**Pattern-Oriented Software Architecture(Patterns for concurrent and Networked Objects)**

목차

[Chapter 1: Concurrent and Networked Objects 1](#_Toc456704209)

[Overview 1](#_Toc456704210)

[1.1 Motivation 1](#_Toc456704211)

[Challenge 1 : Service Access and Configuration 2](#_Toc456704212)

[Challenge 3: Concurrency 3](#_Toc456704213)

[Chapter 2: Service Access and Configuration 4](#_Toc456704214)

[Overview 4](#_Toc456704215)

[Wrapper Facade 5](#_Toc456704216)

[[Example] 5](#_Toc456704217)

[Context 5](#_Toc456704218)

[Problem 5](#_Toc456704219)

[Solution 6](#_Toc456704220)

[Structure 6](#_Toc456704221)

[Dynamic(동작) 7](#_Toc456704222)

[Implementation 7](#_Toc456704223)

# Chapter 1: Concurrent and Networked Objects

## Overview

## 1.1 Motivation

1.2 Challenges of Concurrent and Networked S/W

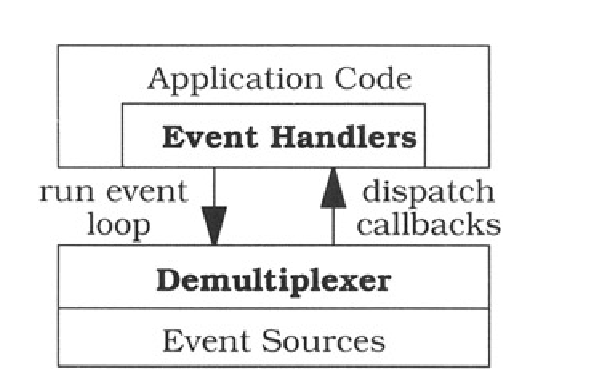
* concurrent & networked services는 시스템의 성능, reliability, scalability, cost-effectiveness를 향상시킨다고 믿는다. 하지만 실제로는 복잡복잡…
* **Collaboration and Connectivity**

### Challenge 1 : Service Access and Configuration

* stand-alone application은 single address space를 가지기에 이것저것 공유하고 접근할 수 있다.
* 하지만, networked application은 다음을 이용해야한다.
* IPC ( for shared memory, pipes, sockets)
* Communication protocols : telnet, ftp, smpt http, ldap 등등등
* Remote Operations : com+, CORBA
* Concurrency service access APIs :
* IPC service access APIs :

Challenge 2: Event Handling

* event-driven application의 3가지 특성 :
* application behivor는 external 또는 internal event로 비동기적으로 발생된다.
* 대부분의 이벤트는 CPU starvation, improve perceived response time, keep h/w device real-time 하기 위해 사용한다.
* 이벤트 처리를 제어하고 이상한 천이를 감지하기 위해서 FSM이 필요하다.



@ event sources : 여러 h/w와 s/w로 부터 이벤트를 감지하고 전달한다.

@ demultiplexer : select()가 그 예시. 이벤트를 기다리고 있다가 해당 handler callback으로 전달해준다.

@ event handlers : application code와 함께 이벤트 처리. callback의 응답으로 application-specific proccessing을 한다. 그래서 ‘**inversion of control**’이라고 부른다. (callback사용으로인해 처리 순서가 달라졌기 때문!)

* chapter 3에서 다룬다!!

### Challenge 3: Concurrency

* 여러개의 스레드, 프로세스가 그들의 서비스들을 동시에 수행할 수 있도록 해주는 정책 및 메커니즘
* **프로세스는 units of protection and resource allocation**
* **Thread는 unit of execution!**
* 이런 multi-thread programming(parallel processing)의 장점 !
* improve performance transparently by using parallel processing
* improve performance explicitly
* improve perceived response time
* simplify application design
* 일반적인 multi-threading에서 발생하는 문제로는 **race condition**과 **deadlock이** 있다!
* 일반적으로 adapter, wrapper facade(패턴)가 사용된다.
* Chatper 5.에서 다룬다!

# Chapter 2: Service Access and Configuration

## Overview

* 서비스와 컴포넌트들을 설정하고 접근하는 application programming interface(APIs)를 효율적으로 설계하기 위한 4가지 패턴을 소개!
* wrapper facade
* Component Configurator
* Interceptor
* Extension Interface

**[wrapper facade]**

* non-object-oriendted API가 제공하는 함수와 데이터들을 encapsulate.
* 나중에 reactor, proactor, acceptor-connector, strategized locking, active object, monitor object 등에 사용된다!

**[Component Configurator]**

* application이 run-time에 component implementation을 수정, 재컴파일 없이! link, unlink할 수 있도록 해준다.
* service configuration, service evolution의 측면에서 사용.
* Extension Interface, Interceptor가 사용한다 !

