used by the executor
- Add state which concrete commands need during their execution to the context
- Define & implement the creator
- Define the context
- Implement specific command functionality in subclasses:
  - Implementing the execute() operation, adding necessary attributes

```
public class DownloadService
    extends IntentService {

    ...

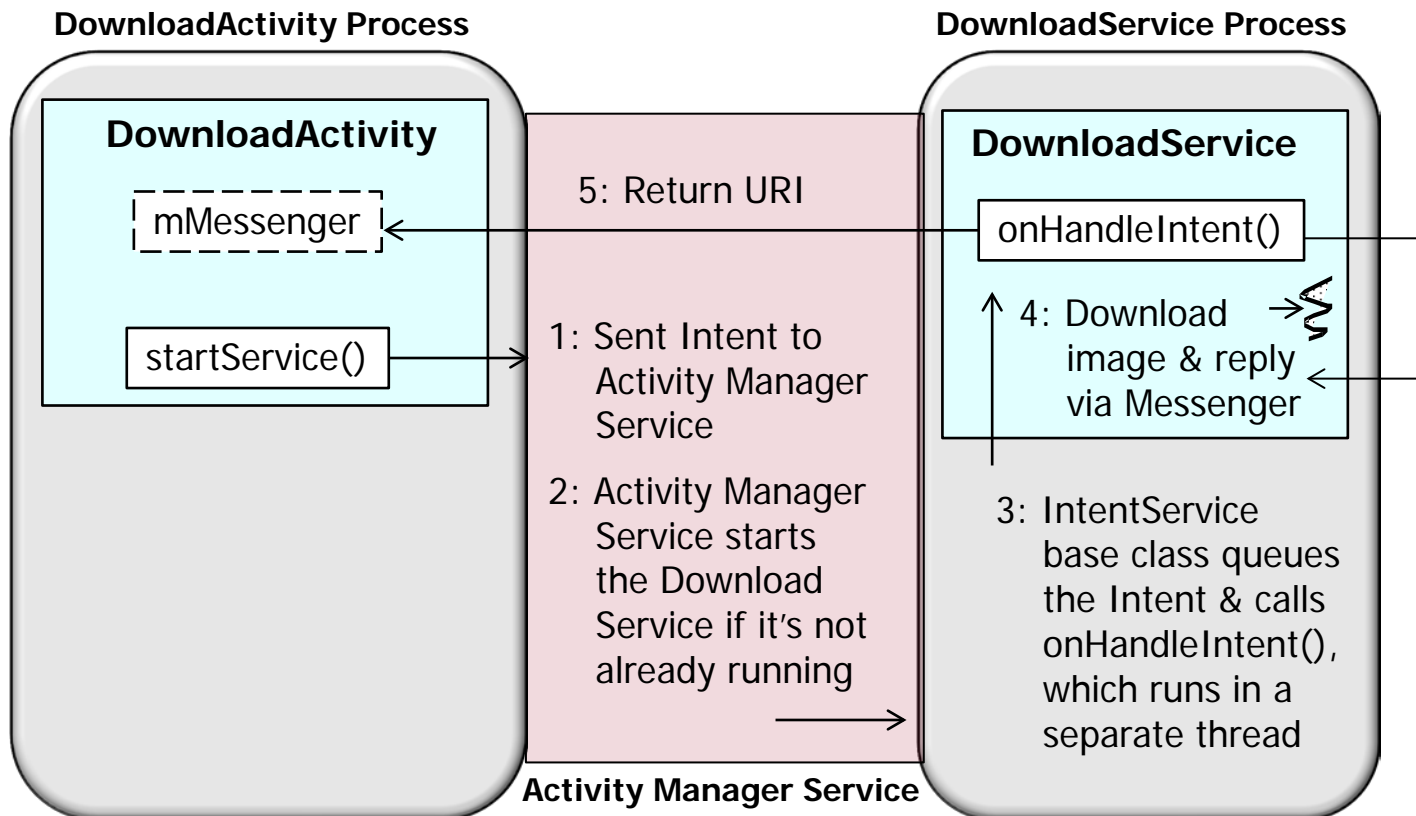
    protected void
        onHandleIntent(Intent intent)
    {
        downloadImage(intent);
    }

    ...
}
```

# Command Processor      POSA1 Design Pattern

## Applying Command Processor in Android

(Some) steps involved in the Android implementation of *Command Processor* pattern




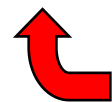
Other patterns are involved here: *Activator*, *Messaging*, *Result Callback*, etc.

# Command Processor      POA1 Design Pattern

## Applying Command Processor in Android

(Portion of) the DownloadActivity implemented using the *Command Processor* pattern

```
public void runDownloadImage(View view) {  
    URL url = new URL(image_url.getText().toString())  
    Intent intent = new Intent(this,  
        Make Intent "command"  DownloadService.class);  
  
    intent.putExtra(DownloadService.MESSENGER,  
                    new Messenger(handler))  
    intent.putExtra(DownloadService.URL, url);  
  
    startService(intent);  
}
```



**Issue request to command processor**



This code runs in the DownloadActivity process



# Command Processor      POA1 Design Pattern

## Applying Command Processor in Android

(Portion of) the DownloadService implemented using the *Command Processor* pattern

 **Command processor executes the request**

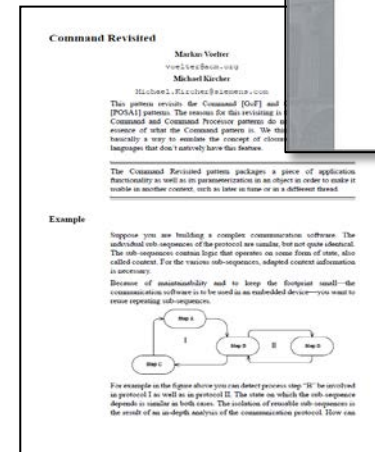
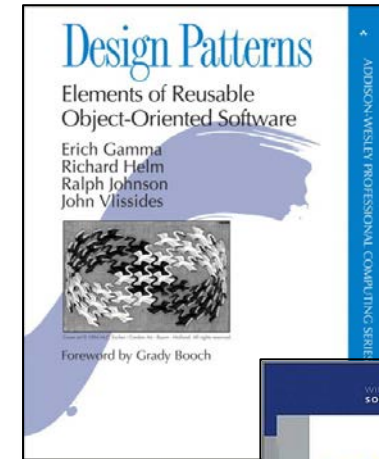
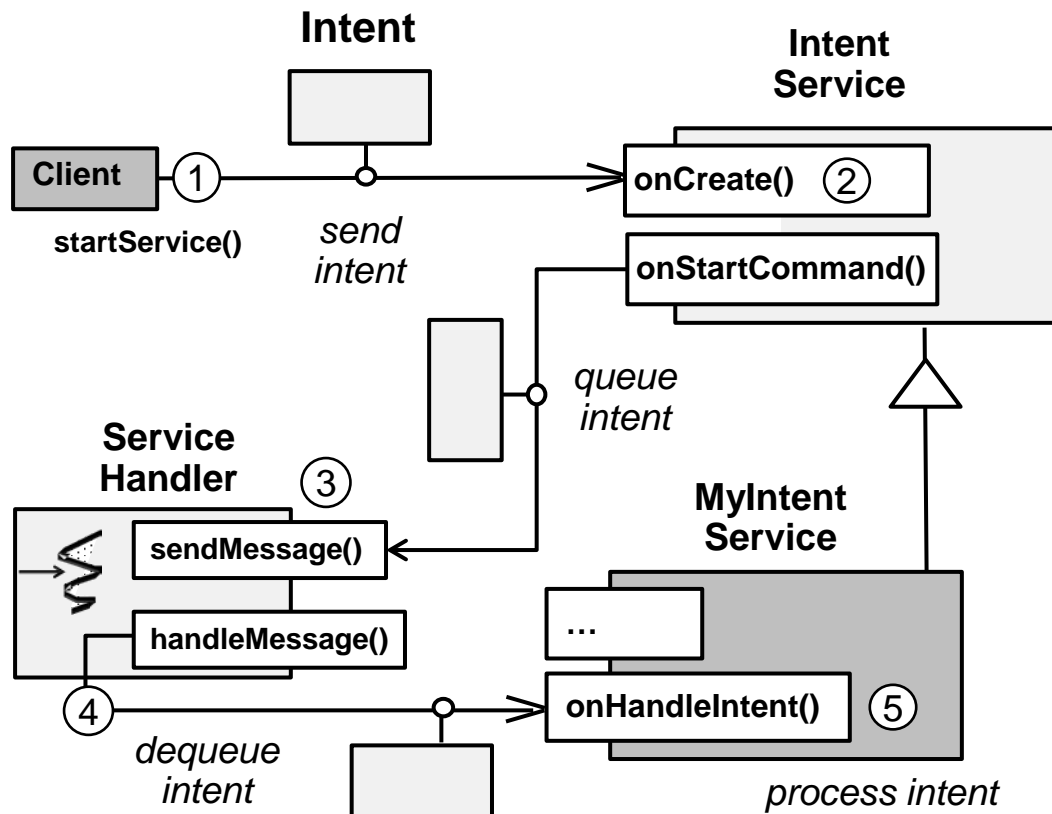
```
public class DownloadService extends IntentService {  
    ...  
  
    protected void onHandleIntent(Intent intent) {  
        Bundle extras = intent.getExtras();  
        URL url = (URL)extras.get(URL);  
        Messenger messenger (Messenger)extras.get(MESSANGER);  
  
        // Download image at designated URL & send  
        // reply back to Activity via messenger callback  
        downloadImage(url, messenger);  
    }  
}
```

This code runs in a thread in the DownloadService process



# Summary

- The Android Intent Service framework implements the *Command Processor* pattern & is used to process Intents in a background Thread

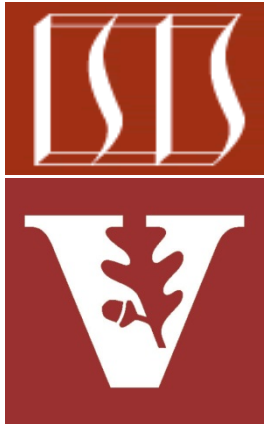


# Android Services & Local IPC: Overview of Bound Services

Douglas C. Schmidt

[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)

[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)



