아키텍처 패턴과 시스템 설계

송태국
넥스트리 소프트㈜ 부사장/CTO
tsong@nextree.co.kr

POC 설계 - 목차

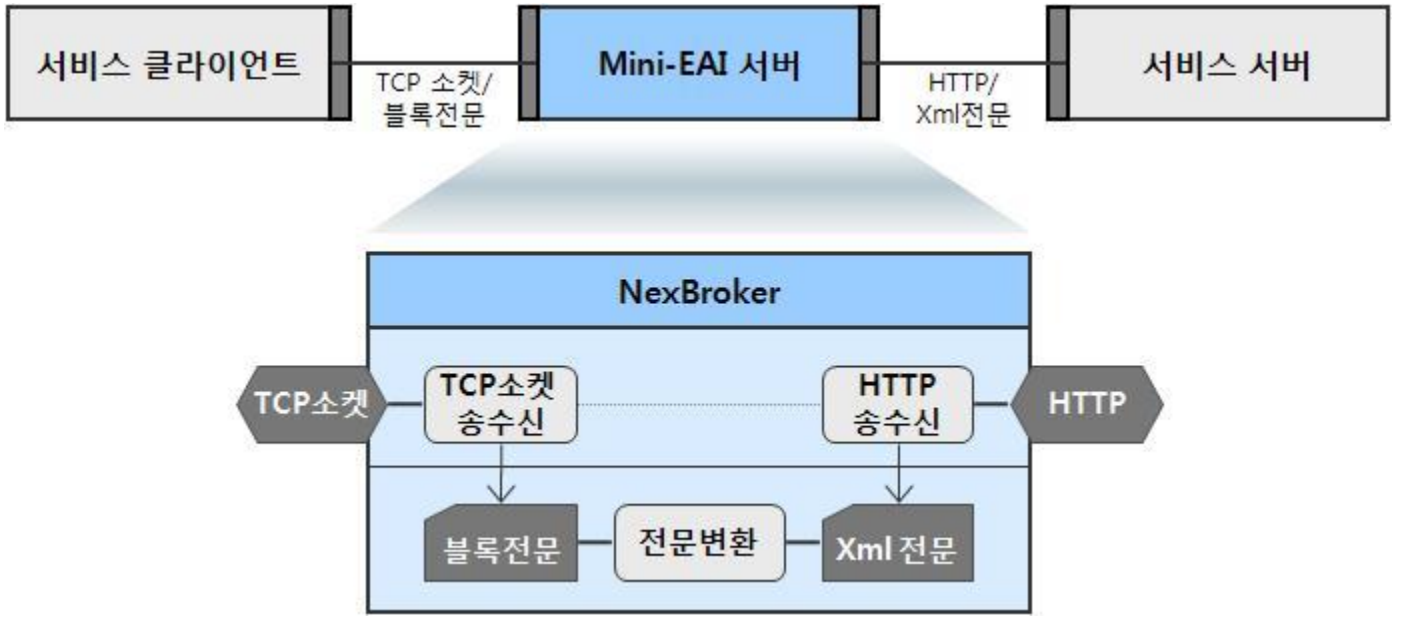
1. 요구사항
2. 분석
3. 설계
4. 구현
5. 실행
6. 개선
7. 질의 응답 및 토론

1. 요구사항 - 문제 기술서

- ✓ 전사 차원의 시스템 구축 프로젝트 수행 중, 대외 통신은 HTTP 프로토콜과 Xml 전문을 표준으로 함
- ✓ 레거시 시스템 현황
  - 메임프레임에서 실행되던 코볼로 작성된 서비스 애플리케이션이 신규 시스템으로 이전
  - 서비스를 활용하던 레거시 애플리케이션들은 신규 시스템의 서비스를 활용하여야 함
  - 소스코드 유실, 소프트웨어 개발자 부재(C, 델파이, 비주얼베이직) 등의 이유로 신규 시스템으로 직접 연결을 위한 레거시 애플리케이션 업데이트가 제한적임
  - 레거시 시스템은 TCP소켓으로 통신을 하며 블록형식의 전문을 사용함
  - 레거시와 신규 시스템 간의 통신 문제로 중개서버(Mini-EAI서버)가 필요함
- ✓ TCP소켓/블록전문을 HTTP/Xml 전문으로 변환해 줄 Mini-EAI가 필요함
- ✓ 시간이 흐르면 레거시 애플리케이션은 하나 둘 사라질 예정임
- ✓ TCP소켓 , HTTP, SOAP 등의 프로토콜 변화에 따른 확장성이 필요함
- ✓ 블록전문 ←→ 구분자 전문 ←→ Xml 전문 ←→ WSDL 전문 등으로의 변화에 따른 확장이 필요함
- ✓ 30여 종의 레거시 클라이언트 요청을 받아서 신규시스템으로 전달할 수 있어야 함
- ✓ 높은 수준의 가용성 제공하여야 하며, 내부에 존재하므로 전문 보안 및 암호화 요구는 없음

1. 요구사항 – 컨텍스트 이해(1/2)

- ✓ 이기종성 극복을 위한 중간 매개체가 필요함
- ✓ Mini-EAI 서버 역할을 하는 NexBroker가 필요함
- ✓ 프로토콜 및 전문 변환(transformation)이 주요 역할임



1. 요구사항 – 컨텍스트 이해(1/2)

NEXTREE

✓ 회사의 로드맵

- 회사는 향후 5년 동안 SOA 환경을 완전히 정착시키고자 한다.
- 따라서 HTTP/Xml 기반의 신규 시스템들을 3년 후 WSDL/SOAP 기반으로 변경한다.
- 다양한 프로토콜을 사용하는 클라이언트들은 Mini-EAI를 확장하여 2년간 사용한다.
- 5년 후에는 전사의 모든 시스템을 서비스 호출 및 제공이 가능한 시스템으로 구축한다.
- 전사에 SOA가 정착한 후 NexBroker는 폐기하지 않고, 외부에 도입한 시스템과 본사 시스템을 연결하는 다리 역할을 한다.

✓ NexBroker 비전

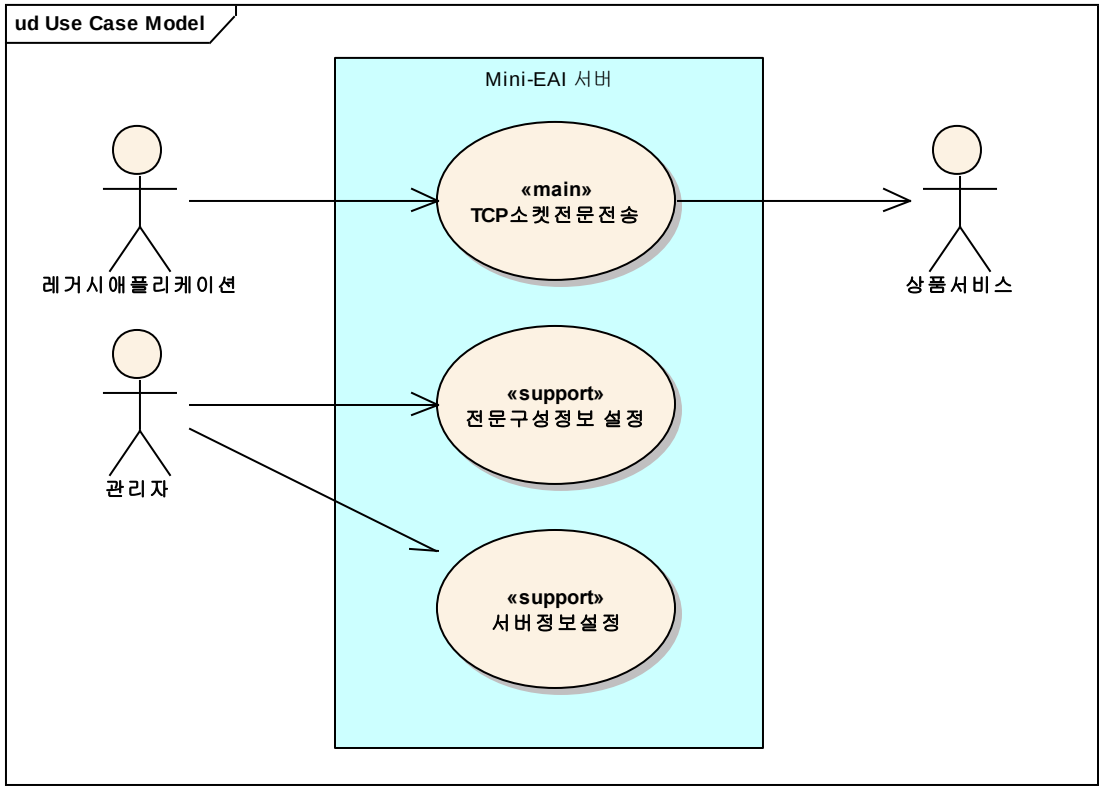
- 현재 TCP소켓 ←→ HTTP, 블록전문 ←→ Xml전문 기능에 충실한다.
- 회사의 로드맵에 따라 다음과 같은 확장 가능성을 갖춘다.
- TCP소켓 ←→ SOAP, 블록전문 ←→ WSDL전문 변환
- HTTP ←→ SOAP, Xml전문 ←→ WSDL전문 변환
- 회사의 규모 확장에 따른 트랜잭션을 감당할 수 있도록 설계한다.

1. 요구사항 – 유스케이스

NEXTREE

✓ Mini-EAI 서버(NexBroker)의 주요 유스케이스

- TCP소켓전문 전송(우선순위:1) – Elaboration에서 아키텍처 프로토타입에 사용
- 전문구성정보 설정(우선순위:2)
- 서버정보설정(우선순위:2)



1. 요구사항 – 유스케이스 명세서

NEXTREE

✓ 유스케이스 : TCP소켓전문 전송

- 유형: Main
- 수준: 사용자 수준
- 흐름:

1. Mini-EAI 서버는 TCP 소켓 블록전문 수신 준비를 하고 대기한다.
2. 레거시 애플리케이션은 TCP소켓전문을 송신한다.
3. Mini-EAI 서버는 TCP 소켓 전문을 수신한다.
    3.1 블록형식의 소켓전문을 Xml 로 변환한다.
    3.2 상품 서버로 서비스를 요청한다.
    3.3 응답 Xml을 블록형식의 소켓전문으로 변환한다.
    3.4 레거시 애플리케이션으로 전문을 전송한다.
4. 레거시 애플리케이션은 TCP소켓전문을 수신한다.
5. Mini-EAI 서버는 대기상태로 들어간다.

1. 요구사항 – 유스케이스 명세서

NEXTREE

✓ 문제 기술서와 고객 인터뷰를 통해서 다음 품질 속성을 식별함

- 가용성(availability) – 중개서버의 특성상 높은 수준의 가용성을 확보하여야 함
- 확장성(extendability) – 향후 사용하게 될 프로토콜 및 전문을 수용할 수 있어야 함
- 수행성능(performance) – 월말 마감처리 등 작업이 집중될 경우 처리할 수 있어야 함
- 사용의 편의성(useability) – 서버 정보 설정 및 블록전문정보 설정이 용이해야 함

문제기술서에서 비기능 요구사항 부문

1. 레거시와 신규 시스템 간의 통신 문제로 중개서버(Mini-EAI서버)가 필요함
2. TCP소켓/블록전문을 HTTP/Xml 전문으로 변환해 줄 Mini-EAI가 필요함.
3. 시간이 흐르면 레거시 애플리케이션은 하나 둘 사라질 예정임.
4. TCP소켓 → HTTP → SOAP 등의 프로토콜 변화에 따른 확장성이 필요함
5. 블록전문 → 구분자 전문 → Xml 전문 → WSDL 전문 등으로의 변화에 따른 확장이 필요함
6. 30여 종의 레거시 클라이언트 요청을 받아서 신규시스템으로 전달할 수 있어야 함
7. 높은 수준의 가용성 제공하여야 함
8. 내부에 존재하므로 전문에 대한 보안 및 암호화 요구는 없음

2. 분석 – 객체 식별

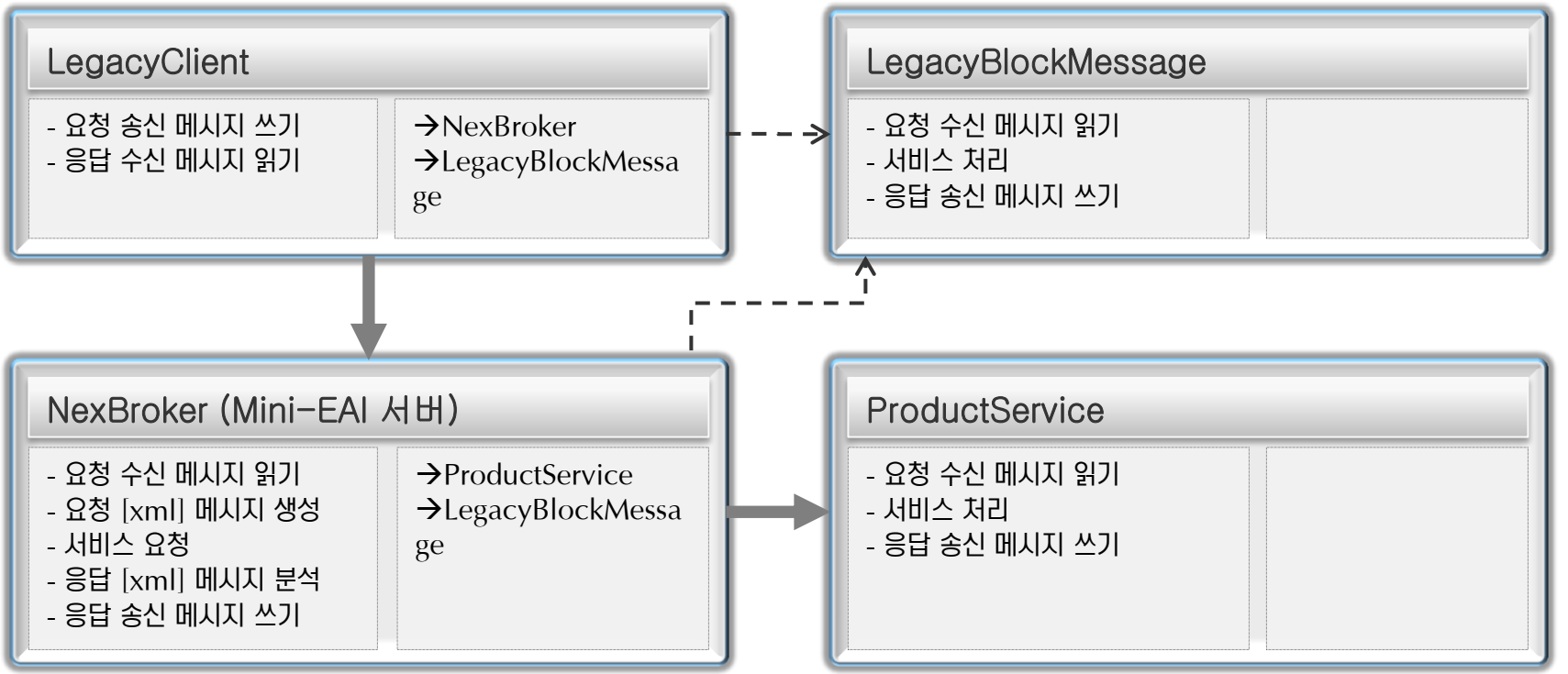
NEXTREE

✓ 내부 객체

- NexBroker(Mini-EAI 서버)
- LegacyBlockMessage – 전문 정보 저장 객체

✓ 외부 객체

- LegacyClient, ProductService

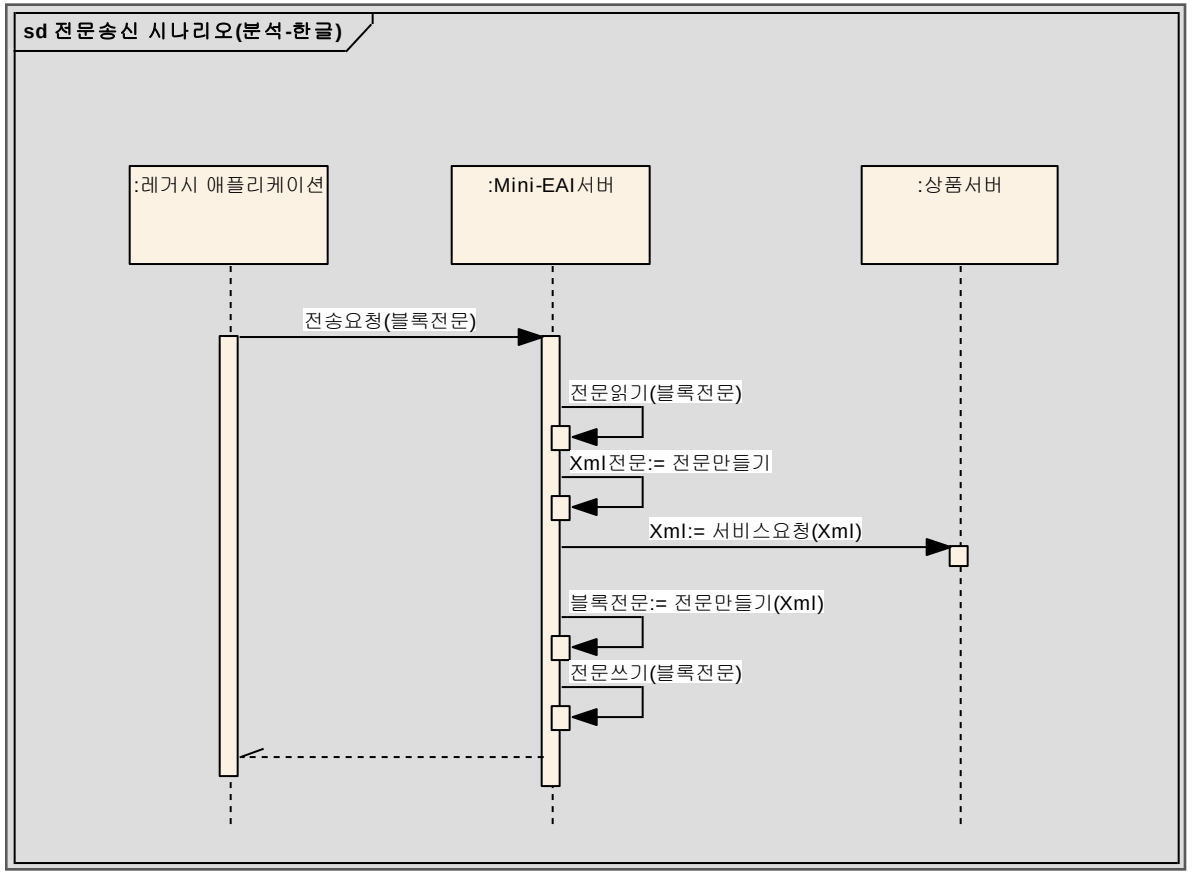


2. 분석 – 객체 행위 분석

NEXTREE

✓ 유스케이스, 문제기술서 등의 정보로부터 확인한 객체들 간의 정보 흐름, 즉, 객체의 행위를 시퀀스 다이어그램으로 표현함

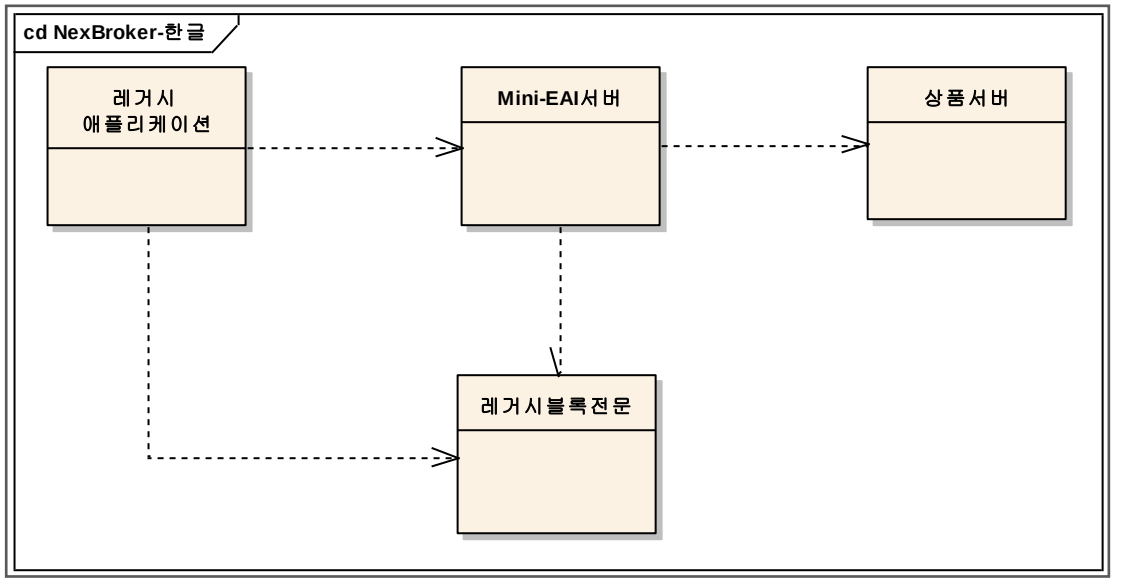
1. Mini-EAI 서버는 TCP 소켓 블록전문 수신 준비를 하고 대기한다.
2. 레거시 애플리케이션은 TCP소켓전문을 송신한다.
3. Mini-EAI 서버는 TCP 소켓 전문을 수신한다.
    3.1 블록형식의 소켓전문을 Xml 로 변환한다.
    3.2 상품 서버로 서비스를 요청한다.
    3.3 응답 Xml을 블록형식의 소켓전문으로 변환한다.
    3.4 레거시 애플리케이션으로 전문을 전송한다.
4. 레거시 애플리케이션은 TCP소켓전문을 수신한다.
5. Mini-EAI 서버는 대기상태로 들어간다.



2. 분석 – 객체 관계 분석

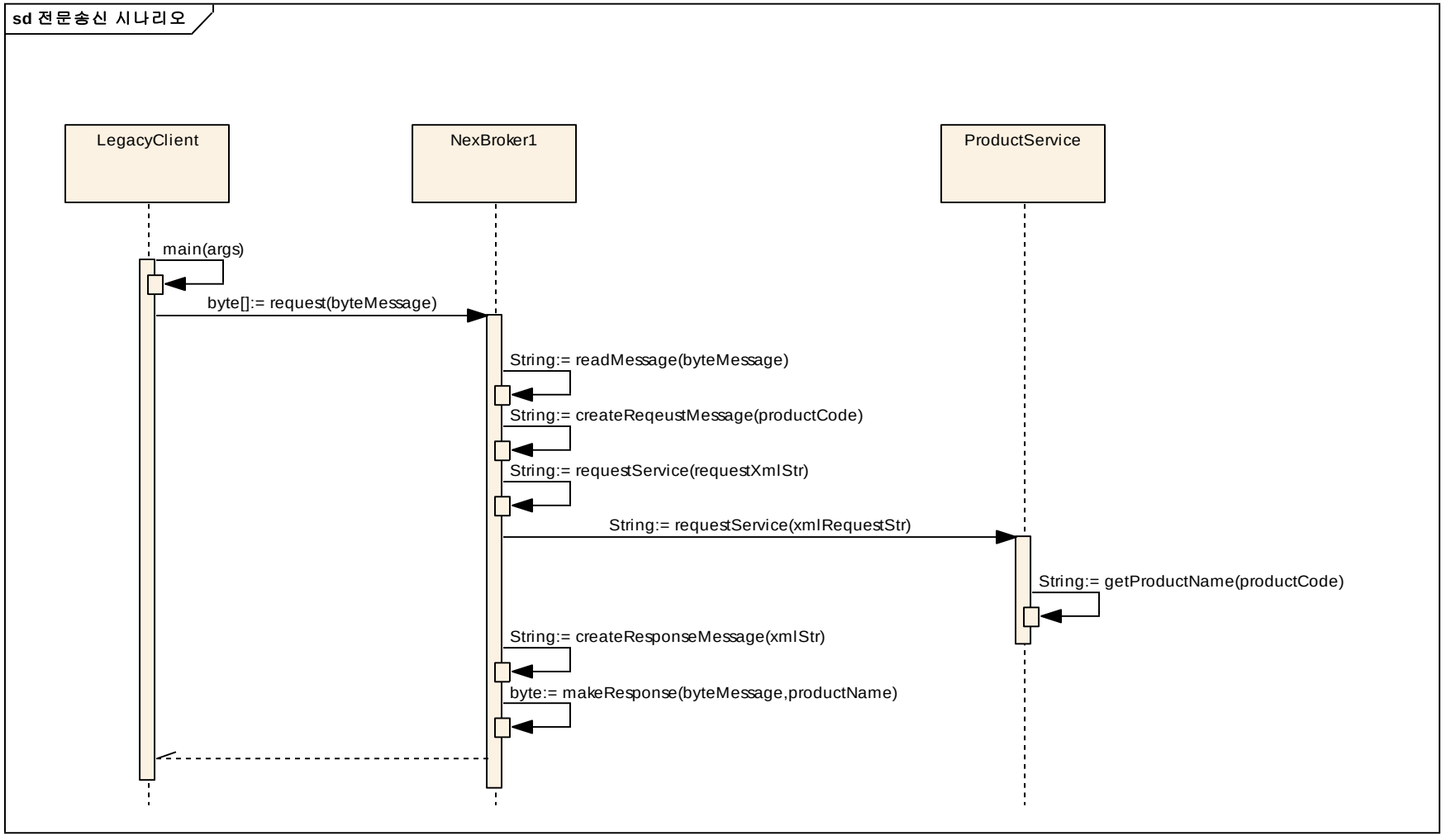
- ✓ 행위 모델링 결과 객체들 간의 메시지 흐름에 참여한 객체를 클래스로 식별하고, 메시지의 흐름을 기준으로 클래스 간의 관계를 구성함
- ✓ 분석 모델링은 사용자가 볼 것이라는 가정하에서, 즉 사용자의 관점에서의 모델링 이므로 기술적인 내용은 최대한 배제하고 구성함
- ✓ 사용자와의 의사교환을 위해 클래스, 메소드 이름을 한글로 표현해도 됨

1. Mini-EAI 서버는 TCP 소켓 블록전문 수신 준비를 하고 대기한다.
2. 레거시 애플리케이션은 TCP소켓전문을 송신한다.
3. Mini-EAI 서버는 TCP 소켓 전문을 수신한다.
    3.1 블록형식의 소켓전문을 Xml 로 변환한다.
    3.2 상품 서버로 서비스를 요청한다.
    3.3 응답 Xml을 블록형식의 소켓전문으로 변환한다.
    3.4 레거시 애플리케이션으로 전문을 전송한다.
4. 레거시 애플리케이션은 TCP소켓전문을 수신한다.
5. Mini-EAI 서버는 대기상태로 들어간다.



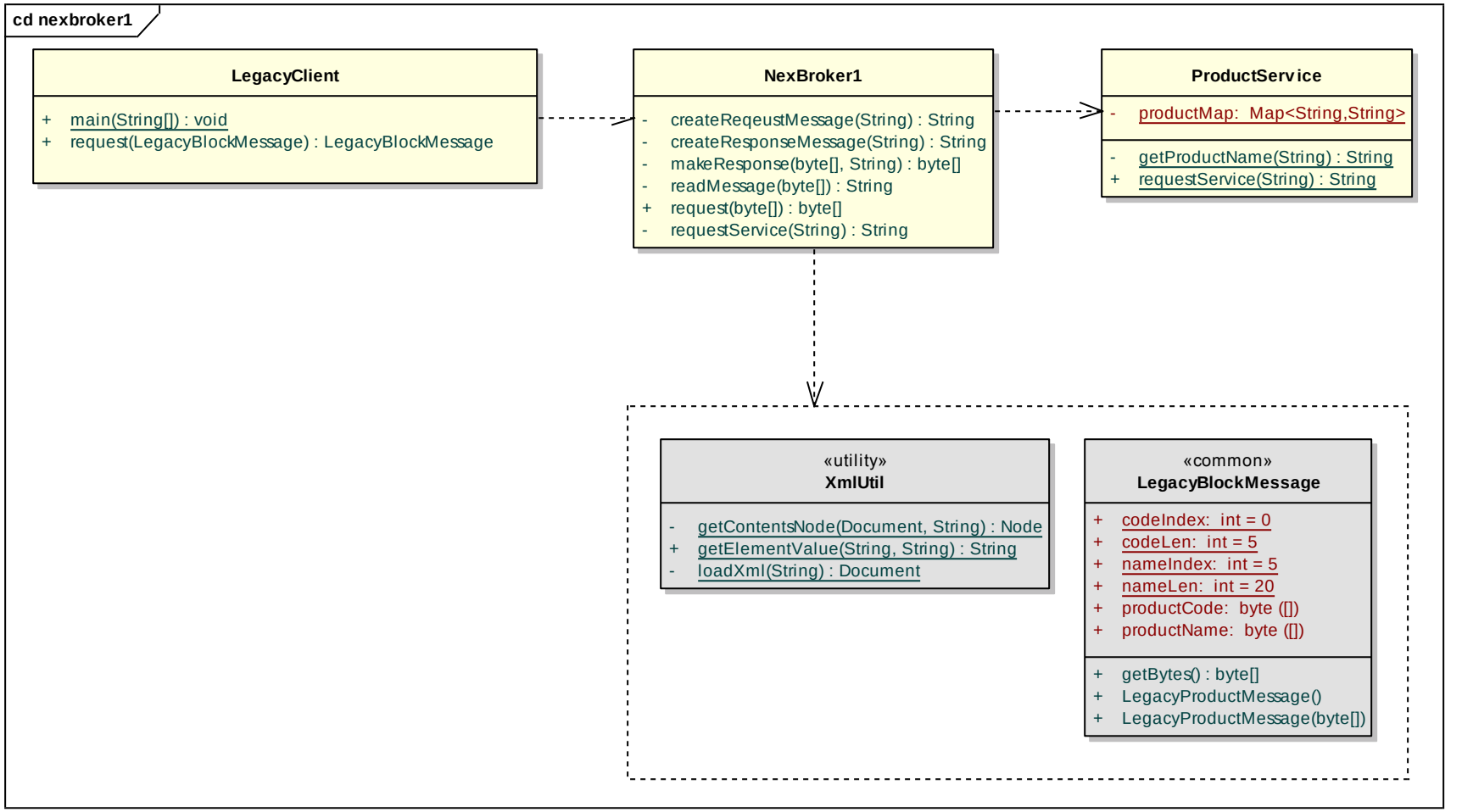
3. 설계 – 객체 행위 설계

✓ 분석 수준의 행위 모델을 설계 수준으로 끌어내림



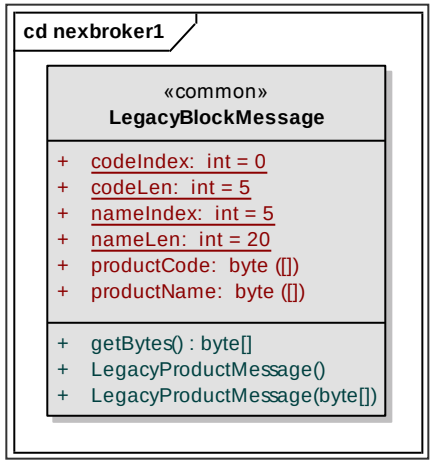
3. 설계 – 객체 관계 설계

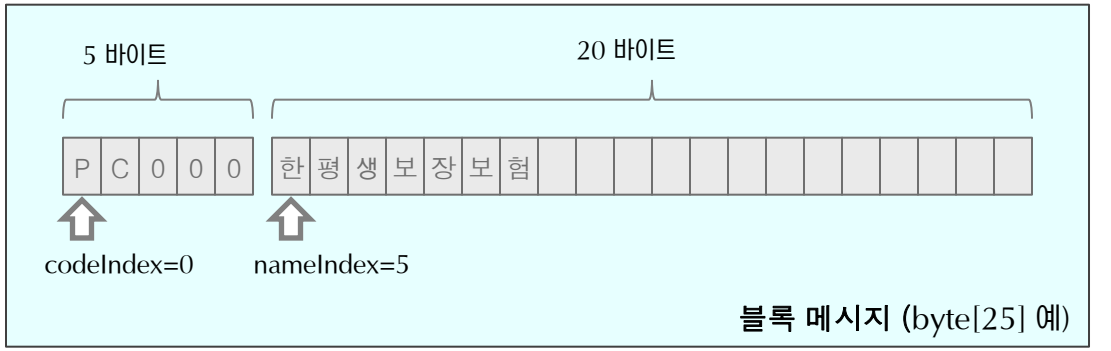
✓ 분석 수준의 클래스 다이어그램을 설계 수준으로 끌어 내림



4. 구현 – 블록 메시지 (1/2)