**[Interceptor]**

* framework에 서비스를 등록하고, 특정 이벤트가 발생했을 때 자동적으로 실행될 수 있도록 한다!
* Interceptor는 초기 프레임워크가 개발되었을 때에는 모르거나, 설정되지 않았던 서비스들에 대해서 대비할 수 있도록 해준다. (개발 이후에 쉽게 추가할 수 있도록 해줌)

**[Extension Interface]**

* interface의 팽창과 기존의 component의 기능 들을 수정할 때 breakage를 예방해준다.
* 1개의 component에 여러 extension interface가 붙을 수 있다.
* 각 패턴에서 제시한 challenge외에 아래와 같은 challenge들이 또 존재한다.
* remote service의 접근을 ‘local proxies’로 중재!
* 서비스의 생명주기를 관리하고 분산 시스템에 서비스를 위치시키는 것 !

## Wrapper Facade

### [Example]

* low level api는 운영체제, 컴파일러에 따라 다르다.
* direct programming of low-level apis make the code unnecessarily hard to understand, debug, port, maintain.
* 플랫폼 종속적인 declarations을 하지 말고,(mutex, socket type등과 같은것들) 개별적인 configuration header files로 만들어라 !

### Context

* non-object-oriented api가 제공하는 서비스와 메커니즘을 사용해라 !

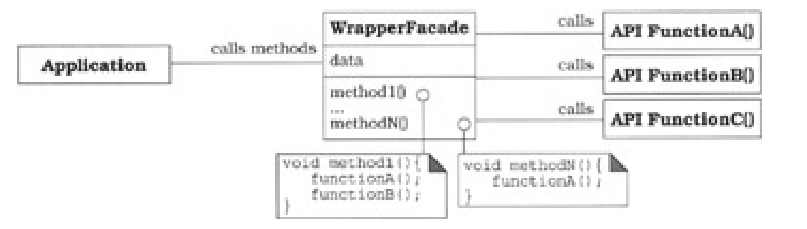
### Problem

* higher-level 수준의 object-oriented 언어들은 common programming error의 가능성을 낮춰준다.
* 하지만, low-level의 function-based api를 직접 사용하게 되면 에러의 가능성이 높아진다 !
* 그리고 low-level로 짜게 되면 유지 보수가 힘들어진다. 그렇다고 해서 conditional compilation을 통한 플랫폼 특정적인 구현은 복잡도를 더 증가시킨다.

### Solution

* non-object-oriented API를 직접 사용하는것을 피해라
* 해당 non-object oriented 함수와 데이터를 쓸 때에는, ‘wrapper facade’ 클래스를 만들어서 해당 함수와 데이터를 포장해라.

### Structure



* Wrapper Facade 패턴에 2가지 방법이 사용된다.

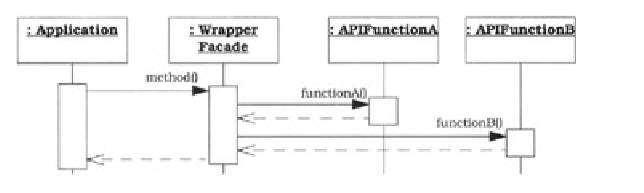
1. Functions

* non-object-oriented API가 존재하는 building blocks.
* standalone service, mechanism을 제공한다.

1. wrapper facade

* set of one or more object-oriented classes
* 여기에 있는method들은 one or more function들에 요청을 전달한다.(기능을 위한 해당 함수들을 호출한다)

### Dynamic(동작)



Wrapper Facade안에 논리적인 Method들이 있다.

이 Method들은 standalone 기능을 제공하기 위하여 관련된 여러 low-level api들을 수행해준다.

### Implementation(p.55)

* wrapper facade의 구현은 **‘acceptor-connector, strategized locking, thread-specific storage, monitor object’** 등에 쓰인다!

1. identify the cohesive abstractions and relation among existing APIs.
   1. Create cohesive classes
   2. Coalesce multiple individual functions into a single method

* Wrapper façade class에 여러 개의 함수들을 묶어서 적은 수의 method를 만든다.
  1. Automate creation and destruction operations
* Lower level api는 때때로 개발자에게 data structure를 만들고, 해제하는 일을 시키는데, 이는 에러 유발 가능성이 크다.(까먹ㅋ)
* 그래서 이러한 기능들을 포함시켜라.
  1. Select the level of indirection
  2. Determine where to encapsulate any platform-specific variation.

1. Conditional compilation

* Platform-specific variation은 conditional compilation과 같은 것들로 캡슐화 되어야 한다.
* 가능하면 platform-specific 요소 없도록 작성.
* Conditional compilation은 여러 개의 wrapper façade class 중에서 필요한 것을 선택해준다.
* developer에게 직접적으로 노출되지 않으면 사용되어도 괜찮다.

1. Separate directories

* 각 platform별로 폴더를 만들어서 관리한다.
* Conditional compilation을 최소화!