Android & Java Concurrency: the Active Object Pattern (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Module

- Understand the *Active Object* pattern

See [en.wikipedia.org/wiki/Active_object](en.wikipedia.org/wiki/Active_object)
Challenge: Invoking Methods in Another Thread

Context

- Android clients that access objects running in separate threads of control

A “client” is any Android code that invokes a object’s method
Challenge: Invoking Methods in Another Thread

Context
- Android clients that access objects running in separate threads of control, e.g.,
  - A background thread invoking sendMessage() on a Handler associated with the UI thread

```java
Handler.sendMessage(msg);

void handleMessage(Message msg) {
    switch (msg.what) {
    case SET_PROGRESS_BAR_VISIBILITY:
        progress.setVisibility((Integer) msg.obj);
        break;
    ...
```
Challenge: Invoking Methods in Another Thread

Context

- Android clients that access objects running in separate threads of control, e.g.,
  - A background thread invoking `sendMessage()` on a Handler associated with the UI thread
  - More generally, any threads that interact via Handlers/Messages

```java
void handleMessage(Message msg)
{
  ...
}
```
Challenge: Invoking Methods in Another Thread

Problems

• Leveraging the parallelism available on a hardware & software platform (relatively) transparently

```java
Handler.sendMessage(msg);
```

```java
void handleMessage(Message msg) {
    ...
}
```
Challenge: Invoking Methods in Another Thread

Problems

- Leveraging the parallelism available on a hardware & software platform (relatively) transparently
- Ensuring that processing-intensive methods invoked on an object do not block the entire process

```java
Handler.sendMessage(msg);

void handleMessage(Message msg) {
    ...
}
```
Challenge: Invoking Methods in Another Thread

Problems

- Leveraging the parallelism available on a hardware & software platform (relatively) transparently
- Ensuring that processing-intensive methods invoked on an object do not block the entire process
- Simplifying programming of synchronized access to shared objects
Challenge: Invoking Methods in Another Thread

Solution

- Apply the *Active Object* pattern to decouple method invocation on the object from method execution.
Challenge: Invoking Methods in Another Thread

Solution

• Apply the Active Object pattern to decouple method invocation on the object from method execution
  • Method invocation should occur in the client’s thread of control
Challenge: Invoking Methods in Another Thread

Solution

- Apply the *Active Object* pattern to decouple method invocation on the object from method execution
  - Method invocation should occur in the client’s thread of control, whereas method execution should occur in a separate thread
Challenge: Invoking Methods in Another Thread

Solution

- Apply the Active Object pattern to decouple method invocation on the object from method execution
  - Method invocation should occur in the client’s thread of control, whereas method execution should occur in a separate thread
  - The client should appear to invoke an ordinary method
Challenge: Invoking Methods in Another Thread

Solution

- Apply the *Active Object* pattern to decouple method invocation on the object from method execution
  - Method invocation should occur in the client’s thread of control, whereas method execution should occur in a separate thread
  - The client should appear to invoke an ordinary method
    - i.e., the client (& the method execution itself) shouldn’t need to manipulate synchronization mechanisms explicitly
Intent & Applicability of the Active Object Pattern
Intent

- Define service requests on components as the units of concurrency & run service requests on a component in different thread(s) from the requesting client thread.
Intent

- Define service requests on components as the units of concurrency & run service requests on a component in different thread(s) from the requesting client thread
- Enable the client & component to interact asynchronously to produce & consume service results
Active Object

Client thread

Handler interface

method_1

method_2

Client

Handler implementation

POS A2 Concurrency

Scheduler

Active object thread

Handler implementation

method_2

method_1

Applicability
Active Object

POS A2 Concurrency

Applicability

- When an object’s interface methods should define its concurrency boundaries
Active Object & POSA2 Concurrency

**Applicability**

- When an object’s interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
Active Object

Applicability
- When an object’s interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object’s methods may block for a long time during their execution
Active Object

Applicability

- When an object’s interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object’s methods may block for a long time during their execution
- When multiple client method requests can run concurrently on an object
Active Object