- ✓ C, 비주얼베이직, 델파이 등으로 작성된 레거시 클라이언트는 TCP 소켓을 사용
- ✓ 소켓을 통해 바이트 열로 구성된 전문을 주고 받음
- ✓ 블록전문은 바이트열의 상대적 위치(offset) 값으로 구분함
- ✓ NexBroker1 개발에 사용할 전문은 두 가지 정보를 포함하는 간단한 것임





블록 메시지 (byte[25] 예)

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
<product-code>PC001</product-code>
<product-name>한평생보장보험</product-name>
</message>
```

Xml 메시지

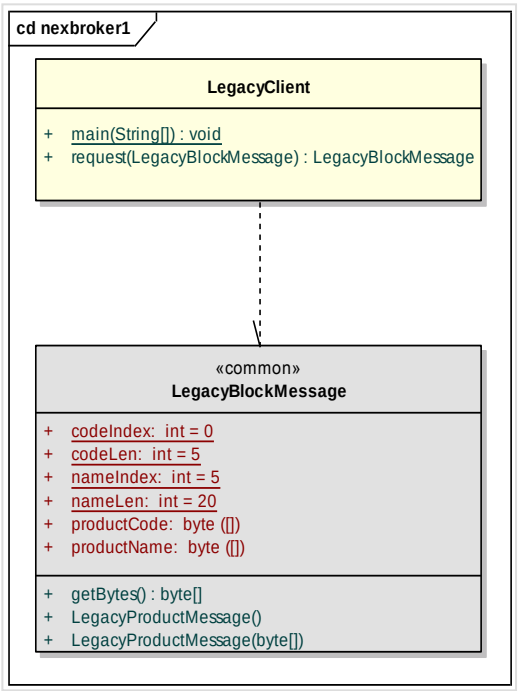
4. 구현 – 블록 메시지 (2/2)

- ✓ 개발의 편의를 위해 블록메시지 객체를 설계한다.
  - C 언어의 struct와 같은 역할
  - 인덱스와 오프셋 정보 포함(OO)
  - 필드는 두 개로 고정(첫걸음)
- ✓ LegacyBlockMessage 의 기능
  - 바이트 메시지로 초기화
  - 메시지 값을 바이트로 리턴함

```
public class LegacyBlockMessage {
	public static int codeIndex = 0;
	public static int codeLen = 5;
	public static int nameIndex = 5;
	public static int nameLen = 20;
	public byte[] productCode;
	public byte[] productName;

	public LegacyBlockMessage(byte[] byteMessage) {
		...
		this.productCode = new byte[codeLen];
		for (int i=0; i<codeLen; i++) {
			productCode[i] = byteMessage[i];
		}
		this.productName = new byte[nameLen];
		for (int i=0; i<nameLen; i++) {
			productName[i] = byteMessage[i+codeLen];
		}
	}

	public byte[] getBytes() {
		byte[] allMessageBytes = new byte[codeLen + nameLen];
		for (int i=0; i<codeLen; i++) {
			allMessageBytes[i] = productCode[i];
		}
		for (int i=0; i<nameLen; i++) {
			allMessageBytes[i+codeLen] = productName[i];
		}
		return allMessageBytes;
	}
}
```

4. 구현 – 레거시 클라이언트 코드

✓ C, 비주얼베이직, 델파이 등으로 개발된 레거시 애플리케이션을 Java로 시뮬레이션 함



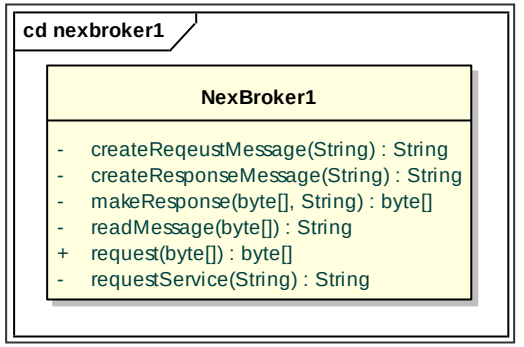

```
public class LegacyClient {

	public static void main(String args[]) {
		int count = 2;
		LegacyBlockMessage responseMessage = null;
		for (int i=0; i<count; i++) {
			LegacyClient client = new LegacyClient();
			String productCode = "PC00" + (i+1);
			LegacyBlockMessage legacyMessage = new LegacyBlockMessage();
			byte[] codeBytes = productCode.getBytes();
			for (int j=0; j<LegacyBlockMessage.codeLen; j++) {
				legacyMessage.productCode[j] = codeBytes[j];
			}
			responseMessage = client.request(legacyMessage);
			String productName = responseMessage. productName;
			System.out.println("-----------------------------------------------");
			System.out.println("서비스결과: 상품명 - " + (productName));
			System.out.println("-----------------------------------------------");
		}
	}

	public LegacyBlockMessage request(LegacyBlockMessage legacyMessage) {
		NexBroker1 nexBroker = new NexBroker1();
		byte[] responseBytes = nexBroker.request(legacyMessage.getBytes());

		return (new LegacyBlockMessage(responseBytes));
	}
}
```

4. 구현 - NexBroker1: 컨트롤 영역

NEXTREE

✓ NexBroker1은 여러가지 일(task)을 수행함

- 블록메시지 읽기, 요청 Xml 메시지 생성, 서비스 요청, 응답 Xml 메시지 분석...
- 모든 메소드는 한 가지 태스크만 수행함, 아래 request는 제어(control) 작업을 수행함




```
package com.nextree.nexbroker1;

public class NexBroker1 {

    public byte[] request(byte[] byteMessage) {

        // 요청 수신 메시지 읽기
        String productCode = this.readMessage(byteMessage);

        // 요청 xml 메시지 생성
        String requestXmlStr = this.createReqeustMessage(productCode);

        // 서비스 요청
        String responseXmlStr = this.requestService(requestXmlStr);

        // 응답 xml 메시지 분석
        String productName = this.createResponseMessage(responseXmlStr);

        // 응답 송신 메시지 쓰기
        byte[] responseBytes = this.makeResponse(byteMessage, productName);

        return responseBytes;
    }
    ....
}
```

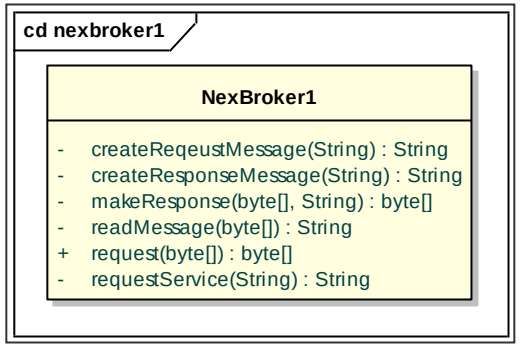
메소드 이름: LegacyClient가 NexBroker1에게 서비스를 요청(request)한다.

제어 작업

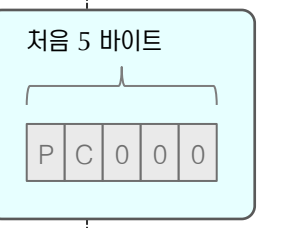
4. 구현 - NexBroker1: 컨트롤 영역

✓ 메시지 읽기(readMessage) 태스크

- 서비스 요청은 다양한 프로토콜과 전문을 통해 올 수 있으므로 읽기 자체가 주요작업임
- 100미터 수준에서는 읽기 다양성을 고려하지 않음, 지정 바이트 메시지 읽기에 충실함.



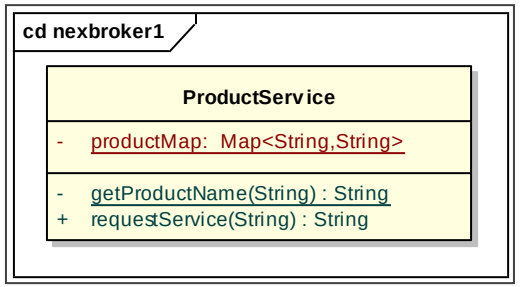

```
package com.nextree.nexbroker1;

public class NexBroker1 {
....
    private String readRequestMessage (byte[] byteMessage) {
                int codeLen = LegacyBlockMessage.codeLen;
                byte[] productCode = new byte[codeLen];
                for (int i=0; i<codeLen; i++) {
                            productCode[i] = byteMessage[i];
                }

                return (new String(productCode));
    }
    private String createReqeustMessage (String productCode) {
                StringBuilder builder = new StringBuilder();
                builder.append("<message>");
                builder.append("<product-code>");
                builder.append(productCode);
                builder.append("</product-code>");
                builder.append("<product-name>");
                builder.append("");
                builder.append("</product-name>");
                builder.append("</message>");
                return builder.toString();
    }
    ....
}
```



4. 구현 – 서비스 서버

✓ 더미 서비스 클래스

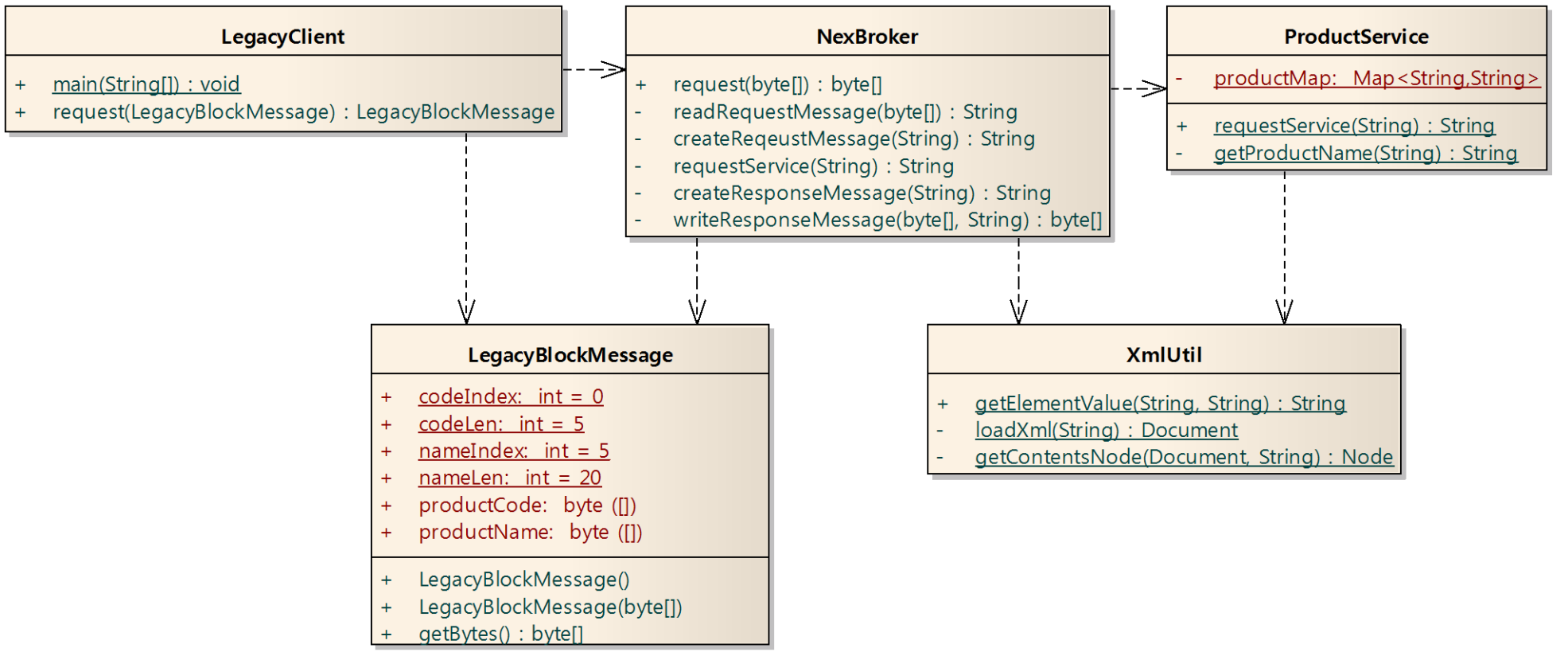
- Xml로 요청을 받고, 처리 결과를 Xml로 응답함.
- [토론]이 수준에서 예외처리를 해야 하는가?




```
public class ProductService {
    private static Map<String,String> productMap;
    static {
            productMap = new HashMap<String,String>();
            productMap.put("PC001", "가족행복보험");
            productMap.put("PC002", "성장기 건강보험");
            ...
    }
    public static String requestService (String xmlRequestStr) {
            String productCode = XmlUtil.getElementValue(xmlRequestStr, "product-code");
            String productName = getProductName(productCode);
            StringBuilder builder = new StringBuilder();
            builder.append("<message>");
            builder.append("<product-code>");
            ...
            builder.append("</message>");
            return builder.toString();
    }
    private static String getProductName(String productCode) {
            String productName = productMap.get(productCode);
            if (productName == null) {
                productName = "상품없음";
            }
            return productName;
    }
}
```

4. 구현 – 서비스 서버

- ✓ 아래 모델을 구현합니다. (LegacyBlockMessage, XmlUtil 소스는 제시함)
- ✓ 자바 패키지: 자유롭게 구성



5. 실행

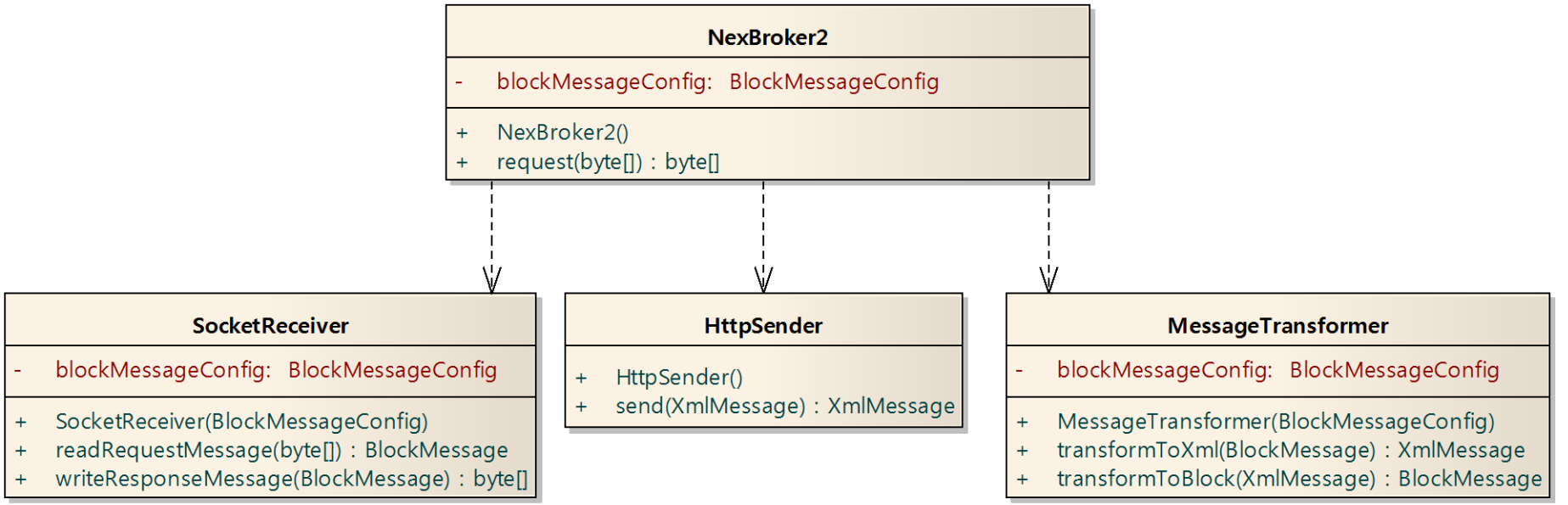
- ✓ LegacyClient를 실행함
- ✓ 참고
  - NexBroker1이 모든 일을 수행함
  - 협업을 통한 문제해결이 필요한 부분
- ✓ 서비스 실행 결과

```
----------------------------------------
서비스결과: 상품명 - 가족행복보험
----------------------------------------
----------------------------------------
서비스결과: 상품명 - 성장기 건강보험
----------------------------------------
```

6. 개선 - 구조

✓ 협업을 통한 작업수행하도록 개선합니다.

- NexBroker1이 수행하는 작업을 나누어서 수행함
- NexBroker2로 이름 변경하고 패키지이름 역시 변경함

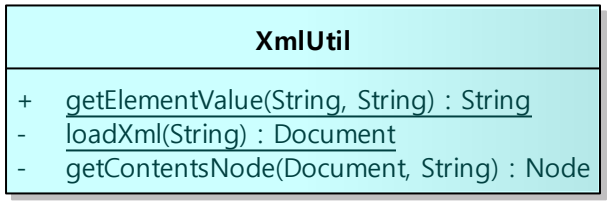


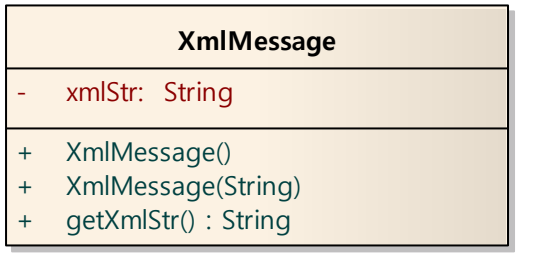
6. 개선 – common 패키지

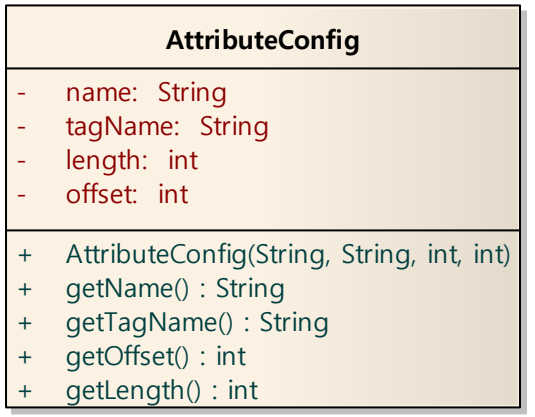
NEXTREE

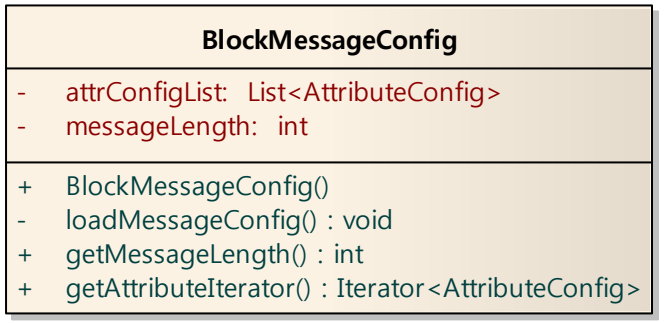
✓ common 패키지

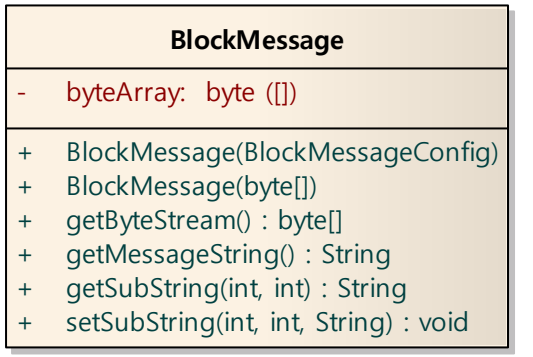
- 아래 클래스 정보는 모드 common 패키지에 소속됨
- 블록 메시지를 보다 자연스럽게 처리하기 위해 개선한 클래스들
- 개발하지 않음, common 패키지의 모든 파일 제시함







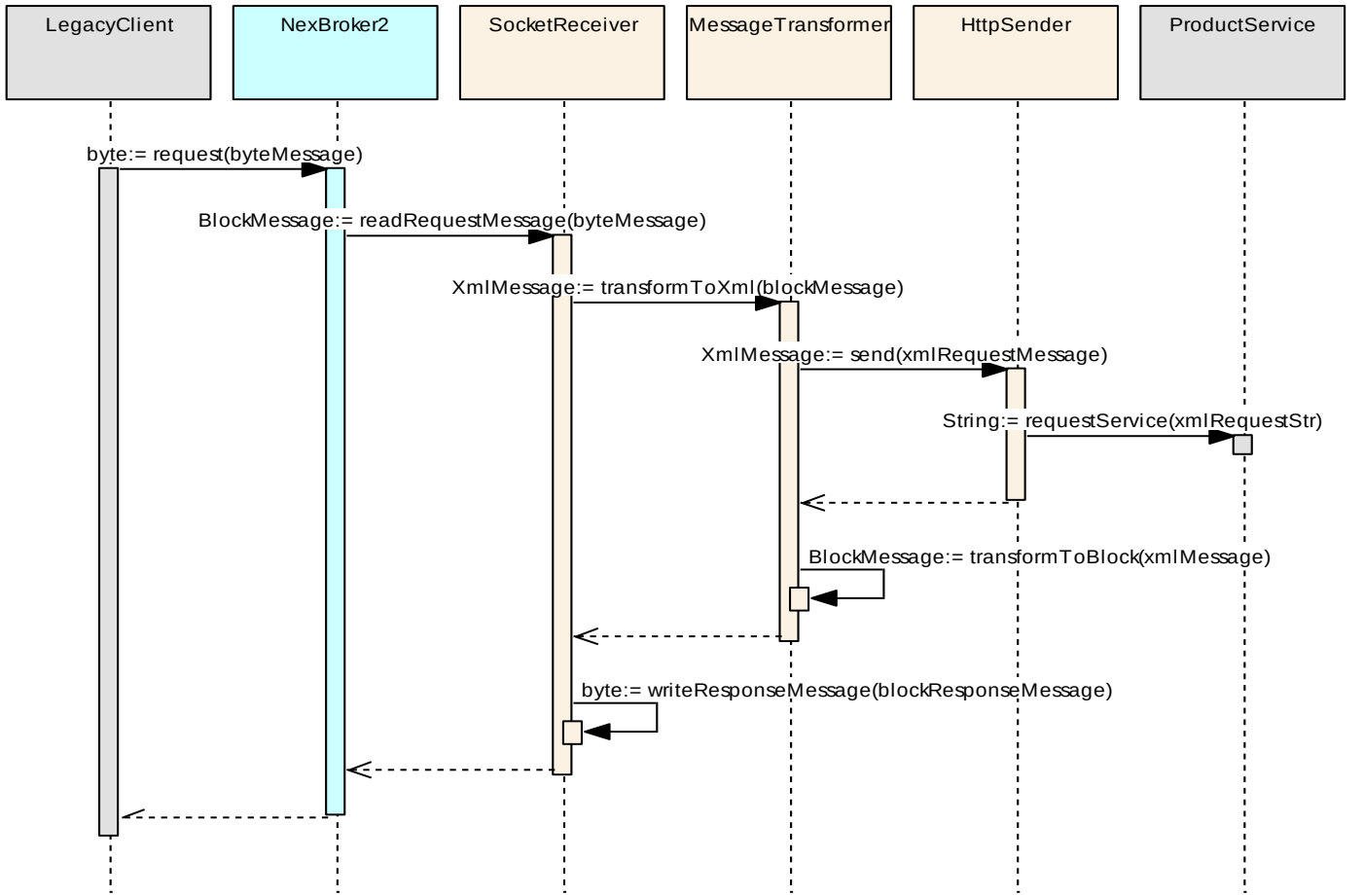




6. 개선 – 행위

✓ 시퀀스 다이어그램

- 새로 등장한 객체들과 협업하는 모습



6. 개선 – 행위

NEXTREE

✓ NexBroker2의 request() 메소드

```
public byte[] request(byte[] byteMessage) {

        // [가상] 소켓을 읽은 후 블럭메시지 리턴
        SocketReceiver socketReceiver = new SocketReceiver(this.blockMessageConfig);
        BlockMessage blockRequestMessage = socketReceiver.readRequestMessage(byteMessage);

        // 블록메시지를 Xml 메시지로 변환
        MessageTransformer transformer = new MessageTransformer(this.blockMessageConfig);
        XmlMessage xmlRequestMessage = transformer.transformToXml(blockRequestMessage);

        // 서비스 서버로 서비스 요청
        HttpSender httpSender = new HttpSender();
        XmlMessage xmlResponseMessage = httpSender.send(xmlRequestMessage);

        // Xml메시지를  블록 메시지로 변환
        BlockMessage blockResponseMessage = transformer.transformToBlock(xmlResponseMessage);

        // [가상] 소켓으로 전송
        byte[] responseBytes = socketReceiver.writeResponseMessage(blockResponseMessage);

