

Android Concurrency: Overview of Patterns in Android Communication Frameworks

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

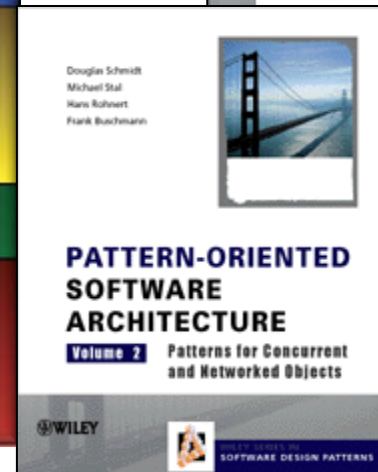
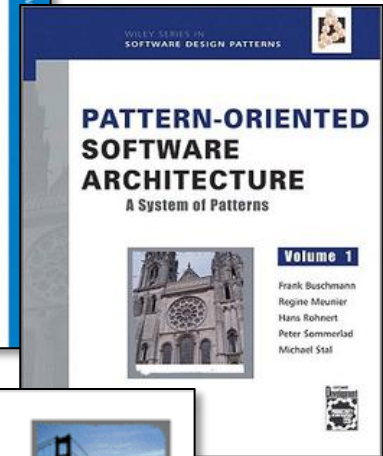
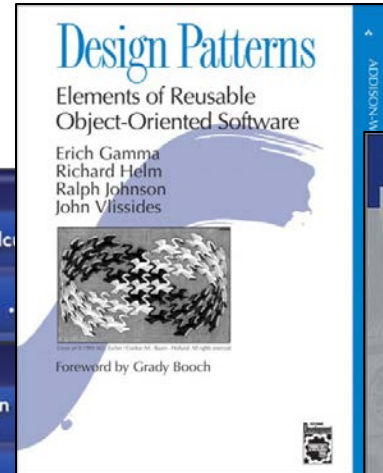
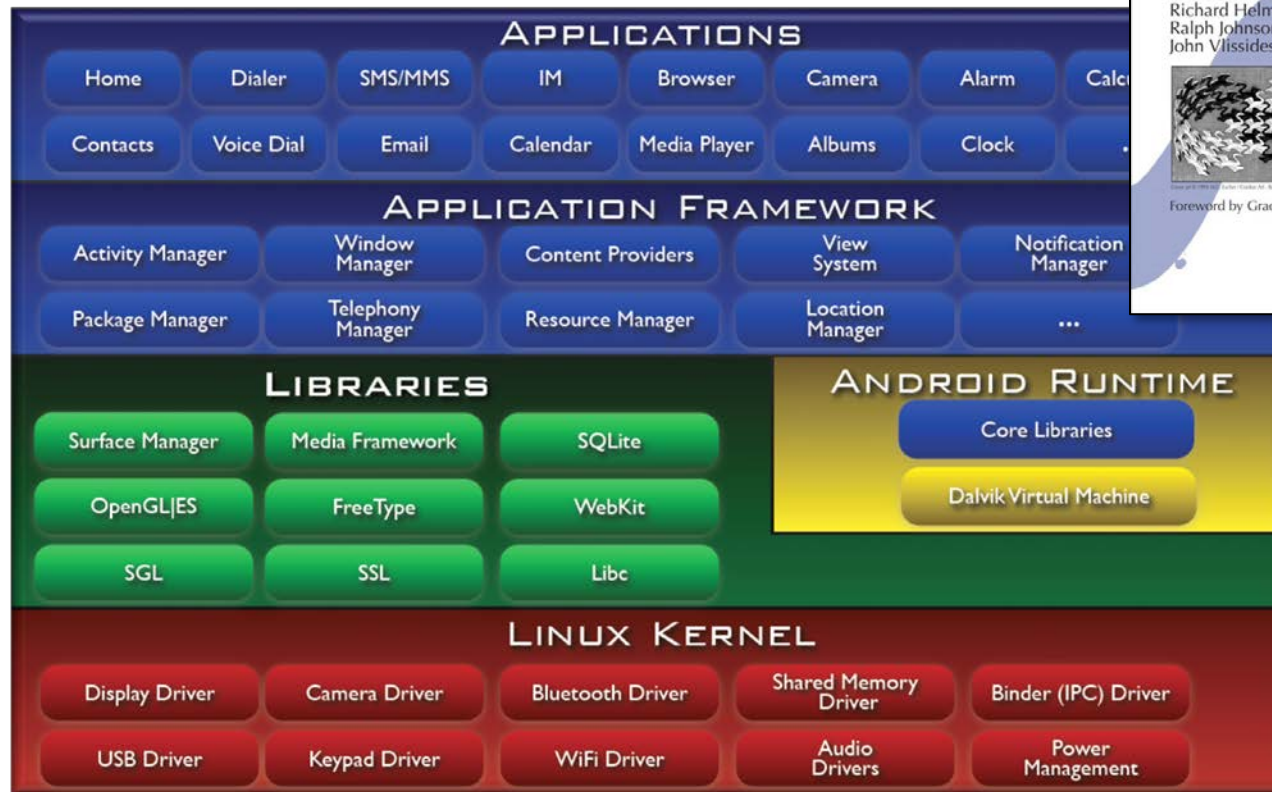
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives in this Part of the Module