Professor of Computer Science

Institute for Software  
Integrated Systems

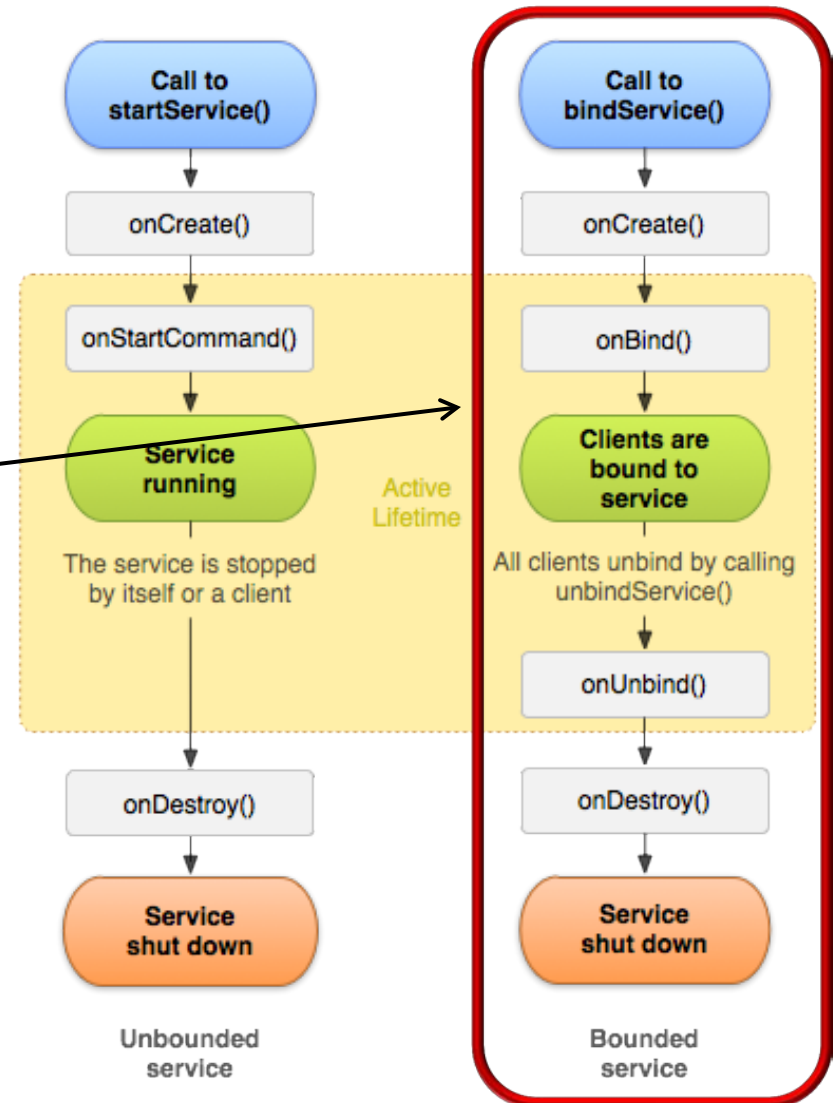
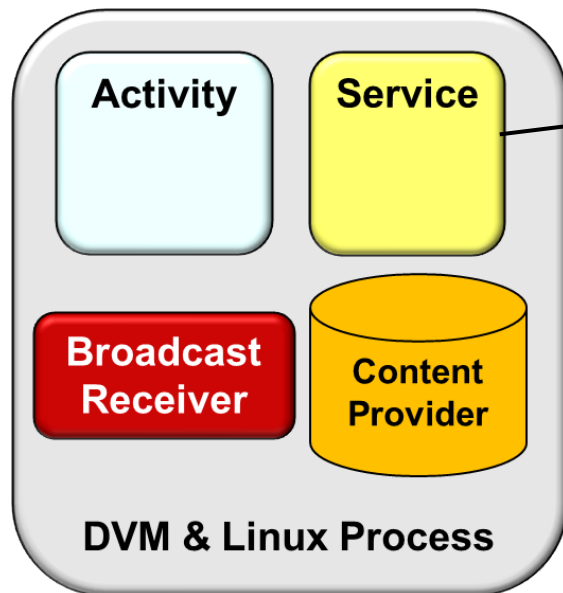
Vanderbilt University  
Nashville, Tennessee, USA





# Learning Objectives in this Part of the Module

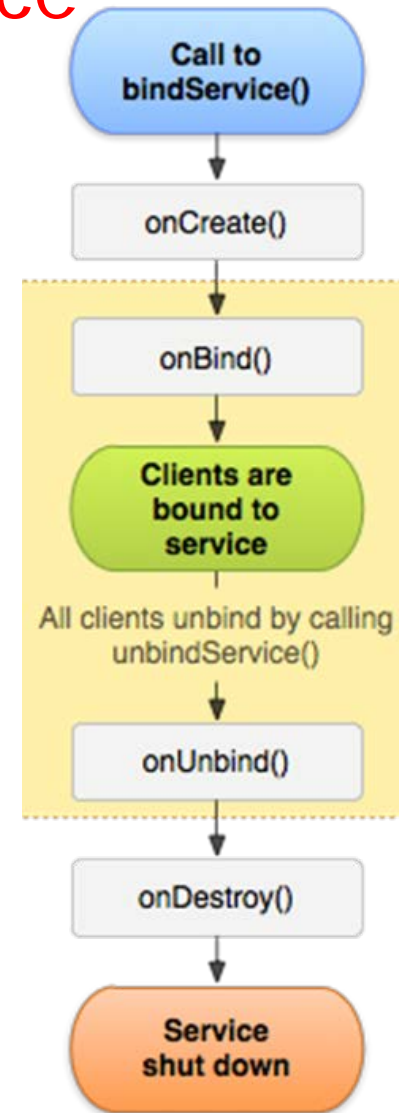
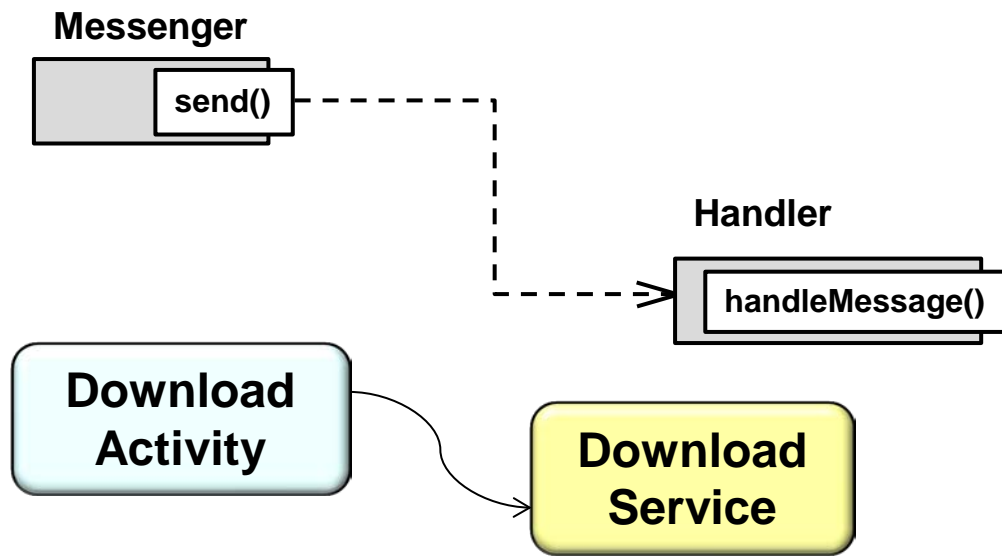
- Understand how what a Bound Service is & what hook methods it defines to manage its various lifecycle states



We'll emphasize commonalities & variabilities in our discussion

# Interfacing with a Bound Service

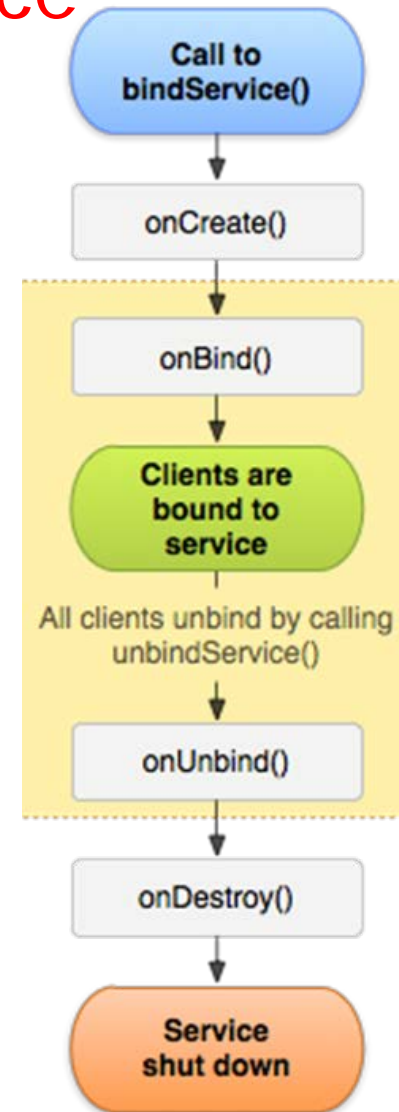
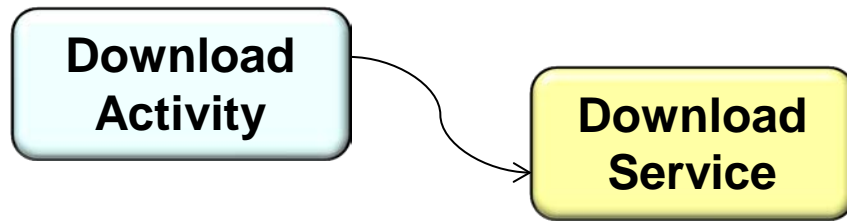
- A Bound Service offers components an interface that clients can use to interact with the Service
- This interface can be generic
  - e.g., using Messengers & Handlers for inter- or intra-process communication



# Interfacing with a Bound Service

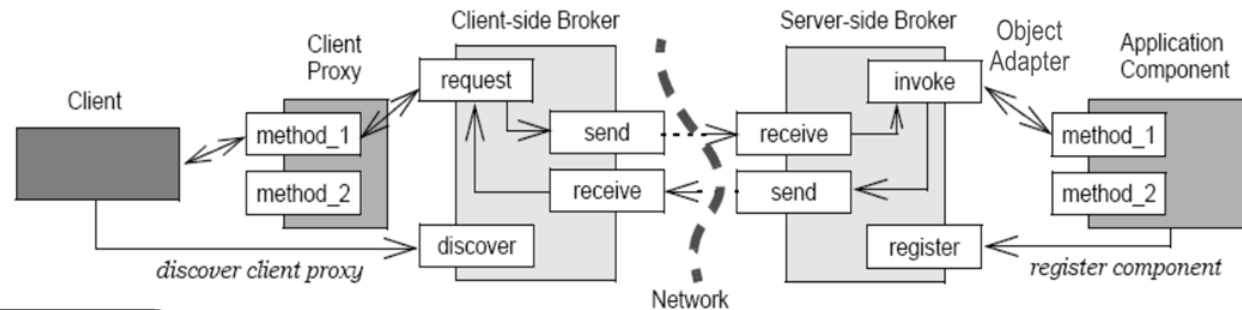
- A Bound Service offers components an interface that clients can use to interact with the Service
  - This interface can be generic
  - This interface can also be specific
    - e.g., using the Android Interface Definition Language (AIDL) for inter- or intra-process communication

```
interface IDownload {  
    String downloadImage(in Uri uri);  
}
```