        return responseBytes;
}
```

7. 질의 응답 및 토론

- ✓ 질의 응답
- ✓ 토론

❖넥스트리소프트(주)
❖송태국 (tsong@nextree.co.kr)
❖부사장/CTO

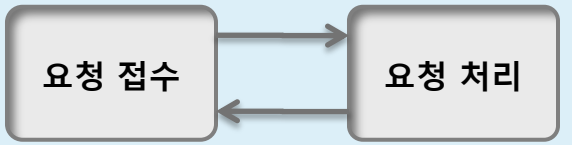
아키텍처 설계 - 목차

1. 시스템 특성 정의
2. Reactor 패턴 이해
3. 구조 설계
4. 구현
5. 질의응답 및 토론

1. 시스템 특성 정의 - 개요

✓ NexBroker 의 시스템 특성을 결정하기 위한 아키텍트의 직관으로 추상화 시행

핵심-외부 요청 접수 및 처리



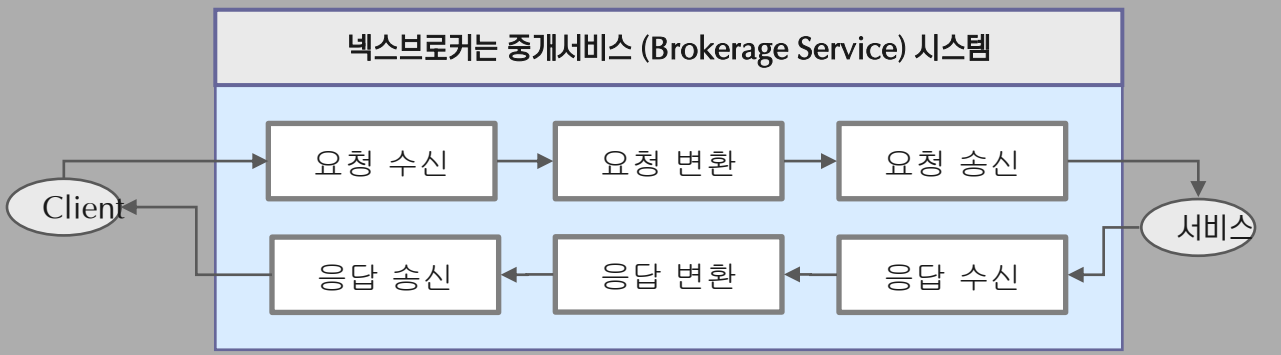
이벤트를 접수하는 일과 접수한 이벤트를 처리하는 일로 크게 작업을 나눈다. 이러한 구조를 다루는 아키텍처 패턴은 Reactor (POSA) 이다. 시스템을 요청을 접수하는 모듈과 요청을 처리하는 모듈로 나눈다.

추상화 방향

아주 먼 곳에서 보았을 때 어떻게 보이는가?

핵심 - 연속적인 작업 처리

수신, 변환, 송신 등의 일련의 작업으로 구성한다. 처리와 전달의 연속이다. 이런 구조를 다루는 아키텍처 패턴은 필터-파이프이다. 필터가 처리하며 처리 결과는 파이프가 다음 필터로 전송한다. 필터는 처리 결과가 어느 필터로 가는 지 알 필요가 없다.

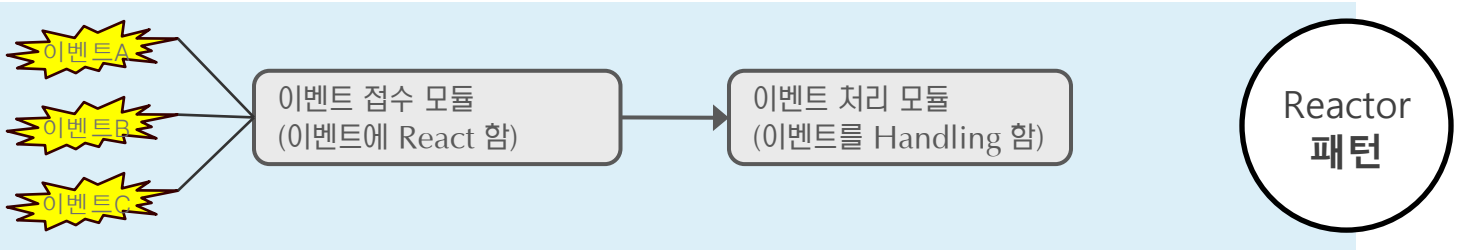


1. 시스템 특성 정의 - 추상화

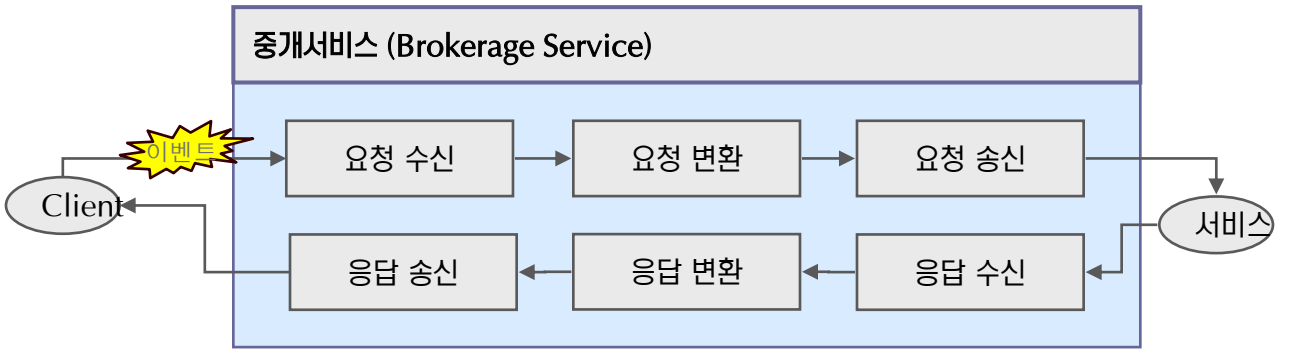
✓ 추상화를 통해 살펴 본 NexBroker의 특성 또는 본질

- 추상화 방향: 외부에서 발생한 이벤트를 중심으로 시스템을 바라 봄

추상화 결론: NexBroker는 외부의 이벤트를 접수하고 처리하는 시스템이다.



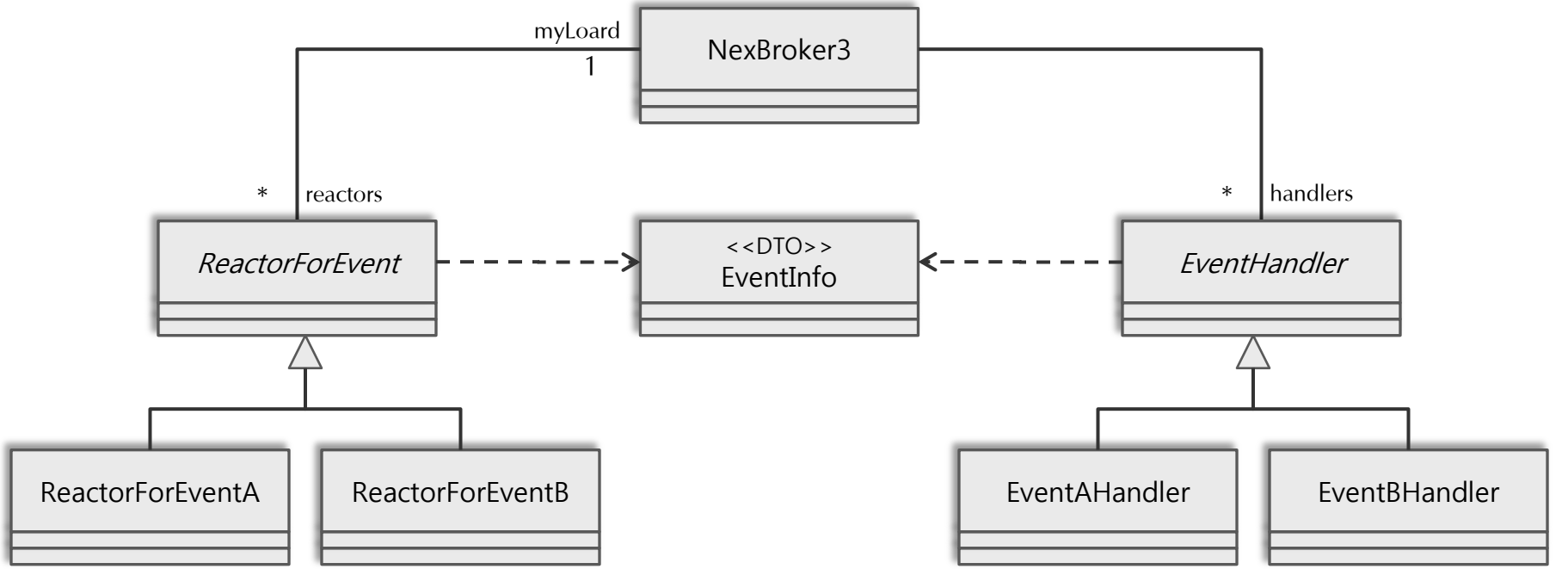
1. 외부에 클라이언트로부터 서비스 요청 이벤트가 TCP 포트에서 발생하면,
2. 포트로 들어온 메시지를 포트로부터 요청 수신, 요청 Xml 변환, HTTP로 요청 송신,
응답변환, 응답 소켓포트로 송신하는 일련의 태스크를 수행함.



1. 시스템 특성 정의 - 기본구조

✓ Reactor 중심의 이벤트 핸들링 구조

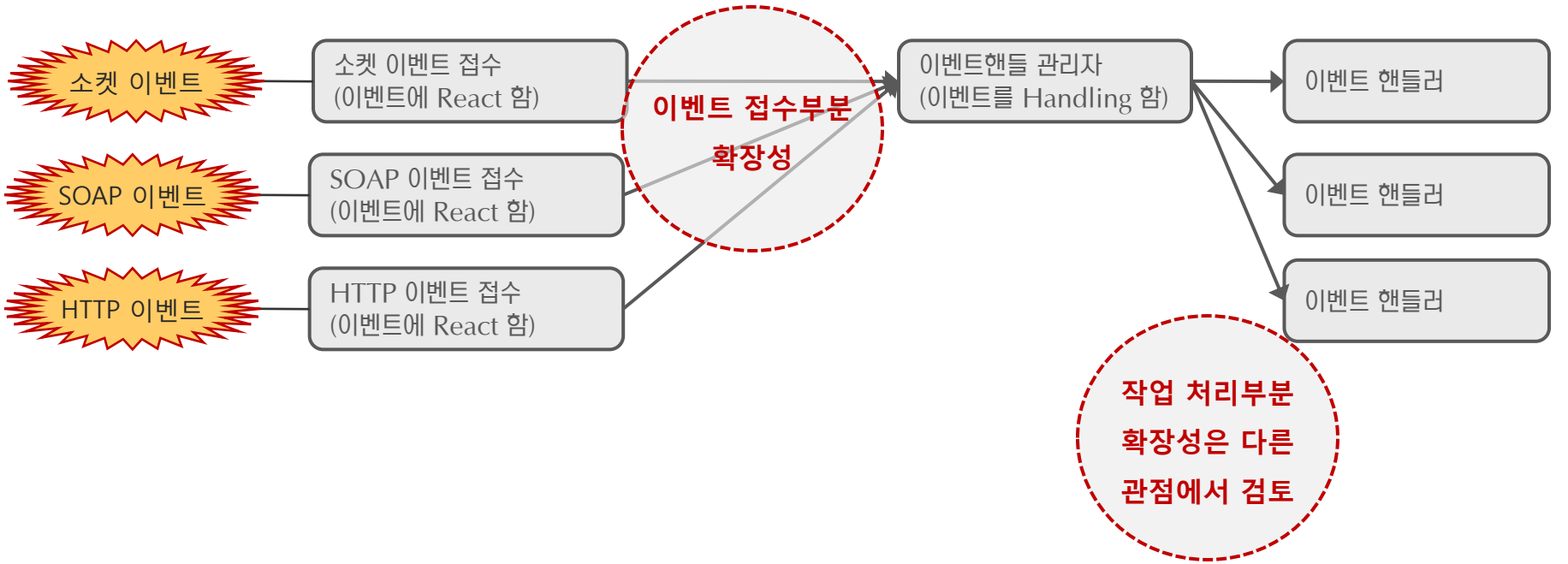
- EventA 가 발생하면 ReactorForEventA는 결과를 NexBroker3에게 보고를 함
- NexBroker3는 EventA를 위한 핸들러인 EventAHandler를 찾아서 결과를 전달함



1. 시스템 특성 정의 - 검토

✓ 추상화 방향의 적절성을 위한 확장 검토

- 이벤트를 접수하고 이벤트를 처리하는 구조와 관련있는 품질속성은 확장성임
- 이벤트 접수-이벤트 처리라는 기본 구조는 프로토콜 확장성을 가짐
- 향후 다양한 클라이언트 요청 이벤트에 대한 구조적인 확장성 있음
- 작업처리 부분에 대한 확장은 이벤트 핸들러 내부 구조의 이슈에 해당함

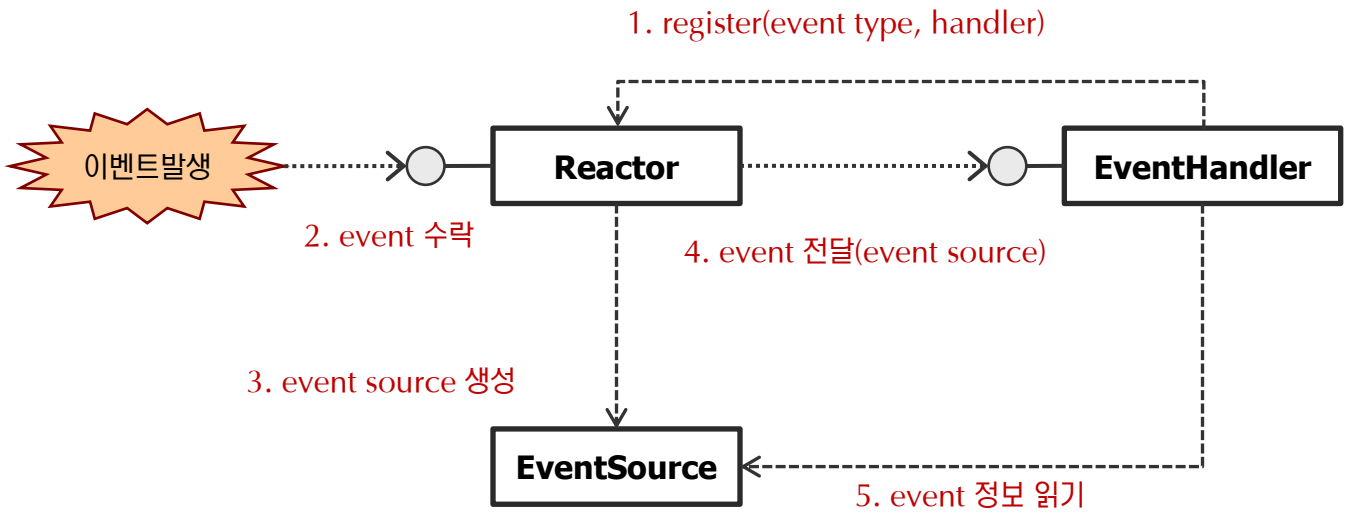


2. Reactor 패턴 이해

NEXTREE

✓ 리액터 패턴 동작구조

- 핸들러 처리가 길어질 경우 블로킹을 줄이기 위해 리액터가 멀티스레드 객체여야 함
- 등록 -> 이벤트 발생 -> 수락 -> 전달 흐름을 가짐

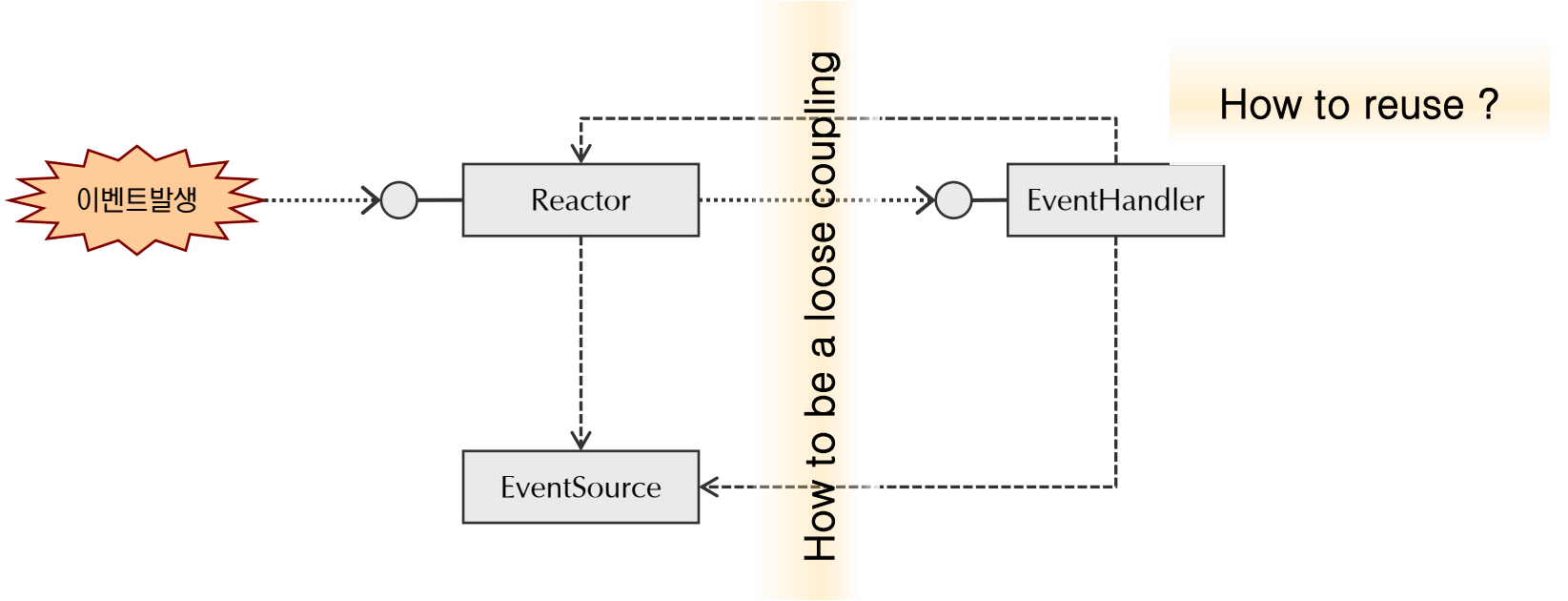


2. Reactor 패턴 이해

NEXTREE

✓ 리액터 패턴 관련 패턴

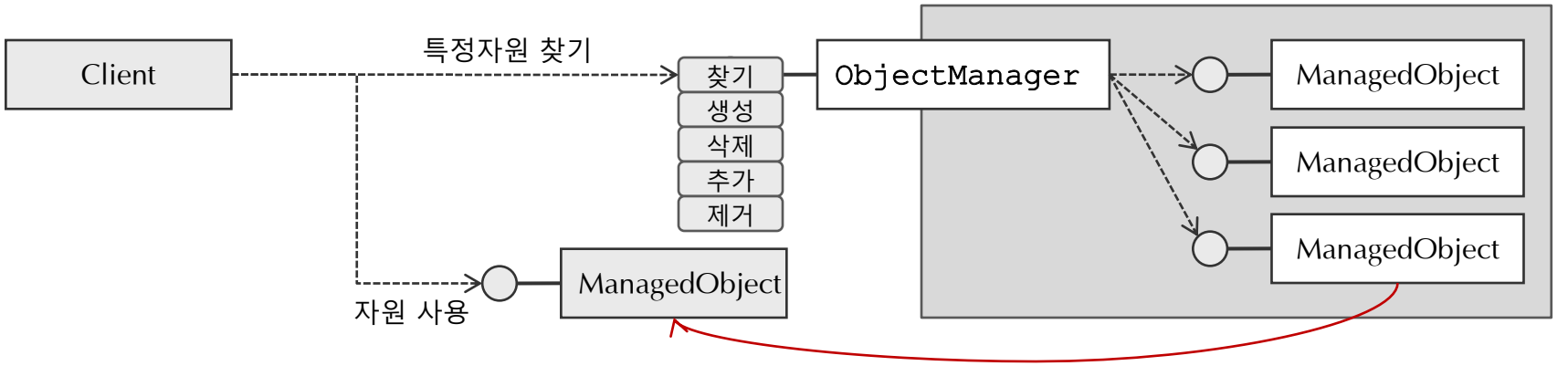
- 다수의 핸들러 객체를 다루어야 함 – Object Manager
- EventSource와의 인터페이스를 단순화 함 – Wrapper Facade
- EventHandler 인터페이스 단순화 함 – Explicit Interface



2. Reactor 패턴 이해

✓ Reactor 관련 패턴들 – Object Manager

- 한 번 사용한 핸들러 객체를 버리지 않고 재사용하는 것이 성능에 유리함
- 핸들러 객체의 라이프사이클 관리와 접근통제를 위한 전문가 객체가 필요함
- ManagedObject를 Pool 이나 Cache에 저장할 수 있음

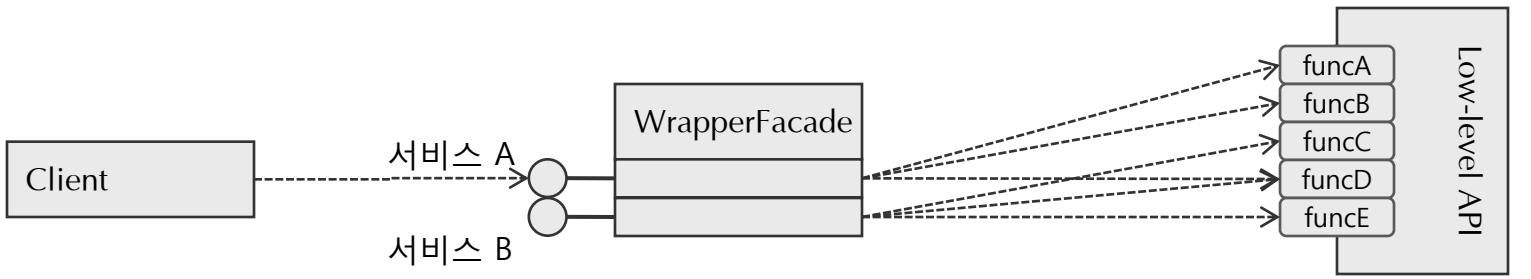


2. Reactor 패턴 이해

NEXTREE

✓ Reactor 관련 패턴들 – Wrapper Facade

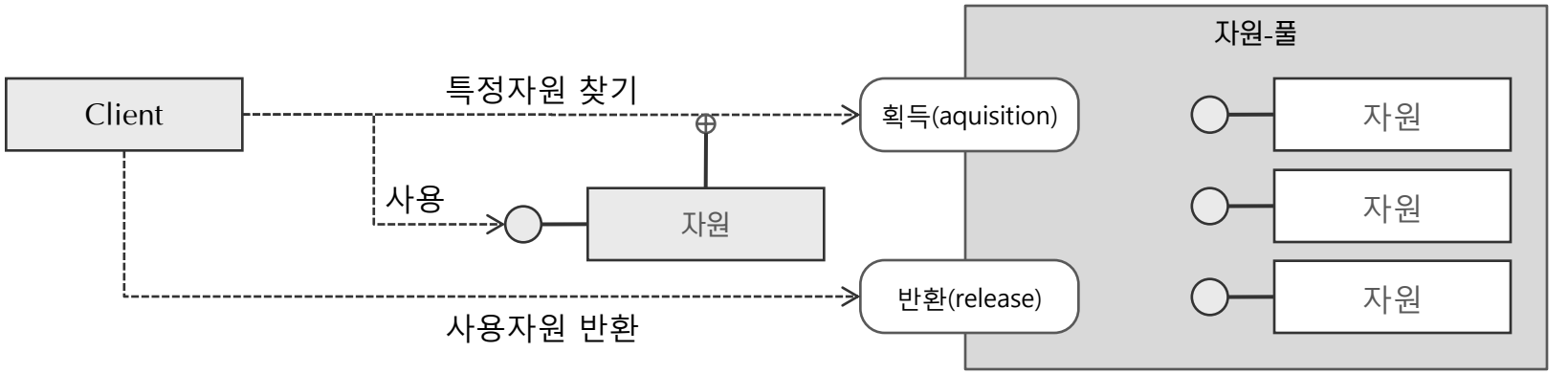
- 복잡한 세부 API 접근은 loose coupling 원칙을 깸
- 그 결과로 이식성(portability) 및 유지보수성이 떨어짐
- 응집도 높은(cohesive) WrapperFacade는 두 영역 간의 정보개방을 최소화 하여줌



2. Reactor 패턴 이해

✓ Reactor 관련 패턴들 – Resource Pool

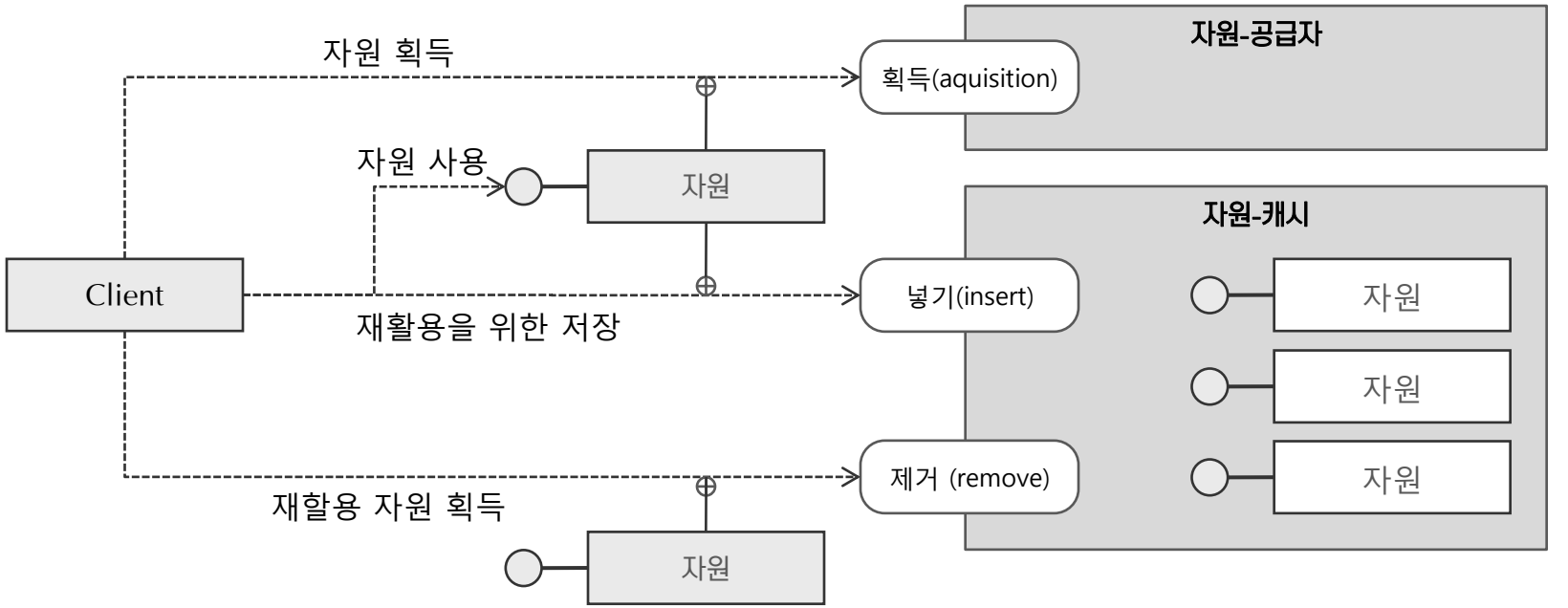
- 제한된 자원에 대한 신속한 획득과 해제가 필요함
- 예, Connection Pool, Service Pool
- 성능과 범위성이 중요한 애플리케이션은 효율적이고 예측가능한 자원 접근이 필요함
- 자원을 반복적으로 생성하기 보다는 메모리 상의 자원-풀에 자원의 개수를 유지함
- 필요할 경우 자원-풀을 찾아서 가져오고, 다 사용한 후 반환함
- 자원 획득,접근,관리에 대한 지식을 클라이언트에게 숨김



2. Reactor 패턴 이해

✓ Reactor 관련 패턴들 – Resource Cache

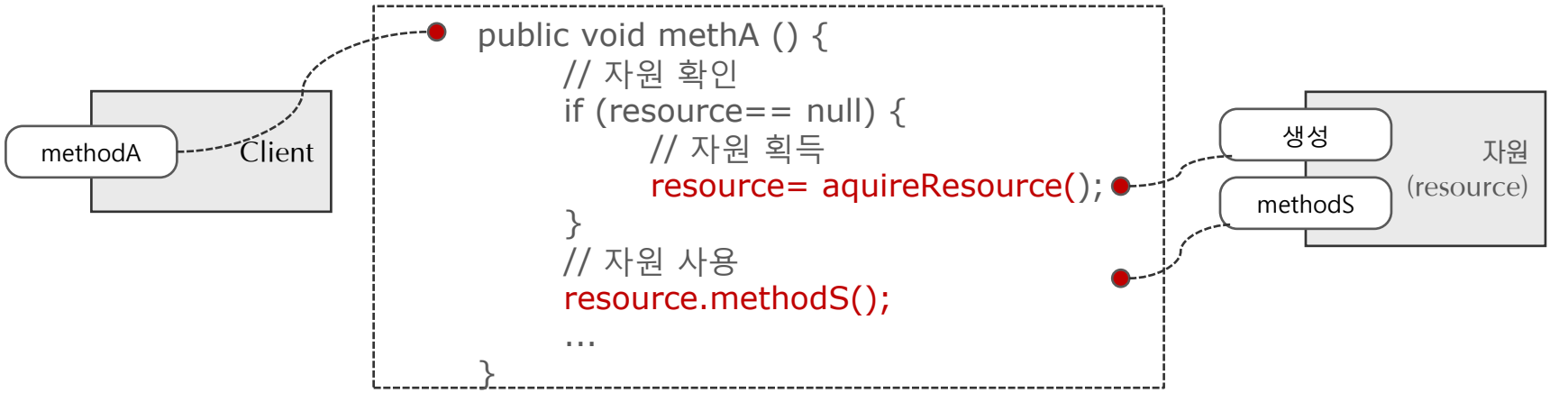
- 동일한 자원에 반복적으로 접근하는 자원에 대한 초기화, 폐기 비용을 최소화 해야함
- 캐시는 이미 획득한 자원에 대한 재활용임
- 캐시는 시간과 공간의 트레이드-오프임, 공간을 제공하여 시간을 절약하고자 함
- 캐시는 제한된 공간이므로 자원을 어떤 방식으로 제거할 것인가에 대한 전략이 필요함



2. Reactor 패턴 이해

✓ Reactor 관련 패턴들 – Lazy Acquisition

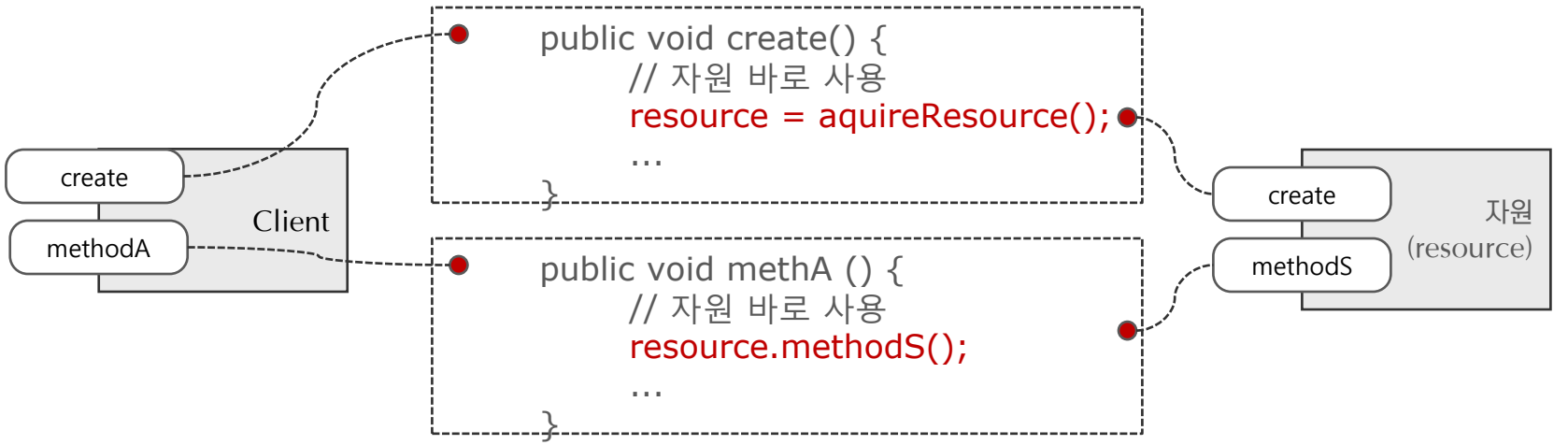
- 객체 생성 및 자원 획득이 높은 처리량과 가용성 요구에 부응해야 함
- 초기 자원 획득(= start-up 또는 초기화)비용을 줄여야 가용성 확보 가능
- 사용하지도 않을 객체에 대한 획득은 자원 낭비임
- 따라서 사용할 시점에 자원을 획득함(내일 할 일을 오늘 걱정하지 말라...)
- 문제점: 자원을 확인하는 절차 필요하고 초기 실행시간 예측이 어려움
- 문제점: Virtual Proxy와 같은 중간 객체나 상태를 나타내는 속성이 필요함



2. Reactor 패턴 이해

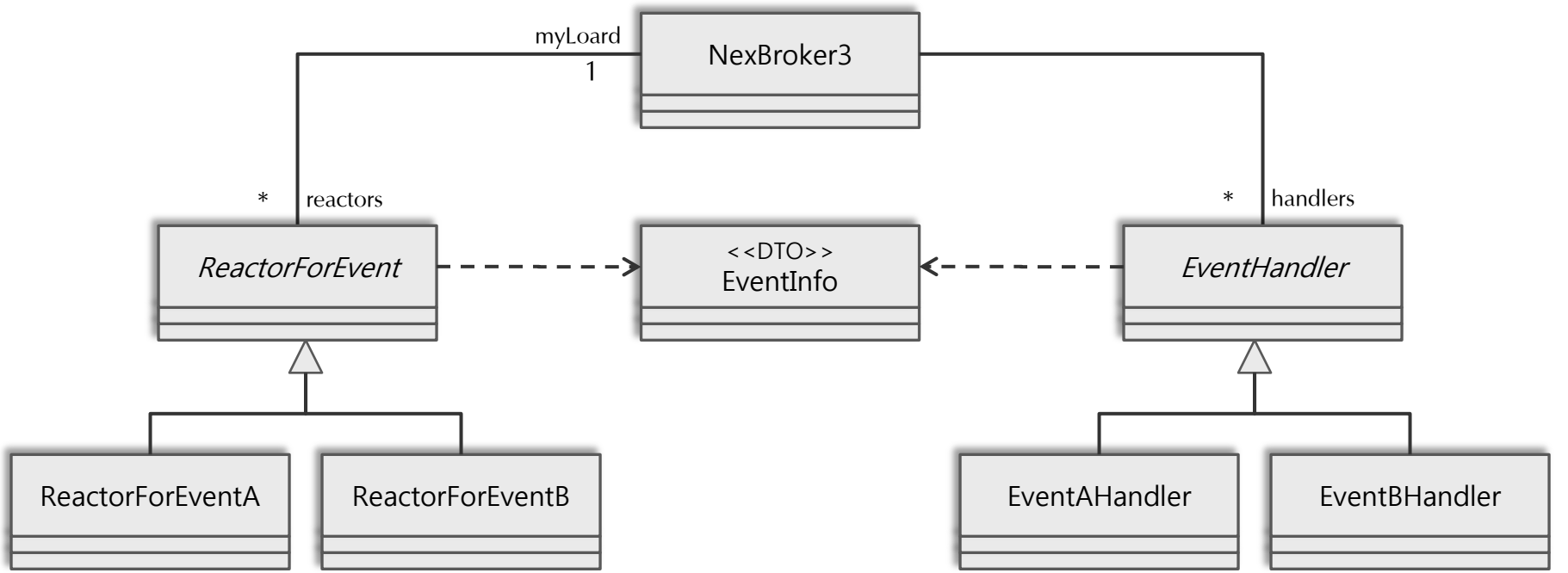
✓ Reactor 관련 패턴들 – Eager Acquisition

- 높은 예측가능성(preditability)과 수행성능이 필요할 경우가 있음
- 실행시간에 높은 자원획득 비용을 지불할 수 없은 경우가 있음
- Lazy Acquisition과 반대상황
- 미리 자원을 모두 준비하고 필요한 상황에서 자원확인 절차 없이 바로 사용함