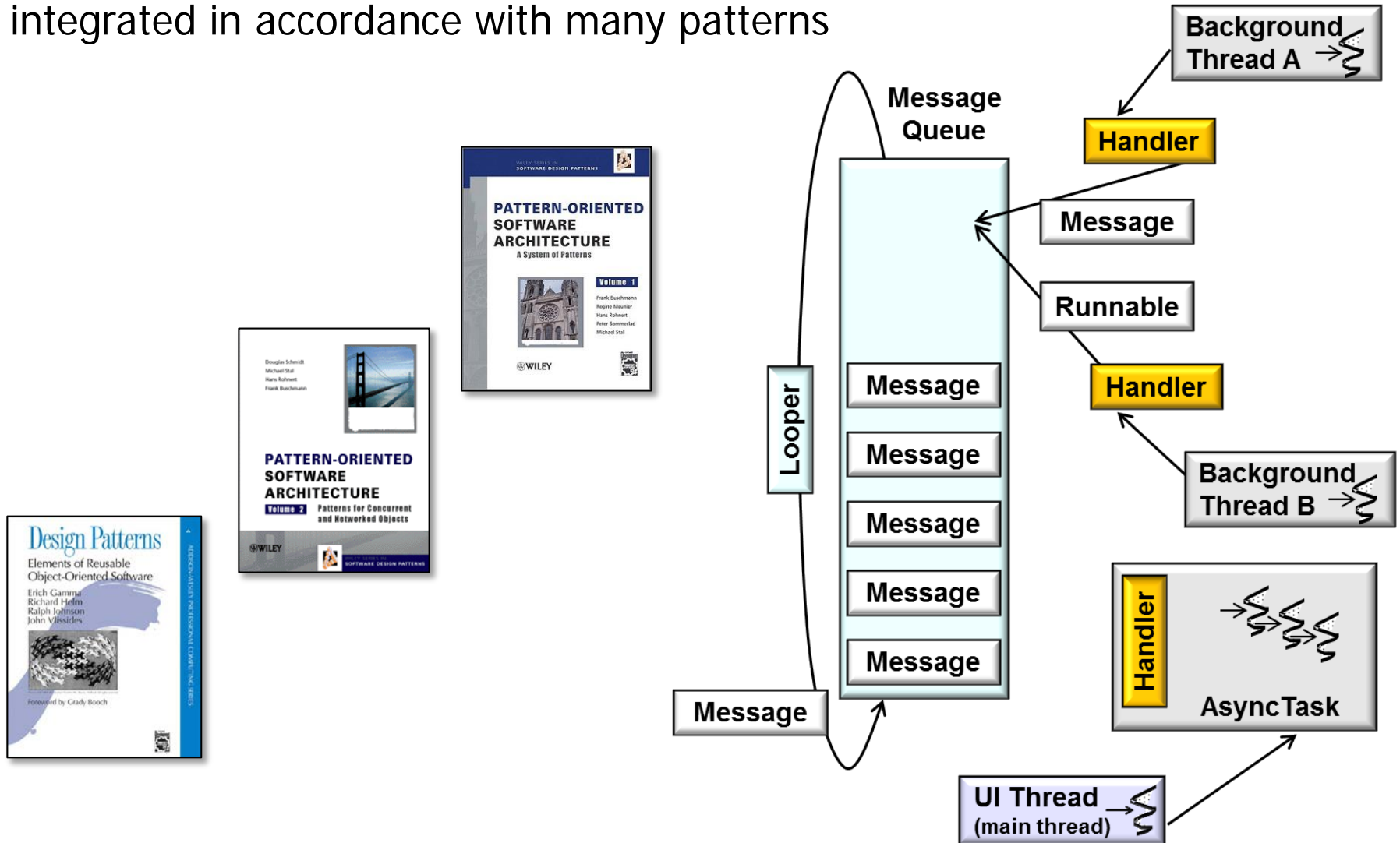
- Understand how *patterns* & *frameworks* help improve the structure & functionality of Android's concurrency & communication middleware used by Applications & Services



Patterns in Android Communication Frameworks (Part 1)