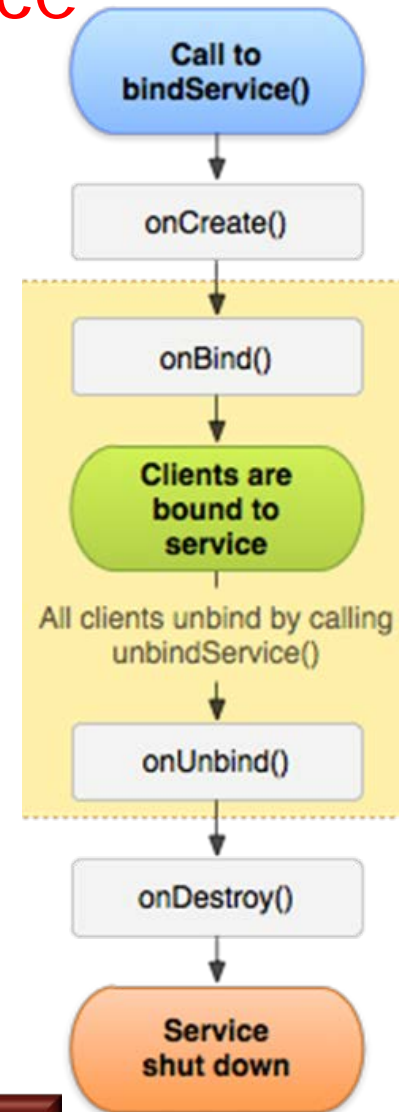
# Interfacing with a Bound Service

- A Bound Service offers components an interface that clients can use to interact with the Service
  - This interface can be generic
  - This interface can also be specific
- Both approaches use the Binder RPC mechanism
  - This implements the *Broker & Proxy* patterns



Download Activity

Download Service



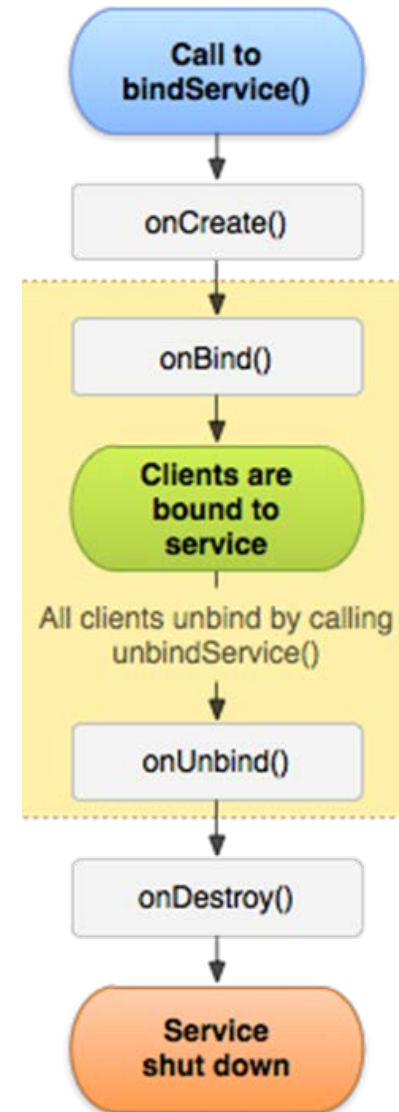
# Launching a Bound Service

- A Bound Service allows App components to bind to it by calling `bindService()` to create a "persistent" connection

```
Intent intent = new  
    Intent(IDownloadSync.class.getName());  
bindService(intent, this.conn,  
            Context.BIND_AUTO_CREATE);
```

**Download  
Activity**

**Download  
Service**



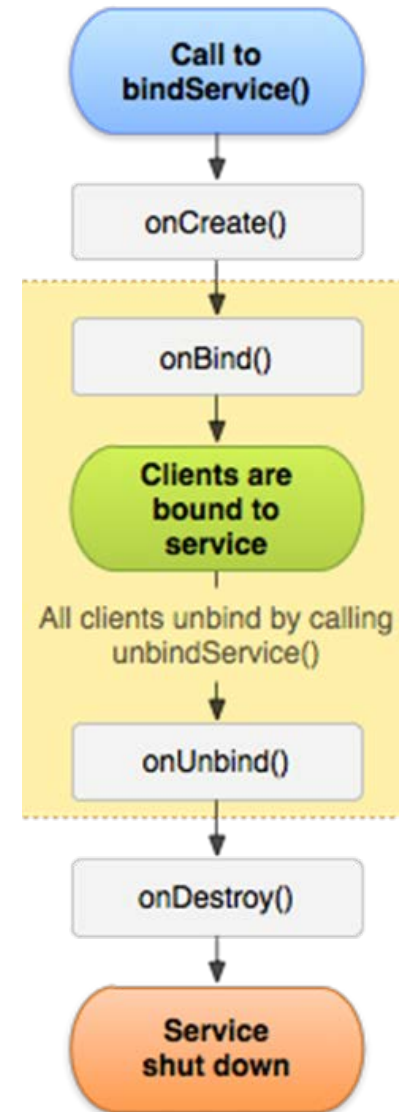
# Launching a Bound Service

- A Bound Service allows App components to bind to it by calling `bindService()` to create a “persistent” connection
- The client must provide `ServiceConnection` object to monitor the connection with the Service

```
Intent intent = new  
    Intent(IDownloadSync.class.getName());  
bindService(intent, this.conn,  
            Context.BIND_AUTO_CREATE);
```

Download  
Activity

Download  
Service



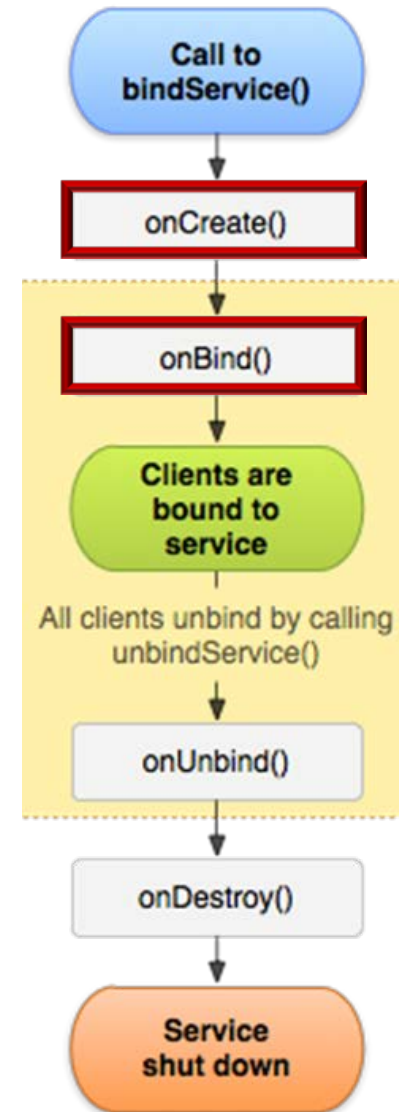
## Connecting a Bound Service

- When the client calls `bindService()` Android starts the Service & invokes the Service's `onCreate()` & `onBind()` hook methods
- If the Service isn't already running it will be activated

```
IDownload.Stub binder;  
Class DownloadService extends Service {  
    public void onCreate(Bundle savedInstanceState) {  
        binder = new IDownload.Stub() {  
            public String downloadImage(String urlValue)  
            { /* ... */ }  
        }  
    }  
}
```

Download  
Activity

Download  
Service





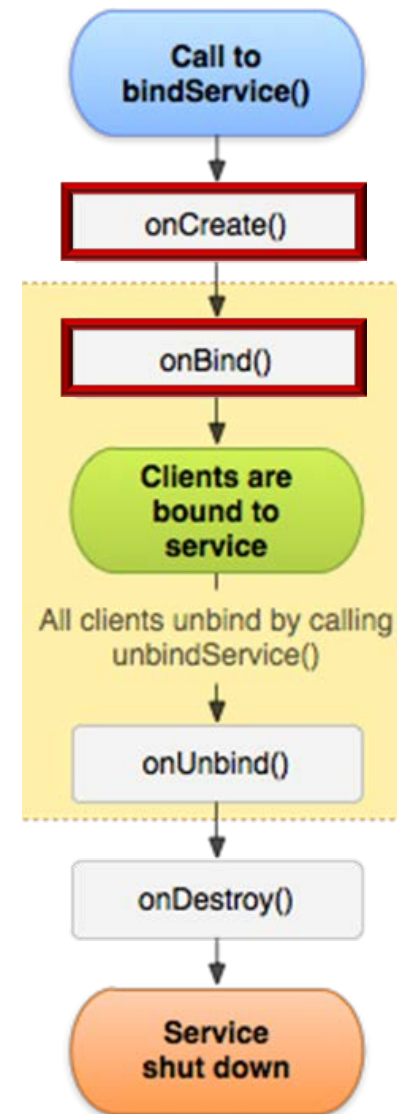
# Connecting a Bound Service

- When the client calls `bindService()` Android starts the Service & invokes the Service's `onCreate()` & `onBind()` hook methods
  - If the Service isn't already running it will be activated
- `onBind()` returns an `IBinder` object that defines the API for communicating with the Bound Service

```
...  
public IBinder onBind(Intent intent)  
{  
    return this.binder;  
}
```

**Download  
Activity**

**Download  
Service**



An interesting callback-driven protocol is used to establish a connection



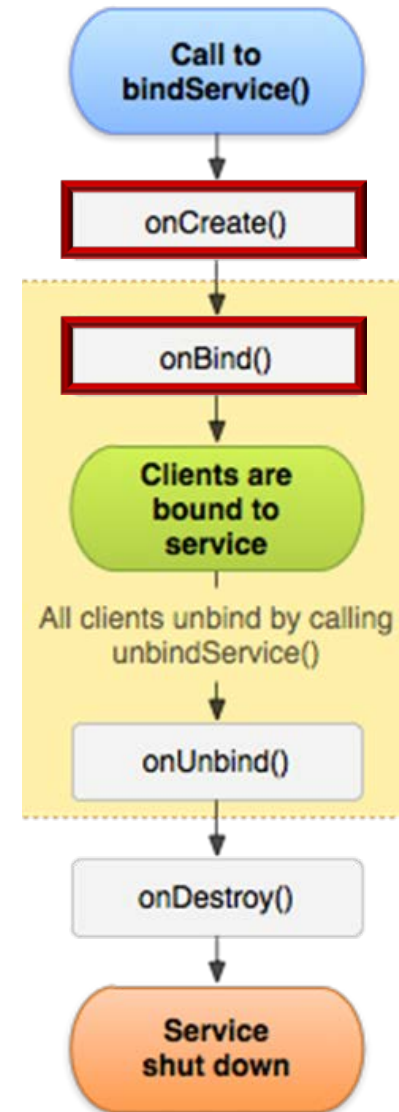
## Connecting a Bound Service

- When the client calls `bindService()` Android starts the Service & invokes the Service's `onCreate()` & `onBind()` hook methods
- The client needs to implement an `onServiceConnected()` hook method to get a proxy to the `IBinder`

```
Idownload syncSvc;  
ServiceConnection conn = new ServiceConnection() {  
    public void onServiceConnected  
        (ComponentName className, IBinder iSvc) {  
        syncSvc = IDownload.Stub.asInterface(iSvc);  
    }  
}
```