3. 구조 설계

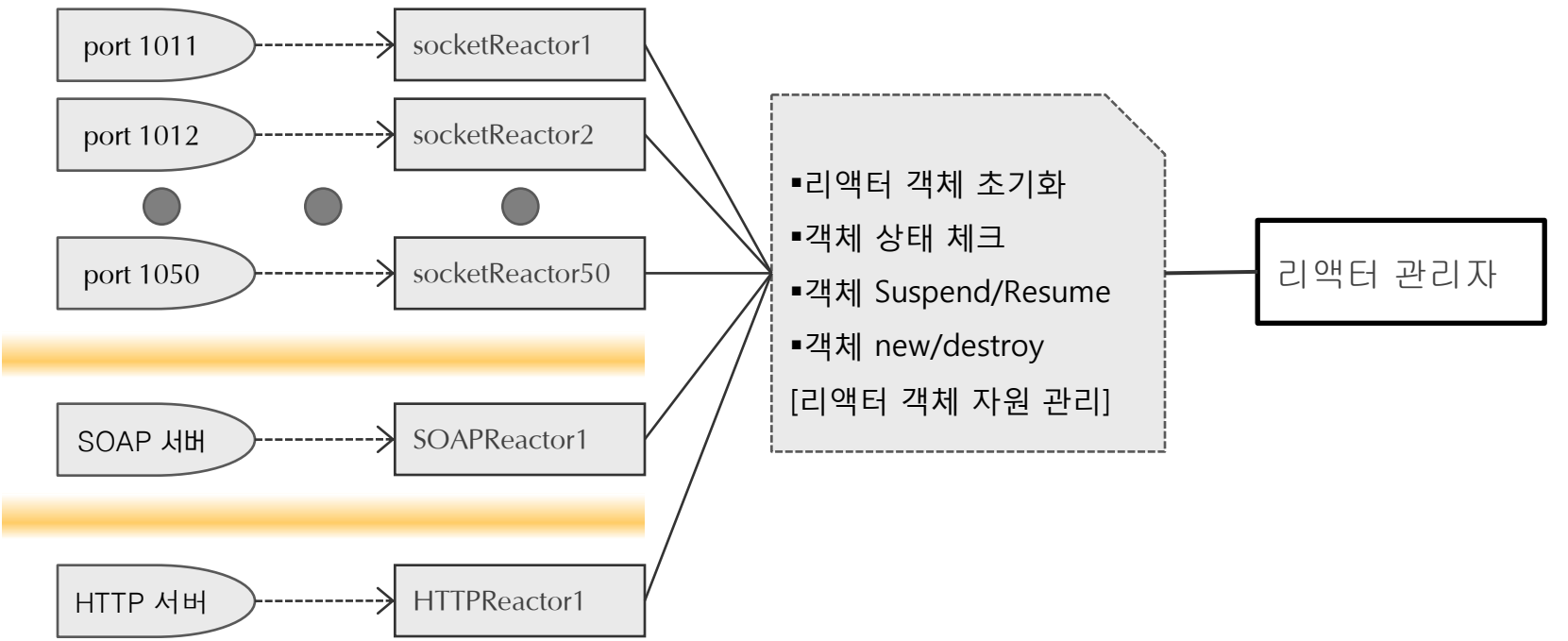
- ✓ Reactor 패턴을 NexBroker의 백본으로 결정함
- ✓ 품질속성 요구를 반영하면서 기본 구조를 점차 개선하여 감



3. 구조 설계

✓ 가용성과 성능에 대한 설계 고려 – 리액터 관리자

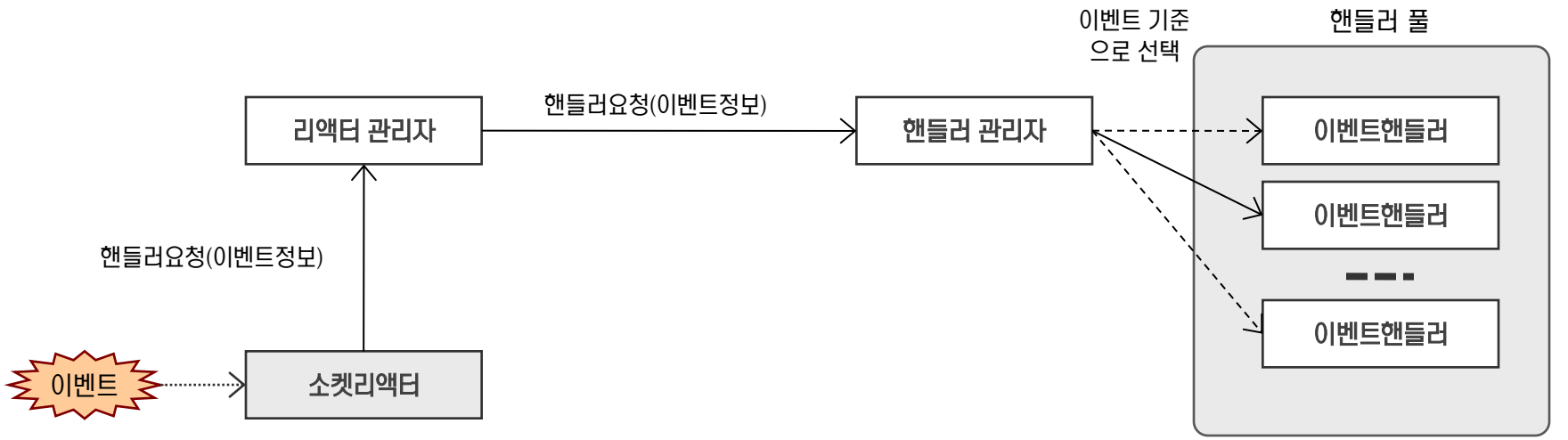
- 다수의 이벤트 소스로 부터 이벤트 발생함
- 효율적인 자원 사용 – 이벤트 리액터 객체생성 최소화
- 리액터 객체 상태 모니터링



3. 구조 설계

✓ 효율적인 자원획득을 위한 설계 고려 – 이벤트 핸들러 관리자

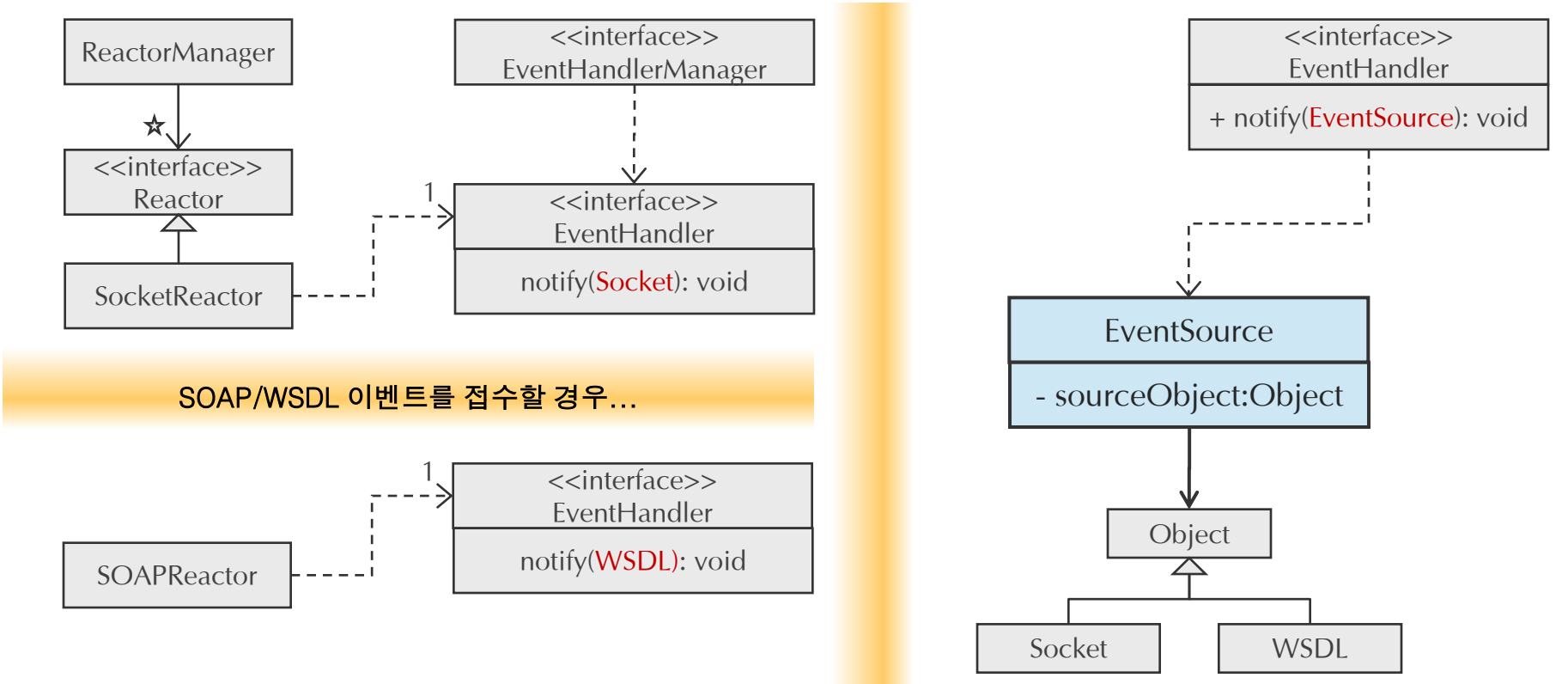
- 리액터로부터 이벤트핸들러(자원) 요청이 있을 경우,
- 이벤트에 해당하는 핸들러를 선택해야 하고
- 한 번 사용한 이벤트핸들러를 재사용해야 하고
- 이벤트 핸들러 사용현황을 관리해야 함 (사용되지 않는 핸들러는 제거해야 함)
- 이벤트 핸들러라는 자원을 효율적으로 관리하기 위한 관리자가 필요함



3. 구조 설계

✓ 확장성 및 Loose coupling을 위한 설계 고려 - 이벤트소스

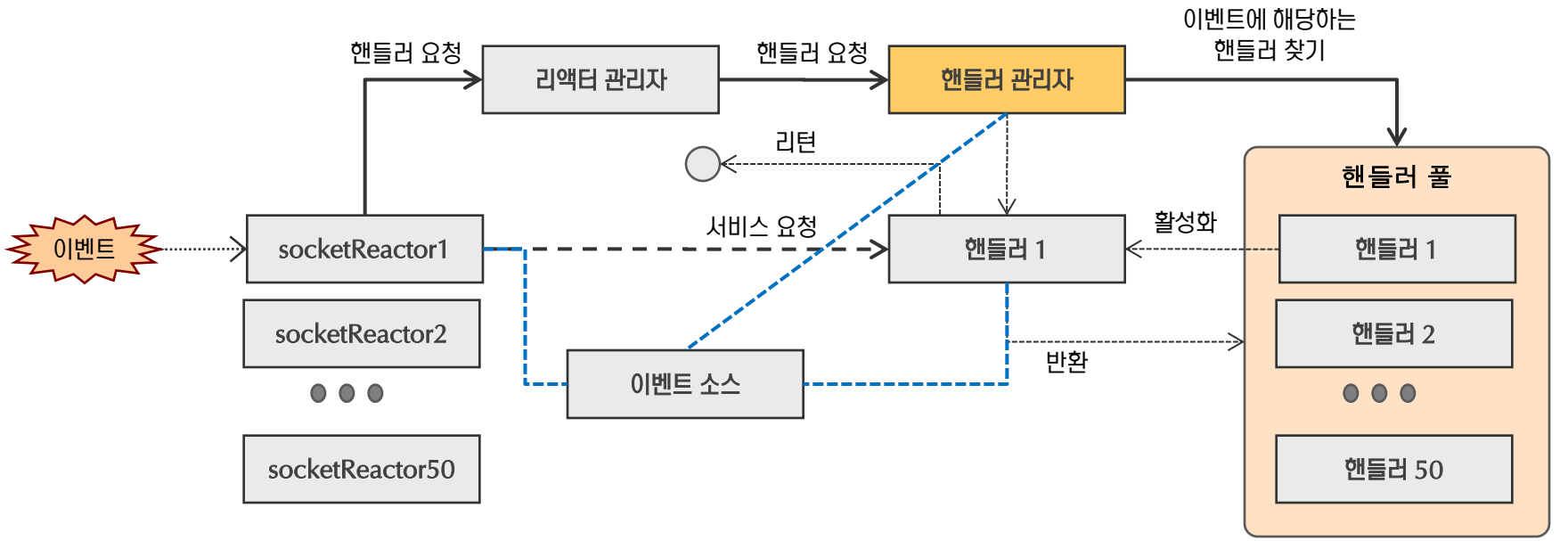
- 소켓 이벤트 외에 추가로 다른 유형의 이벤트가 존재 가능함
- 리액터 영역과 이벤트처리 영역간에 느슨한 결합(loose coupling)이 필요함
- EventSource객체로 다양한 이벤트 객체를 래핑함



3. 구조 설계

✓ 가용성과 성능에 대한 설계 고려 – 리액터 관리자/핸들러 관리자/이벤트소스

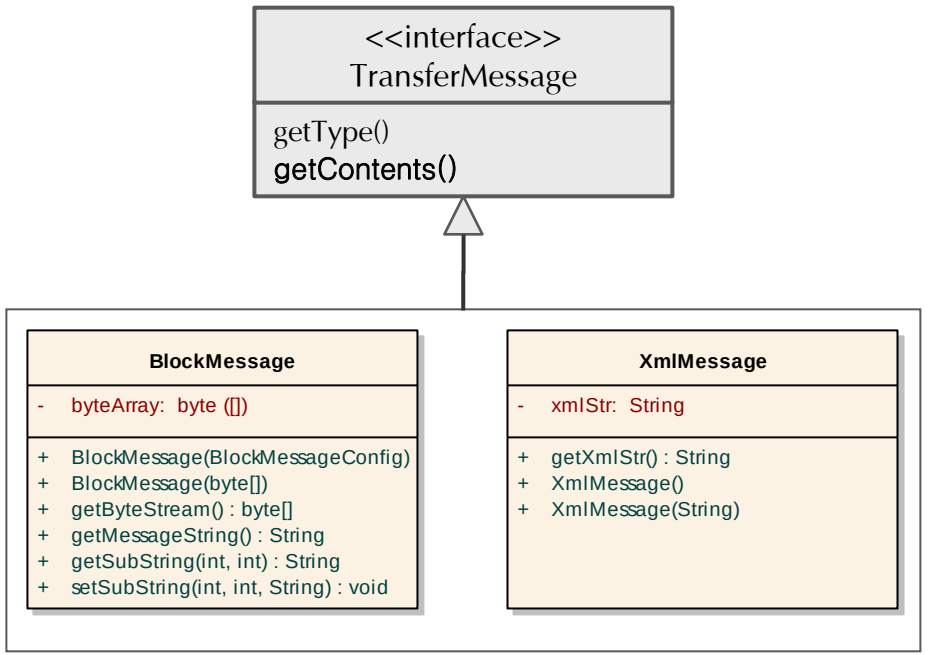
- 다수의 이벤트 소스로 부터 이벤트 핸들링 요청
- 효율적인 자원 사용 – 이벤트 핸들러 객체생성 최소화
- 이벤트 핸들러 객체 상태 모니터링이 필요할 수 있음



3. 구조 설계

✓ 메시지 일반화

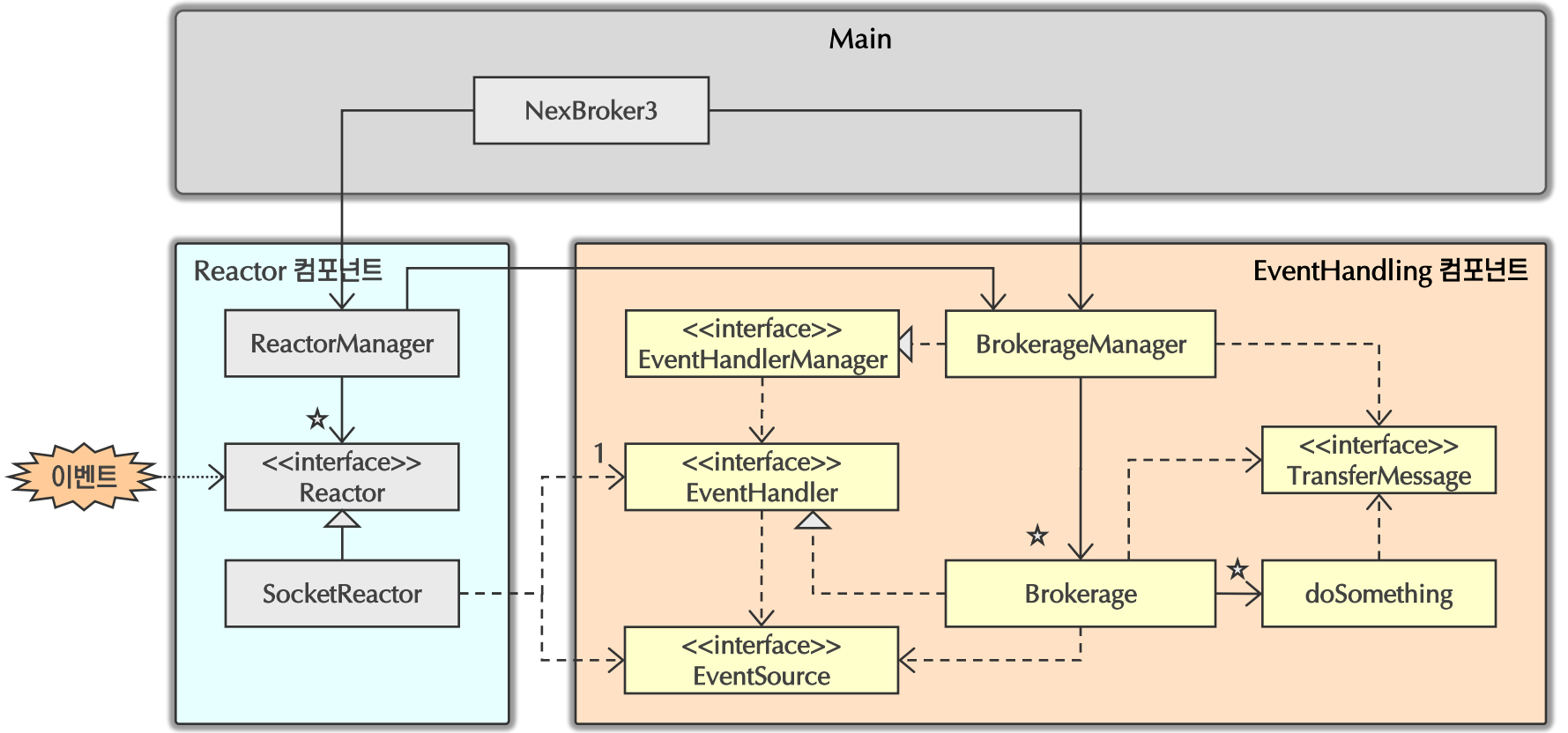
- 블록메시지와 Xml메시지를 묶어서 표현할 수 있다면, 즉 일반화 할 수 있다면 메시지 처리 설계에 유연성을 얻을 수 있음
- 일반화할 때 적절한 이름은 무엇인가?
- 일반화할 때 반드시 가져야 하는 공통 메소드는 무엇인가?
- 일반화를 추상클래스로 할 것인가 아니면 인터페이스로 할 것인가?



3. 구조 설계

✓ 제 2 수준 추상화를 기반으로 NexBroker 구조 설계

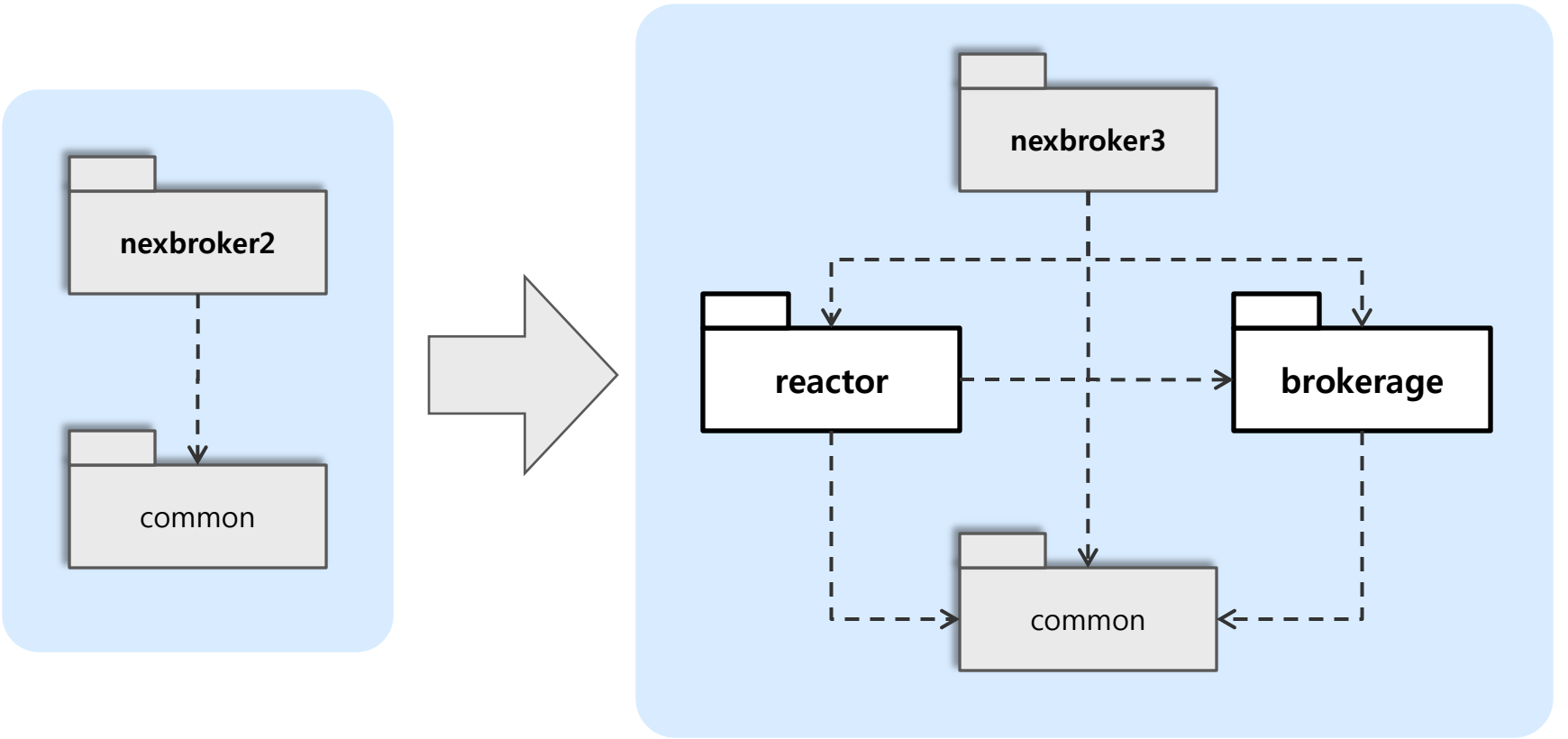
- 한 건의 중개요청 이벤트 처리를 Brokerage(중개) 객체로 표현
- Brokerage는 TransferMessage를 매개변수로 받고 결과로 리턴하는 일련의 TransferTask 들의 연속적인 실행임



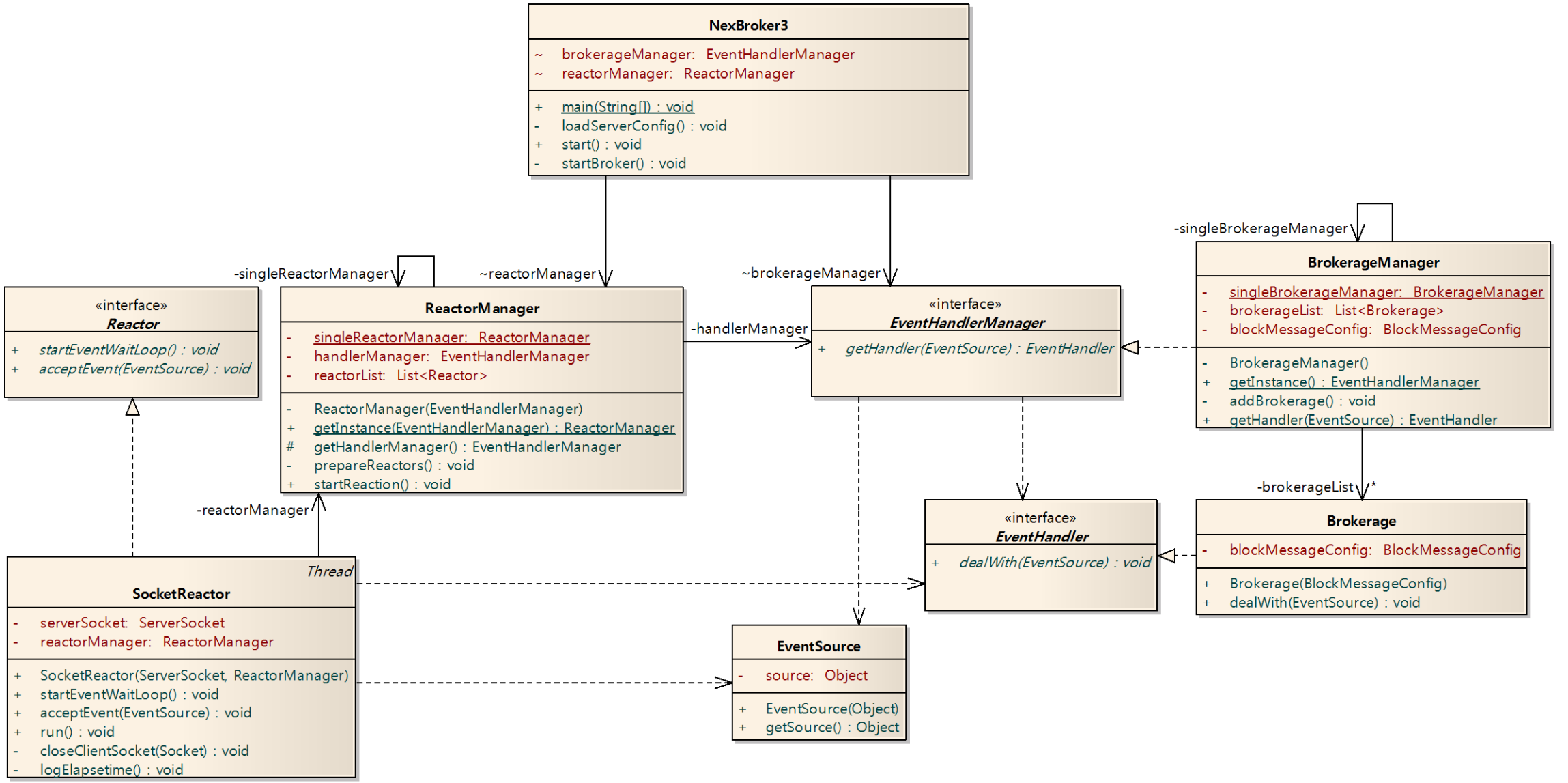
3. 구조 설계

✓ 패키지 설계

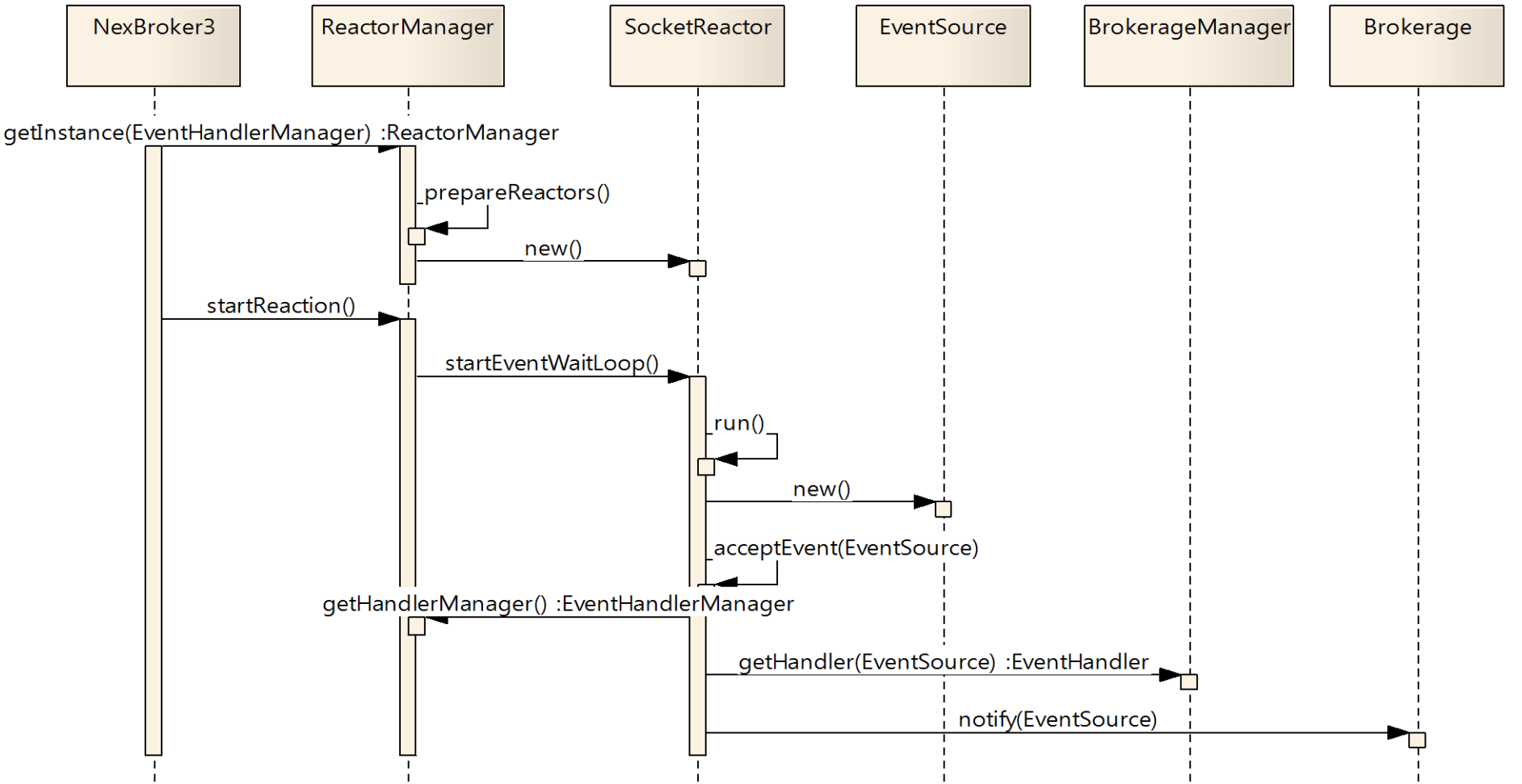
- 구조설계를 반영하여 패키지를 설계함
- 이벤트를 접수하는 객체 들을 담은 reactor 패키지 추가
- 이벤트를 처리하는 객체 들을 담은 brokerage 패키지 추가



4. 구현 – 구조 모델


NexBroker3
~ brokerageManager: EventHandlerManager
~ reactorManager: ReactorManager
+ main(String[]) : void
- loadServerConfig() : void
+ start() : void
- startBroker() : void
-singleReactorManager
~reactorManager
~brokerageManager
-singleBrokerageManager
«interface»
Reactor
+ startEventWaitLoop() : void
+ acceptEvent(EventSource) : void
ReactorManager
- singleReactorManager: ReactorManager
- handlerManager: EventHandlerManager
- reactorList: List<Reactor>
- ReactorManager(EventHandlerManager)
+ getInstance(EventHandlerManager) : ReactorManager
# getHandlerManager() : EventHandlerManager
- prepareReactors() : void
+ startReaction() : void
-handlerManager
«interface»
EventHandlerManager
+ getHandler(EventSource) : EventHandler
BrokerageManager
- singleBrokerageManager: BrokerageManager
- brokerageList: List<Brokerage>
- blockMessageConfig: BlockMessageConfig
- BrokerageManager()
+ getInstance() : EventHandlerManager
- addBrokerage() : void
+ getHandler(EventSource) : EventHandler
-brokerageList
*
-reactorManager
Thread
SocketReactor
- serverSocket: ServerSocket
- reactorManager: ReactorManager
+ SocketReactor(ServerSocket, ReactorManager)
+ startEventWaitLoop() : void
+ acceptEvent(EventSource) : void
+ run() : void
- closeClientSocket(Socket) : void
- logElapsetime() : void
«interface»
EventHandler
+ dealWith(EventSource) : void
Brokerage
- blockMessageConfig: BlockMessageConfig
+ Brokerage(BlockMessageConfig)
+ dealWith(EventSource) : void
EventSource
- source: Object
+ EventSource(Object)
+ getSource() : Object

4. 구현 – 행위 모델


NexBroker3
ReactorManager
SocketReactor
EventSource
BrokerageManager
Brokerage
getInstance(EventHandlerManager) :ReactorManager
prepareReactors()
new()
startReaction()
startEventWaitLoop()
run()
new()
acceptEvent(EventSource)
getHandlerManager() :EventHandlerManager
getHandler(EventSource) :EventHandler
notify(EventSource)

4. 구현 – 소스코드

NEXTREE

✓ SocketReactor

- run()과 acceptEvent() 메소드

