Android & Java Concurrency: The Command Processor Pattern (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
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: Performing a Long-Duration Operation

Context

- Synchronous method calls in an Activity’s UI Thread can block client for extended periods
Challenge: Performing a Long-Duration Operation

Context

• Synchronous method calls in an Activity’s UI Thread can block client for extended periods
  
  • e.g., a two-way `downloadImage()` call would block the UI thread in the `DownloadImageActivity` while web server retrieves image

Activity makes a blocking two-way call in the UI Thread
Problems

- Android generates an “Application Not Responding” (ANR) dialog if an App doesn’t respond to user input within a short time (~3 seconds)

See developer.android.com/training/articles/perf-anr.html for more on ANRs
Challenge: Performing a Long-Duration Operation

Problems

- Android generates an “Application Not Responding” (ANR) dialog if an App doesn’t respond to user input within a short time (~3 seconds)
  - e.g., calling a (potentially) lengthy operation like `downloadImage()` in the UI Thread is problematic
Challenge: Performing a Long-Duration Operation

Solution

- Apply *Command Processor* pattern to download image in background Thread & display image in UI Thread

[Web Server](http://www.dre.vanderbilt.edu/~schmidt/PDF/CommandRevisited.pdf) has more info
Challenge: Performing a Long-Duration Operation

Solution

- Apply *Command Processor* pattern to download image in background Thread & display image in UI Thread
- Start a background Thread to perform long-duration download

Download image in a background thread
Challenge: Performing a Long-Duration Operation

Solution

- Apply *Command Processor* pattern to download image in background Thread & display image in UI Thread
  - Start a background Thread to perform long-duration download
  - Create a “display image” request as Runnable command object

Create a *DisplayImage Runnable*
Challenge: Performing a Long-Duration Operation

Solution

- Apply *Command Processor* pattern to download image in background Thread & display image in UI Thread
  - Start a background Thread to perform long-duration download
  - Create a “display image” request as Runnable command object
  - Pass Runnable from background Thread to the UI Thread

*Call runOnUiThread() to return DisplayImage Runnable to UI Thread*
Challenge: Performing a Long-Duration Operation

Solution

- Apply *Command Processor* pattern to download image in background Thread & display image in UI Thread
  - Start a background Thread to perform long-duration download
  - Create a “display image” request as Runnable command object
  - Pass Runnable from background Thread to the UI Thread
  - Dispatch the Runnable command to display image in UI Thread

*HaMeR framework calls run() method on DisplayImage Runnable in the UI Thread context*
Intent & Applicability of the Command Processor Pattern
Intent
• Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context
Intent

- Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context
- e.g., at a later point in time, in a different process or thread, etc.

The Gang-of-Four book contains a description of the similar *Command* pattern
Applicability
Applicability

- When there’s a need to decouple the decision of what piece of code should be executed from the decision of when/where this should happen.
**Command Processor**  
**POSA1 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/where this should happen
- e.g., specify, queue, & execute service requests on a component at different times or in different contexts
Command Processor  POSA1 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/where this should happen
- When there’s a need to ensure enhancements to the services offered by a component don’t break existing code
Command Processor POSA1 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/where this should happen
- When there’s a need to ensure enhancements to the services offered by a component don’t break existing code
- When additional capabilities must be implemented consistently for all requests to a service
Applicability

- When there’s a need to decouple the decision of what piece of code should be executed from the decision of when/where this should happen
- When there’s a need to ensure enhancements to the services offered by a component don’t break existing code
- When additional capabilities must be implemented consistently for all requests to a service
  - Examples of these capabilities include undo/redo & persistence
Structure of the Command Processor Pattern
Command Processor POSA1 Design Pattern

Structure & Participants

- **Command**
  - `execute()`

- **Concrete Command**
  - `parameters...`
  - `execute()`

- **Creator**
  - `<<creates>>`
  - passes command to **Executor**

- **Executor**
  - uses parameter from **Execution Context**

- **Execution Context**
Command Processor POSA1 Design Pattern

Structure & Participants

- Creator
- Executor
- Runnable
- Concrete Command
- Execution Context

- Creator <<creates>> Concrete Command
- Executor passes command to Creator
- Executor uses parameter from Concrete Command
- Concrete Command uses parameter from Execution Context
- Runnable execute()
Command Processor  