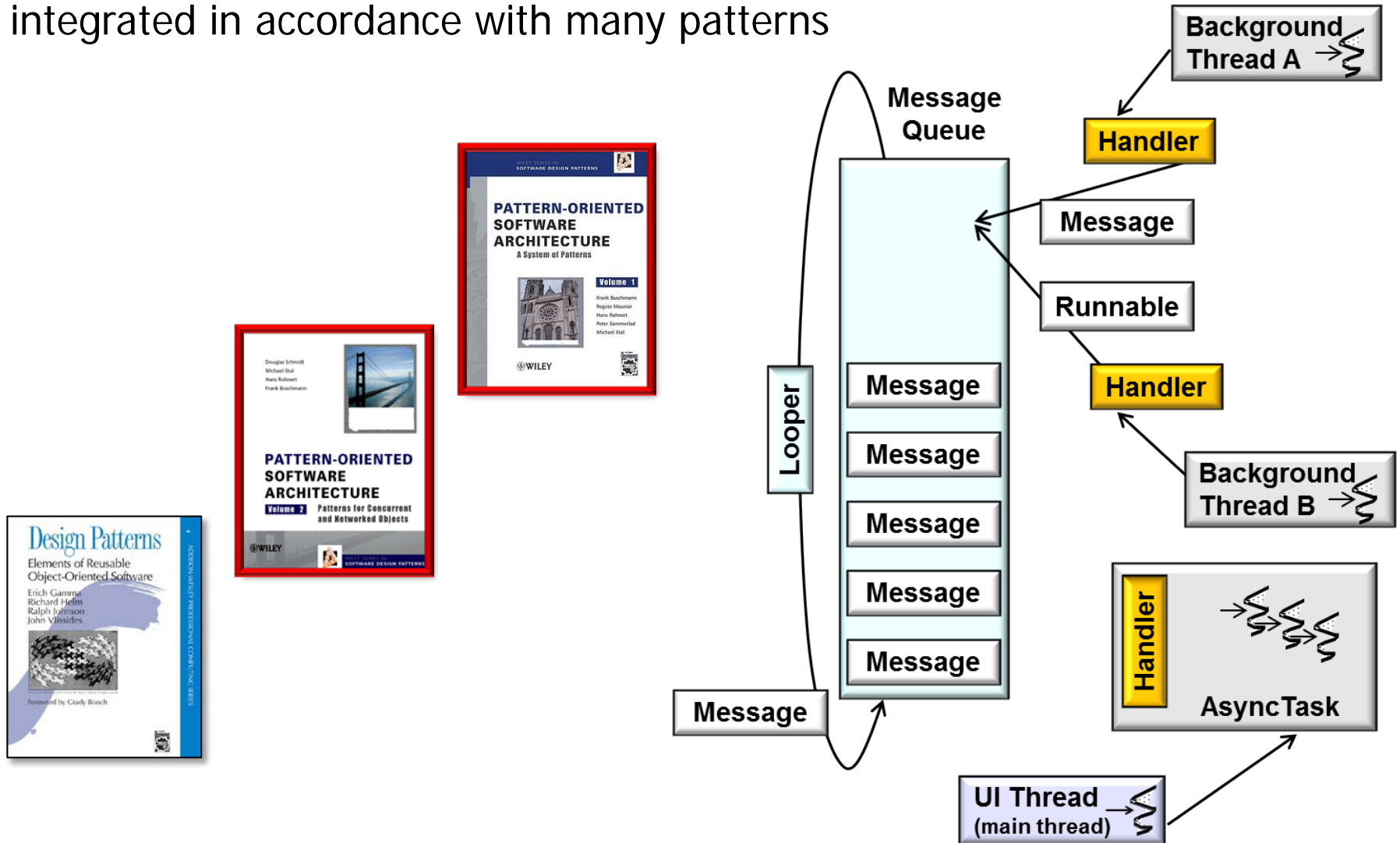
Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns



Patterns in Android Communication Frameworks

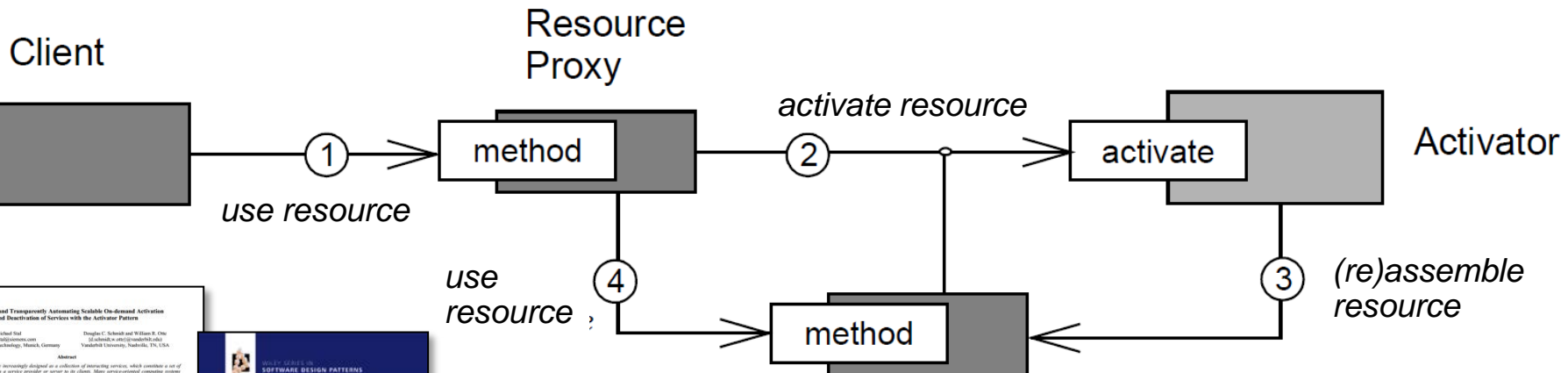
- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns