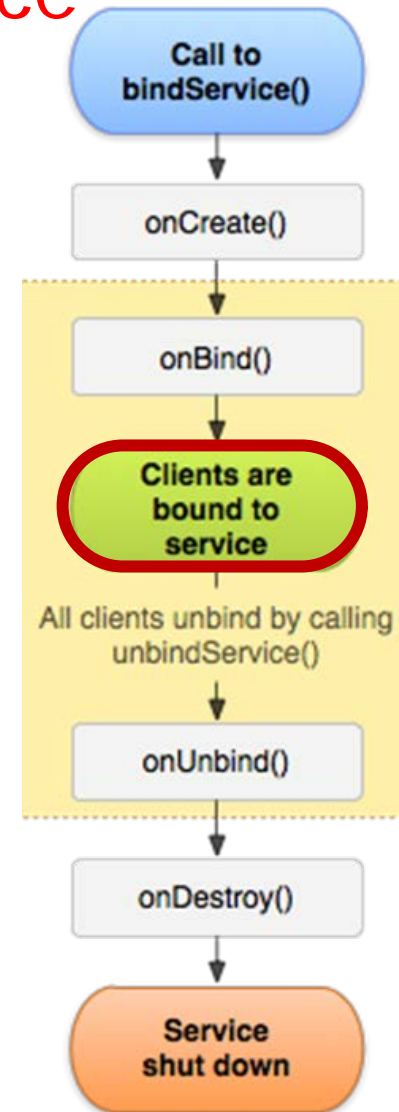
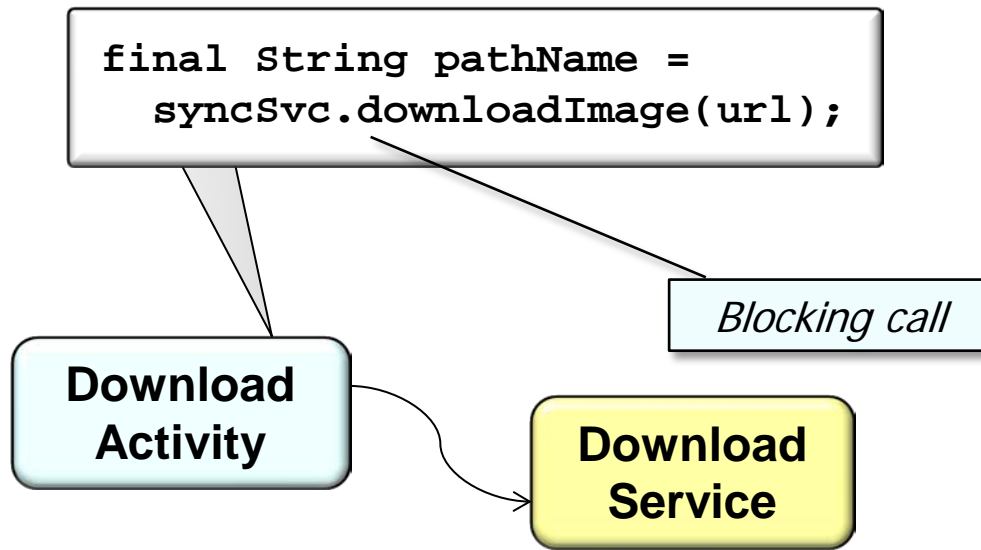
Download  
Activity

Download  
Service



# Interfacing with a Bound Service

- A Bound Service offers components an interface that clients can use to interact with the Service, e.g.:
  - Sending requests
  - Getting results &
  - Conversing across processes via IPC



# Interfacing with a Bound Service

- A Bound Service offers components an interface that clients can use to interact with the Service, e.g.:
  - Sending requests
  - Getting results &
  - Conversing across processes via IPC

```
IDownload.Stub binder = new IDownload.Stub()
{
    public String downloadImage(String url) {
        return downloadImageFromURL(url,
            DOWNLOADED_FILE_NAME_PREFIX);
    }
}
```

Download  
Activity

Download  
Service

*Run in a separate  
thread of control*



# Stopping a Bound Service

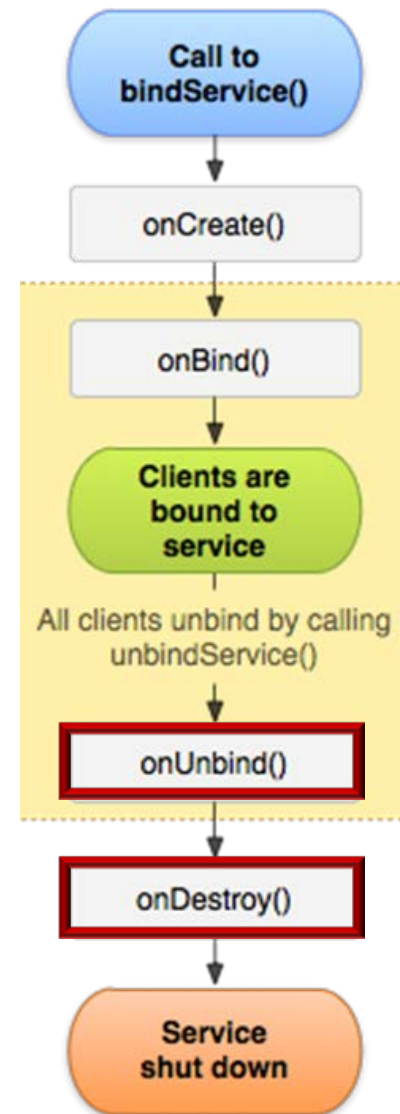
- When a Bound Service is launched, it has a lifecycle that depends on the component(s) that access it
  - i.e., it doesn't run in the background indefinitely

*When a client is done interacting with the service, it calls `unbindService()` to unbind*