```
...
public void acceptEvent(EventSource eventSource) {

   EventHandlerManager handlerManager = reactorManager.getHandlerManager();
   // identify handler
   EventHandler handler = handlerManager.getHandler(eventSource);
   handler.notify(eventSource);
}

public void run() {
      while (true){
         Socket clientSocket = null;
         try {
            synchronized(this.serverSocket) {
               clientSocket = serverSocket.accept();
            }

            this.acceptEvent(new EventSource(clientSocket));

         } catch (SocketException se) {
         // 외부에서 서버소겟을 닫았을 경우 발생함. 서버 소켓이 닫기면 밖으로 빠져나감
         } catch (IOException ex) {
            ...
         } finally {
            closeClientSocket(clientSocket);
            logElapsetime();
         }
      }
   }
```

4. 구현 – 소스코드

NEXTREE

✓ ReactorManager

- prepareReactor(), startReaction() 메소드

```
private static ReactorManager singleReactorManager;
private EventHandlerManager handlerManager;
private List<Reactor> reactorList;
...
private void prepareReactors() {
    int portNum = 1200;// 후에 config 파일로부터 가져옴

    try {
       ServerSocket serverSocket = new ServerSocket(portNum);
       SocketReactor socketReactor = new SocketReactor(serverSocket, this);
       this.reactorList.add(socketReactor);
    } catch (IOException ioe) {
    }
}

public void startReaction() {
    Iterator<Reactor> reactorIter = reactorList.iterator();
    while (reactorIter.hasNext()) {
       Reactor reactor = reactorIter.next();
    reactor.startEventWaitLoop();
    }
}
```

4. 구현 – 소스코드

NEXTREE

✓ Brokerage 클래스

- dealWith() 메소드
- NexBroker2에서 NexBroker2의 처리로직이 Brokerage로 이동함, 구조의 변화에 따른 기능블록 위치의 변화임

```
public void dealWith(EventSource eventSource) {

    // 트랜스퍼 메시지 획득
    SocketWorker socketWorker = new SocketWorker(blockMessageConfig);
    TransferMessage receivedMessage = socketWorker.read(eventSource);

    // 블록메시지를 Xml 메시지로 변환
    MessageTransformer transformer = new MessageTransformer(this.blockMessageConfig);
    XmlMessage xmlRequestMessage = transformer.transformBlockToXml((BlockMessage)receivedMessage);

    // 서비스 서버로 서비스 요청
    HttpSender httpSender = new HttpSender();
    XmlMessage xmlResponseMessage = httpSender.send(xmlRequestMessage);

    // Xml메시지를  블록 메시지로 변환
    BlockMessage blockResponseMessage = transformer.transformXmlToBlock(xmlResponseMessage);

    // 이벤트 소스로 결과 리턴
    socketWorker.write(eventSource, (BlockMessage)blockResponseMessage);
}
```

4. 구현 – 소스코드

NEXTREE

✓ LegacyClient클래스

- request() 메소드 업데이트
- 객체 호출에서 TCP 소켓쓰기/읽기로 변경함
- 포트번호:1200번 고정

```
public void request(LegacyProductMessage legacyMessage) {

 try {
      Socket socket = new Socket("localhost", 1200);
      BufferedWriter bufferedWriter = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));
      BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

      bufferedWriter. write(new String(legacyMessage.getBytes())); // size 20
      bufferedWriter.flush();
      // System.out.println("Sended string: " + (new String(legacyMessage.getBytes())));
      // System.out.print("Received string: ");

      while (!bufferedReader.ready()) {}
      System.out.println("--------------------------------------------");
      System.out.println("서비스결과: 상품명 - " + (bufferedReader.readLine()));
      System.out.println("--------------------------------------------");

      bufferedReader.close();
   } catch(Exception e) {
      System.out.print("Whoops! It didn't work!\n");
   }
}
```

5. 질의 응답 및 토론

- ✓ 질의 응답
- ✓ 토론

❖넥스트리소프트(주)
❖송태국 (tsong@nextree.co.kr)
❖부사장/CTO

아키텍처 개선 I - 목차

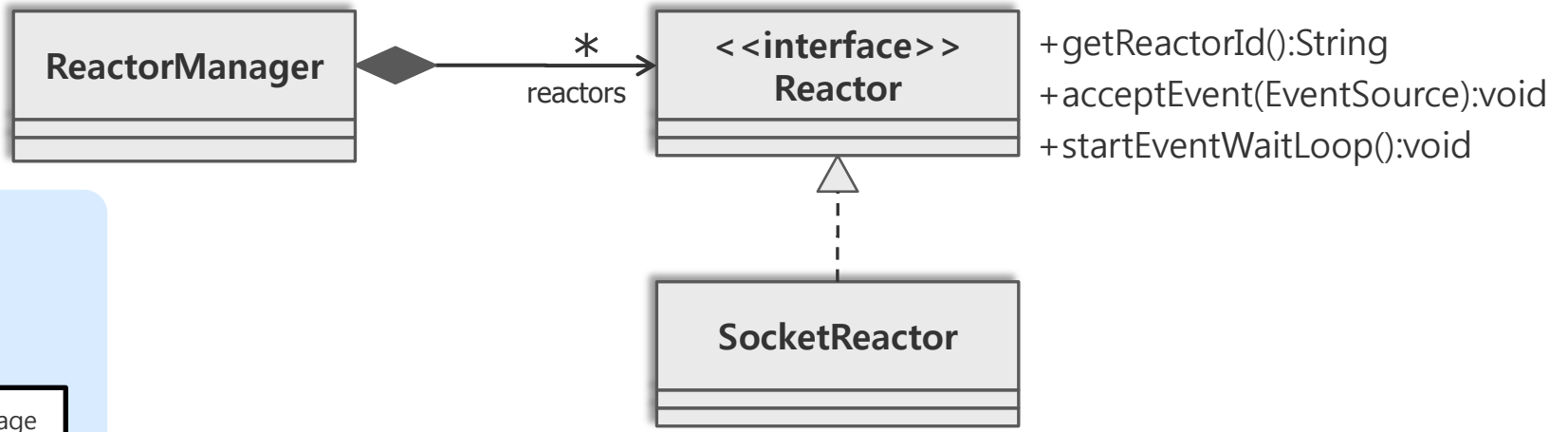
1. Reactor 확장
2. 질의 응답 및 토론

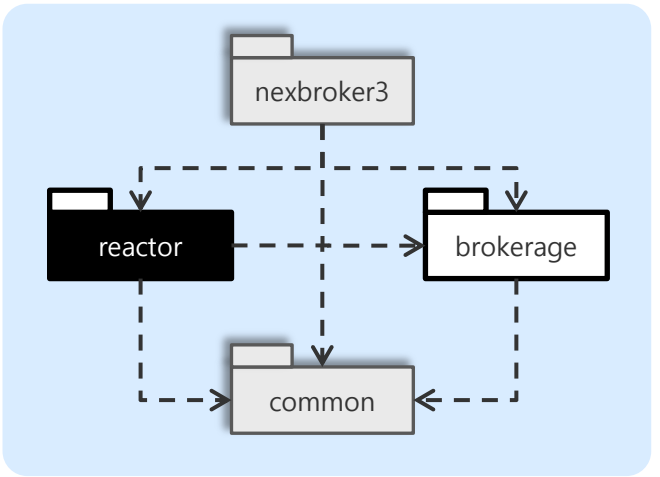
1. Reactor 확장 - 기본구조

NEXTREE

✓ 이벤트접수를 위한 객체 설계

- 패키지 이름: reactor
- 각 프로토콜 별로 접수 객체인 리액터가 존재함
- 리액터는 접수 효율을 높이기 위해 여러 개를 준비함
- 리액터가 여러 개이므로 이를 관리하기 위한 관리자가 필요함, ReactorManager
- 이것들이 전부일까? 객체들의 대화(conversation)를 들어보자...



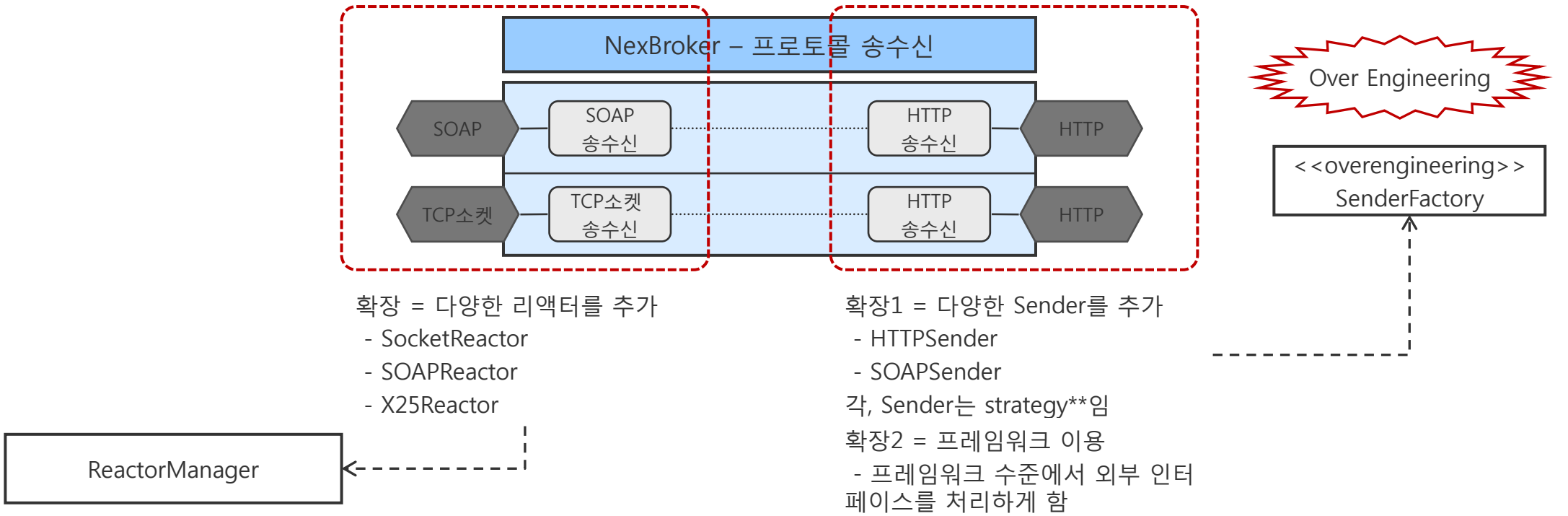


1. Reactor 확장 – 확장 포인트

NEXTREE

✓ 이벤트 접수 부분 확장성

- 현재는 SocketReactor이나 미래는 HttpReactor, SoapReactor 등으로 확장 가능해야 함
- Reactor 유형
- 대기형 - SocketReactor
- 호출형 – HttpReactor, 대기는 HttpServer가 대신 수행해 줌



1. Reactor 확장 – 객체들의 대화

NEXTREE

✓ 리액터 생성 중에 오고가는 객체들 간의 대화

리액터관리자:	접수 준비를 해야겠네... 먼저 구성정보를 읽고... (불만: 그런데 이것을 누가 대신해주면 안될까?), 흠, 포트는 2222번에 대기형 소켓리액터 20개....
리액터:	리액터를 만들자...(불만: 내가 꼭 이런 것 까지 해야할까?), 어쨌든 20개를 다 만들었군. 모두 대기시켜야지... 리액터1 너 대기상태로 들어가라. 예, 대기상태로 들어갑니다. [나머지 20 개에 대해서 대기명령을 반복한다...] 다시 구성정보를 읽고...(여전히 불만스럽다.), 흠

1. Reactor 확장 – 객체들의 대화

NEXTREE

✓ 다음 시나리오는 리액터관리자가 리액터 준비를 위해 상호 대화하는 모습임

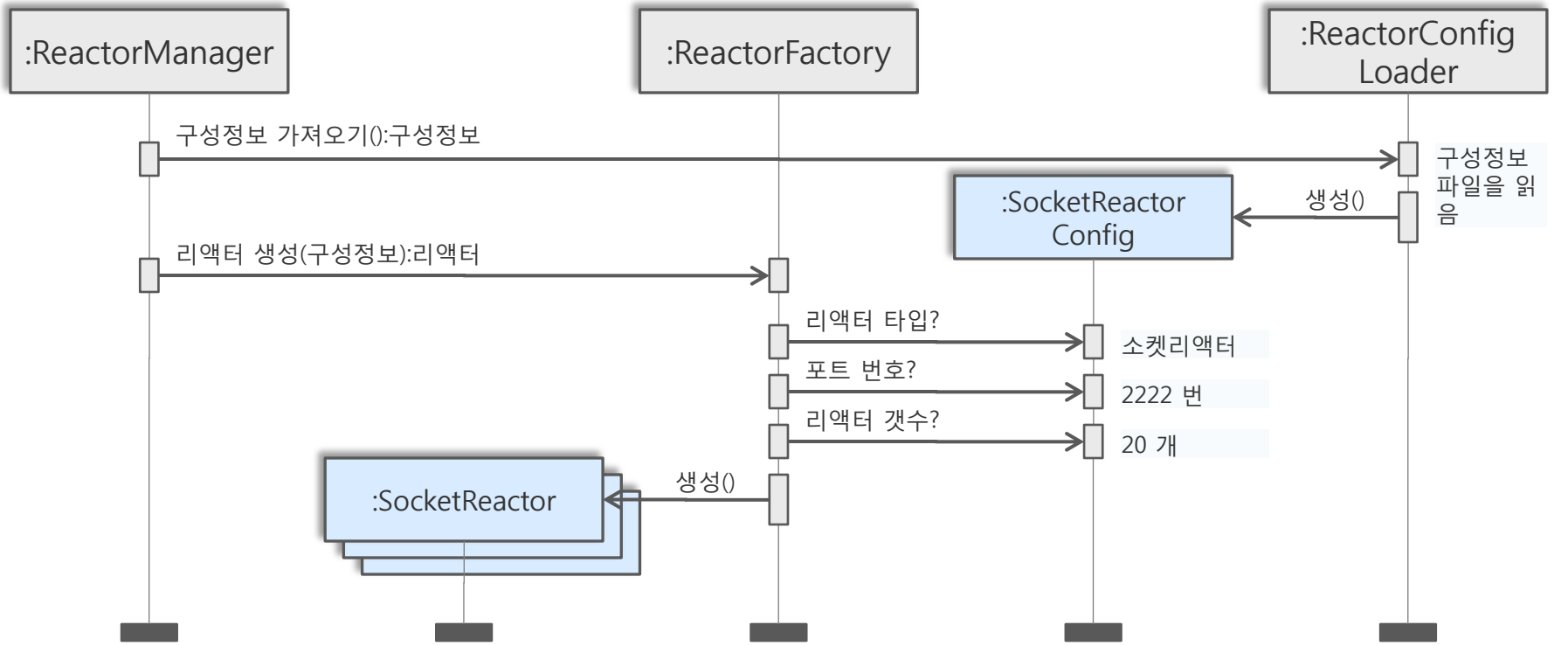
- 구성정보 가져오는 방법, 리액터 관리자와 리액터 팩토리 간의 대화를 주의깊게 관찰

리액터관리자:	리액터구성 로더야, 리액터 구성정보를 다오.
리액터구성로더:	(열심히 구성정보 파일을 읽은 후) 여기 리액터 구성정보가 있어요.
리액터 관리자:	(첫번째 리액터구성정보를 꺼내들고는) 리액터 팩토리야, 이 구성정보를 이용하여 리액터를 만들어 다오.
리액터 팩토리:	예, (구성정보에게 묻는다.) 너 무슨 구성정보니 ?
소켓리액터구성정보:	전, 소켓리액터를 위한 구성정보인데요.
리액터 팩토리:	흠, 알았어. 소켓리액터라... 소켓리액터를 만들어야지. 포트번호가 몇 번이야 ?
소켓리액터구성정보:	포트번호는 2010인데요.
리액터 팩토리:	그럼 리액터를 몇 개 만들라고 했지?
소켓리액터구성정보:	50개요.
리액터 팩토리:	그래? 소켓리액터 50개를 만들어야겠군. (50개를 만든 후 리액터 관리자에게) 리액터 모두 만들었습니다.
리액터 관리자:	고마워 리액터 팩토리. 이녀석들에게 나를 알려줘야겠군. 그래야 핸들러가 필요할 때 나에게 요청할 수 있지. (첫번째 리액터를 찾아서) 핸들러 필요하면 내게 말해 줘, 알았지?.
소켓리액터:	네, 알았어요.
	... (전체 리액터에게 핸들러 정보를 보유한 자신 알려주기를 반복한다.)
리액터 관리자:	(다음 리액터구성정보를 꺼내들고는) 리액터 팩토리야, 이 구성정보를 이용하여 리액터를 만들어 다오.
	.... (모든 리액터구성정보에 대해 반복한다.)

1. Reactor 확장 – 시퀀스 분석

✓ 주요 객체 간의 메시지 전송

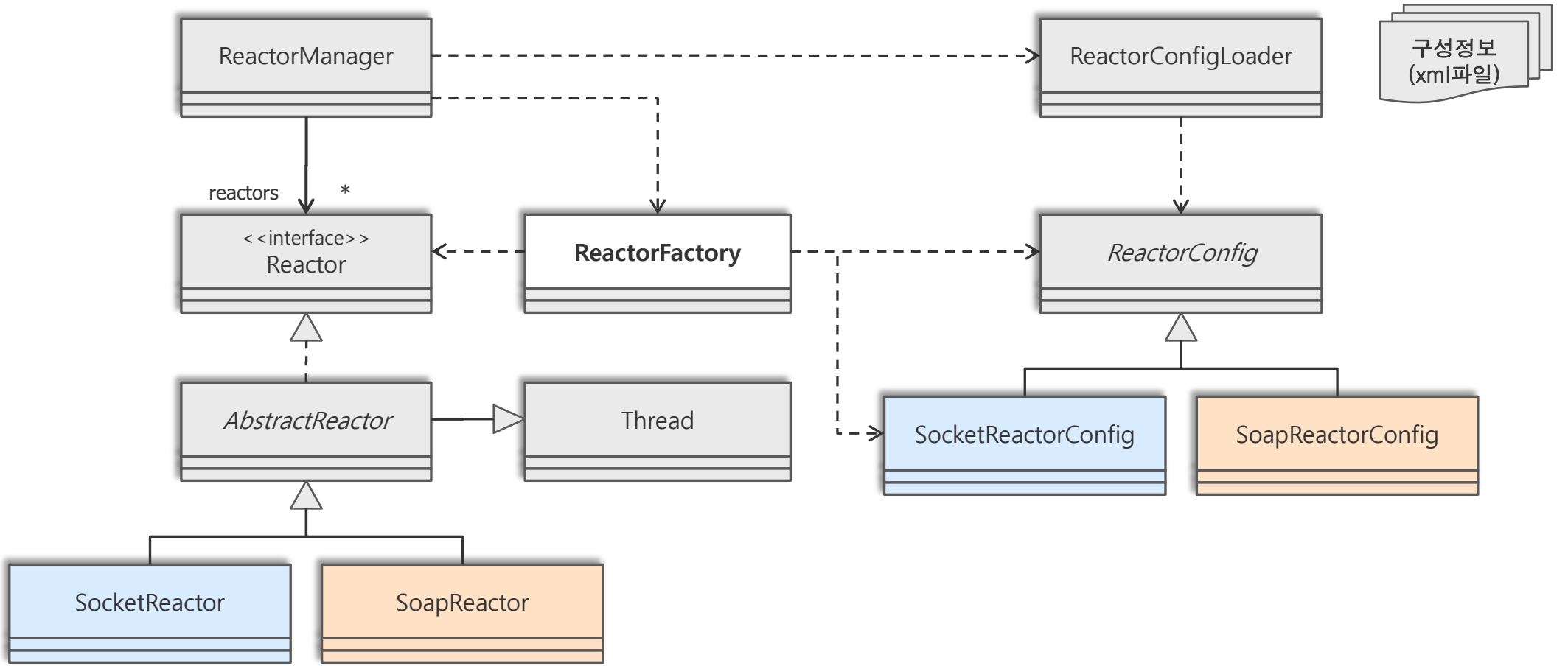
- 리액터관리자: 구성정보 객체를 획득한 후 이를 기반으로 리액터 생성 요구함
- 구성정보로더: 구성정보를 읽은 후 구성정보 객체를 생성함
- 리액터팩토리: 구성정보를 근거로 필요한 개수 만큼 리액터를 생성함



1. Reactor 확장 – 주요 클래스들

✓ 리액터 준비 시나리오에 참여하는 객체들의 클래스는 다음과 같음.

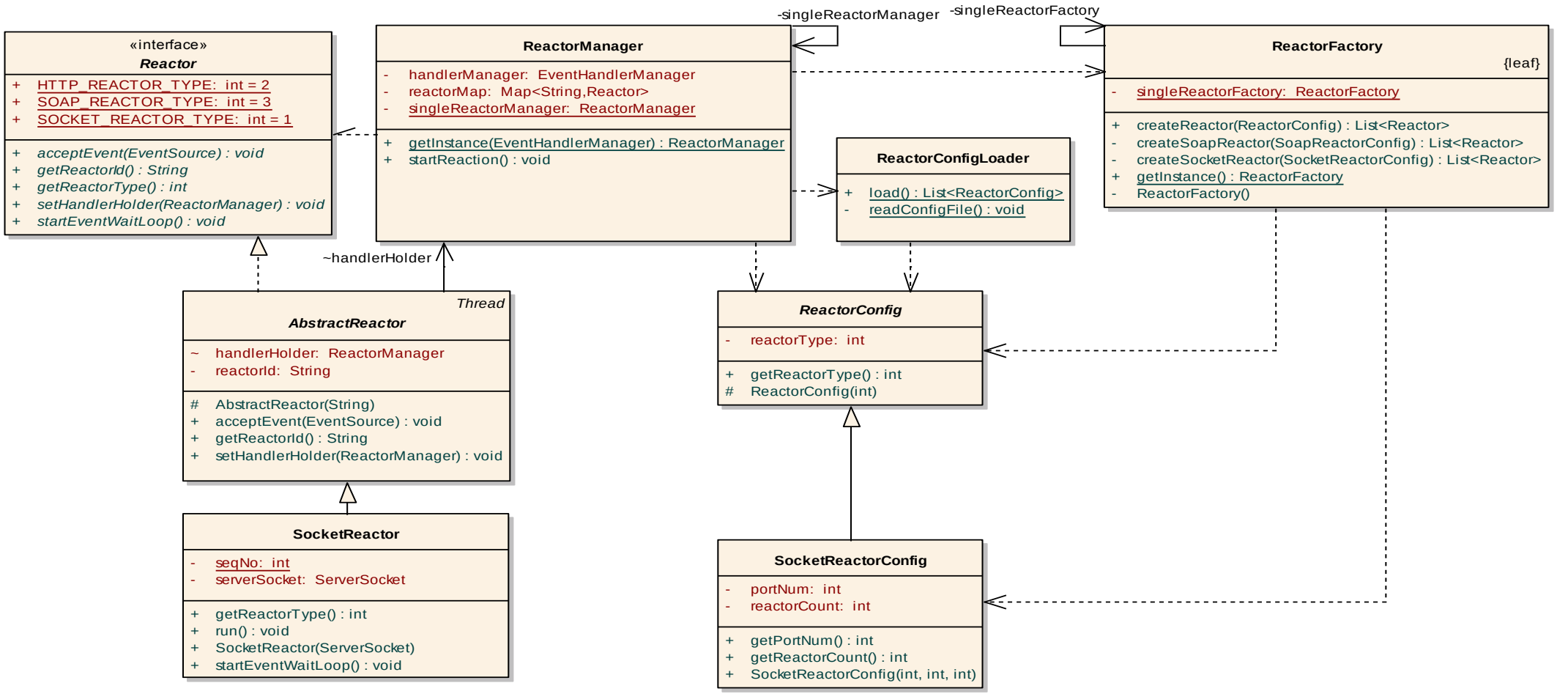
- 구조를 유지하는 핵심 클래스: ReactorFactory



1. Reactor 확장 – 클래스 다이어그램

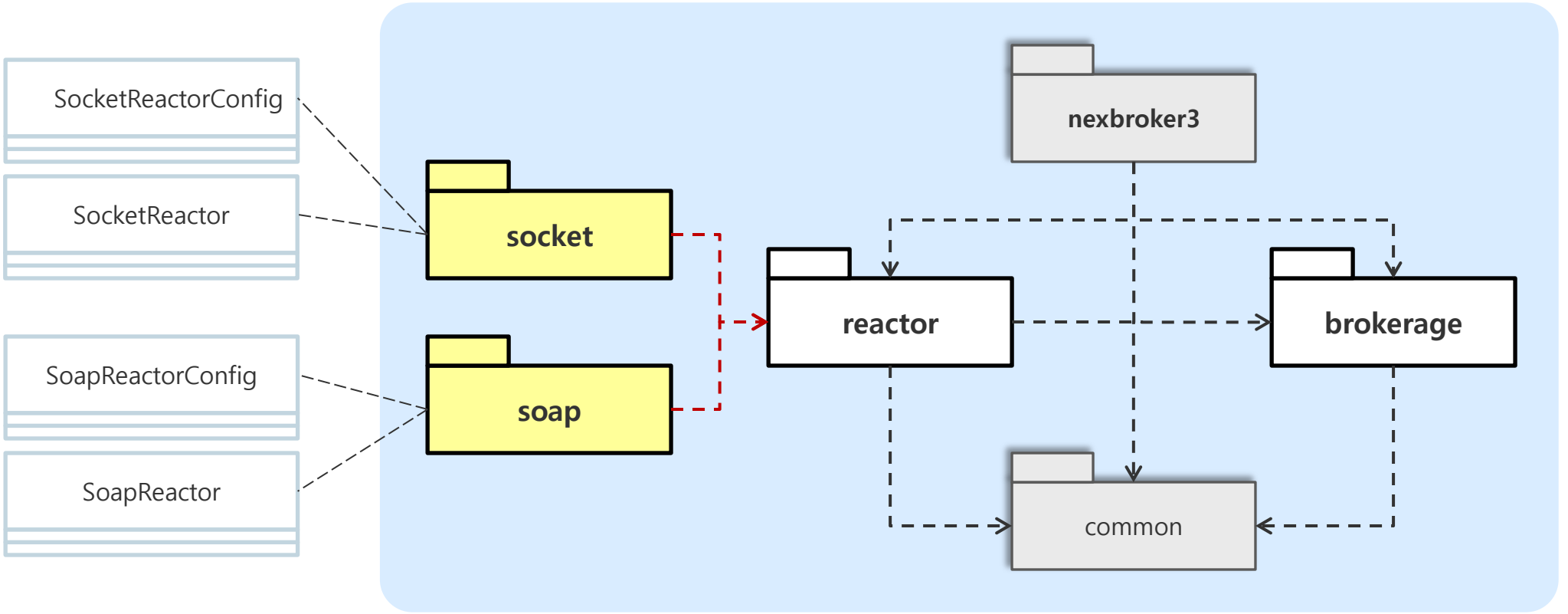
✓ 지금까지 설계한 내용을 클래스 다이어그램으로 표현하면 다음과 같음

- 확장시 변경이 필요한 클래스는 ReactorFactory 임 (붉은 점선 박스 참조)



1. Reactor 확장 – 패키지 확장

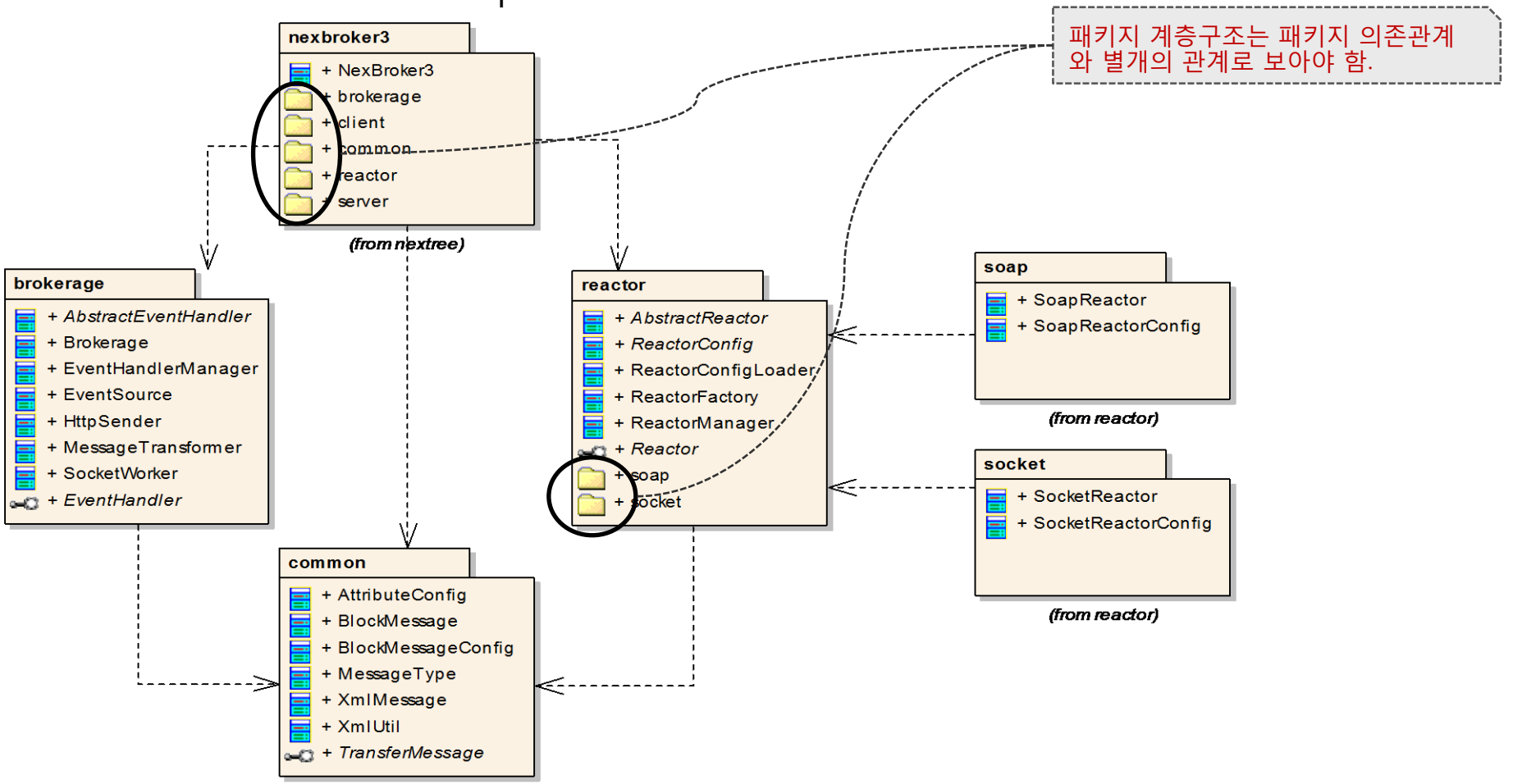
- ✓ 확장부분에 해당하는 클래스들은 독립 패키지로,
  - ✓ com.nextree.nexbroker3.reactor
  - ✓ com.nextree.nexbroker3.reactor.socket
  - ✓ com.nextree.nexbroker3.reactor.soap



1. Reactor 확장 – 패키지 다이어그램

NEXTREE

- ✓ 확장부분에 해당하는 클래스들은 독립 패키지로,
  - ✓ com.nextree.nexbroker3.reactor
  - ✓ com.nextree.nexbroker3.reactor.socket
  - ✓ com.nextree.nexbroker3.reactor.soap



1. Reactor 확장 – 패키지 다이어그램

NEXTREE

✓ 새로운 리액터 추가 시 절차(큰 변경없이 수월하게 추가할 수 있음),

- HttpReactorConfig 설계/개발
- HttpReactor 설계/개발
- ReactorConfig 파일 업데이트
- ReactorFactory 클래스 수정

Problem !!
reactor 패키지는 인프라에 해당하는 패키지임, 확장에도 흔들림이 없어야 함, 그런데!!

reactor

1. HttpReactorConfig 설계/개발

HttpReactorConfig

2. HttpReactor 설계/개발

HttpReactor

3. ReactorConfig 파일 업데이트

HttpReactorConfig 파일

4. RectorFactory 클래스 수정(붉은글자 부분)