Many patterns are based on material in the POSA books

Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Activator* – Automates scalable on-demand activation & deactivation of service execution contexts to run services accessed by many clients without consuming excessive resources



Efficiently and Transparently Automating Scalable On-demand Activation and Deactivation of Services with the Activator Pattern

Michael Fay
michael.fay@intel.com
Intel Corporation, Santa Clara, CA, USA

Douglas C. Schmidt
schmidt@cs.vanderbilt.edu
Vanderbilt University, Nashville, TN, USA

Abstract

Computing systems are increasingly designed as a collection of interacting services, which constitute a set of functionalities offered by a service provider to a client. These service-oriented computing systems have constraints on the resources they allocate and manage. In these systems, certain types of services should consume resources only when they are accessed by clients and clients should be decoupled from when services are located, how they are deployed, and how their lifecycle is managed. The activator pattern provides an effective means to efficiently and transparently automate scalable on-demand activation and deactivation of services accessed by many clients. This paper motivates the need for the activator pattern, describes the structure and dynamics of contextual implementations of the pattern, and examines the benefits and limitations of applying the pattern to services in resource-constrained computing systems.

1. Introduction

Despite the decreasing costs and increasing size of state memory in computing hardware, many service-oriented systems have constraints on the resources they can efficiently manage. For example, the memory constraints in battery-powered devices, such as devices using Google Android or Apple iOS, require careful programming techniques to ensure applications do not consume excessive power (Battaglini) or exhaust resources, which can result in the device or OS to be subject to being shut down or to be in a degraded operating state and the effective life of the device. Similarly, in the cloud, the limited capacity of the underlying hardware and the effective life of the hardware (e.g., the number of times a server can be rebooted) are also constraints. The activator pattern provides an effective means to efficiently and transparently automate scalable on-demand activation and deactivation of services accessed by many clients. This paper motivates the need for the activator pattern, describes the structure and dynamics of contextual implementations of the pattern, and examines the benefits and limitations of applying the pattern to services in resource-constrained computing systems.

A typical system in the literature of on-demand resource allocation strategy for a particular set of services is shown in Figure 1. The "System Load" strategy in the diagram depicts the current memory load of the controller that lets the on-demand resource allocation strategy to be applied to the system load and the "Service Load" strategy in the diagram depicts the current memory load of the service that lets the on-demand resource allocation strategy to be applied to the service load. The "Service Load" strategy in the diagram depicts the current memory load of the service that lets the on-demand resource allocation strategy to be applied to the service load. The "Service Load" strategy in the diagram depicts the current memory load of the service that lets the on-demand resource allocation strategy to be applied to the service load.