*After there are no clients bound to the service it is destroyed*

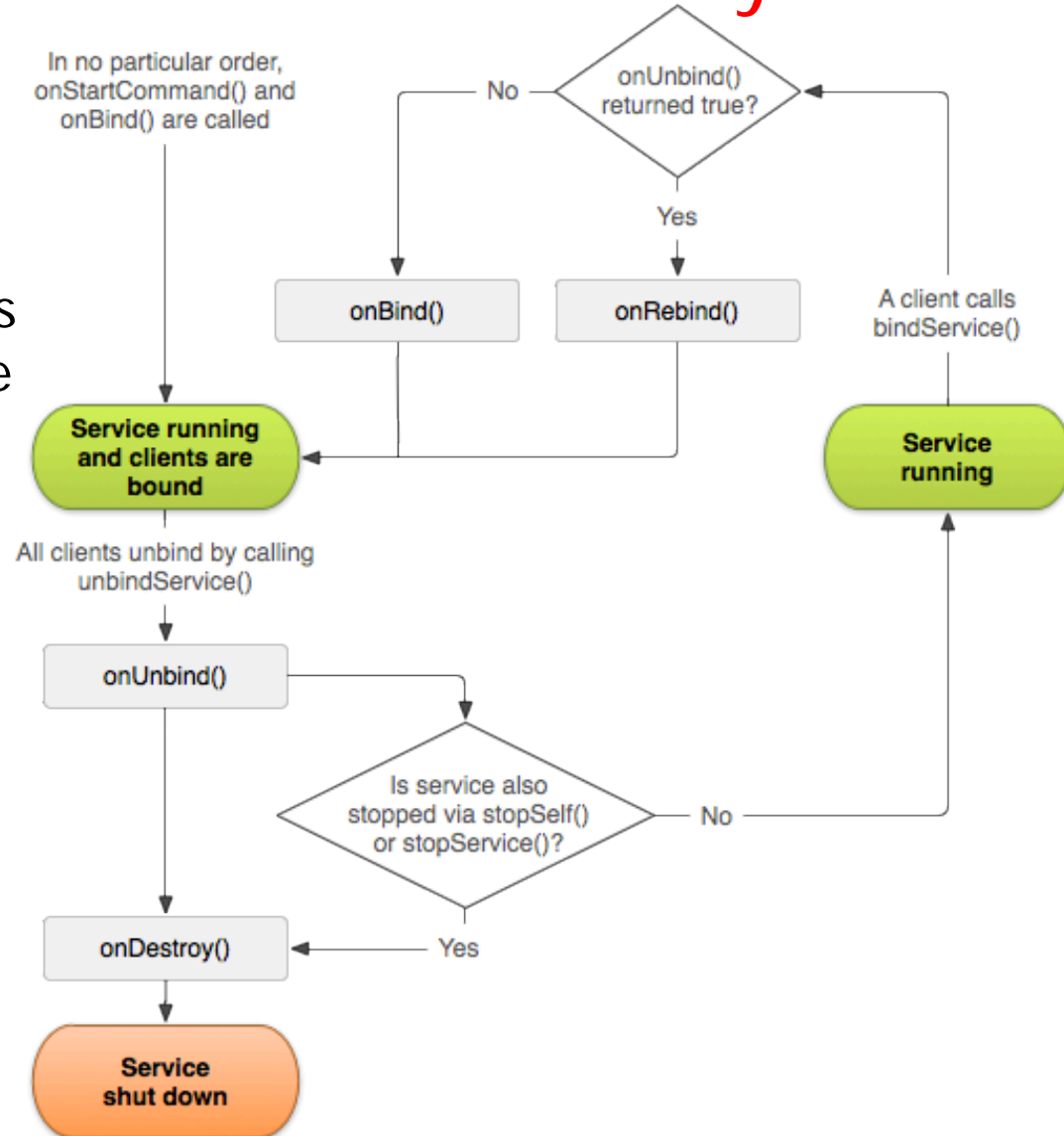
**Download Activity**

**Download Service**



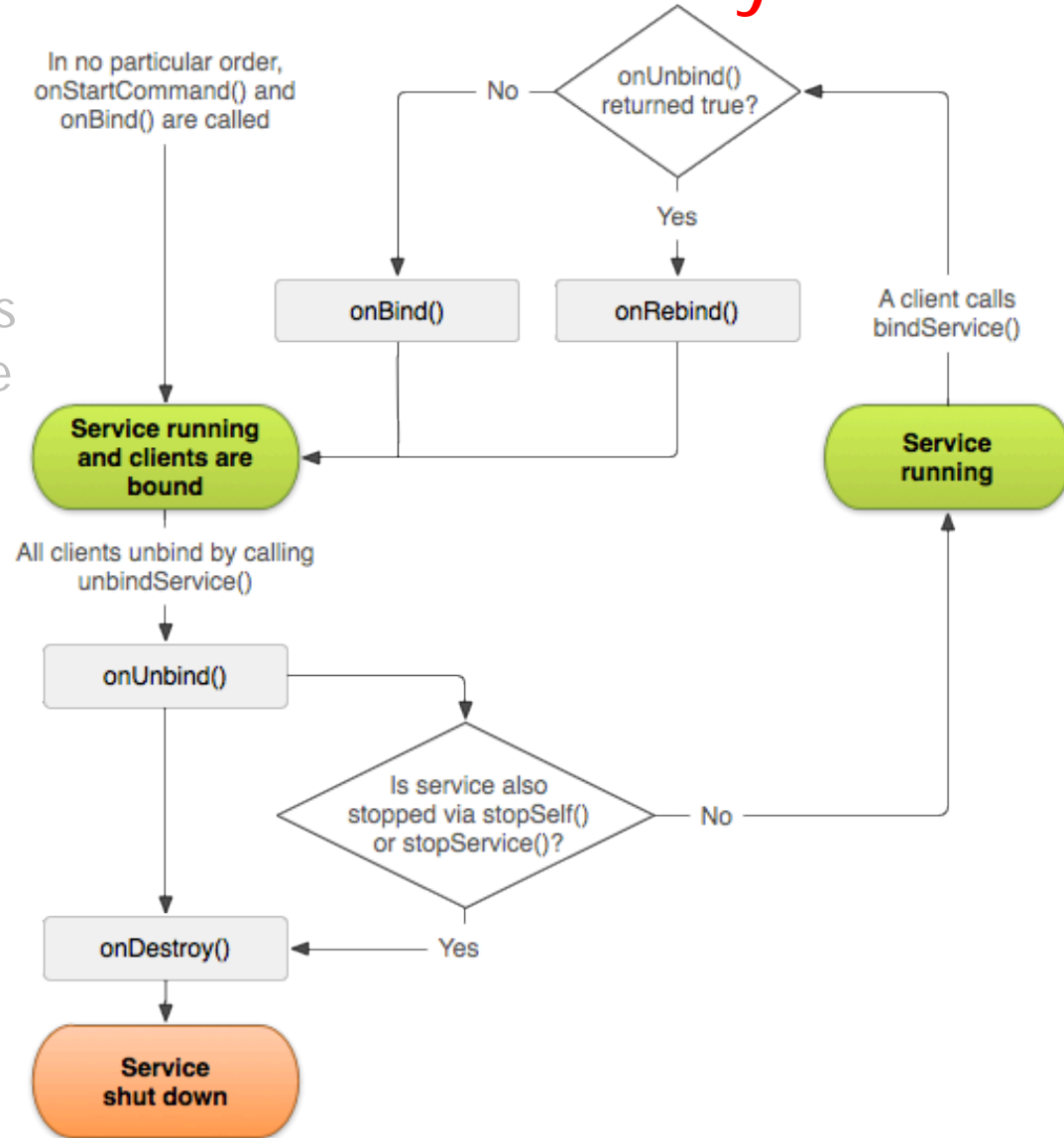
# Hybrid Started & Bound Service Lifecycles

- It is possible to define “hybrid” models that combine Bound & Started Services
- If a Bound Service implements `onStartCommand()` it won't be destroyed when it is unbound from all clients



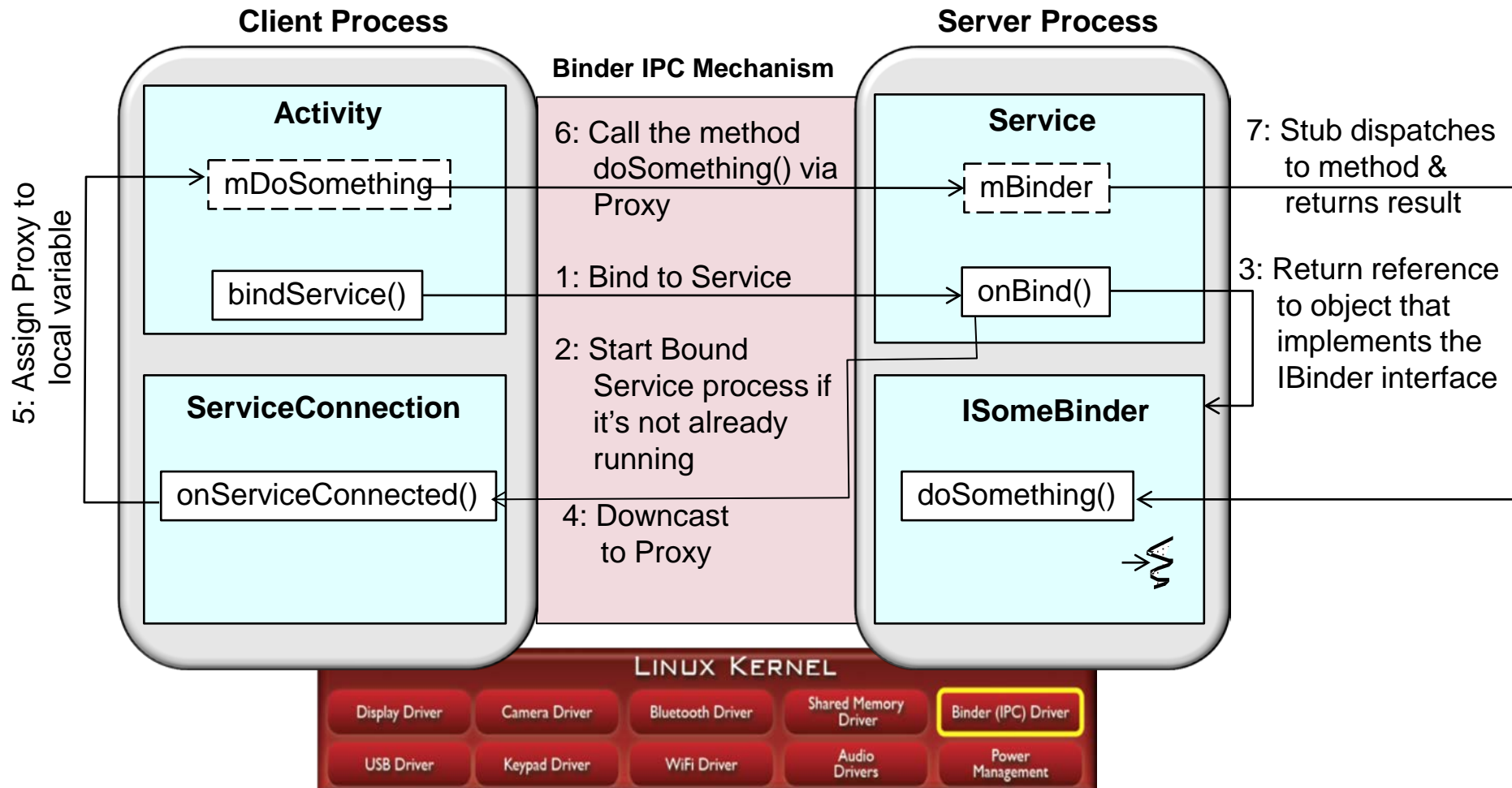
# Hybrid Started & Bound Service Lifecycles

- It is possible to define “hybrid” models that combine Bound & Started Services
  - If a Bound Service implements `onStartCommand()` it won't be destroyed when it is unbound from all clients
- If you return true when the system calls `onUnbind()` the `onRebind()` method will be called the next time a client binds to the Service
  - Instead of receiving a call to `onBind()`



# Protocol for Bound Service Interactions

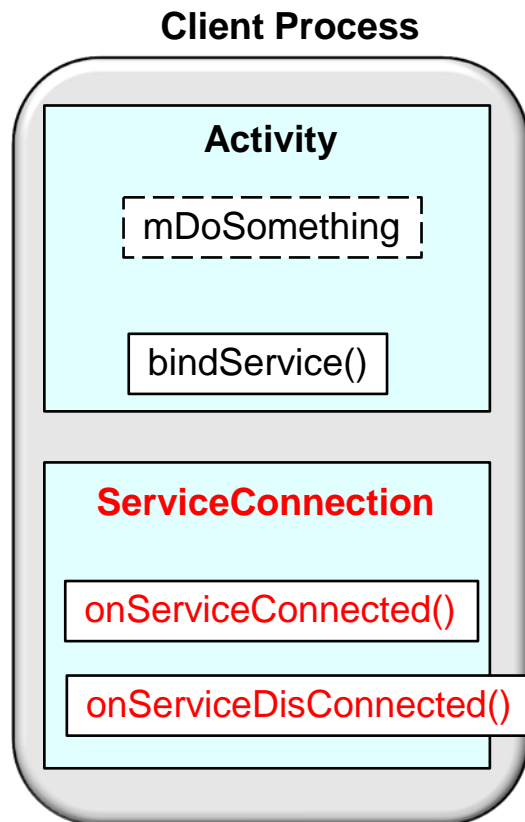
- A protocol is used to interact between Activities & Bound Services



Many patterns are involved here: *Broker, Proxy, Activator, Adapter*, etc.