Applicability

- When an object’s interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object’s methods may block for a long time during their execution
- When multiple client method requests can run concurrently on an object
- When method invocation order might not match method execution order
Active Object

Applicability
- When an object’s interface methods should define its concurrency boundaries
- When objects should be responsible for method synchronization & scheduling transparently, without requiring explicit client intervention
- When an object’s methods may block for a long time during their execution
- When multiple client method requests can run concurrently on an object
- When method invocation order might not match method execution order

Note the similarities between Active Object & Monitor Object wrt applicability
Structure of the Active Object Pattern
Active Object

Structure & Participants

Proxy
  method_1()
  method_N()

Handler.sendMessage()

Future

Scheduler
  dispatch()
  insert()

Activation List
  insert()
  remove()

Servant
  method_1()
  method_N()

Client

Concrete MethodRequest 1

Concrete MethodRequest N
Active Object

POS A2 Concurrency

Structure & Participants

- **Proxy**
  - method_1()
  - method_N()

- **Scheduler**
  - dispatch()
  - insert()

- **Activation List**
  - insert()
  - remove()

- **Future**

- **MethodRequest**
  - can_run()
  - call()

- **Servant**
  - method_1()
  - method_N()

- **Message**

- **Client**

- **Concrete MethodRequest 1**

- **Concrete MethodRequest N**
Active Object

Structure & Participants

Proxy
- method_1()
- method_N()

Scheduler
- dispatch()
- insert()

MessageQueue
- Activation List
  - insert()
  - remove()

Future
- <<obtain result from>>

MethodRequest
- can_run()
- call()

Servant
- method_1()
- method_N()

Client

Concrete MethodRequest 1

Concrete MethodRequest N
### Active Object & POSA2 Concurrency

#### Structure & Participants

- **Client**
- **Proxy**
  - `method_1()`
  - `method_N()`
- **Future**
- **MethodRequest**
  - `can_run()`
  - `call()`
- **Scheduler**
  - `dispatch()`
  - `insert()`
- **Activation List**
  - `insert()`
  - `remove()`
- **Concrete MethodRequest 1**
- **Concrete MethodRequest N**
- **Looper**
- **Servant**
  - `method_1()`
  - `method_N()`
Structure & Participants

- **Proxy**
  - Method: `method_1()`, `method_N()`
- **Scheduler**
  - Method: `dispatch()`, `insert()`
- **Activation List**
  - Methods: `insert()`, `remove()`
- **Future**
  - Methods: `can_run()`, `call()`
  - <<obtain result from>>
  - <<write to>>
- **Servant**
  - Methods: `method_1()`, `method_N()`
- **Servant**
  - `Servant`
- **Client**
- **Concrete MethodRequest 1**
- **Concrete MethodRequest N**
- **MyHandler**
Active Object

POSA2 Concurrency

Structure & Participants

- **Proxy**
  - method_1()
  - method_N()
- **Scheduler**
  - dispatch()
  - insert()
- **Activation List**
  - insert()
  - remove()
- **Future**
  - <<invoke>>
- **MethodRequest**
  - can_run()
  - call()
  - <<execute>>
  - <<maintain>>
- **Servant**
  - method_1()
  - method_N()
- **Client**
  - <<obtain result from>>
- **Concrete MethodRequest 1**
- **Concrete MethodRequest N**

- **MyActivity**
Dynamics of the Active Object Pattern
Active Object

POS A2 Concurrency

Dynamics

Client invokes a method call on a proxy

: Client

: Proxy

: Future

: Scheduler

: Activation List

: Servant

method()

<<create>>

Future

insert()

Future

<<create>>

Method Request

insert()

can_run()

remove()

Result

<<write to future>>

dispatch()

<<read from future>>

method()
Active Object

Proxy converts method call into method request, passes to scheduler, & returns future to client

Dynamics

: Client
: Proxy
: Scheduler
: Activation List
: Servant

Method Request
insert()

Future
insert()
dispatch()

Result
<<read from future>>

Result
can_run()
call()
method()
Active Object

Dynamics

Scheduler enqueues method request
Scheduler dequeues method request at some point & runs it on the servant in a separate thread.
Clients can obtain result from futures via polling, or callbacks.
Consequences of the Active Object Pattern
Active Object POSA2 Concurrency