PATTERN-ORIENTED
SOFTWARE
ARCHITECTURE

A Pattern Language for
Distributed Object Computing

Volume 4



Frank Buschman

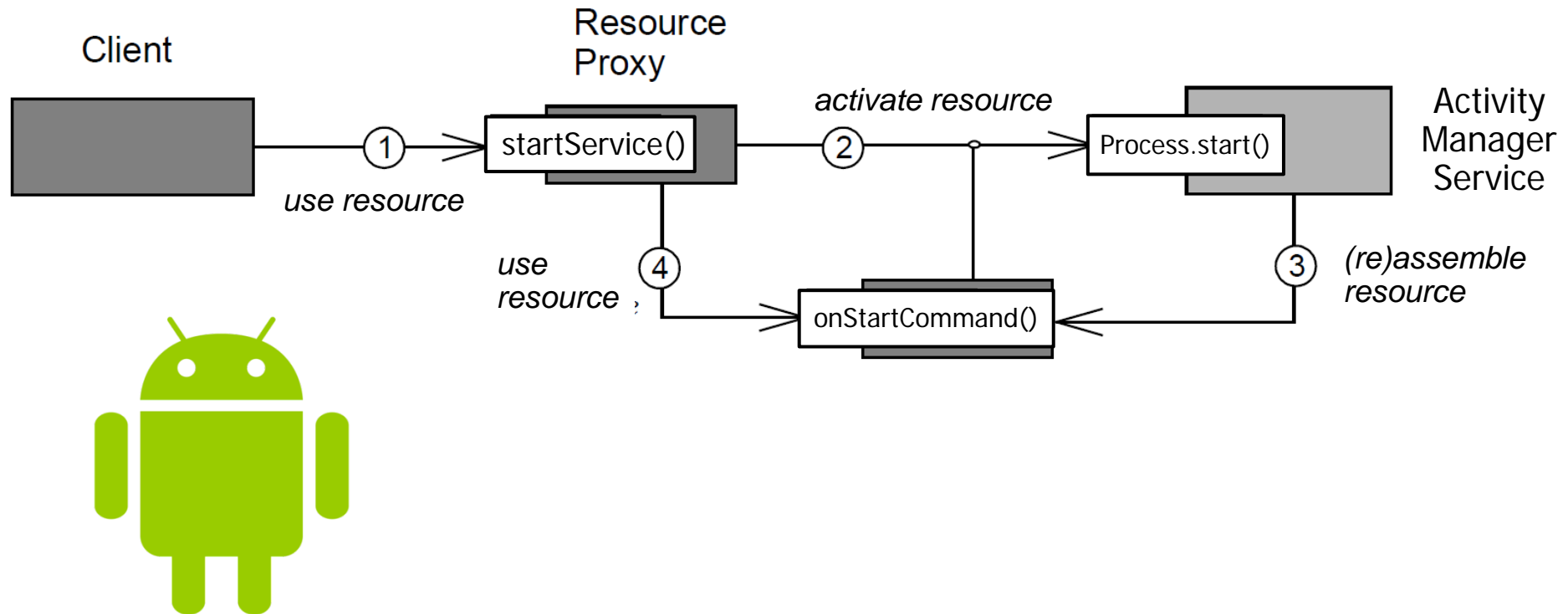
Erkay Hartney

Douglas C. Schmidt

See www.dre.vanderbilt.edu/~schmidt/PDF/Activator.pdf

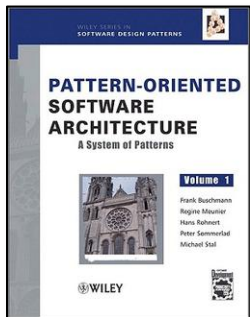
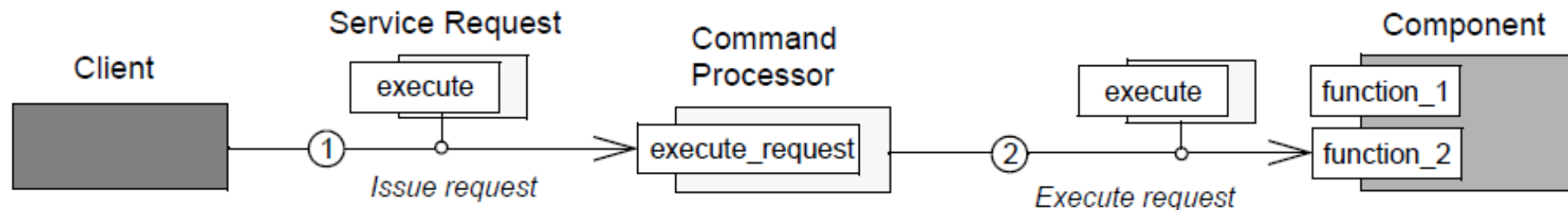
Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Activator* – Automates scalable on-demand activation & deactivation of service execution contexts to run services accessed by many clients without consuming excessive resources



Patterns in Android Communication Frameworks

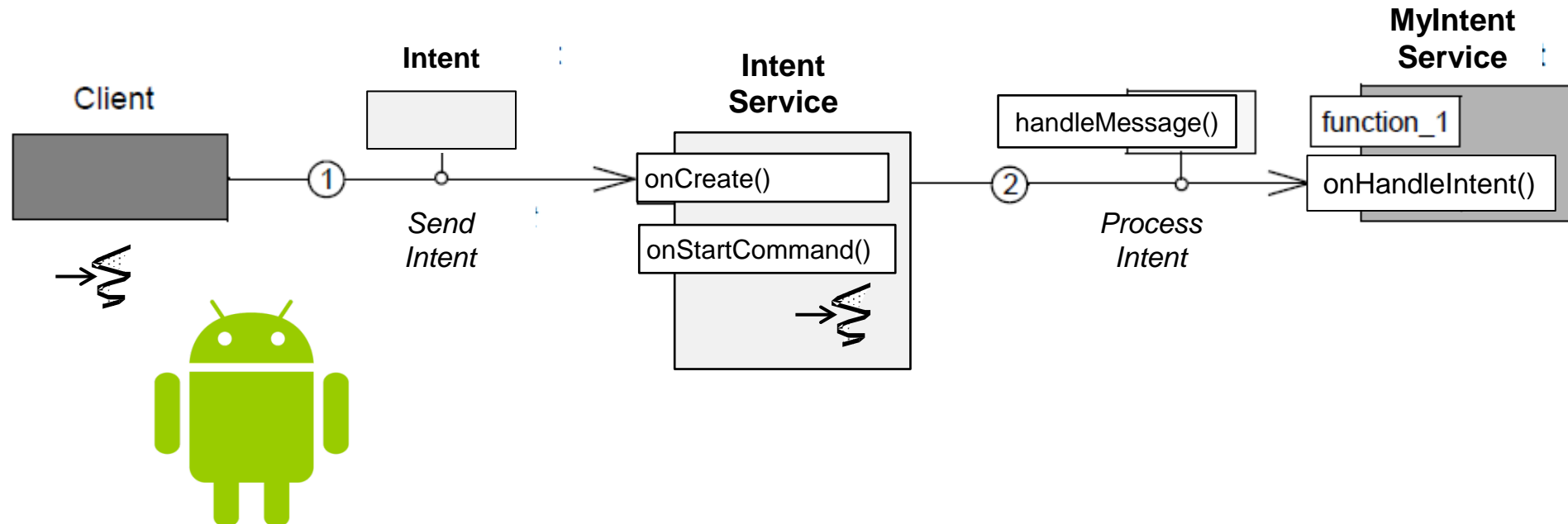
- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Command Processor* – Package a piece of application functionality—as well as its parameterization in an object—to execute it in another context
 - e.g., at a later point in time, in a different process or thread, etc.