```
public List<Reactor> createReactor(ReactorConfig reactorConfig) {
        List<Reactor> resultReactors = new ArrayList<Reactor>();
        int reactorType = reactorConfig.getReactorType();

        switch(reactorType) {
                case Reactor.SOCKET_REACTOR_TYPE:
                        resultReactors = createSocketReactor((SocketReactorConfig)reactorConfig);
                        break;
                case Reactor.HTTP_REACTOR_TYPE:
                        resultReactors = createSoapReactor((HttpReactorConfig)reactorConfig);
                        break;
        }

        return resultReactors;
}

private List<Reactor> createHttpReactor(SoapReactorConfig httpReactorConfig) {
        ....
        return resultReactor;
}
```

1. Reactor 확장 – 완전한 확장(1/5)

NEXTREE

✓ 완전성(Integrity) 요구사항: reactor 패키지는 reaction의 인프라 패키지이므로 확장 시에도 내부 클래스에는 전혀 변화가 없어야 함.

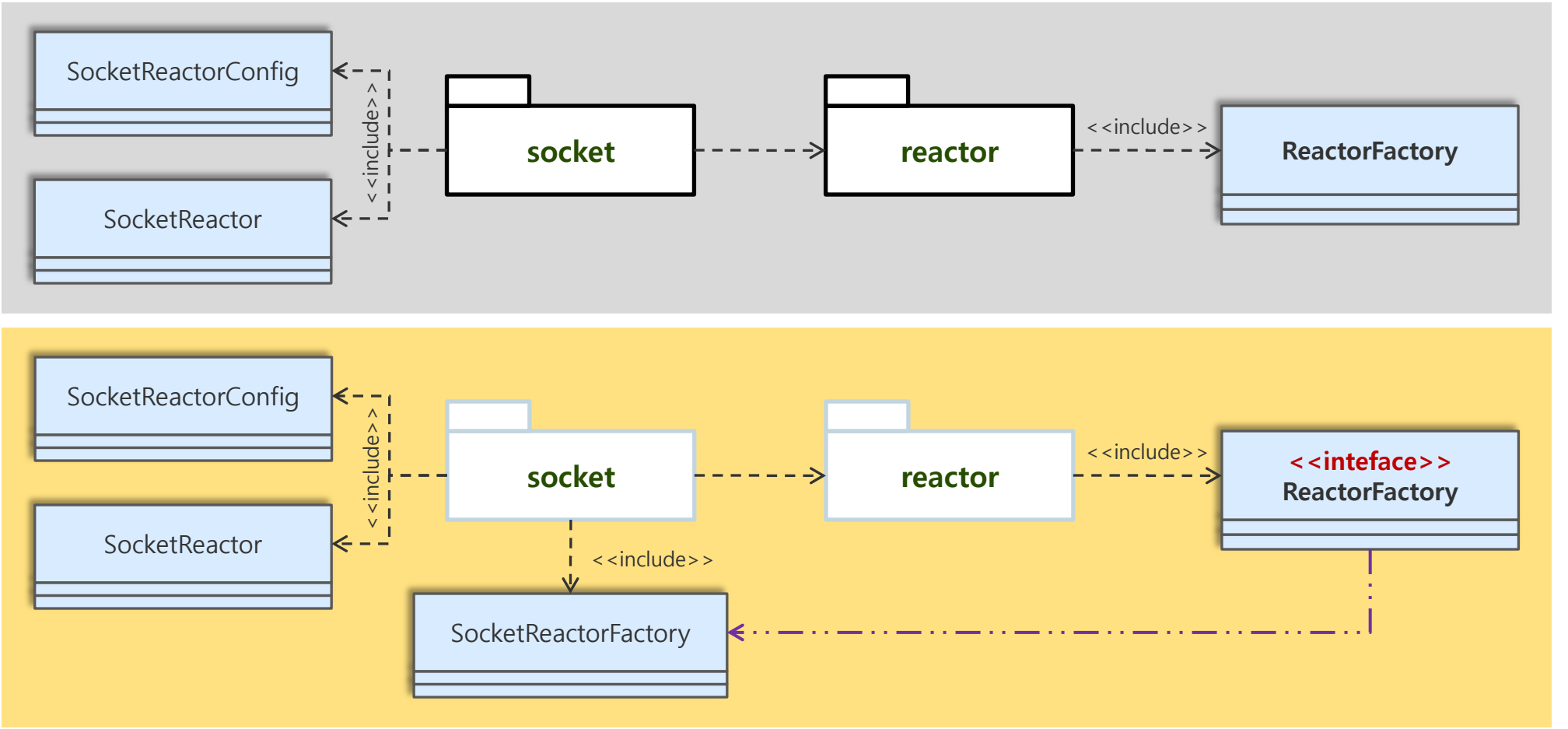
- 이를 위해서 ReactorFactory를 인터페이스로 변경
- 확장(예, Socket 확장)에 대해 다음 세 가지를 한 세트로 추가
  - SocketReactor
  - SocketReactorConfig
  - SocketReactorFactory
- socket 패키지 속에 세 클래스를 둠
- 확장 팩토리 이름을 구성정보에 설정(공통이므로, RecactorConfig에 추가)
- getFactoryName() 서비스 추가
- ReactorManager가 reflection 기능을 이용하여 팩토리 생성
- Class factoryClass = Class.forName(factoryName);
- Object factoryObject = c.newInstance();
- 생성된 팩토리에게 리액터 생성을 요청

1. Reactor 확장 – 완전한 확장(2/5)

NEXTREE

✓ 리팩토링

- ReactorFactory를 인터페이스로 변경
- 각 확장별로 팩토리 클래스 생성: SocketReactorFactory

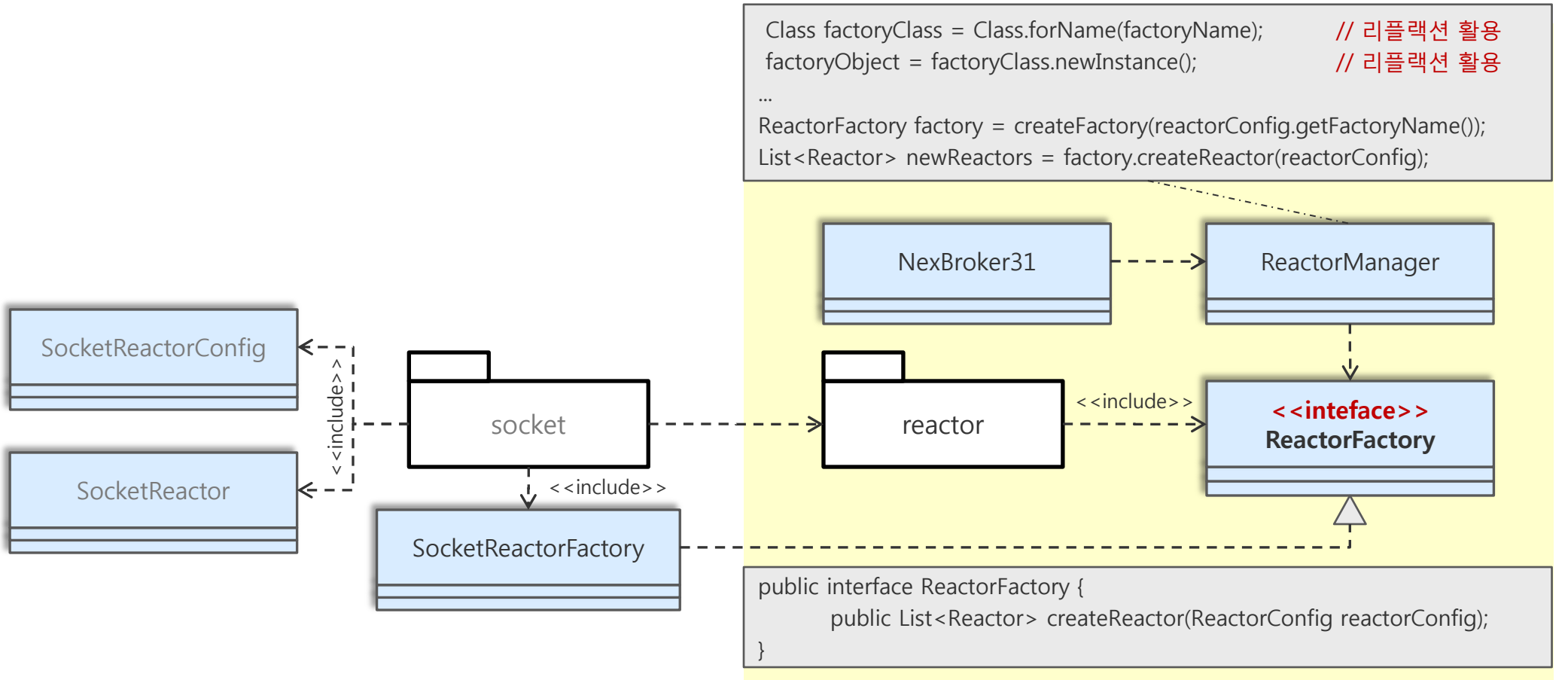


1. Reactor 확장 – 완전한 확장(3/5)

NEXTREE

✓ 리팩토링 2

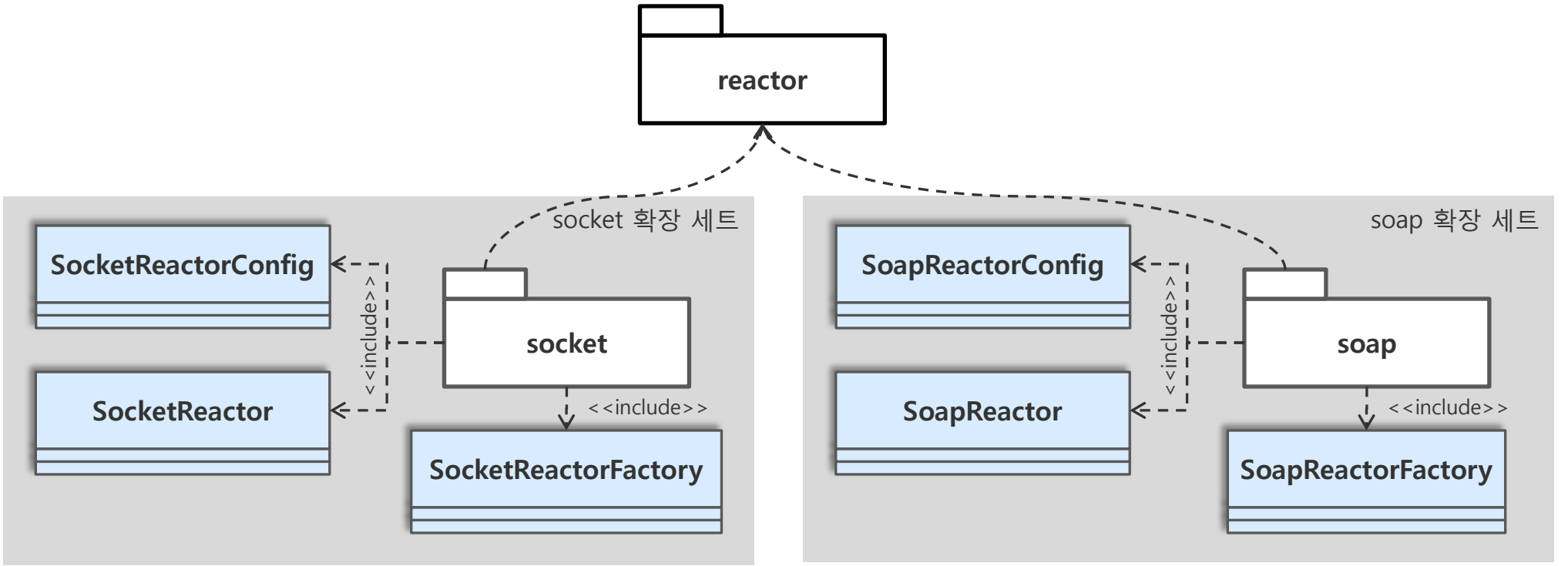
- ReactorManager는 ReactorFactory 만을 보게함, 리플랙션 활용하여 팩토리 메소드 호출
- ReactorFactory의 서비스는 인퍼페이스 메소드로 변환



1. Reactor 확장 – 완전한 확장(4/5)

✓ 리팩토링 3

- Reactor, ReactorConfig, ReactorFactory 세 클래스 하나의 확장 세트가 됨

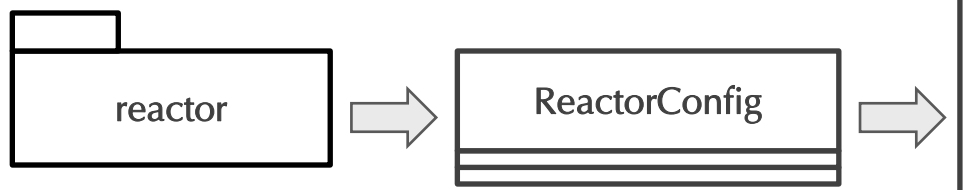


1. Reactor 확장 – 완전한 확장(5/5)

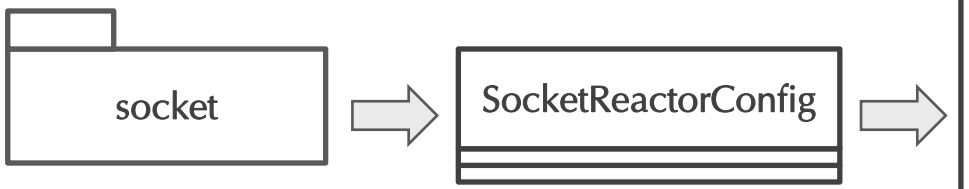
NEXTREE

✓ 리팩토링 4

- 확장 팩토리 이름을 구성정보에 설정
- 모든 확장에 공통이므로, ReactorConfig에 추가
- getFactoryName() 서비스 추가




```
public abstract class ReactorConfig {
        private int reactorType;
        private String factoryName;
        protected ReactorConfig(int reactorType, String factoryName) {
                this.reactorType = reactorType;
                this.factoryName = factoryName;
        }
        ...
        public String getFactoryName() {
                return this.factoryName;
        }
}
```




```
public class SoapReactorConfig extends ReactorConfig {
        private int reactorCount;
        public SoapReactorConfig(int reactorType,
                                String factoryName, int reactorCount) {
                super(reactorType, factoryName);
                this.reactorCount = reactorCount;
        }
        ...
}
```

1. Reactor 확장 – 요약

- ✓ Late binding: 팩토리 이름을 제공하고, 필요한 시점에서 인스턴스 화 하여 사용함
- ✓ 세 개의 확장 세트 클래스를 제공하고, 구성정보에 팩토리 이름을 제공하면, reactor 패키지에는 어떤 변경도 없이 확장이 가능함
- ✓ 원래 설계 버전에서 확장에 따른 약간의 코드 수정 역시 변경이 크다고 할 수 없음
- ✓ 따라서, 완전한 확장은 약간은 과도한 설계(over-engineering)로 느껴짐
- ✓ 하지만, 어떤 상황에서는 실시간에 애플리케이션 정지없이 확장을 해야 할 경우가 있는데 이럴 경우에 적절한 설계라고 할 수 있음

넘치는 것이 무엇인지 알아야, 모자라는 것이 무엇이고, 적절한 것이 무엇인지 알 수 있다.

두려워 하지 말고 과도한 설계를 해 보자...

2. 질의 응답 및 토론

- ✓ 질의 응답
- ✓ 토론

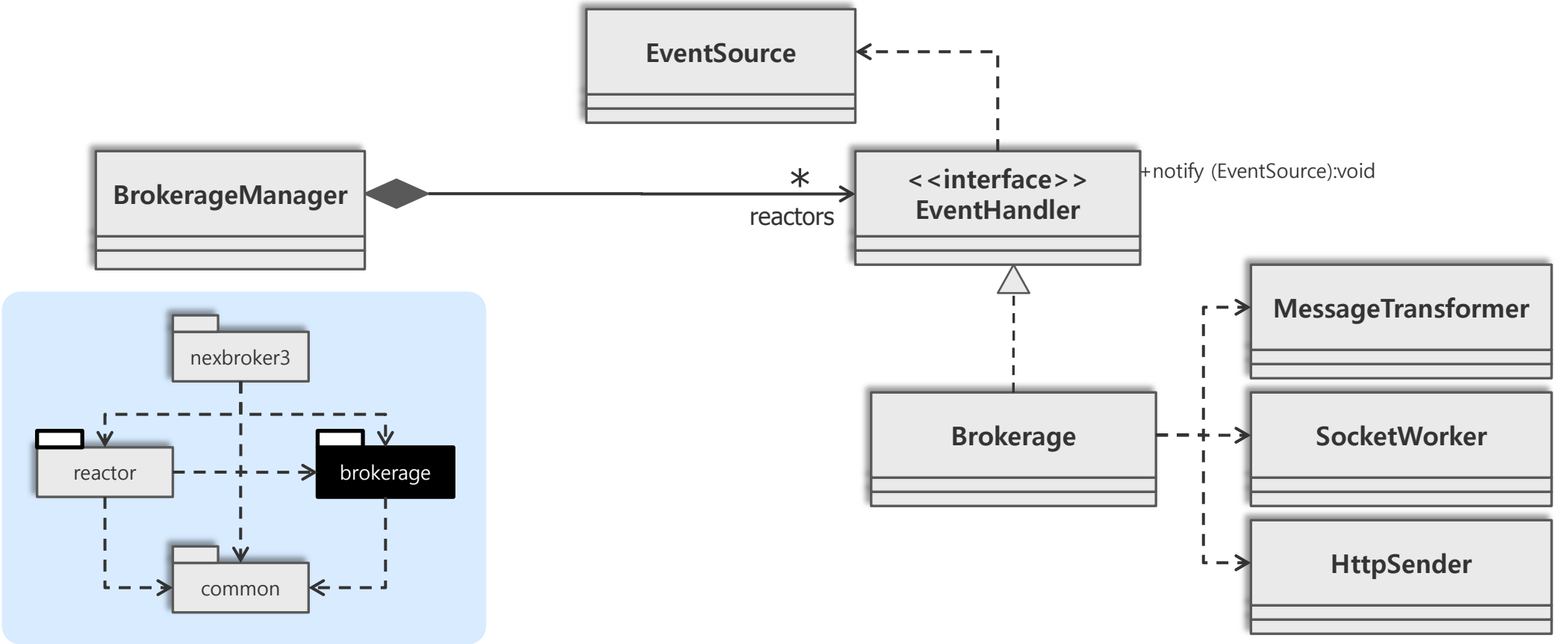
❖넥스트리소프트(주)
❖송태국 (tsong@nextree.co.kr)
❖부사장/CTO

아키텍처 개선 II - 목차

1. Brokerage 확장
2. 질의 응답 및 토론

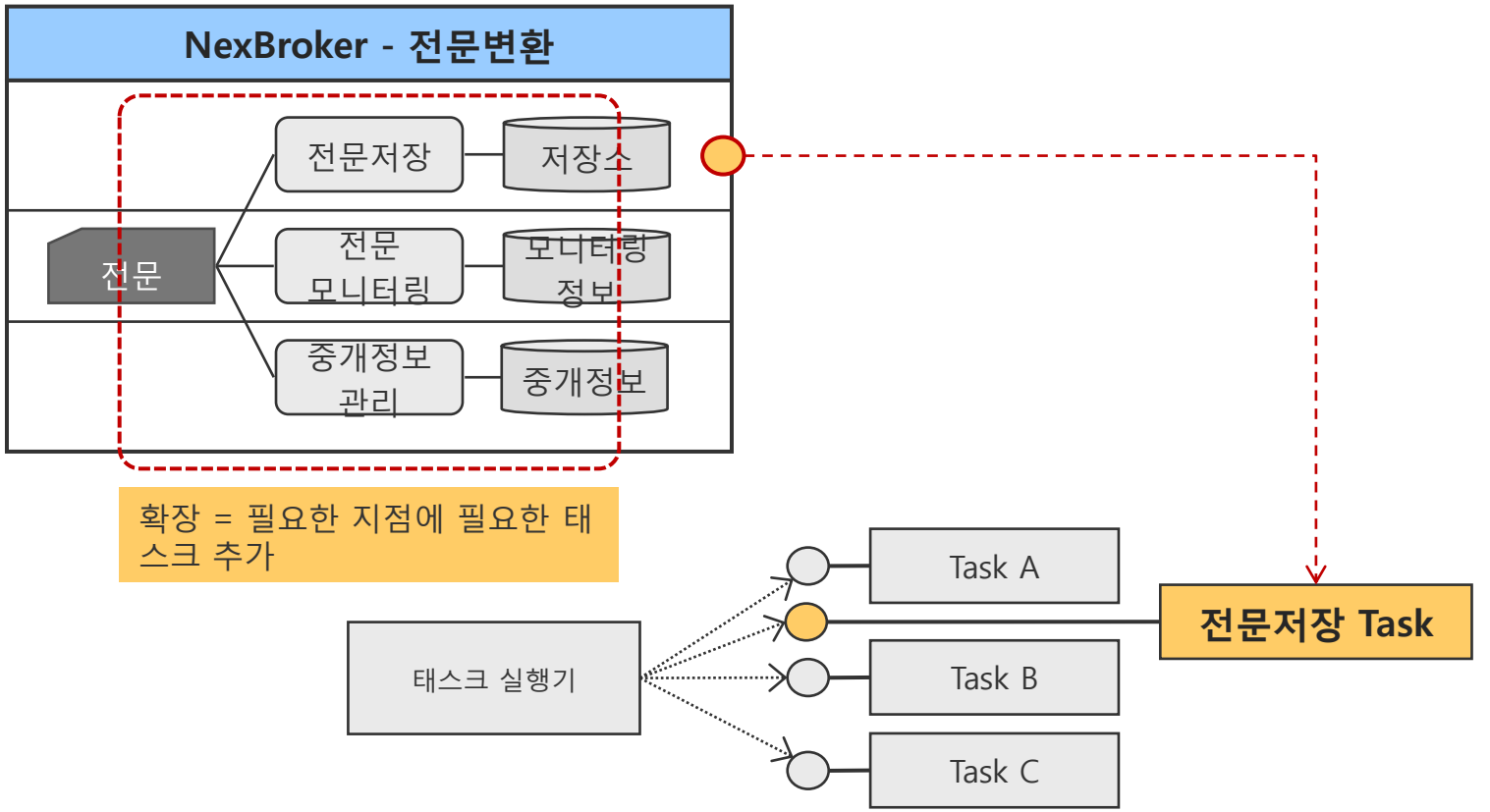
1. Brokerage 확장 – 기본 구조

- ✓ NexBroker3에서 설계한 기본구조가 확장성을 충분히 가질 수 있는가?
- ✓ 새로운 작업이 추가되었을 경우, Brokerage 의 처리 부분에 추가하여야 함.
- ✓ Brokerage는 핵심 클래스이므로 코드 변경을 지양하여야 함.



1. Brokerage 확장 – 품질속성 검토

- ✓ Brokerage 영역의 확장이라 함은 새로운 태스크를 추가할 수 있음을 의미함
- ✓ 현행 구조가 확장성을 보장할 수 있는 구조인가?



1. Brokerage 확장 – 설계 이슈(1/2)

NEXTREE

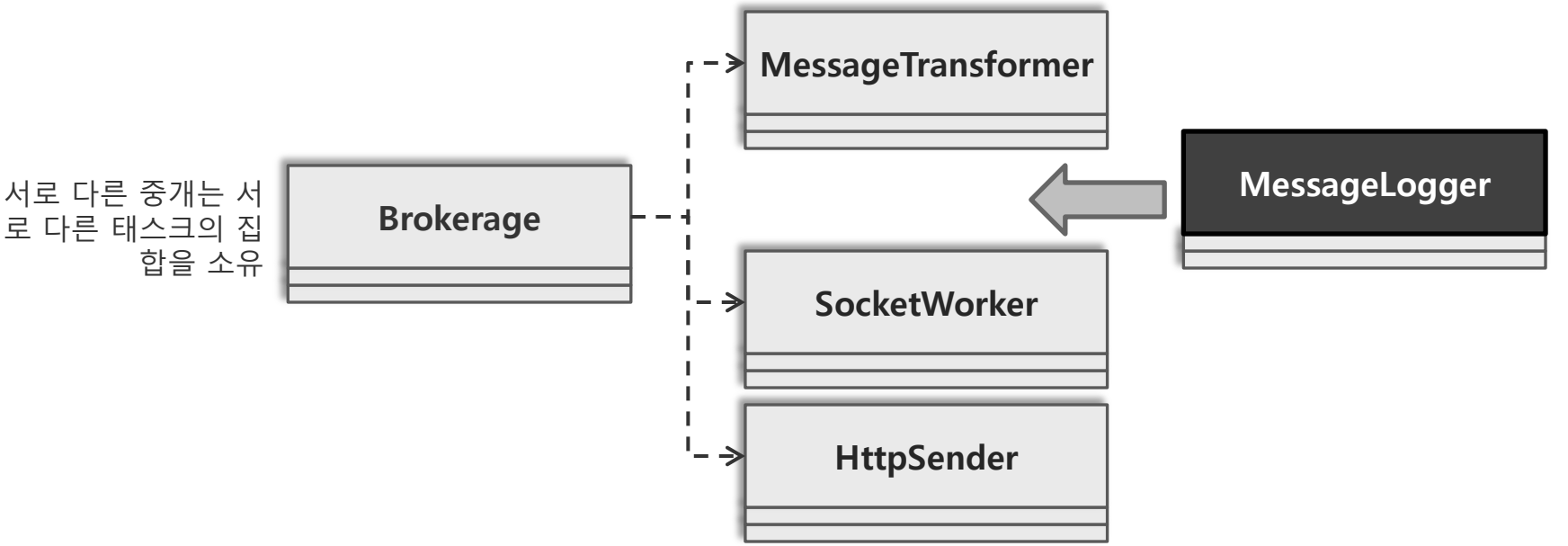
- ✓ 중개(Brokerage) 객체 재사용
  - 중개는 반복적으로 사용하는 객체인데, 매번 생성,삭제하는 것은 비효율적임
  - Pool에 담아 두고 필요할 때 사용하고 반환하여야 함
- ✓ 중개 객체를 누가 만들 것인가?
  - 중개 객체는 일련의 작업을 처리하는 객체를 안고 있으므로 생성을 위한 지식이 필요함

이벤트핸들러관리자:	재사용 가능한 중개(brokerage) 객체 50개를 만들어야지... (불만: 왜 내가 중개객체를 직접만들어야지, 여러 가지를 만들려니 알아야 할 것도 많고...) 모두 만들었군, 흠 이 객체를 저장소에 넣어두고 필요할 때 해당 객체를 찾아서 써야지... (불만: 왜 내가 저장소를 관리하고 필요한 객체를 직접 찾아야 하지, 누가 해 주면 편할텐데....) 흠... 어쨌거나 준비를 다 하고 나니 홀가분하군.... 이제 핸들러 요청을 기다리자... (대기)
리액터관리자:	핸들러 관리자, 이 이벤트소스에 해당하는 핸들러 하나만 주시오.
이벤트핸들러관리자:	잠시만, 이벤트 소스 넌 타입이 뭐지 ?
이벤트소스:	소켓타입입니다.
이벤트핸들러관리자:	내 저장소를 뒤져서 소켓타입 핸들러를 찾을 때까지 기다려 주시오. (불만: 왜 내가 이 일을 직접 해야 하지... ㅠㅠ) 옳지 여기 있군, 자 소켓핸들러 여기 있소.
리액터 관리자:	고맙소, 핸들러 관리자...

1. Brokerage 확장 – 설계 이슈(2/2)

NEXTREE

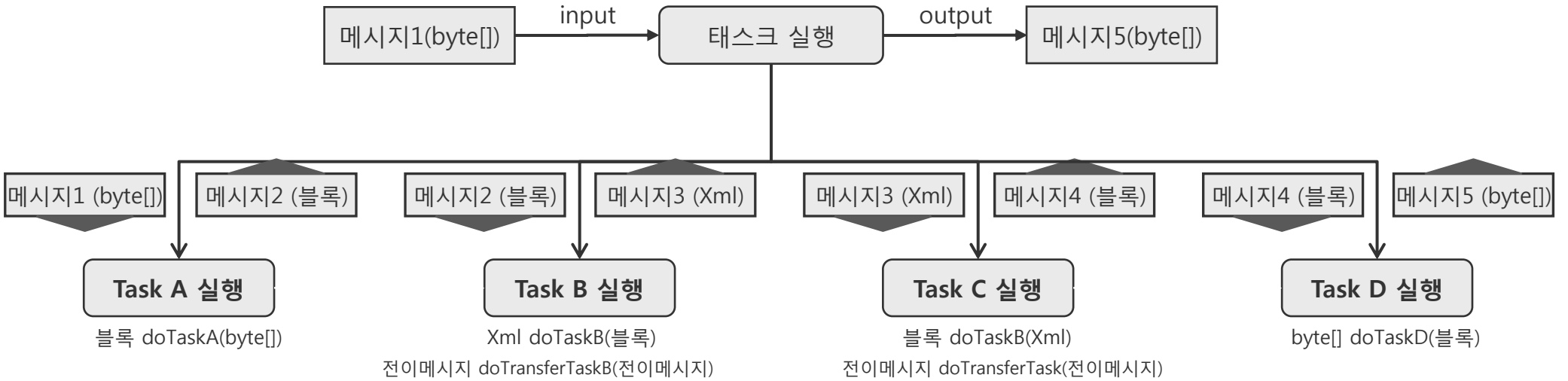
- ✓ 중개 작업 처리 – 절차 일반화
  - 단지 여러 개의 태스크인가?
- ✓ 중개 작업 처리 – 추가 작업
  - 한 건의 중개는 여러 태스크를 연속적으로 수행하는 것임
  - 태스크는 추가되거나 삭제될 수 있음
  - 중개 별로 서로 다른 태스크를 가지고 있음

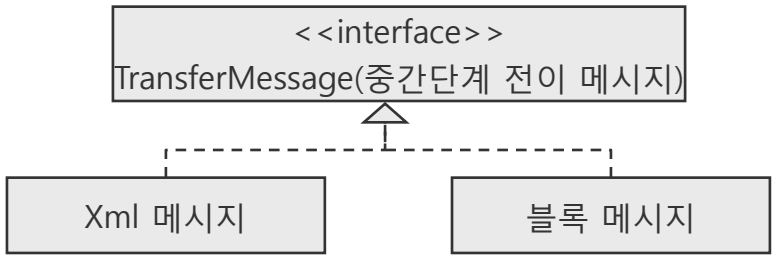


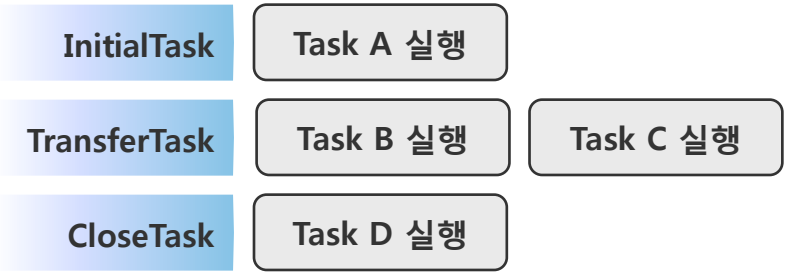
1. Brokerage 확장 – “일련의 작업”에 대한 고찰

✓ 태스크와 메시지 추상화

- 태스크가 단일 인터페이스 doTask()를 낼 수 없음 – byte[] 메시지 때문
- 태스크의 유형이 세 가지로 분류되며, 메시지는 TransferMessage로 추상화 함

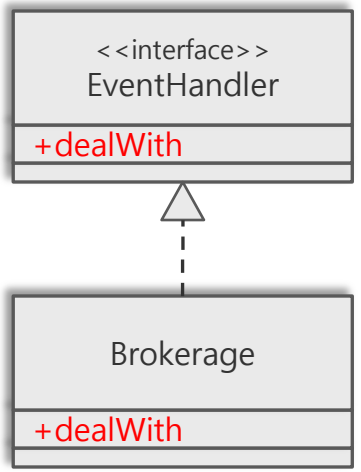






1. Brokerage 확장 – 중개절차 통일(1/3)

- ✓ NexBroker3의 notify 메소드는 알고리즘으로 구성하였음
- ✓ 중개 알고리즘의 변화는 곧 Brokerage.dealWith()의 변경을 의미함
- ✓ 템플릿 메쏘드를 이용하여 일련의 태스크 속에서 표준화 절차를 찾은 다음, 모든 유형의 중개(Brokerage)가 함께 사용할 수 있도록 함

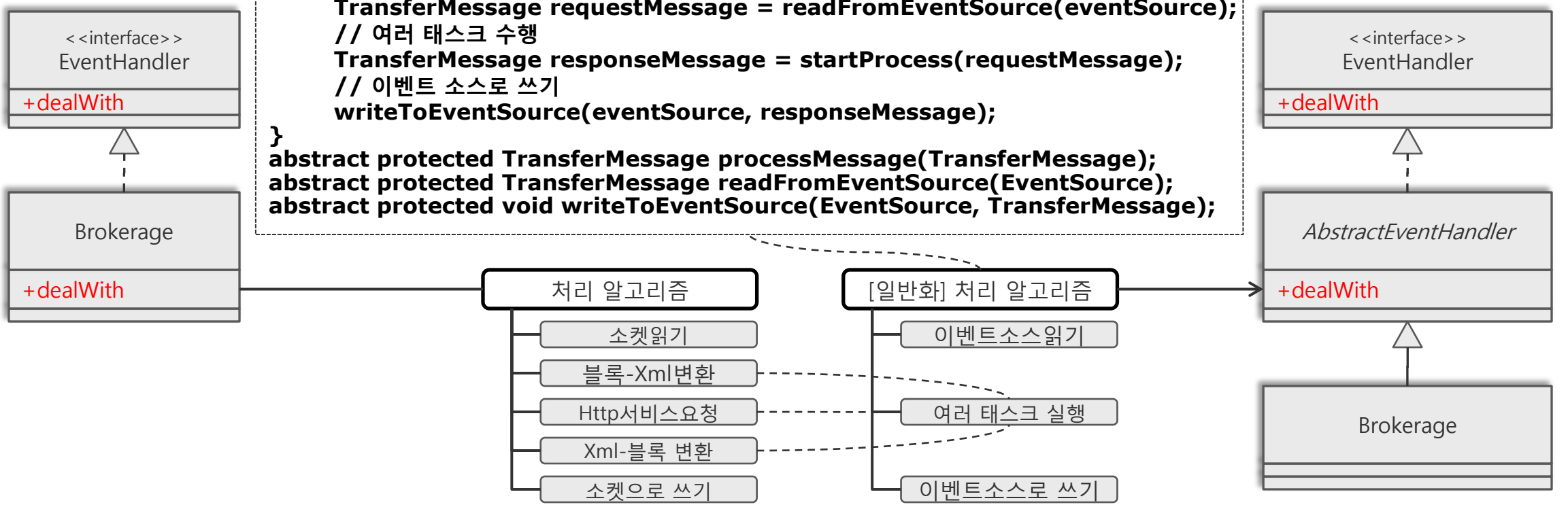



