Android & Java Concurrency: 
The Command Processor Pattern (Part 2)

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 how *Command Processor* is applied in Android & Java.
Implementation

- Define an interface for Command execution

```java
interface Runnable {
    void run();
}
```

## Implementation

- Define an interface for Command execution
  - e.g., define a method to execute the Command if processing can be localized to a single method

```java
interface Runnable {
    void run();
}
```

Implementation

• Define an interface for Command execution

• Add state & methods needed by Concrete Commands during their execution

```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);
            }
        };
    ...
```
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
  • e.g., implement a Command interface, extend a Command class, etc.

```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);
        }
    };
    runOnUiThread(downloadRunnable);
    ...}
```
Android & Java Concurrency: the Command Processor Pattern (Part 2)

Command Processor

**Implementation**

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
  - Often use patterns like Abstract Factory or Factory Method

---

```java
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);
        }
    };
```

---
Implementation

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
  - Often use patterns like Abstract Factory or Factory Method
- Determine a mechanism for passing Command to Executor

```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);
            }
        };
    ...
}
```

Pass the Runnable Command to the HaMeR framework

developer.android.com/reference/android/app/Activity.html#runOnUiThread(java.lang.Runnable)
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
  • Provide the run-time environment for processing the Command object

```
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);
            }
        };
}
```

Access context state
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
• Implement the Executor
  • Receive Command from Creator

```java
public class Handler {
    ...

    public final boolean post(Runnable r) {
        return sendMessageDelayed(getPostMessage(r), 0);
    }
}
```

`runOnUiThread()` uses the `Handler.post()` method to execute a command in the UI Thread
**Implementation**
- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
- Receive Command from Creator

```java
public class Handler {
    ...
    public final boolean post(Runnable r) {
        return sendMessageDelayed(getPostMessage(r), 0);
    }

    private final Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
    }
}
```

Internally, Runnable commands are encapsulated within Messages
Implementation

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
- Receive Command from Creator

```java
public class Handler {
    ...  
    public final boolean post(Runnable r) {
        return sendMessageDelayed (getPostMessage(r), 0);  
    }

    private final Message getPostMessage(Runnable r) {  
        Message m = Message.obtain();  
        m.callback = r;  
        return m;  
    }
}
```
**Implementation**

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator

```java
public class Handler {
    ...
    public final boolean post(Runnable r) {
        return sendMessageDelayed(getPostMessage(r), 0);
    }
}
```
## Implementation

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator
  - Enqueue Command for processing

```java
public class Handler {
    ...
    public final boolean sendMessageAtTime(
        Message msg,
        long uptimeMillis) {
        ...
        MessageQueue queue = mQueue;
        ...
        msg.target = this;
        queue.enqueueMessage(
            msg, uptimeMillis);
        ...
    }
}```
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
• Implement the Executor
  • Receive Command from Creator
  • Enqueue Command for processing

```java
public class Handler {
    ...
    public final boolean sendMessageAtTime(
        Message msg,
        long uptimeMillis)
    {
        ...
        MessageQueue queue = mQueue;
        ...
        msg.target = this;
        queue.enqueueMessage(msg, uptimeMillis);
        ...
```
Command Processor      POSA1 Design Pattern

**Implementation**

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator
  - Enqueue Command for processing

```java
public class Handler {
    ...
    public final boolean sendMessageAtTime(
        Message msg,
        long uptimeMillis) {
        ...
        MessageQueue queue = mQueue;
        ...
        msg.target = this;
        queue.enqueueMessage(
            msg, uptimeMillis);
        ...
    }
}
```
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
• Implement the Executor
  • Receive Command from Creator
  • Enqueue Command for processing

```java
public class MessageQueue {
    ...
    final boolean enqueueMessage (Message msg, long when) {
        final boolean needWake;
        synchronized (this) {
            Message p = mMMessages;
            ...
            if (needWake) {
                nativeWake(mPtr);
            }
            ...
        }
    }
}
```

Determine where to enqueue the Message in the doubly-linked list

frameworks/base/core/java/android/os/MessageQueue.java has the source
Command Processor  POSA1 Design Pattern

Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
• Implement the Executor
  • Receive Command from Creator
  • Enqueue Command for processing

```java
public class MessageQueue {

    final boolean enqueueMessage (Message msg,
                                      long when) {

        final boolean needWake;
        synchronized (this) {
            Message p = mMMessages;
            ...
            if (needWake) {
                nativeWake(mPtr);
                ...
        }
}

frameworks/base/libs/utils/Looper.cpp has the source code
Implementation

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator
  - Enqueue Command for processing
  - Dequeue Command

```java
public class Looper {
    ...
    final MessageQueue mQueue;

    public static void loop() {
        Looper me = myLooper();
        ...
        MessageQueue queue = me.mQueue;
        for (;;) {
            Message msg = queue.next();
            ...
            msg.target.
            dispatchMessage(msg);
            ...
    }
```

frameworks/base/core/java/android/os/Looper.java has the source
## Implementation

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator
  - Enqueue Command for processing
  - Dequeue Command

```java
public class Looper {
    ...
    final MessageQueue mQueue;

    public static void loop() {
        Looper me = myLooper();
        ...
        MessageQueue queue = me.mQueue;

        for (;;) {
            Message msg = queue.next();
            ...
            msg.target.
                dispatchMessage(msg);
            ...
        }
    }
}
```
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
• Implement the Executor
  • Receive Command from Creator
  • Enqueue Command for processing
  • Dequeue Command

```java
public classLooper {
  ...
  final MessageQueue mQueue;

  public static void loop() {
    Looper me = myLooper();
    ...
    MessageQueue queue = me.mQueue;

    for (;;) {
      Message msg = queue.next();
      ...
      msg.target.
          dispatchMessage(msg);
      ...
  }
```