See www.dre.vanderbilt.edu/~schmidt/PDF/CommandRevisited.pdf

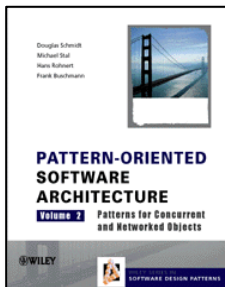
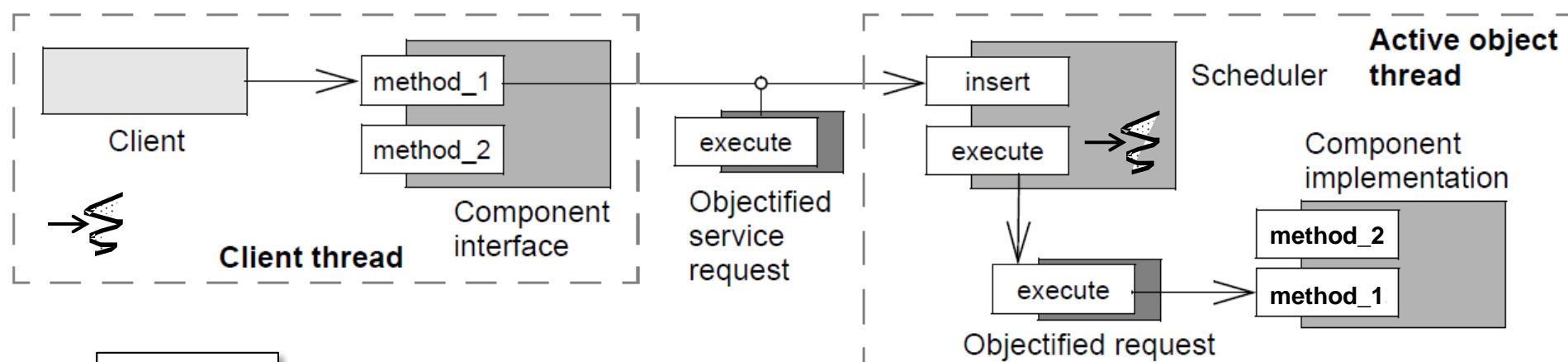
Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Command Processor* – Package a piece of application functionality—as well as its parameterization in an object—to execute it in another context
 - e.g., at a later point in time, in a different process or thread, etc.



Patterns in Android Communication Frameworks

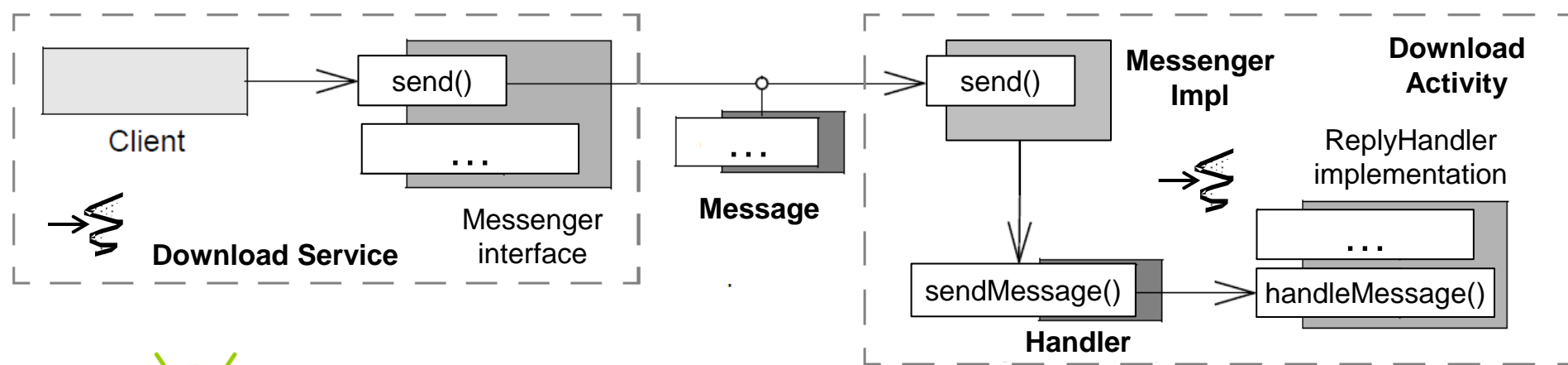
- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - Active Object* – Define service requests on components as the units of concurrency & run service requests on a component in different thread(s) from the requesting client thread