```
public void dealWith(EventSource eventSource) {

        // 태스크1: 이벤트소스 읽기
        SocketWorker socketWorker = new SocketWorker(blockMessageConfig);
        TransferMessage receivedMessage = socketWorker.read(eventSource);

        // 태스크2: 블록메시지를 Xml 메시지로 변환
        MessageTransformer transformer = new MessageTransformer(this.blockMessageConfig);
        XmlMessage xmlRequestMessage = transformer.transformBlockToXml((BlockMessage)receivedMessage);

        // 태스크3: 서비스 서버로 서비스 요청
        HttpSender httpSender = new HttpSender();
        XmlMessage xmlResponseMessage = httpSender.send(xmlRequestMessage);

        // 태스크4: Xml메시지를  블록 메시지로 변환
        BlockMessage blockResponseMessage = transformer.transformXmlToBlock(xmlResponseMessage);

        // 태스크5: 이벤트 소스로 결과 쓰기
        socketWorker.write(eventSource, (BlockMessage)blockResponseMessage);
}
```

1. Brokerage 확장 – 중개절차 통일(2/3)

- ✓ 중개 처리 프로세스는 결국, 이벤트 소스읽기, 태스크 실행, 이벤트 소스로 쓰기 임
- ✓ 향후 모든 Brokerage가 이 절차를 준수하게 하여야 함
- ✓ Template Method를 이용하여 프로세스를 통일함

```
public void dealWith(EventSource eventSource) {
        // 이벤트 소스 읽기
        TransferMessage requestMessage = readFromEventSource(eventSource);
        // 여러 태스크 수행
        TransferMessage responseMessage = startProcess(requestMessage);
        // 이벤트 소스로 쓰기
        writeToEventSource(eventSource, responseMessage);
}
abstract protected TransferMessage processMessage(TransferMessage);
abstract protected TransferMessage readFromEventSource(EventSource);
abstract protected void writeToEventSource(EventSource, TransferMessage);
```

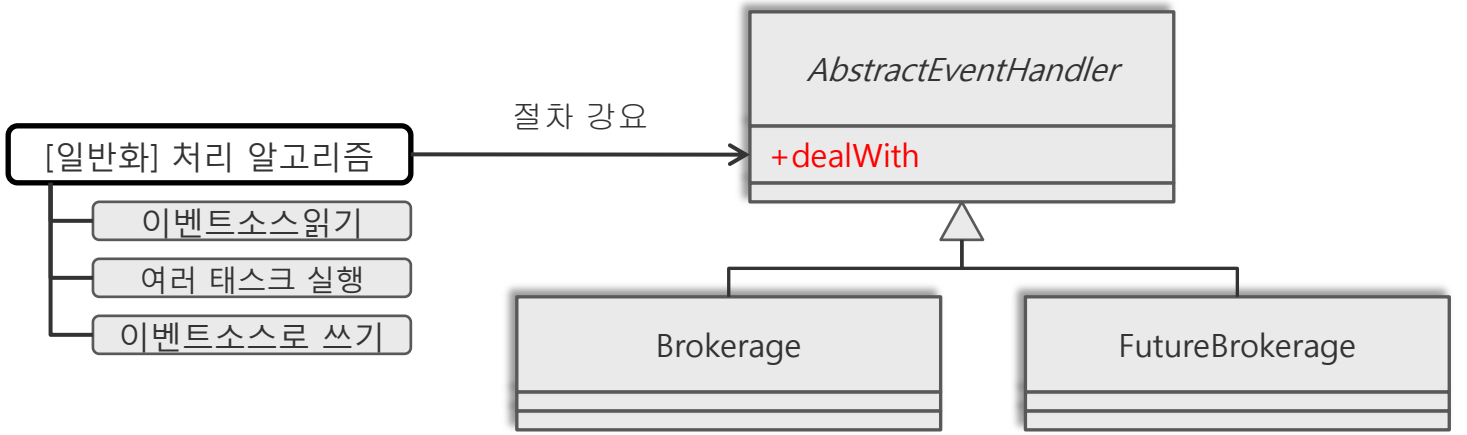


1. Brokerage 확장 – 중개절차 통일(3/3)

NEXTREE

✓ 프로세스 통일을 통해 얻는 것은 ?

- 프로세스의 명확한 식별
- 미래의 이벤트 핸들러에게 동일한 처리 절차 강요 – 처리절차 표준



동일한 절차와
서로 다른 처리 로직을 가짐

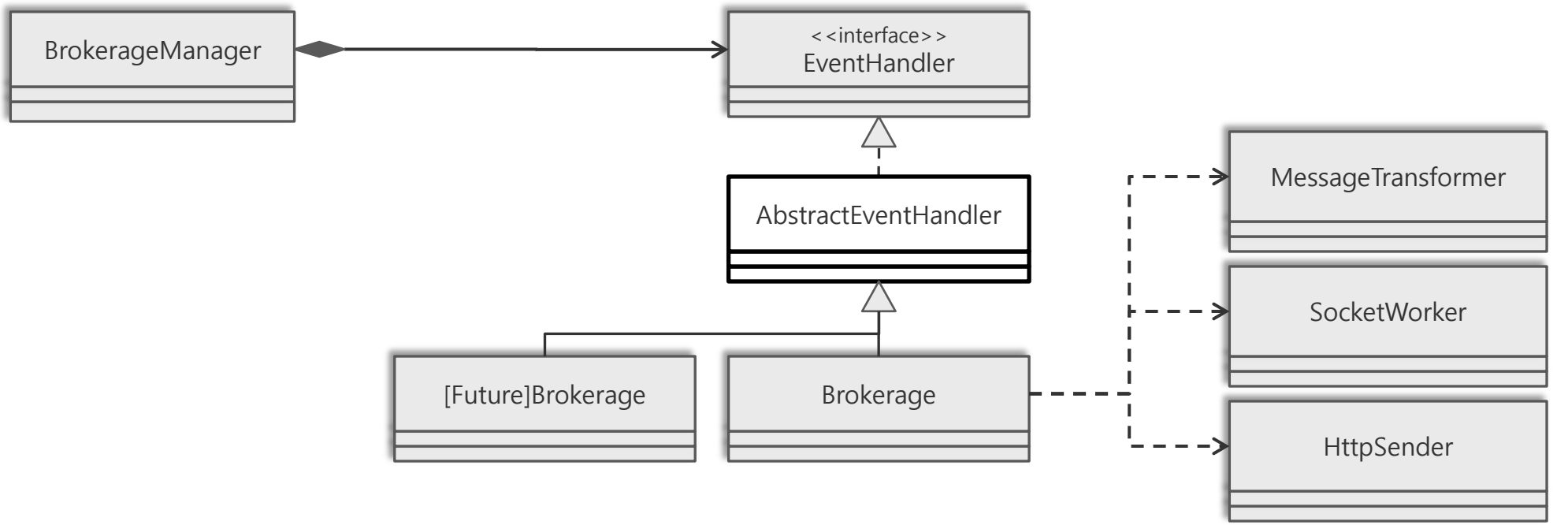
```
protected TransferMessage processMessage(TransferMessage){
        …
}
protected TransferMessage readFromEventSource(EventSource){
        …
}
protected void writeToEventSource(EventSource, TransferMessage){
        …
}
```

```
protected TransferMessage processMessage(TransferMessage){
        …
}
protected TransferMessage readFromEventSource(EventSource){
        …
}
protected void writeToEventSource(EventSource, TransferMessage){
        …
}
```

FutureBrokerage 특정 로직

1. Brokerage 확장 – 지금까지 확장 결과

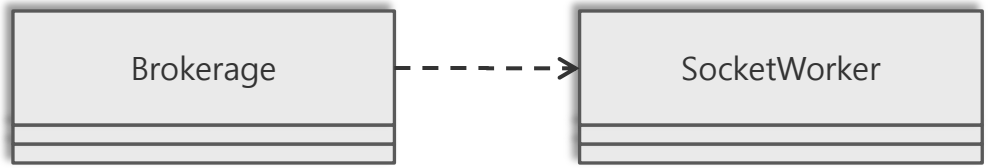
- ✓ 다양할 수 있는 중개절차를 통일하기 위해 Template Method 패턴을 적용
- ✓ Template Method를 담은 AbstractEventHandler 클래스 등장



1. Brokerage 확장 – EventWorker 확장

✓ 이벤트 소스로의 읽기/쓰기

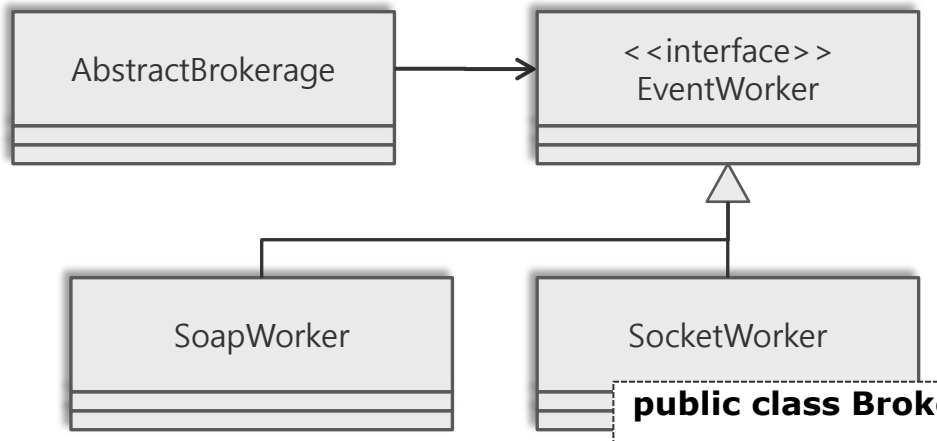
- 현재 버전: SocketWorker가 read/write 메소드를 가지고 있음
- Event에 따라 read/write 하는 Worker 객체가 달라야 함(소켓:소켓워커, 솝:솝워커)
- 따라서, Worker 객체를 일반화 하여야 함.




```
public class Brokerage extends AbstractBrokerage {
        ...
        TransferMessage processMessage(TransferMessage) {
                ...
        }
        protected TransferMessage readFromEventSource(EventSource) {
                SocketWorker worker = new SocketWorker(blockMessageConfig);
                return worker.read(eventSource);
        }

        protected void writeToEventSource(EventSource, TransferMessage) {
                SocketWorker worker = new SocketWorker(blockMessageConfig);
                return worker.write(eventSource);
        }
}
```

1. Brokerage 확장 – EventWorker 확장

- ✓ EvenetWorker 객체로의 일반화를 통해,
  - Brokerage 객체는 어떤 타입의 EventWorker도 수용할 수 있음
  - Brokerage는 완전히 일반화 됨



+read(EventSource):TransferMessage
+write(EventSource, TransferMessage): void

```
public class Brokerage extends AbstractBrokerage {
        ...
        public Brokerage (EventWorker eventWorker) {
                this.eventWorker = eventWorker;
                ...
        }
        protected TransferMessage readFromEventSource(EventSource) {
                return eventWorker.read(eventSource);
        }

        protected void writeToEventSource(EventSource, TransferMessage) {
                return eventWorker.write(eventSource, transferMessage);
        }
}
```

1. Brokerage 확장 – 소스코드 변화 확인

✓ 단순 알고리즘으로 처리한 NexBroker4에서 절차를 일반화한 버전

nexbroker4.Brokerage.dealWith()

```
public void dealWith(EventSource eventSource) {

        // 이벤트 소스 읽기
        SocketWorker socketWorker = new SocketWorker(blockMessageConfig);
        TransferMessage receivedMessage = socketWorker.read(eventSource);

        // 블록메시지를 Xml 메시지로 변환 ----------------------------
        MessageTransformer transformer = new MessageTransformer(this.blockMessageConfig);
        XmlMessage xmlRequestMessage = transformer.transformBlockToXml((BlockMessage)receivedMessage);
        // 서비스 서버로 서비스 요청-----------------------------------
        HttpSender httpSender = new HttpSender();
        XmlMessage xmlResponseMessage = httpSender.send(xmlRequestMessage);
        // Xml메시지를  블록 메시지로 변환 ----------------------------
        BlockMessage blockResponseMessage = transformer.transformXmlToBlock(xmlResponseMessage);

        // 이벤트 소스로 결과 쓰기
        socketWorker.write(eventSource, (BlockMessage)blockResponseMessage);
}
```

nexbroker5.AbstractBrokerage.dealWith()

```
public void dealWith(EventSource eventSource) {

        // 이벤트소스 읽기
        TransferMessage requestMessage = readFromEventSource(eventSource);
        // 메시지 처리 시작
        TransferMessage responseMessage = processMessage(requestMessage);
        // 이벤트 소스로 결과 쓰기
        writeToEventSource(eventSource, responseMessage);
}
```

1. Brokerage 확장 – 소스코드 변화 확인

- ✓ 전체 절차에서 이벤트 소스 읽기/쓰기 로직 분리
- ✓ processMessage 로직에 대한 추가 설계가 필요함
  - 새로운 작업을 추가할 때, 소스코드 변경이 필요하기 때문

nexbroker5.Brokerage.processMessage()외

```
public TransferMessage processMessage(TransferMessage receivedMessage) {
        // 블록메시지를 Xml 메시지로 변환
        MessageTransformer transformer = new MessageTransformer(this.blockMessageConfig);
        XmlMessage xmlRequestMessage = transformer.transformBlockToXml((BlockMessage)receivedMessage);

        // 서비스 서버로 서비스 요청
        HttpSender httpSender = new HttpSender();
        XmlMessage xmlResponseMessage = httpSender.send(xmlRequestMessage);

        // Xml메시지를  블록 메시지로 변환
        BlockMessage blockResponseMessage = transformer.transformXmlToBlock(xmlResponseMessage);

        return blockResponseMessage;
}
protected TransferMessage readFromEventSource(EventSource eventSource) {
        // 이벤트 소스 읽기
        return getEventWorker().read(eventSource);
}
protected void writeToEventSource(EventSource eventSource, TransferMessage transferMessage) {
        // 이벤트 소스로 쓰기
        getEventWorker().write(eventSource, transferMessage);
}
```

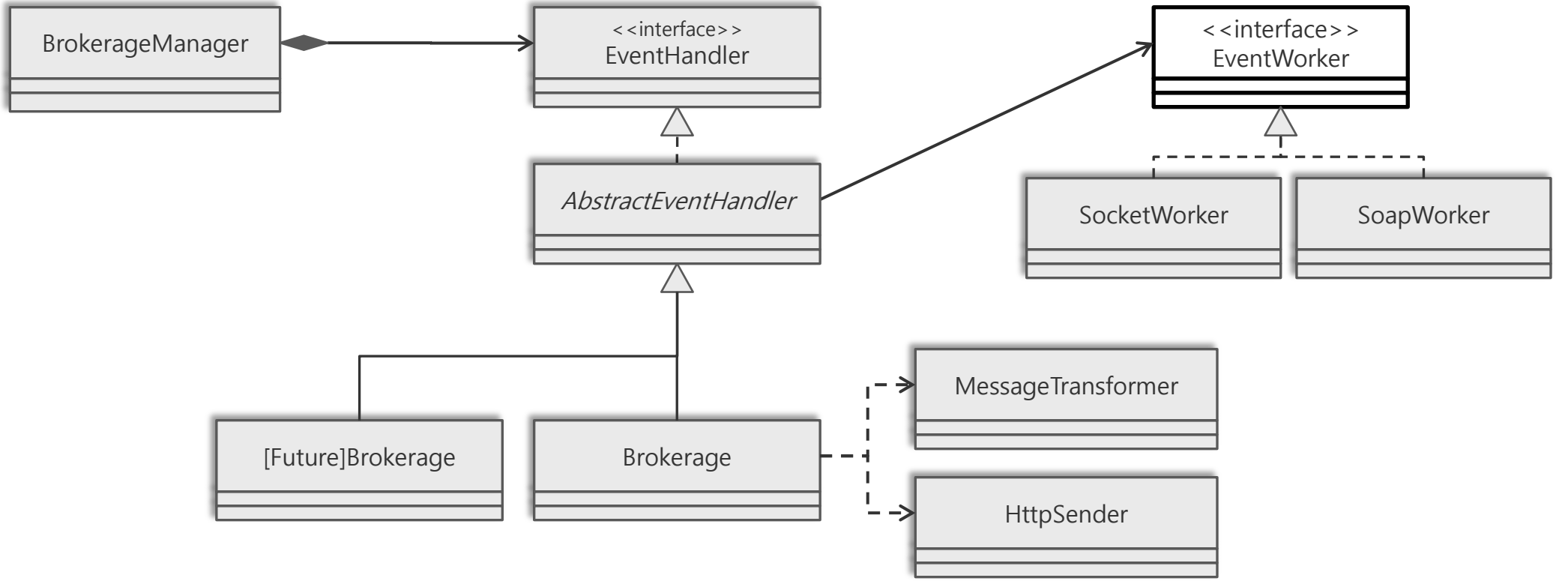
일반화를 통한 분리

1. Brokerage 확장 – 지금까지 확장 결과

NEXTREE

✓ 중개 로직 일부에 대한 일반화를 한 구조

- 처리 로직 일반화 – AbstractEventHandler와 TemplateMethod
- 이벤트소스 작업 일반화 – EventWorker의 read/write 행위

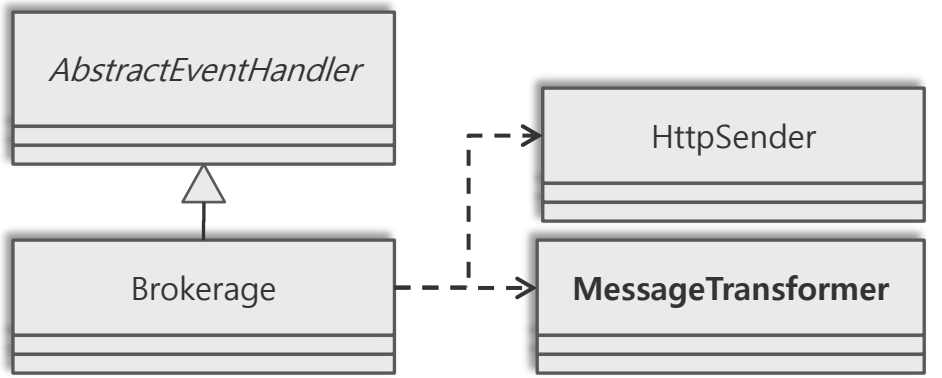


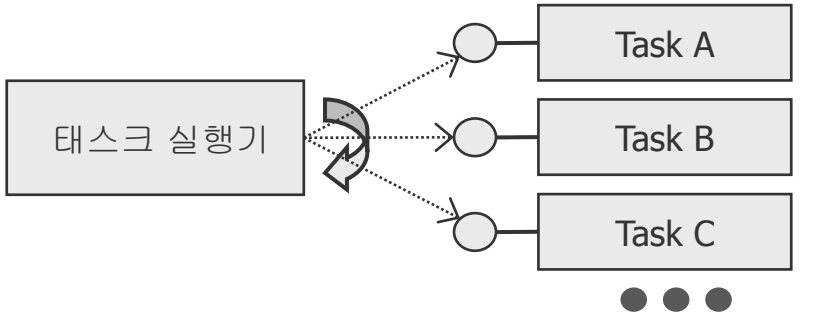
1. Brokerage 확장 – 태스크 추가 메커니즘

NEXTREE

✓ 태스크 추가 가능한 메커니즘을 통해 알고리즘 문제를 해결함

- 태스크 일반화
- 태스크 실행 일반화






```
public TransferMessage processMessage(TransferMessage receivedMessage) {
        // 블록메시지를 Xml 메시지로 변환
        MessageTransformer transformer = new MessageTransformer(this.blockMessageConfig);
        XmlMessage xmlRequestMessage = transformer.transformBlockToXml((BlockMessage)receivedMessage);

        // 서비스 서버로 서비스 요청
        HttpSender httpSender = new HttpSender();
        XmlMessage xmlResponseMessage = httpSender.send(xmlRequestMessage);

        // Xml메시지를  블록 메시지로 변환
        BlockMessage blockResponseMessage = transformer.transformXmlToBlock(xmlResponseMessage);

        return blockResponseMessage;
}
```

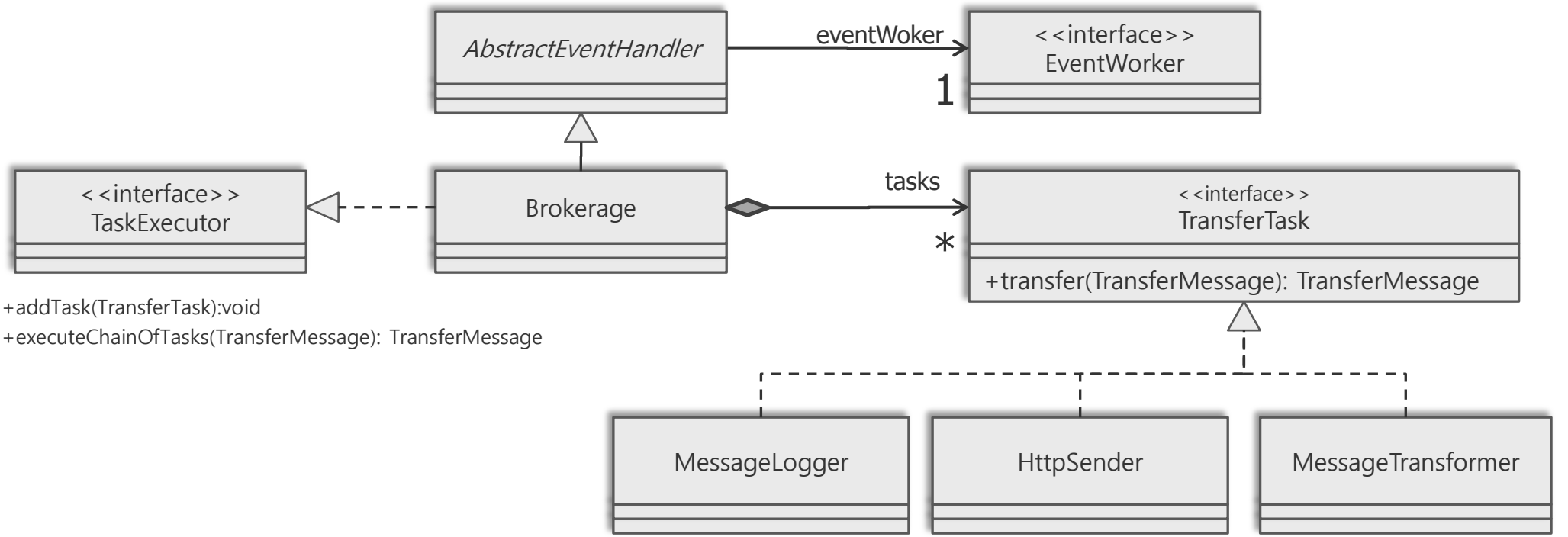
변경이 필요한 알고리즘

1. Brokerage 확장 – 태스크 추가 메커니즘

NEXTREE

✓ 태스크 일반화

- 여러 작업(Tranformation, Sending)은 In/Out으로 연계된 일련의 태스크임
- TransferTask 라는 개념으로 일반화
- TransferTask 인터페이스 도입[transfer(TransferMessage):TransferMessage]

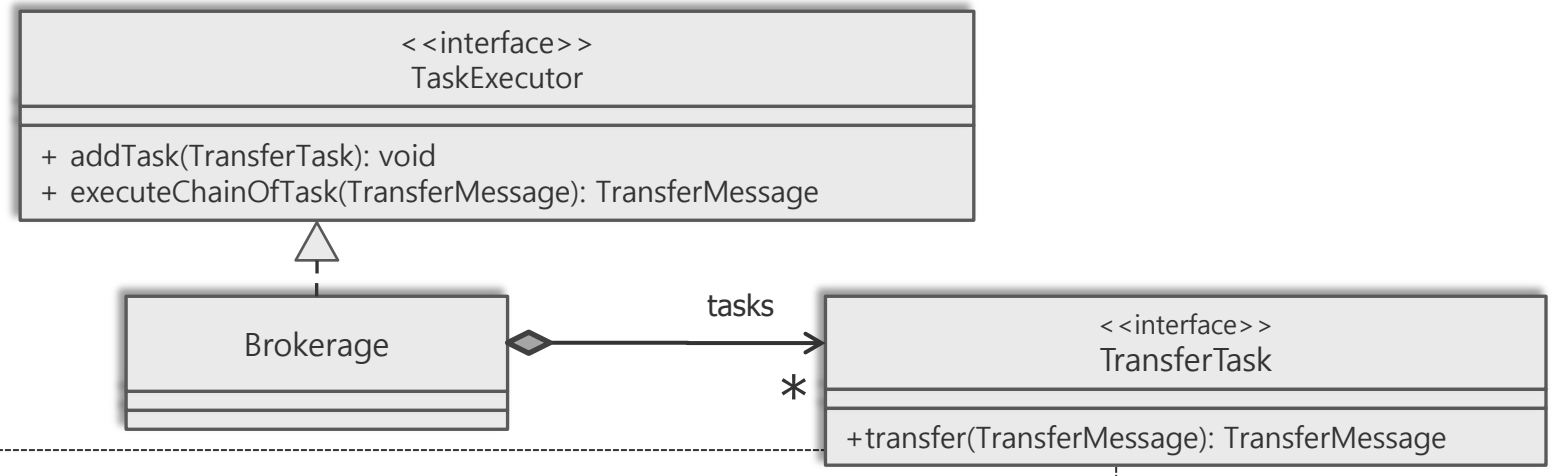


1. Brokerage 확장 – 태스크 추가 메커니즘

NEXTREE

✓ 태스크 실행 일반화

- 갯수가 지정되지 않은 일련의 태스크를 실행함 – executeChainOfTasks()
- 태스크를 추가할 수 있음 – addTask(TransferTask)



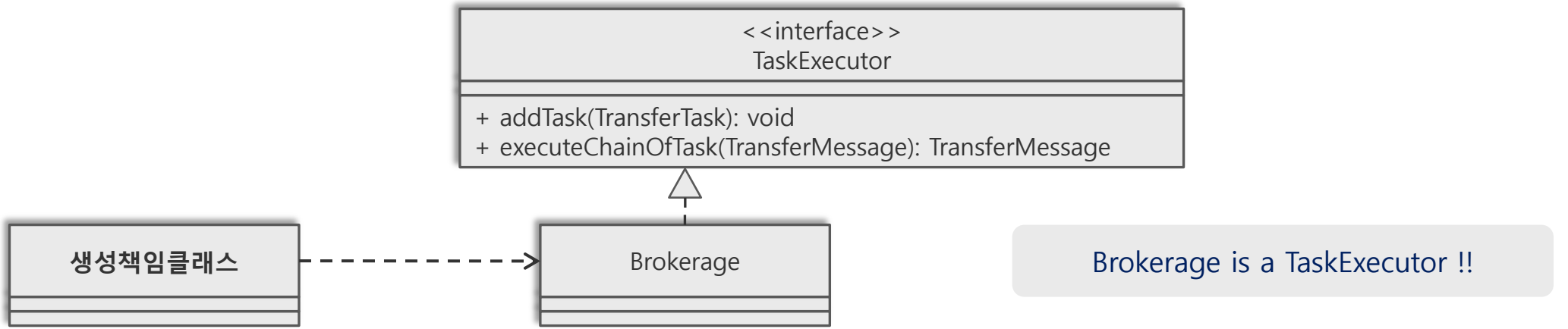