Consequences
Active Object

Consequences

+ Enhances concurrency & simplifies synchronized complexity

```java
Handler.sendMessage(msg);

void handleMessage(Message msg) {
    ...
}
```
Consequences

+ Enhances concurrency & simplifies synchronized complexity, e.g.
  - Client threads & asynchronous method executions can run concurrently

```
void handleMessage(Message msg)
{
    ...
}
```
Consequences

+ Enhances concurrency & simplifies synchronized complexity, e.g.
  - Client threads & asynchronous method executions can run concurrently
  - A scheduler can evaluate synchronization constraints to serialize access to servants

```java
Handler.sendMessage(msg);
void handleMessage(Message msg) {
    ...
}
```
Active Object POSA2 Concurrency

Consequences

+ Transparently leverages available parallelism

```java
Handler.sendMessage(msg);

void handleMessage(Message msg) {
    ...
}
```
Consequences
+ Transparently leverages available parallelism
  • e.g., multiple methods on an active object can execute in parallel if
    • scheduler is configured using a thread pool &
    • supported by the OS & hardware

```java
// Handler class
void handleMessage(Message msg) {
    ...
}
```
Active Object

Consequences

+ Method execution order can differ from method invocation order

```java
Looper

Handler.sendMessageDelayed(msg, delayTime);

Handler

void handleMessage(Message msg)
{
    ...
}

Message Queue

Thread1

Thread2
```
Active Object

Consequences

+ Method execution order can differ from method invocation order

• e.g., methods can execute according to synchronization constraints defined by guards & scheduling policies

```java
Handler.sendMessageDelayed(msg, delayTime);
```

```java
void handleMessage(Message msg) {
    ...
}
```
Active Object

POS A2 Concurrency

Consequences
Active Object

Consequences

- Higher runtime overhead

See www.dre.vanderbilt.edu/~schmidt/PDF/INFOCOM-94.pdf
Active Object

Consequences

- Higher runtime overhead
  - Stemming from

```java
1. Message msg = Handler.obtainMessage
   (SET_PROGRESS_BAR_VISIBILITY, ProgressBar.VISIBLE);
2. Handler.sendMessage (msg);
3. void loop() {
   ... 
   for (;;) {
      Message msg = queue.next();
      ... 
      msg.target.
      dispatchMessage(msg);
      ... 
}
4. void handleMessage(Message msg) {
   switch (msg.what) {
   case SET_PROGRESS_BAR_VISIBILITY:
      { 
      progress.setVisibility ((Integer) msg.obj);
      break;
   }
   ... 
```

POS A2 Concurrency
Active Object

Consequences

- Higher runtime overhead
  - Stemming from
  - Dynamic memory (de)allocation

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

```
1. Message msg = Handler.obtainMessage
   (SET_PROGRESS_BAR_VISIBILITY, ProgressBar.VISIBLE);
2. Handler.sendMessage (msg);
```

```
4. void handleMessage(Message msg) {
   switch (msg.what) {
   case SET_PROGRESS_BAR_VISIBILITY:
      progress.setVisibility ((Integer) msg.obj);
      break;
   ... 
}
```

POS A2 Concurrency
Active Object

Consequences

- Higher runtime overhead
  - Stemming from
    - Dynamic memory (de)allocation
    - Synchronization operations
Active Object

Consequences

- Higher runtime overhead
  - Stemming from
    - Dynamic memory (de)allocation
    - Synchronization operations
    - Context switches

```java
3. void loop() {
    ... 
    for (;;) {
        Message msg = queue.next();
        ... 
        msg.target.
            dispatchMessage(msg);
        ... 
}
```

```java
1. Message msg =
    Handler.obtainMessage
    (SET_PROGRESS_BAR_VISIBILITY,
    ProgressBar.VISIBLE);

2. Handler.sendMessage
    (msg);

4. void handleMessage(Message msg) {
    switch (msg.what) {
        case SET_PROGRESS_BAR_VISIBILITY:
            progress.setVisibility
            ((Integer) msg.obj);
            break;
    }
```