See en.wikipedia.org/wiki/Active_object

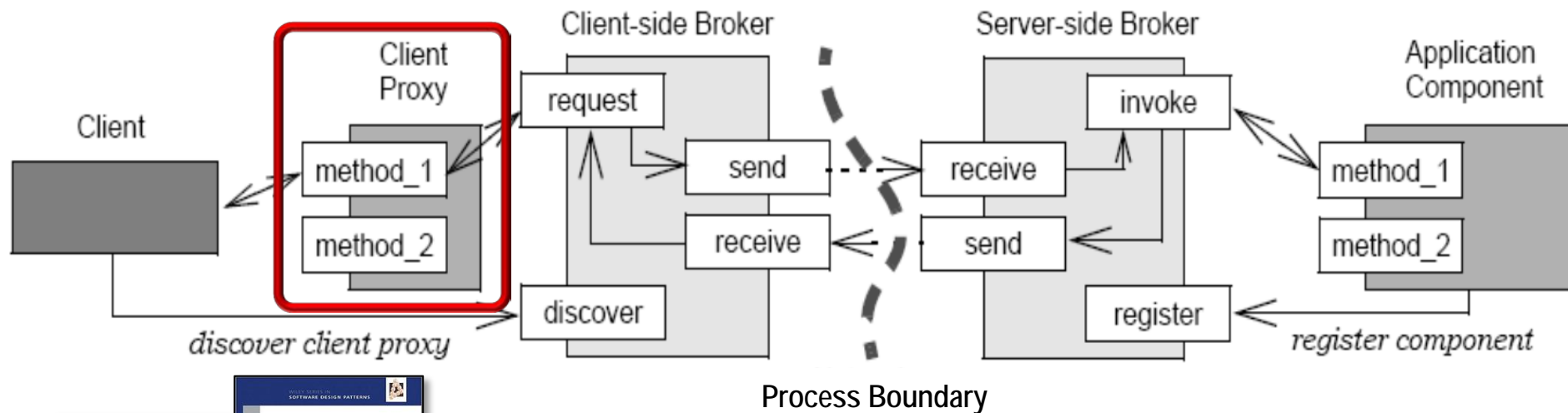
Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - Active Object* – 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



Patterns in Android Communication Frameworks

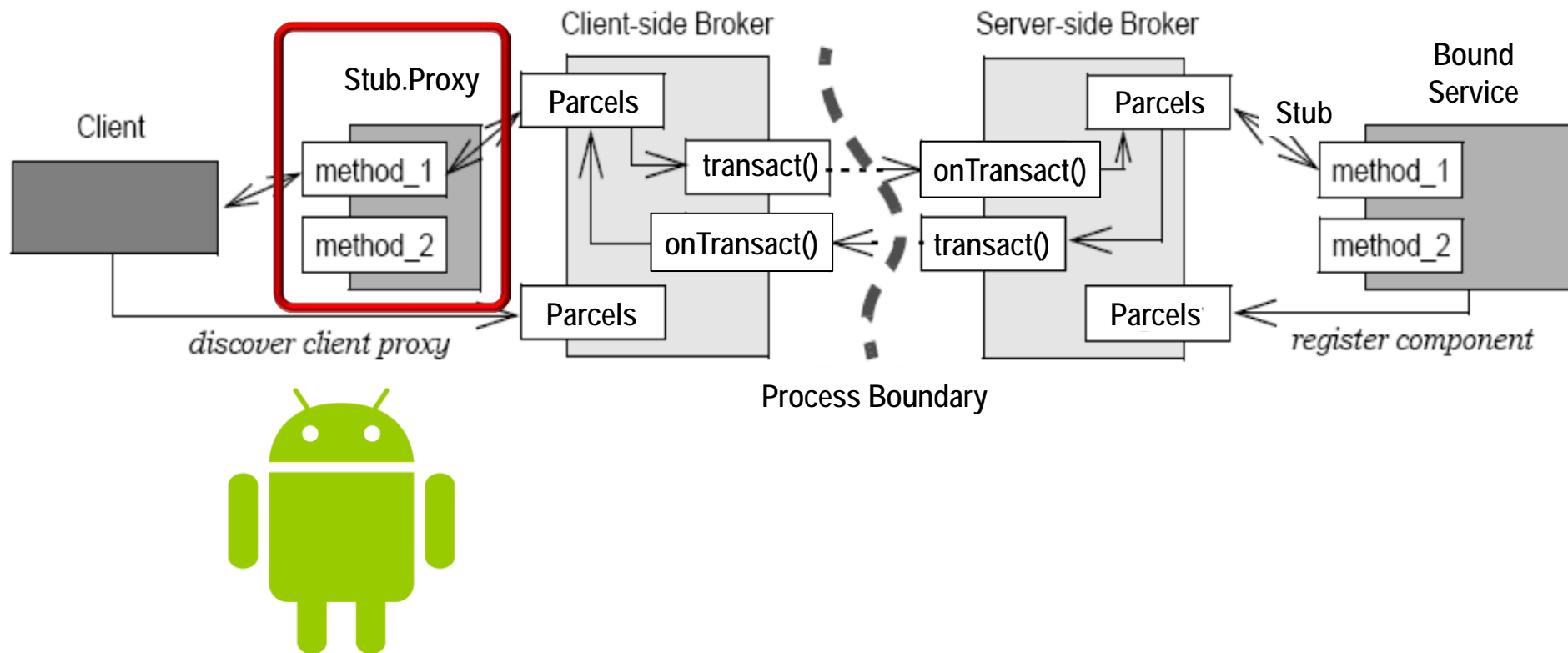
- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - Proxy* – Provide a surrogate or placeholder for another object to control access to it