```
public TransferMessage executeChainOfTasks(TransferMessage transferMessage) {
        Iterator<TransferTask> iter = taskRepository.iterator();
        TransferMessage inputMessage = transferMessage;
        TransferMessage outputMessage = null;
        while (iter.hasNext()) {
                TransferTask task = iter.next();
                outputMessage = task.transfer(inputMessage);
                inputMessage = outputMessage;
        }

        return outputMessage;
}
```

1. Brokerage 확장 – 태스크 추가 메커니즘

NEXTREE

✓ 태스크를 추가하는 시점은?

- Brokerage가 수행하는 태스크에 대한 지식은 팩토리가 가지고 있음
- 따라서, Brokerage 생성책임 객체가 Brokerage객체를 생성할 때 TransferTask를 추가함



Brokerage is a TaskExecutor !!

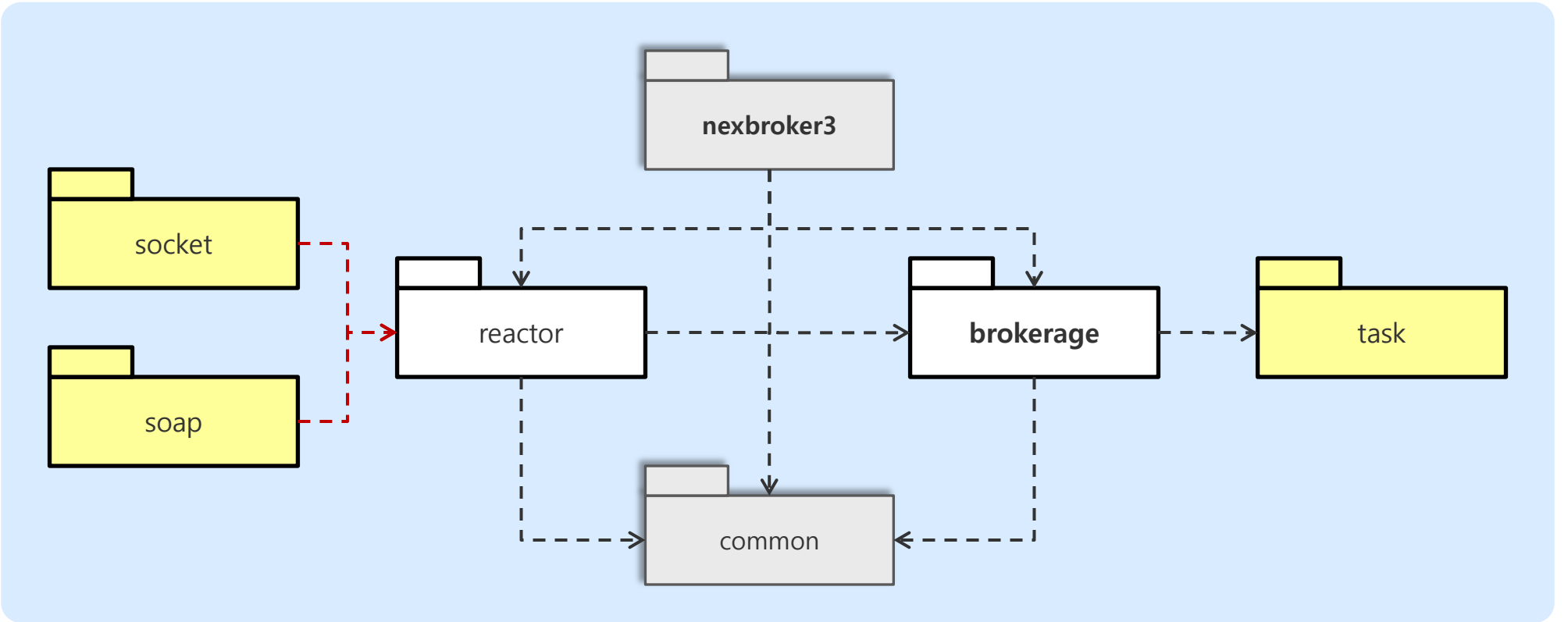
```
class Brokerage생성책임클래스{
        ...
        private Brokerage createSocketBrokerage() {
                EventWorker eventWorker = new SocketWorker(new BlockMessageConfig());
                Brokerage newBrokerage = new Brokerage(eventWorker);

                newBrokerage.addTask(new MessageTransformer(new BlockMessageConfig()));
                newBrokerage.addTask(new HttpSender());
                newBrokerage.addTask(new MessageTransformer(new BlockMessageConfig()));

                return newBrokerage;
        }
}
```

1. Brokerage 확장 – 태스크 추가 메커니즘

NEXTREE

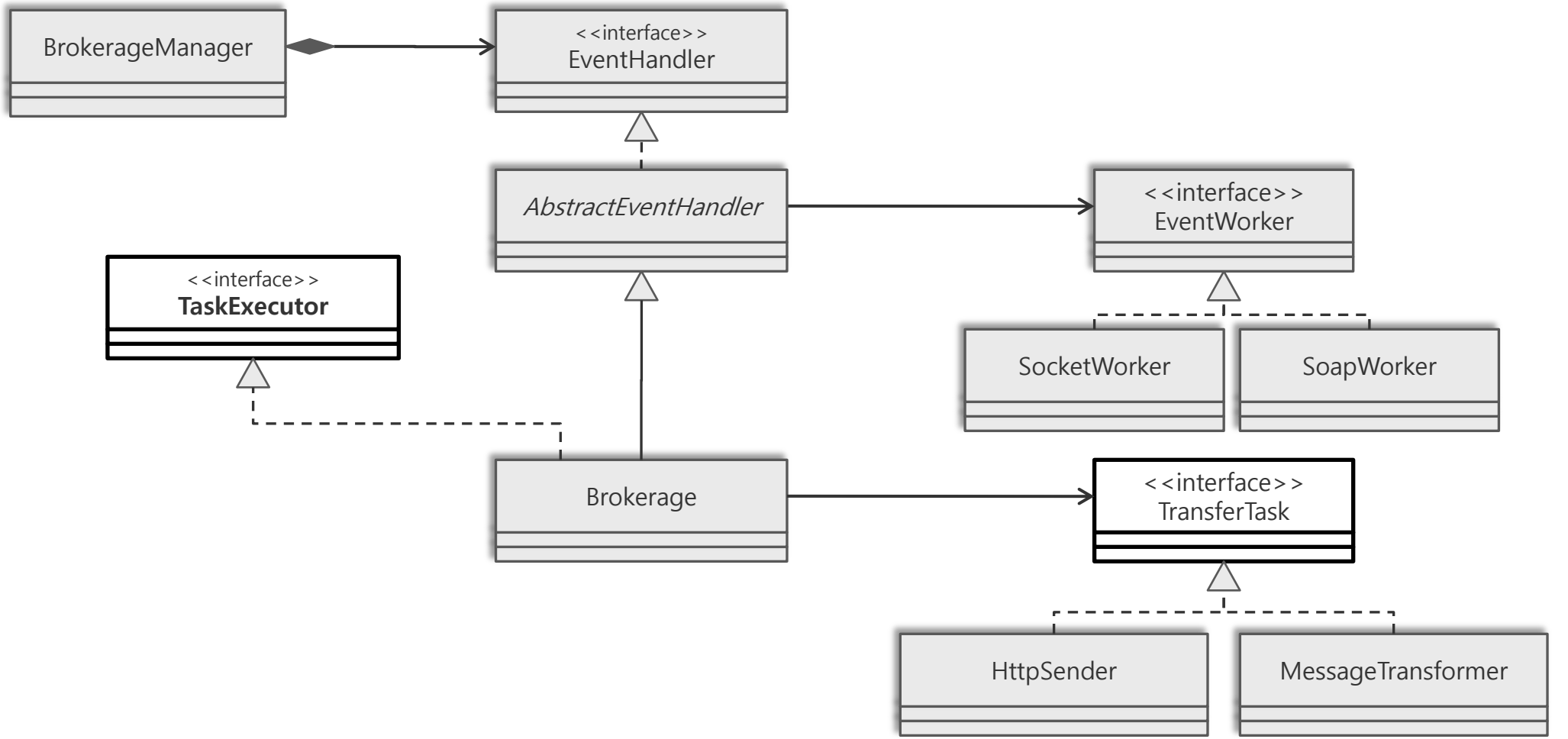
✓ task 패키지 추가

- Task 객체들(HttpSender, MessageTransformer)분리 수용
- 인터페이스 분리 수용
- TransferTask, TaskExecutor



1. Brokerage 확장 – 지금까지 확장 결과

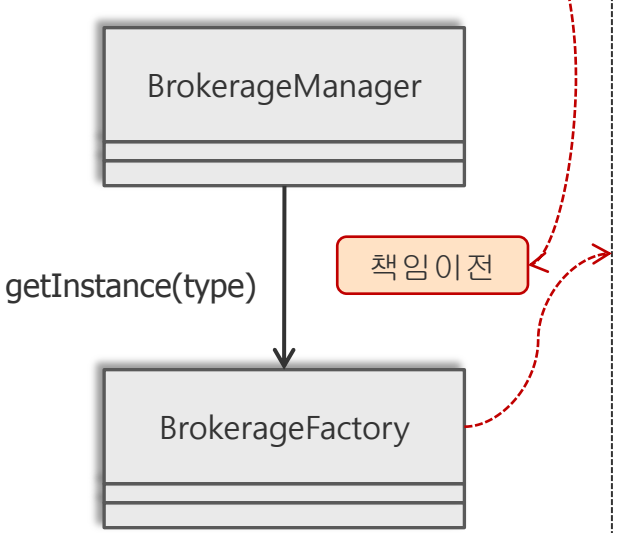
- ✓ 개별 Task 일반화 작업 수행
- ✓ Task 실행 일반화 작업 수행



1. Brokerage 확장 – Brokerage 생성

- ✓ 누가 Brokerage를 만들 것인가?
- ✓ Brokerage를 만들기 위해 알아야 할 것들,
  - 구체적인 EventWorker
  - 필요한 TransferTask 객체와 실행 순서
- ✓ BrokerageFactory

이벤트핸들러관리자:	재사용 가능한 중개(brokerage) 객체 50개를 만들어야지... (불만: 왜 내가 중개객체를 직접만들어야지, 여러 가지를 만들려니 알아야 할 것도 많고...) ...



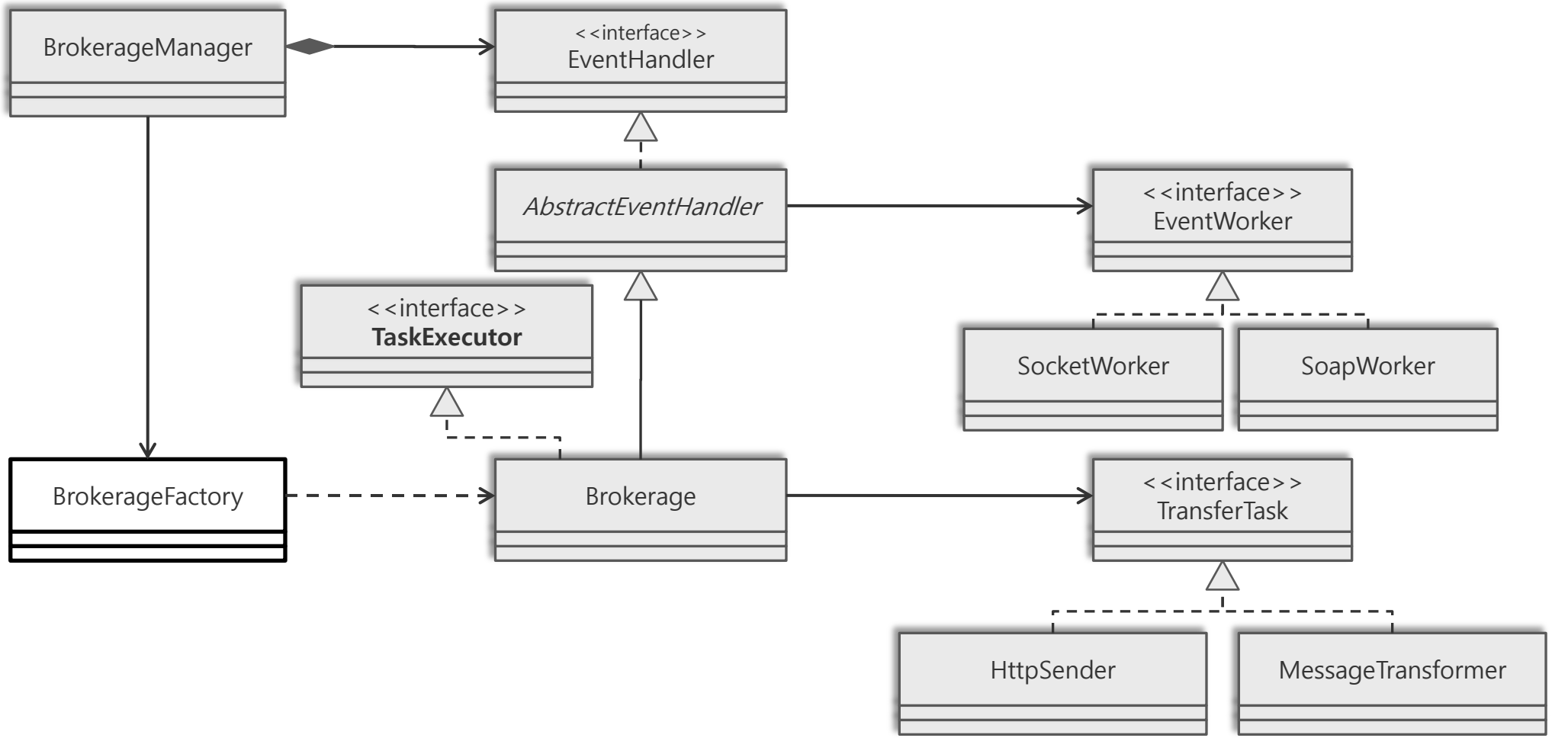

```
public Brokerage getInstance(int type) {
        if (type == EventSource.SOCKET_EVENT) {
                return createSocketBrokerage();
        }
        ...
}

private Brokerage createSocketBrokerage() {
        EventWorker eventWorker = new SocketWorker(new BlockMessageConfig());
        Brokerage newBrokerage = new Brokerage(eventWorker);

        newBrokerage.addTask(new MessageTransformer(new BlockMessageConfig()));
        newBrokerage.addTask(new HttpSender());
        newBrokerage.addTask(new MessageTransformer(new BlockMessageConfig()));

        return newBrokerage;
}
```

1

2

1. Brokerage 확장 – 지금까지 확장 구조

- ✓ BrokerageManager의 불만해결
- ✓ BrokerageFactory가 Brokerage 생성을 책임짐

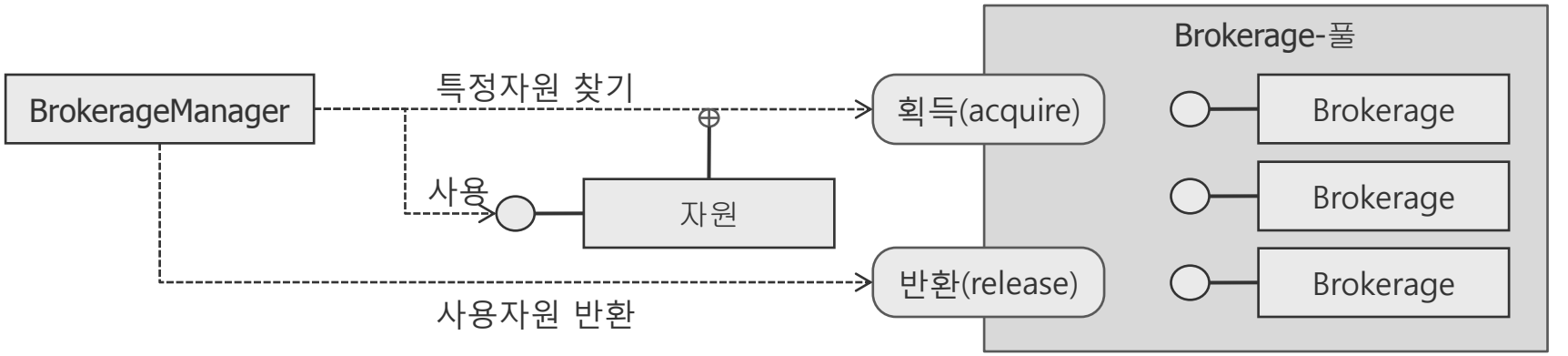


1. Brokerage 확장 – Brokerage 객체 재사용

NEXTREE

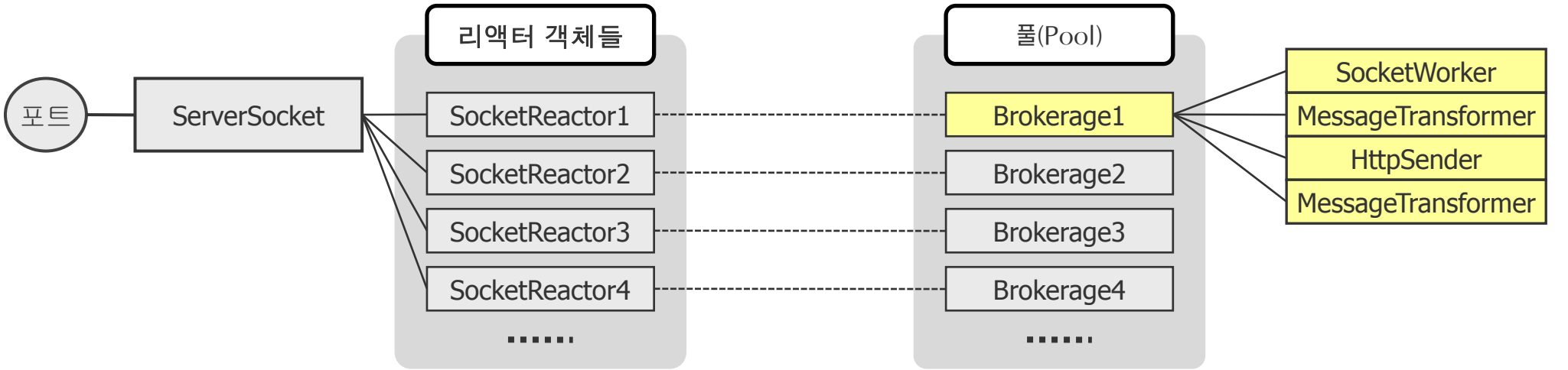
✓ 객체 풀링을 통한 재사용

- ResourcePool 패턴 활용 - BrokeragePool
- Brokerage 객체 획득(acquire) 인터페이스
- Brokerage 객체 반환(release) 인터페이스



1. Brokerage 확장 – Brokerage 객체 재사용

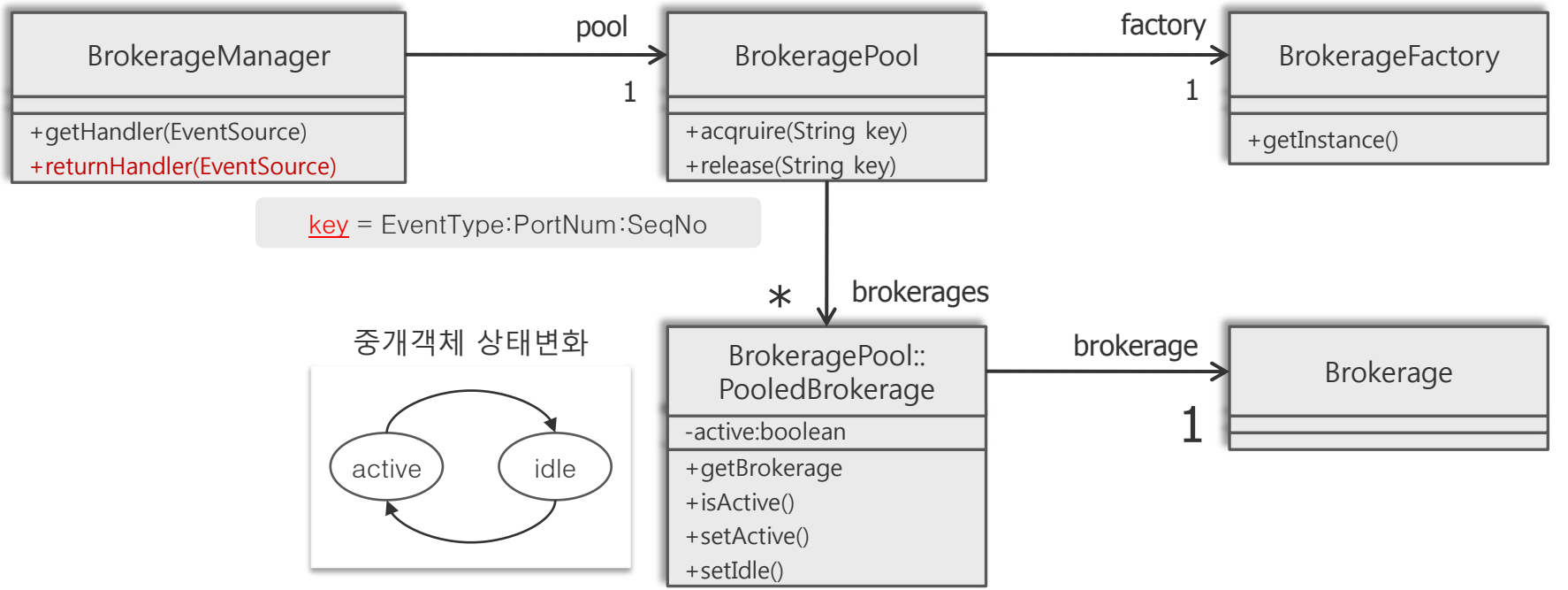
- ✓ Brokerage 객체는 매 요청 시 필요함
  - 매번 생성하게되면, 한 번에 총 5개 객체가 생성함, Pool을 이용한 재사용 필요함
- ✓ ObjectPool에서 객체 매핑 전략
  - 리액터와 Brokerage를 1:1 매핑 (키:이벤트타입:포트번호:순번)
  - Brokerage를 이벤트 별 랜던하게 할당
- ✓ ObjectPool에서 생성 전략
  - 지정 갯수 만큼 미리 Brokerage 생성함
  - 핸들러 요청 시에만 해당 Brokerage 생성



1. Brokerage 확장 – Brokerage 객체 재사용

✓ 풀 객체 설계 및 구조

- Brokerage 객체 생성 책임을 가짐 – BrokerageFactory 보유
- Brokerage 객체의 상태 관리 필요 – PooledBrokerage 등장
- 사용중: active, 대기중: idle
- Brokerage 객체는 리액터 객체와 1:1 매핑- 연결고리는 key값

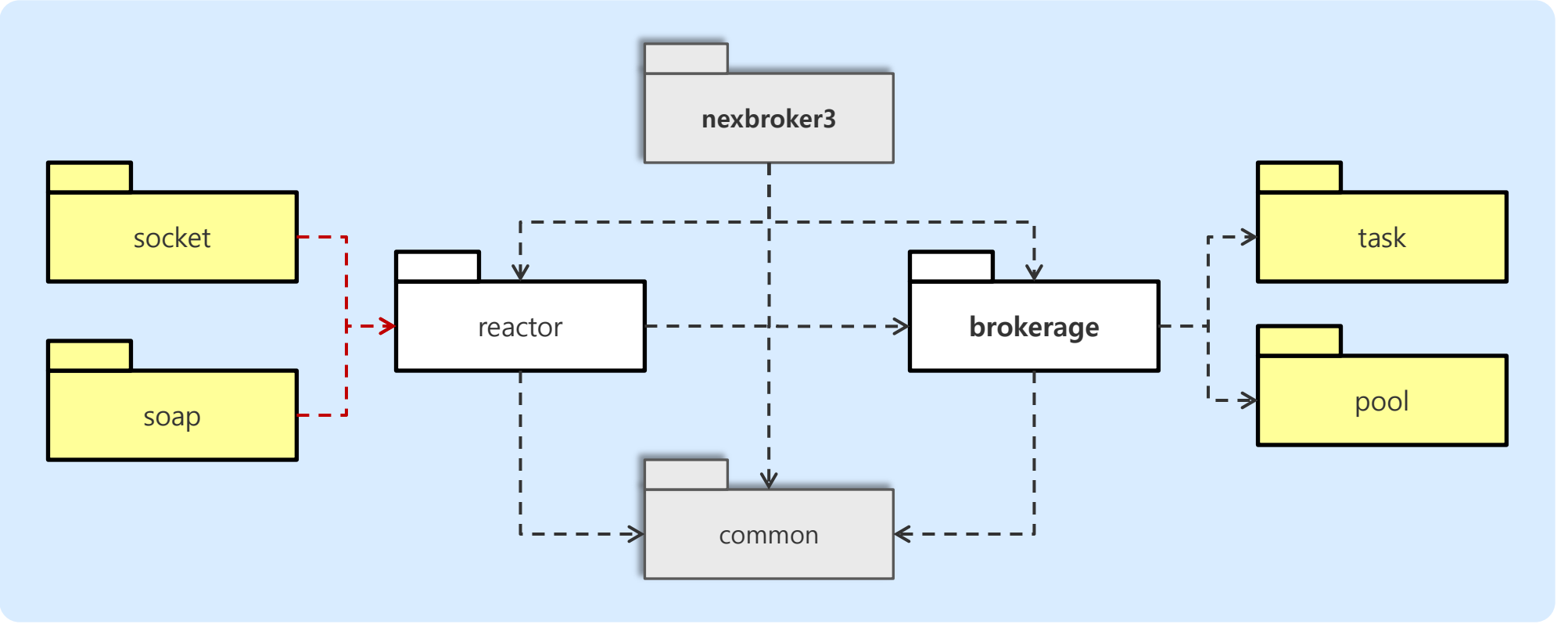


1. Brokerage 확장 – Brokerage 객체 재사용

NEXTREE

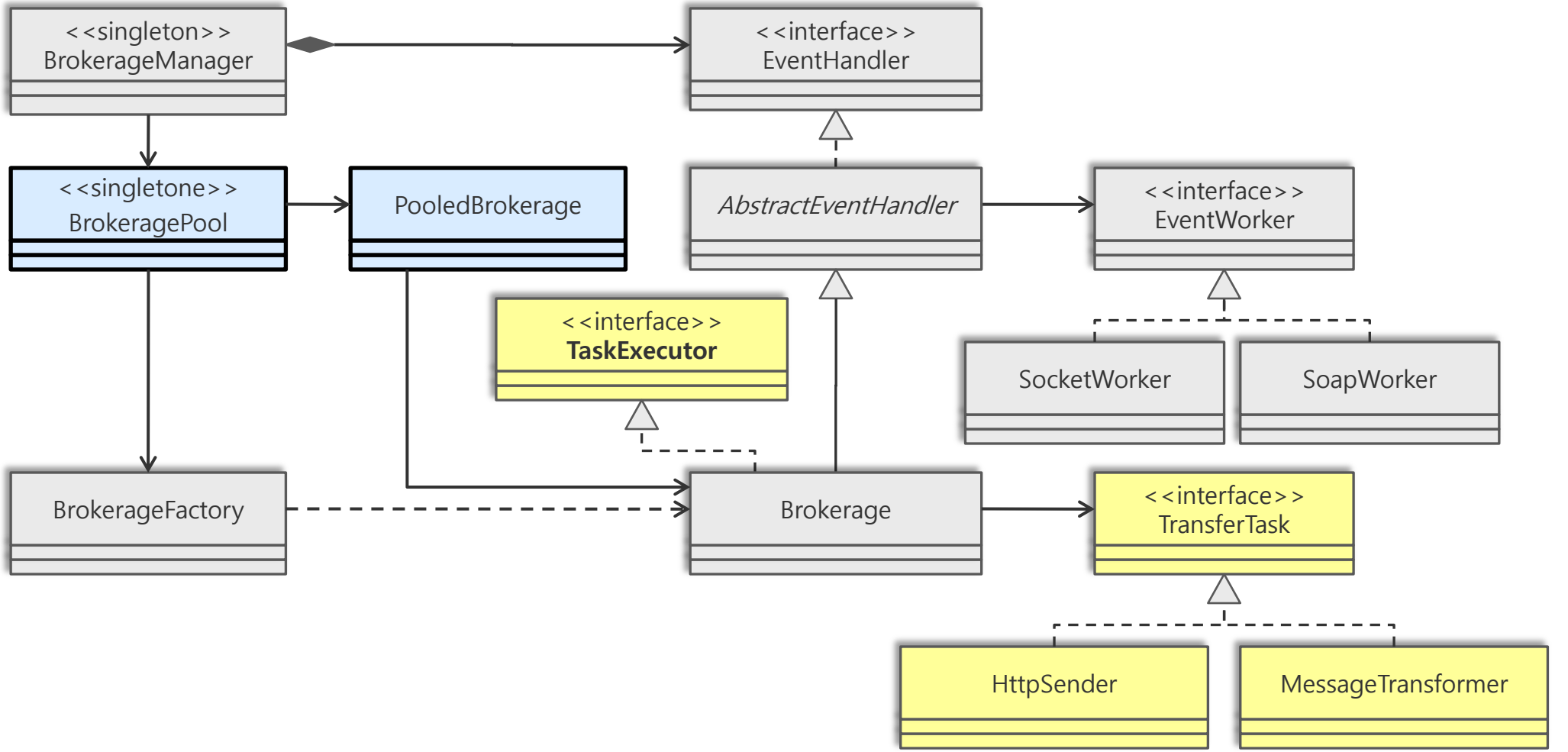
✓ pool 패키지 추가

- BrokeragePool, PooledBrokerage 클래스 추가
- 풀 구현의 다양할 수 있음, 따라서 분리하여 둠
- 직접 구현할 수도 있고, 아파치의 commons 컴포넌트의 ObjectPool을 사용할 수 도 있음



1. Brokerage 확장 – 지금까지 확장 구조

✓ BrokeragePool 추가로 Brokerage 객체 재사용 가능함



2. 질의 응답 및 토론

- ✓ 질의 응답
- ✓ 토론

❖넥스트리소프트(주)
❖송태국 (tsong@nextree.co.kr)
❖부사장/CTO