# Client Bound Service Interactions

- To bind to a service from your client, you must perform the following steps:



1. Implement `ServiceConnection` & override its two callback methods:

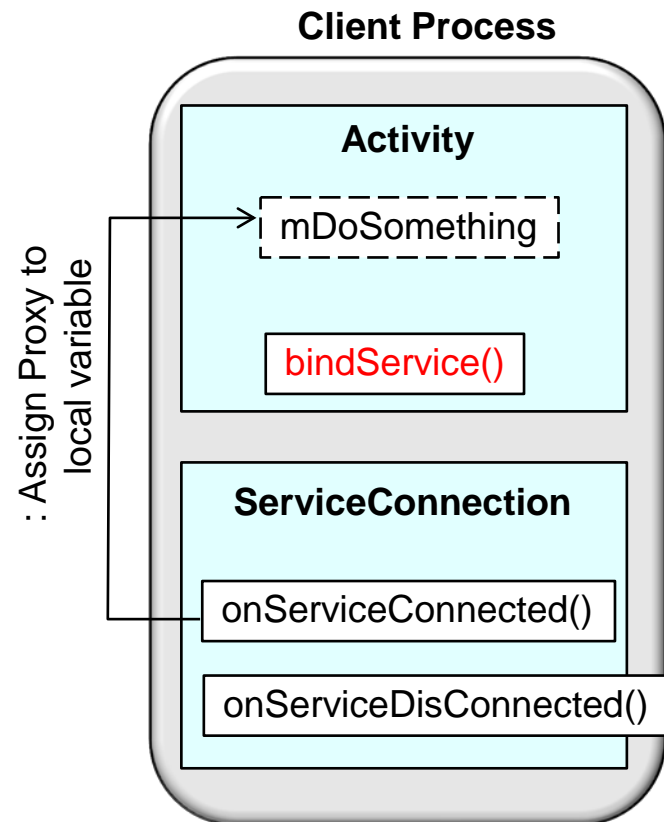
- `onServiceConnected()` – Android calls this to deliver the `IBinder` returned by the service's `onBind()` method
- `onServiceDisconnected()` – Android calls this when the connection to the service is unexpectedly lost
- e.g., when the service has crashed or has been killed (not called with client calls `unbindService()`)



# Client Bound Service Interactions

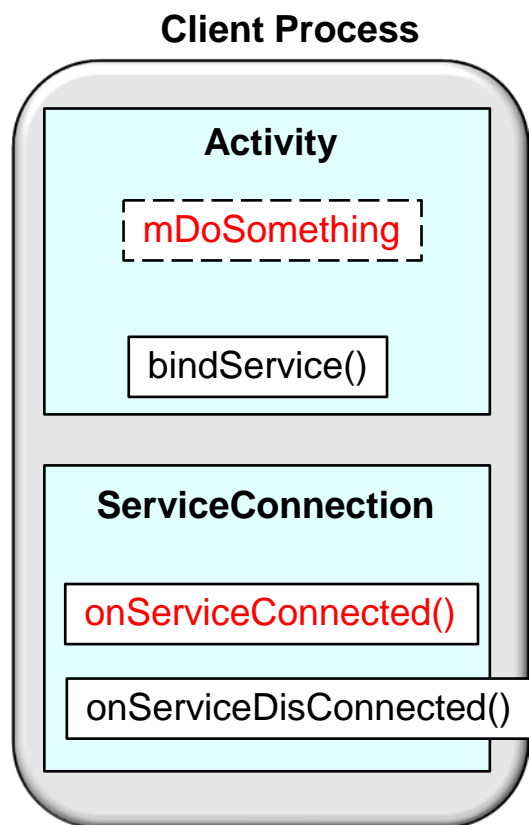
- To bind to a service from your client, you must perform the following steps:

1. Implement `ServiceConnection` & override its two callback methods
2. Call `bindService()`, passing the `ServiceConnection` implementation



# Client Bound Service Interactions

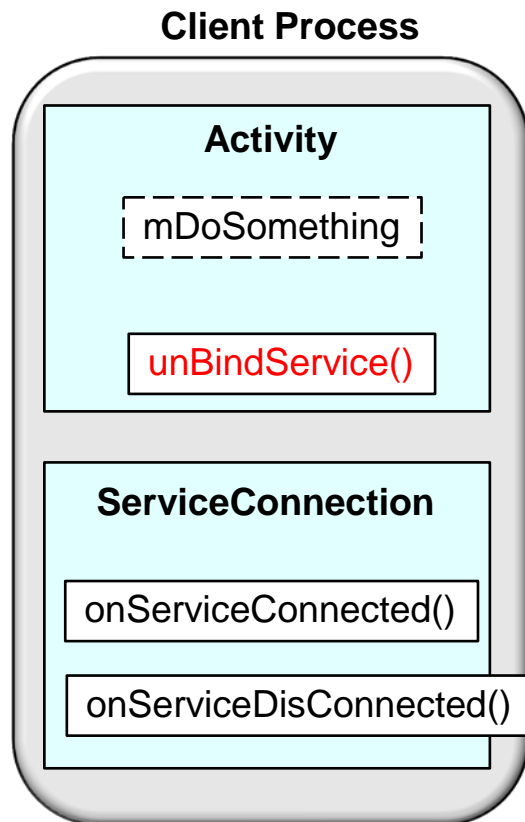
- To bind to a service from your client, you must perform the following steps:



1. Implement `ServiceConnection` & override its two callback methods
2. Call `bindService()`, passing the `ServiceConnection` implementation
3. When the system calls your `onServiceConnected()` callback method, you can begin making calls to the service, using the methods defined by the interface

# Client Bound Service Interactions

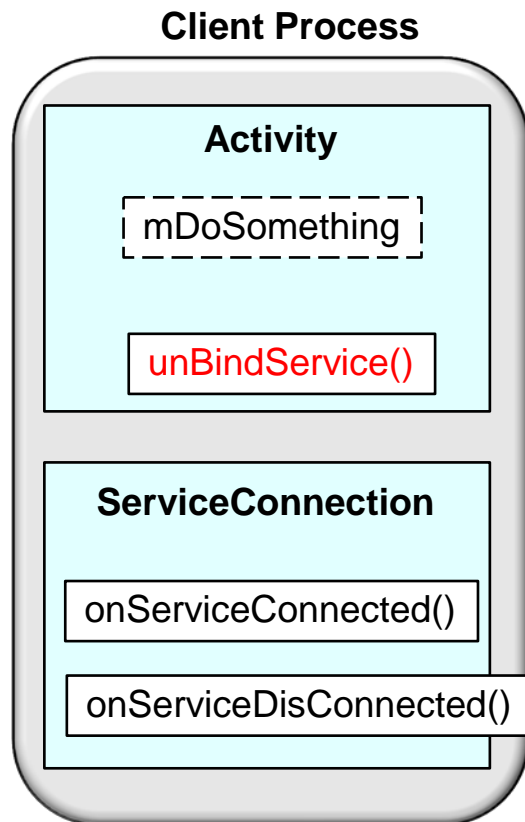
- To bind to a service from your client, you must perform the following steps:



1. Implement `ServiceConnection` & override its two callback methods
2. Call `bindService()`, passing the `ServiceConnection` implementation
3. When the system calls your `onServiceConnected()` callback method, you can begin making calls to the service, using the methods defined by the interface
4. To disconnect from the service, call `unbindService()`
  - When a client is destroyed, it is unbound from the Service automatically

# Client Bound Service Interactions

- To bind to a service from your client, you must perform the following steps:

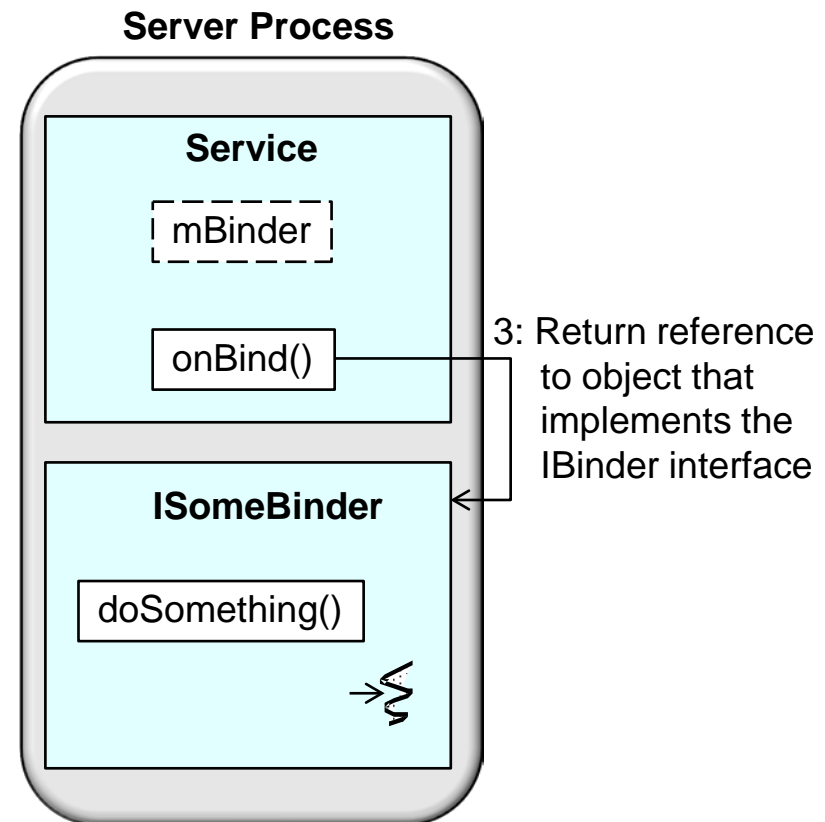


1. Implement `ServiceConnection` & override its two callback methods
2. Call `bindService()`, passing the `ServiceConnection` implementation
3. When the system calls your `onServiceConnected()` callback method, you can begin making calls to the service, using the methods defined by the interface
4. To disconnect from the service, call `unbindService()`

- Always unbind when you're done interacting with the service or when your activity pauses so that the service can shutdown while its not being used

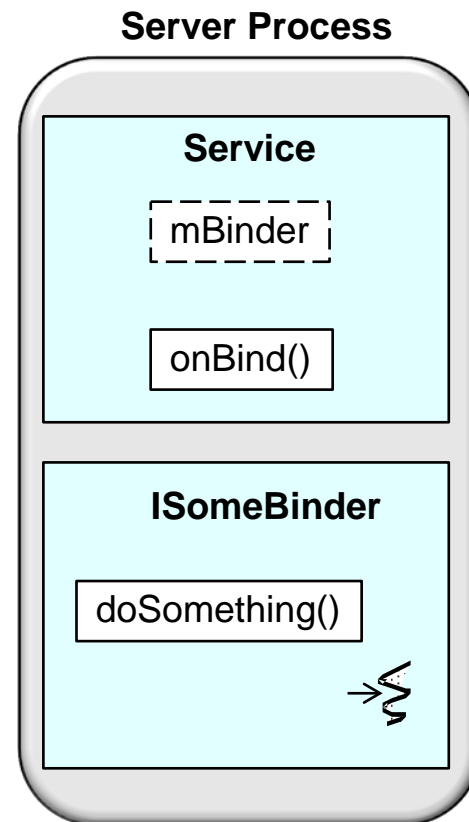
# Server Bound Service Interactions

- A Bound Service must perform the following steps when a client binds to it:
- When Android calls the Service's `onBind()` method it returns an `IBinder` for interacting with the Service



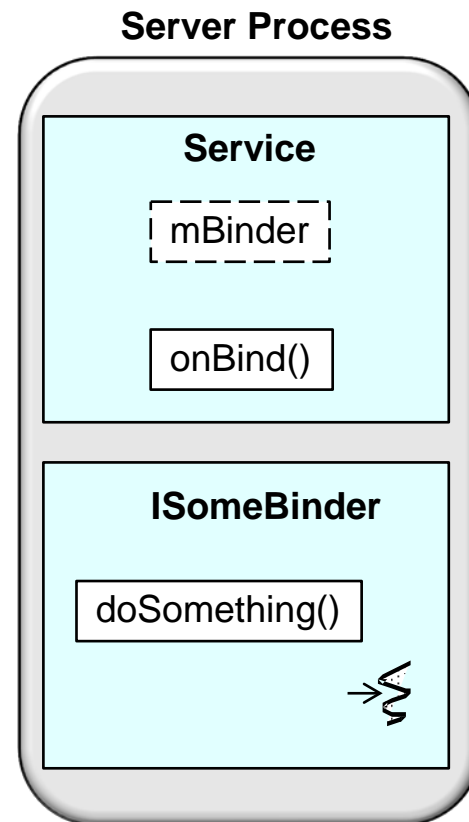
# Server Bound Service Interactions

- A Bound Service must perform the following steps when a client binds to it:
- When Android calls the Service's `onBind()` method it returns an `IBinder` for interacting with the Service
- The binding is asynchronous
  - i.e., `bindService()` returns immediately & does not return the `IBinder` to the client