Command Processor

POSA1 Design Pattern
Implementation

• Define an interface for Command execution
• Add state & methods needed by Concrete Commands during their execution
• Define & implement the Creator
• Define the Execution Context
• Implement the Executor
  • Receive Command from Creator
  • Enqueue Command for processing
  • Dequeue Command
  • Execute Command in the Context

```java
public class Handler {
    ...
    public void dispatchMessage(Message msg) {
        if (msg.callback != null)
            handleCallback(msg);
    ...
```
Command Processor | POSA1 Design Pattern
--- | ---
**Implementation**
- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator
  - Enqueue Command for processing
  - Dequeue Command
  - Execute Command in the Context

```java
public class Handler {
    ...
    public void dispatchMessage (Message msg) {
        if (msg.callback != null)
            handleCallback(msg);
    ...
```
**Command Processor**  
POS A1 Design Pattern

**Implementation**

- Define an interface for Command execution
- Add state & methods needed by Concrete Commands during their execution
- Define & implement the Creator
- Define the Execution Context
- Implement the Executor
  - Receive Command from Creator
  - Enqueue Command for processing
  - Dequeue Command
  - Execute Command in the Context

```java
public class Handler {
    ...
    public void dispatchMessage (Message msg) {
        if (msg.callback != null) {
            handleCallback(msg);
        }
        ...
```
### Command Processor

<table>
<thead>
<tr>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Define an interface for Command execution</td>
</tr>
<tr>
<td>• Add state &amp; methods needed by Concrete Commands during their execution</td>
</tr>
<tr>
<td>• Define &amp; implement the Creator</td>
</tr>
<tr>
<td>• Define the Execution Context</td>
</tr>
<tr>
<td>• Implement the Executor</td>
</tr>
<tr>
<td>• Receive Command from Creator</td>
</tr>
<tr>
<td>• Enqueue Command for processing</td>
</tr>
<tr>
<td>• Dequeue Command</td>
</tr>
<tr>
<td>• Execute Command in the Context</td>
</tr>
</tbody>
</table>

```java
public class Handler {
    ...
    public void dispatchMessage(Message msg) {
        if (msg.callback != null)
            handleCallback(msg);
        ...

    private final void handleCallback(Message message) {
        message.callback.run();
    }
}
```

*This particular run() callback executes in the UI Thread*
Applying Command Processor in Android & Java
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the Command Processor pattern
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the Command Processor pattern

See docs.oracle.com/javase/tutorial/essential/concurrency/executors.html
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the Command Processor pattern
- Client passes execute() a Runnable task

```
Task" is the Java Executor framework term for a "command"
```
Apply the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the Command Processor pattern
- Client passes execute() a Runnable task
- Task executed using a Worker Thread

See libcore/luni/src/main/java/java/util/concurrent/ThreadPoolExecutor.java
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the *Command Processor* pattern
  - Client passes execute() a Runnable task
  - Task executed using a Worker Thread
  - Blocking queue passes tasks from clients to worker Threads
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the Command Processor pattern
  - Client passes execute() a Runnable task
  - Task executed using a Worker Thread
  - Blocking queue passes tasks from clients to worker Threads

- Queues can be strategized
  - Direct handoff
    - Pros – Avoids deadlock from dependencies
    - Cons – Unlimited Threads

```java
public class ThreadPoolExecutor {
    public void execute(Runnable task) {
        // Task is executed using a Worker Thread
    }
}
```
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the **Command Processor** pattern
- Client passes execute() a Runnable task
- Task executed using a Worker Thread
- Blocking queue passes tasks from clients to worker Threads
- Queues can be strategized
  - Direct handoff
  - Unbounded queues
- Pros – Smooth out bursty requests
- Cons – Unlimited resources
Applying the Command Processor pattern in Android & Java

- Java’s ThreadPoolExecutor implements the *Command Processor* pattern
  - Client passes execute() a Runnable task
  - Task executed using a Worker Thread
  - Blocking queue passes tasks from clients to worker Threads

- Queues can be strategized
  - Direct handoff
  - Unbounded queues
  - Bounded queues

- Pros – Limit resources
- Cons – Hard to tune

<Java Class>:
```
<ThreadPoolExecutor
  ThreadPollExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>)
  ThreadPollExecutor(int, int, long, TimeUnit, BlockingQueue<Runnable>, ThreadFactory)
  execute(Runnable): void
  shutdown(): void
  shutdownNow():
  isShutdown(): boolean
  isTerminating(): boolean
  isTerminated(): boolean
  awaitTermination(long, TimeUnit): boolean
  setThreadFactory(ThreadFactory): void
  getThreadFactory():
  setRejectedExecutionHandler(RejectedExecutionHandler): void
  getRejectedExecutionHandler():
  setCorePoolSize(int): void
  getCorePoolSize(): int
  prestartCoreThread(): boolean
  prestartAllCoreThreads(): int
  allowsCoreThread-timeout(): boolean
  allowCoreThread-timeout(boolean): void
  setMaximumPoolSize(int): void
  getMaximumPoolSize(): int
  setKeepAliveTime(long, TimeUnit): void
  getKeepAliveTime(TimeUnit): long
  getQueue():
  remove(Runnable): boolean
  purge(): void
  getPoolSize(): int
  getActiveCount(): int
  getLargestPoolSize(): int
  getTaskCount(): long
  getCompletedTaskCount(): long
  toString()>
```
The Android HaMeR framework implements the *Command Processor* pattern & processes runnable commands in separate Java threads.

Other patterns are involved here: *Thread-Specific Storage, Result Callback, etc.*
End of Android & Java Concurrency: The Command Processor Pattern (Part 2)