POSA1 Design Pattern

Structure & Participants

- **Command**:  
  - `execute()`

- **DisplayImage Runnable**

- **Creator**
  - **passes command to**
  - **<<creates>>**

- **Concrete Command**
  - `execute()`
  - `parameters...`

- **Executor**
  - **uses parameter from**

- **Execution Context**
Command Processor  POSA1 Design Pattern

Structure & Participants

- **Command**: `execute()`
- **Concrete Command**: `parameters...`, `execute()`
- **Executor**: `uses parameter from`
- **Execution Context**: `passes command to`
- **Creator**: `<<creates>>`
- **Download Runnable**
Command Processor POSA1 Design Pattern

Structure & Participants

- **Command**
  - `execute()`

- **Concrete Command**
  - parameters...
  - `execute()`
  - uses parameter from

- **Creator**
  - <<creates>>
  - passes command to

- **Executor**
  - HaMeR Framework
  - uses command to

- **Execution Context**
  - uses parameter from
Command Processor  POSA1 Design Pattern

Structure & Participants

- **Creator**: passes command to **Executor**.
- **Executor**: **Download ImageActivity**
- **Concrete Command**: uses parameter from **Execution Context**
  - **execute()**
  - **parameters...**

**Command**:
- **execute()**
Dynamics of the Command Processor Pattern
Command Processor  POSA1 Design Pattern

Dynamics

Creator

Concrete Command

Execution Context

Executor

<<create>>

set parameters

pass command

execute (execution context)

access context
Command Processor      POSA1 Design Pattern

Dynamics

- **Creator**
  - <<create>>
  - set parameters
  - pass command

- **Concrete Command**

- **Executor**
  - execute (execution context)
  - access context

**Creates a Concrete Command (Runnable) & Java thread & call start() on the thread**
Android & Java Concurrency: the Command Processor Pattern (Part 1)

Command Processor POSA1 Design Pattern

Dynamics

Creator

<<create>>

Concrete Command

set parameters

pass command

Executor

Execution Context

execute (execution context)

access context

HaMeR framework
Call run() to process the DisplayImageRunnable
Consequences of the Command Processor Pattern
Command Processor  POSA1 Design Pattern

Consequences
+ Context- & time-independent execution of the application logic
  * e.g., allows command execution in different thread or after time elapsed

```java
final ImageView iview = ... 

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable()
            { public void run() { iview.setImageBitmap(bitmap); } }
            runOnUIThread(displayImage);
        }
    };

    new Thread(downloadRunnable).start(); ...
```
Command Processor | POSA1 Design Pattern

**Consequences**

+ Context- & time-independent execution of the application logic
  - e.g., allows command execution in different thread or after time elapsed

```java
final ImageView iview = ...;

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable() {
                public void run() {
                    iview.setImageBitmap(bitmap);
                }
            };
            runOnUIThread(displayImage);
        }
    };
    new Thread(downloadRunnable).start();
    ... Create a runnable & execute it in a background thread
```
Command Processor        POSA1 Design Pattern

Consequences
+ Context- & time-independent execution of the application logic
  • e.g., allows command execution in different thread or after time elapsed

    final ImageView iview = ...  

    public void onDownload(View v) {  
        Runnable downloadRunnable = new Runnable() {  
            public void run() {  
                final Bitmap bitmap = downloadImage(URI);  
                final Runnable displayImage = new Runnable() {  
                    public void run() { iview.setImageBitmap(bitmap); } }  
                runOnUIThread(displayImage);  
            }  
        }  
        new Thread(downloadRunnable).start(); ...
## Command Processor - POSA1 Design Pattern

### Consequences

+ Context- & time-independent execution of the application logic
+ Client isn’t blocked for duration of command processing

```java
final ImageView iview = ... 

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable()
            { public void run() { iview.setImageBitmap(bitmap); } };
            runOnUIThread(displayImage);
        }
    };

    new Thread(downloadRunnable).start(); ...
```
Command Processor  