# Server Bound Service Interactions

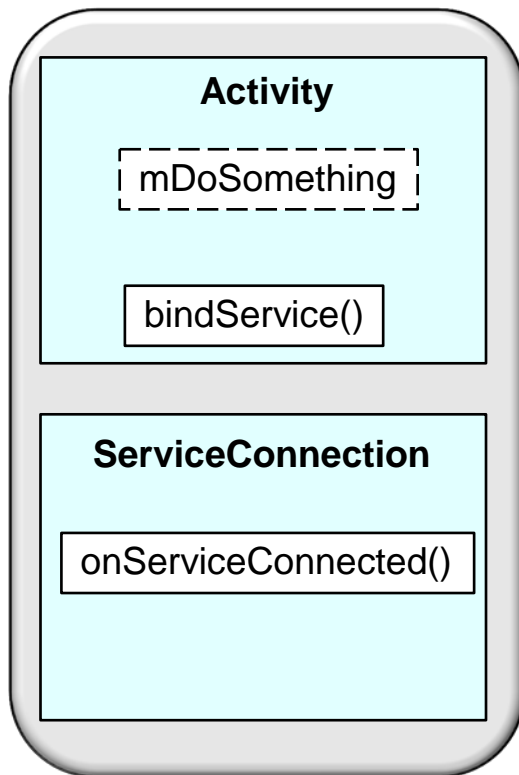
- A Bound Service must perform the following steps when a client binds to it:
- When Android calls the Service's `onBind()` method it returns an `IBinder` for interacting with the Service
  - The binding is asynchronous
  - To receive the `IBinder`, the client must create an instance of `ServiceConnection` & pass it to `bindService()`
  - `ServiceConnection` implements the `onServiceConnected()` callback method that Android invokes to deliver the `IBinder`



# Communicating with Bound Services

- When creating a Bound Service, you must provide an IBinder via an interface clients can use to interact with the Service via one of the following

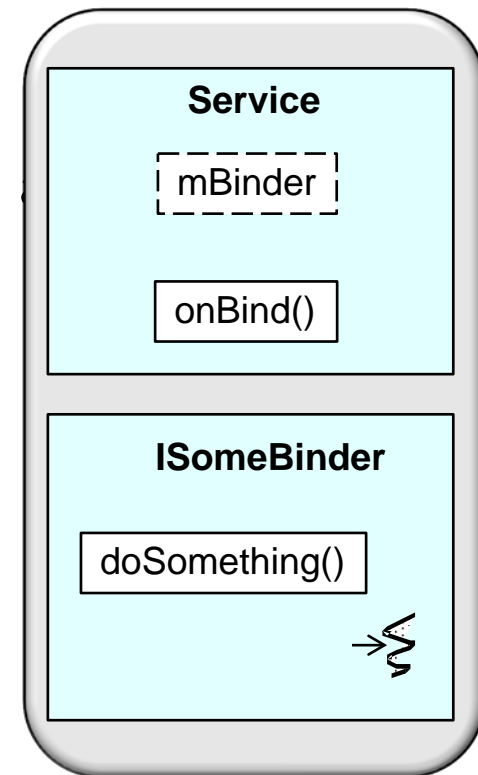
## Client Process



- Extending the Binder class**

- If your service runs in the same process as the client you can extend the Binder class & return from `onBind()`

## Server Process

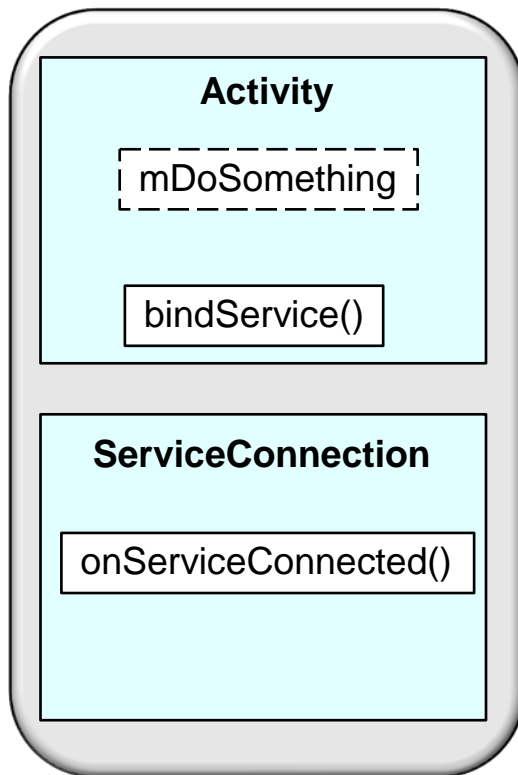




# Communicating with Bound Services

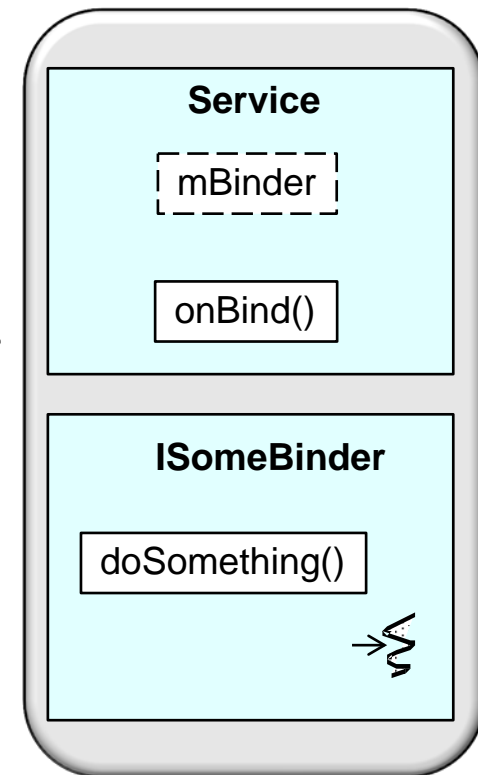
- When creating a Bound Service, you must provide an IBinder via an interface clients can use to interact with the Service via one of the following

## Client Process



- Extending the Binder class
- Using a Messenger**
  - Create an interface for the Service with a Messenger that allows the client to send commands to the Service across processes
  - Doesn't require thread-safe components

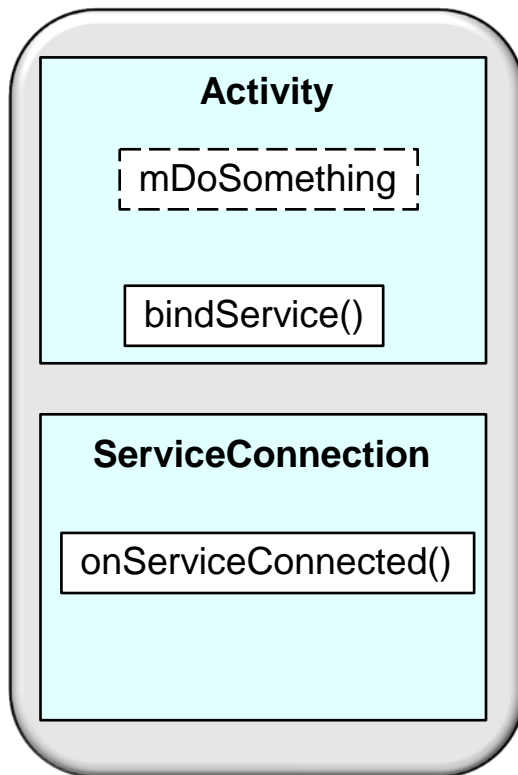
## Server Process



# Communicating with Bound Services

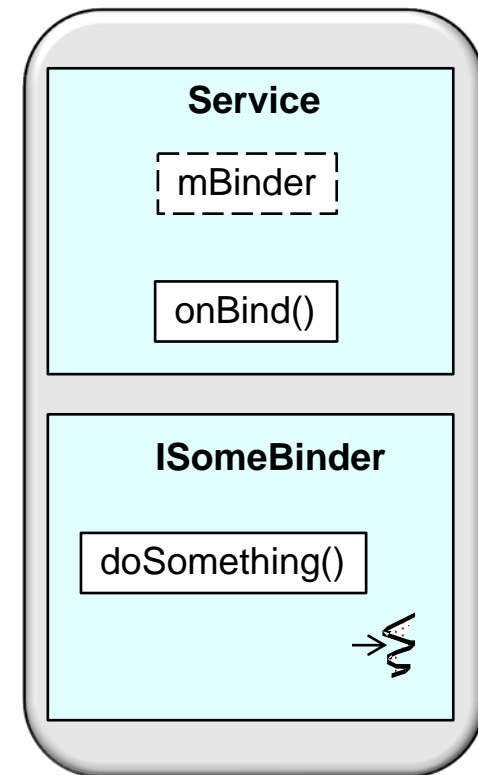
- When creating a Bound Service, you must provide an IBinder via an interface clients can use to interact with the Service via one of the following

## Client Process



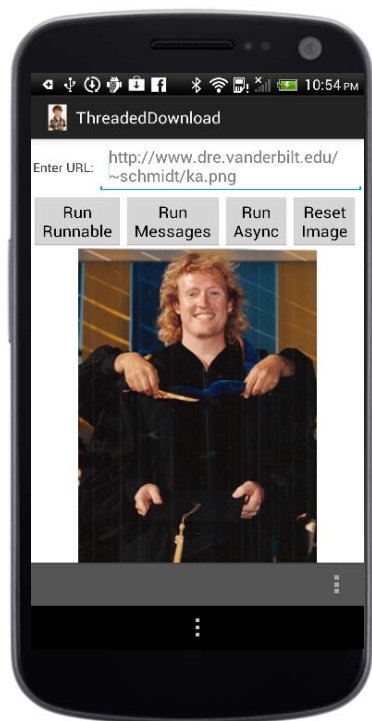
- Extending the Binder class
- Using a Messenger
- Using Android Interface Definition Language (AIDL)**
  - AIDL performs all the work to decompose objects into primitives that Linux can understand & marshal them across processes to perform IPC
  - Does require thread-safe components