POS AO2 Concurrency
Consequences

- Higher runtime overhead
  - Stemming from
    - Dynamic memory (de)allocation
    - Synchronization operations
    - Context switches
    - CPU cache updates

3. void loop() {
   ...  
   for (;;) {
      Message msg = queue.next();
      ... 
      msg.target.
         dispatchMessage(msg);
      ... 
   }

4. void handleMessage(Message msg) {
   switch (msg.what) {
      case SET_PROGRESS_BAR_VISIBILITY:
         {
            progress.setVisibility((Integer) msg.obj);
            break;
         }
   ... 

1. Message msg =
   Handler.obtainMessage
   (SET_PROGRESS_BAR_VISIBILITY, 
    ProgressBar.VISIBLE);

2. Handler.sendMessage(msg);

UI Thread (main thread)  
Looper
Message Queue
Message
Message
Message
Message
Message
Message
Message
Message
Handler
Background Thread
Active Object

POS A2 Concurrency

Consequences
– Complicated debugging

Subsets of *Active Object* are often used to workaround these limitations.
Active Object          POSA2 Concurrency

Consequences

- Complicated debugging
  - e.g., it is harder to debug programs that use concurrency due to non-determinism of the various schedulers

Subsets of *Active Object* are often used to workaround these limitations
Known Uses of the Active Object Pattern
Active Object

Known Uses

- Programming languages based on the Actor model
- A mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation

en.wikipedia.org/wiki/Actor_model has more info
Active Object

Known Uses

• Programming languages based on the Actor model
  • A mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent digital computation
  • In response to a message that it receives, an actor can
    • Make local decisions
    • Create more actors, send more messages
    • Determine how to respond to the next message received

en.wikipedia.org/wiki/Actor_model has more info
Active Object

**Known Uses**

- Programming languages based on the Actor model
- **Active Object** in C++11

```cpp
class Active {
    public:
        typedef function<void()> Message;

        Active(): done(false) {
            th = unique_ptr<thread>(new thread([=]{ this->run(); }));
        }

        ~Active() { send([&]{done = true;}); th->join(); }

        void send(Message m) { mq.send(m); }

    private:
        message_queue<Message> mq; bool done; unique_ptr<thread> th;
        void run(){ while(!done){ Message msg = mq.receive(); msg(); }}
};
```

**Active Object**

**Known Uses**
- Programming languages based on the Actor model
- Active Object in C++11
- The ACE Task framework

---

See [www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf) for more info
Active Object

Known Uses
- Programming languages based on the Actor model
- Active Object in C++11
- The ACE Task framework
- The Java ExecutorService framework
Active Object

Known Uses

- Programming languages based on the Actor model
- Active Object in C++11
- The ACE Task framework
- The Java ExecutorService framework
- The Android HaMeR framework
Summary
Summary

- Clients may need to issue requests on components without blocking until the requests finish executing
Summary

- Clients may need to issue requests on components without blocking until the requests finish executing
  - It should also be possible to schedule the execution of client requests according to certain criteria
    - e.g., request priorities or deadlines
• Clients may need to issue requests on components without blocking until the requests finish executing.

• The *Active Object* pattern helps keep service requests independent so they can be serialized & scheduled transparently to the component & its clients.

See [www.dre.vanderbilt.edu/~schmidt/PDF/Act-Obj.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/Act-Obj.pdf) for *Active Object*
Summary

- Clients may need to issue requests on components without blocking until the requests finish executing.
- The *Active Object* pattern helps keep service requests independent so they can be serialized & scheduled transparently to the component & its clients.
- It’s instructive to compare *Active Object* with *Monitor Object*.
Clients may need to issue requests on components without blocking until the requests finish executing.

The *Active Object* pattern helps keep service requests independent so they can be serialized & scheduled transparently to the component & its clients.

It’s instructive to compare *Active Object* with *Monitor Object*.

*Active Object* is more powerful, but also more complicated (potentially has higher overhead).

See [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf) for more comparisons.
End of Android & Java Concurrency: The Active Object Pattern (Part 1)