POSA1 Design Pattern

Consequences

+ Context- & time-independent execution of the application logic
+ Client isn’t blocked for duration of command processing

```java
final ImageView iview = ...;

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable()
            { public void run() { iview.setImageBitmap(bitmap); } };
            runOnUIThread(displayImage);
        }
    };
    new Thread(downloadRunnable).start();
    // This call doesn’t block the background thread
    new Thread(downloadRunnable).start();
}
```
Consequences

+ Context- & time-independent execution of the application logic
+ Client isn’t blocked for duration of command processing
+ Emulates *closures* in OO languages that don’t natively have this feature

```java
final ImageView iview = ... 

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable() {
                public void run() {
                    iview.setImageBitmap(bitmap);
                }
            };
            runOnUIThread(displayImage);
        }
    };

    new Thread(downloadRunnable).start(); ...
}
```

See [en.wikipedia.org/wiki/Closure_(computer_programming)](en.wikipedia.org/wiki/Closure_(computer_programming))
Command Processor      POSA1 Design Pattern

Consequences

+ Context- & time-independent execution of the application logic
+ Client isn’t blocked for duration of command processing
+ Emulates *closures* in OO languages that don’t natively have this feature

```
final ImageView iview = ... 

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable() {
                public void run() {
                    iview.setImageBitmap(bitmap);
                }
            };
            runOnUIThread(displayImage);
        }
    };
    new Thread(downloadRunnable).start(); ...
```
Consequences
Consequences

- Functionality is driven by the sender of a command rather than the receiver (cf. Active Object)

```java
final ImageView iview = ...

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable() {
                public void run() {
                    iview.setImageBitmap(bitmap);
                }
            };
            runOnUIThread(displayImage);
        }
    };

    new Thread(downloadRunnable).start(); ...
```
Consequences
– Functionality is driven by the sender of a command rather than the receiver (cf. Active Object)

```java
final ImageView iview = ...;

public void onDownload(View v) {
    Runnable downloadRunnable = new Runnable() {
        public void run() {
            final Bitmap bitmap = downloadImage(URI);
            final Runnable displayImage = new Runnable() {
                public void run() {
                    iview.setImageBitmap(bitmap);
                }
            };
            runOnUIThread(displayImage);
        }
    };
    new Thread(downloadRunnable).start(); ...
```
Consequences

– Functionality is driven by the sender of a command rather than the receiver (cf. Active Object)

– Supporting two-way operations requires additional programming effort

```java
final ImageView iview = ...

public void onDownload(View v) {
  Runnable downloadRunnable = new Runnable() {
    public void run() {
      final Bitmap bitmap = downloadImage(URI);
      final Runnable displayImage = new Runnable(){
        public void run() {
          iview.setImageBitmap(bitmap);
        }
      };
      runOnUIThread(displayImage);
    }
  };

  new Thread(downloadRunnable).start(); ...
```
Known Uses of the Command Processor Pattern
Command Processor  POSA1 Design Pattern

Known Uses

- Many UI toolkits
  - InterViews, ET++, MacApp, Swing, AWT, etc.
- Interpreters for command-line shells
- Java `Runnable` interface used by Thread & Executor framework
- Android HaMeR framework
- Android IntentService

```
public abstract class IntentService
    extends Service

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
Summary

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

• Requests & responses needn’t proceed in lock-step
Summary

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

- A command processor acts as a manager for the functionality of a component
Summary

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

- A command processor acts as a manager for the functionality of a component.
  - The component’s clients, as well as the component itself, are freed from organizing the execution of concrete service requests.
Summary

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

- A command processor acts as a manager for the functionality of a component
  - The component’s clients, as well as the component itself, are freed from organizing the execution of concrete service requests
  - This pattern results in a looser coupling between the two parties

*www.dre.vanderbilt.edu/~schmidt/PDF/CommandRevisited.pdf* has more info
End of Android & Java Concurrency: The Command Processor Pattern (Part 1)