## Server Process



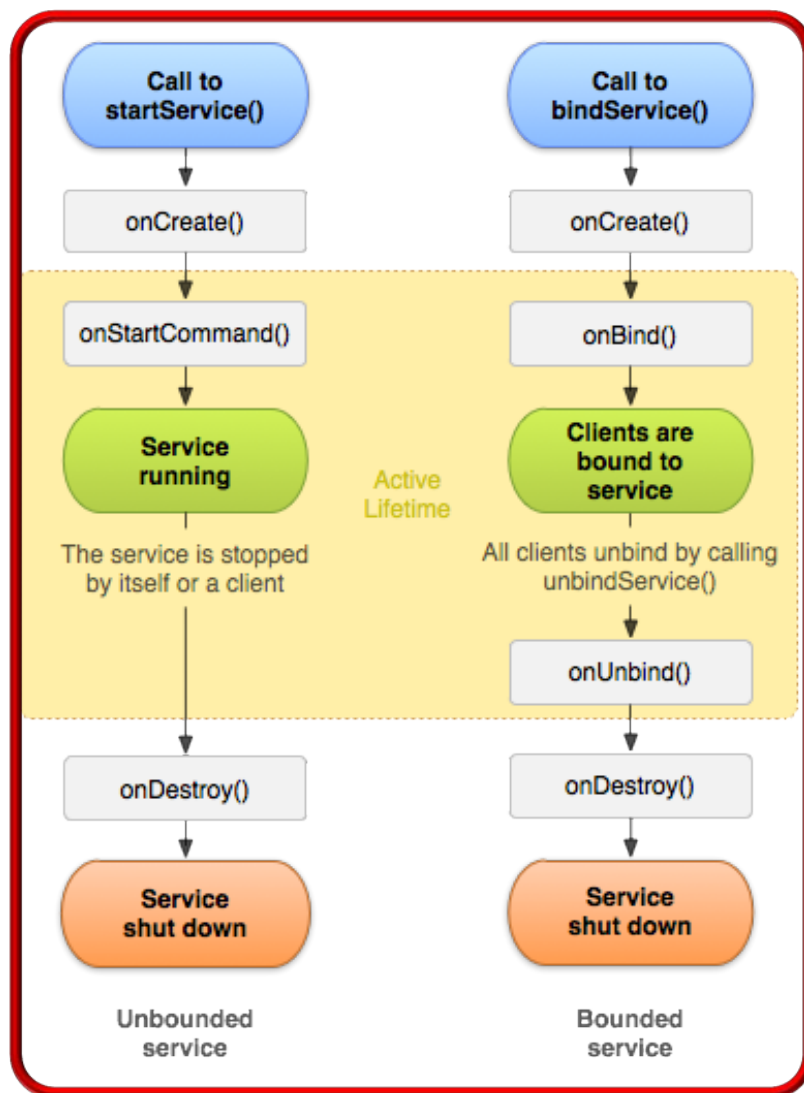
# Summary

- Apps can use Services to implement long-running operations in the background



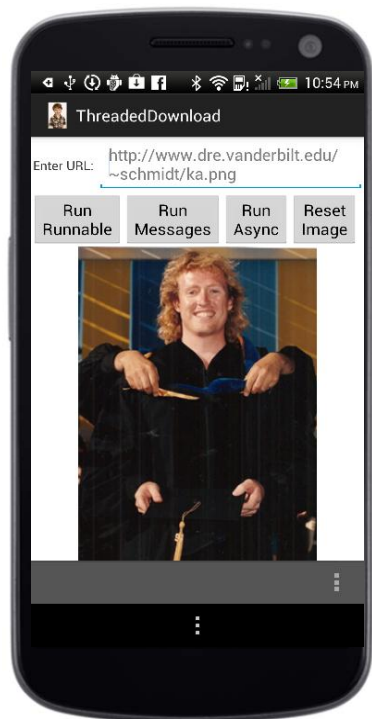
Download Activity

Download Service



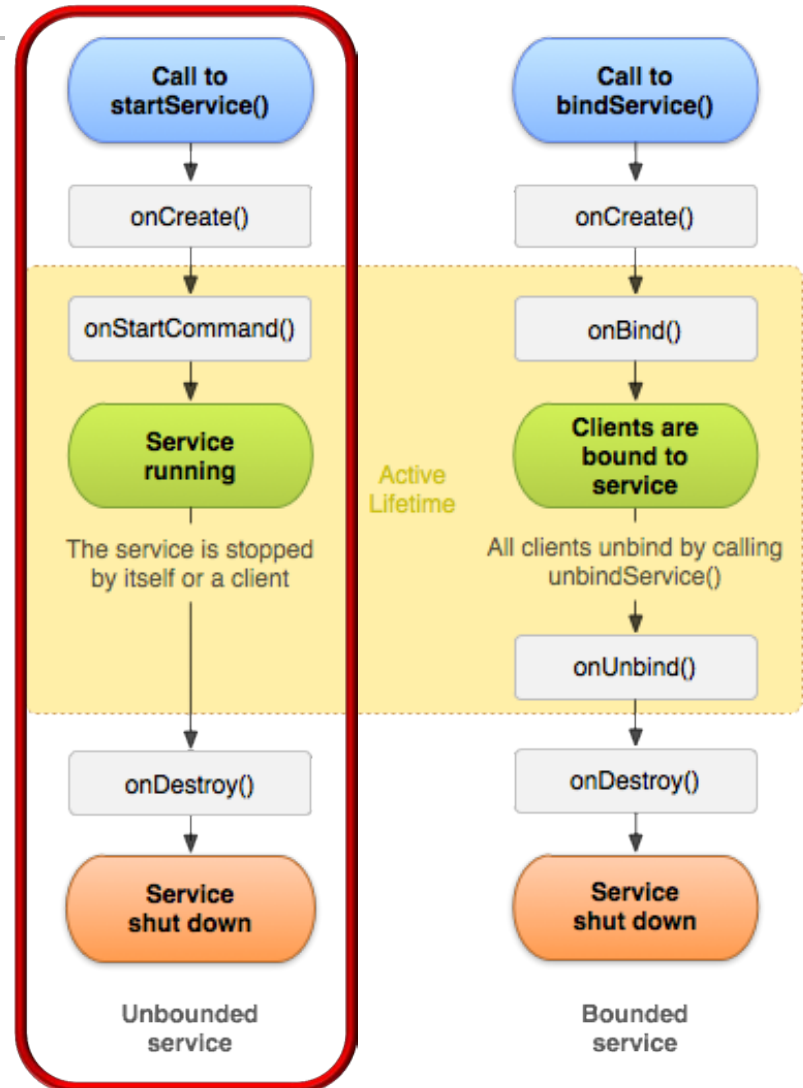
# Summary

- Apps can use Services to implement long-running operations in the background
- Started Services are simple to program



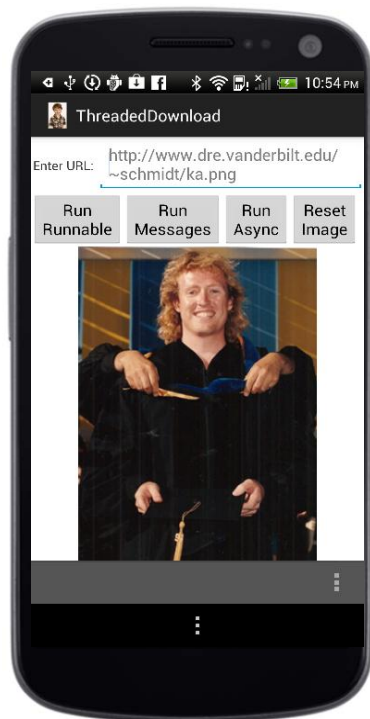
**Download Activity**

**Download Service**



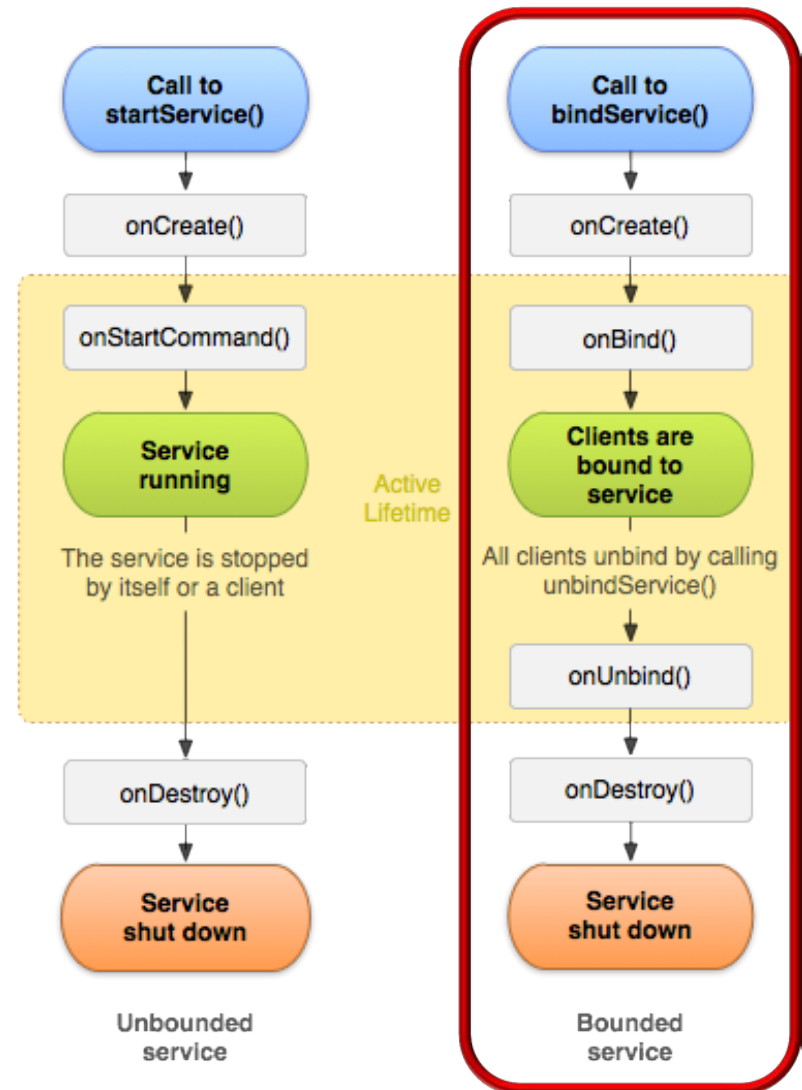
# Summary

- Apps can use Services to implement long-running operations in the background
- Started Services are simple to program
- Bound Services provide more powerful communication models



**Download Activity**

**Download Service**



# Summary

- Apps can use Services to implement long-running operations in the background
- Started Services are simple to program
- Bound Services provide more powerful communication models
- Examples of Android Bound Services:
  - *BluetoothHeadsetService*
    - Provides Bluetooth Headset & Handsfree as Service in Phone App
  - *MediaPlaybackService*
    - Provides "background" audio playback capabilities
  - *Exchange Email Services*
    - Manage email operations, such as sending messages