See en.wikipedia.org/wiki/Proxy_pattern

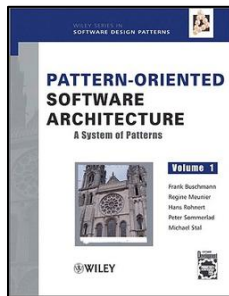
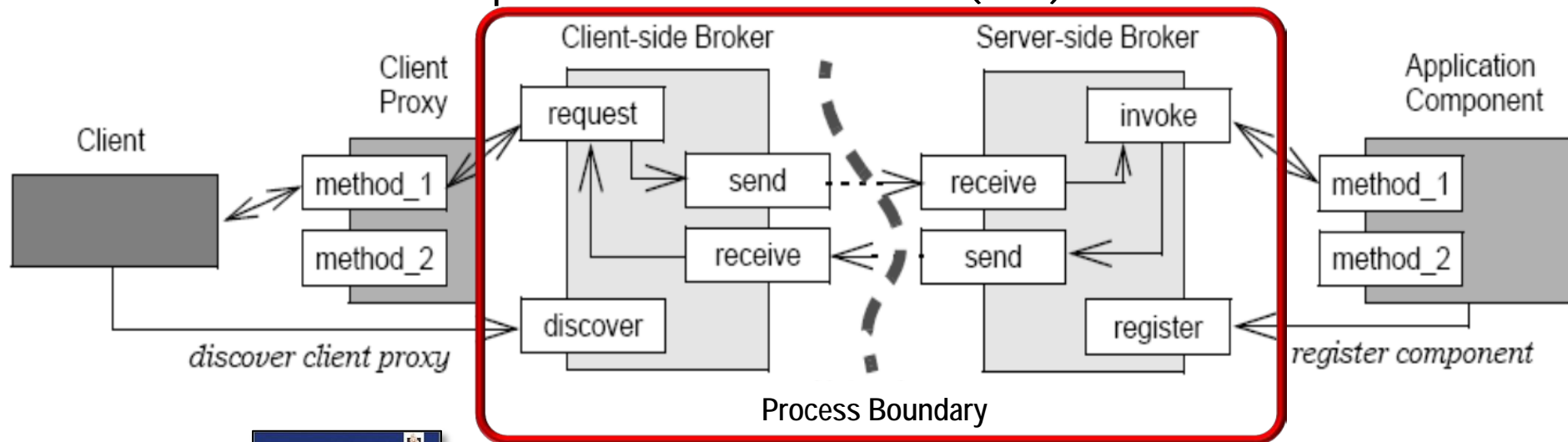
Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Proxy* – Provide a surrogate or placeholder for another object to control access to it



Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Broker* – Connect clients with remote objects by mediating invocations from clients to remote objects, while encapsulating the details of local and/or remote inter-process communication (IPC)



See www.kircher-schwanninger.de/michael/publications/BrokerRevisited.pdf

Patterns in Android Communication Frameworks

- Android's communication frameworks are designed, implemented, & integrated in accordance with many patterns
 - *Broker* – Connect clients with remote objects by mediating invocations from clients to remote objects, while encapsulating the details of local and/or remote inter-process communication (IPC)

