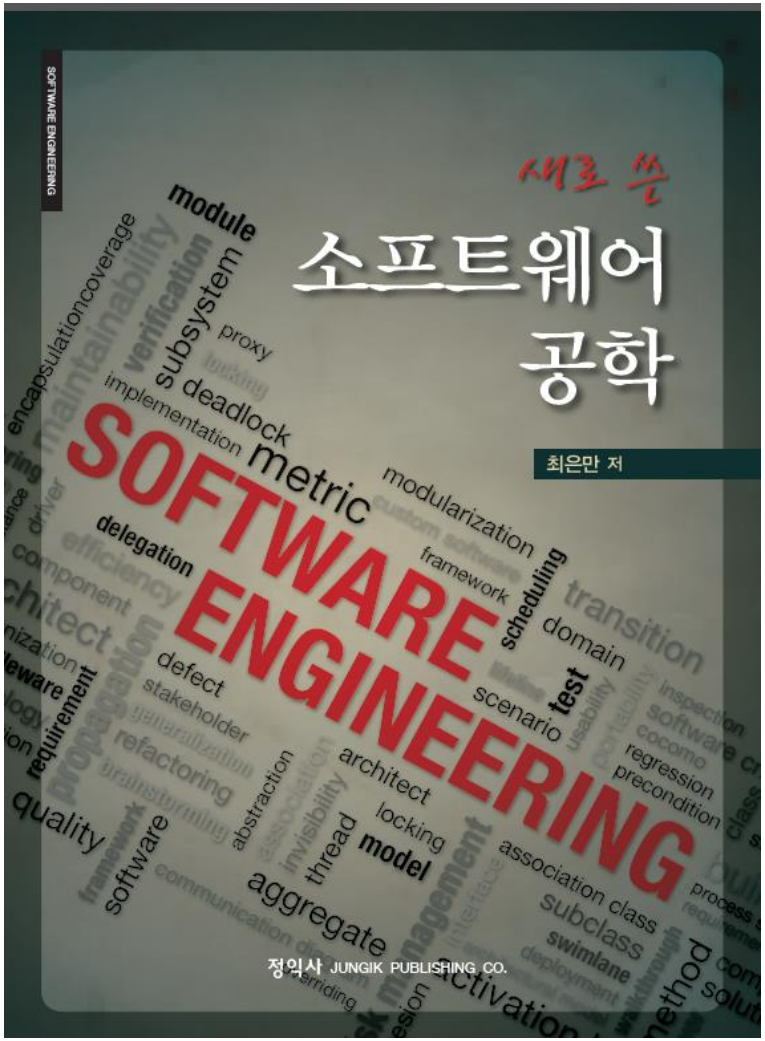
소프트웨어 공학

Lecture #7: 상세 설계



최은만 저

6차 개정판

학습 목표

- 설계 패턴
- 클래스 설계
- 사용자 인터페이스 설계
- 데이터 설계

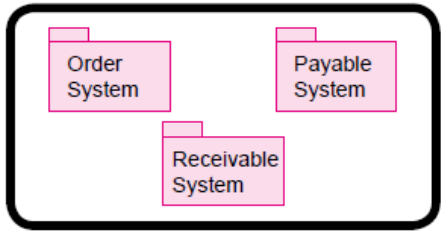
상세 설계

- 상세 설계 목적
  - 설계 모델을 작동하는 소프트웨어로 변환
  - 아키텍쳐-모듈 사이의 추상 수준의 갭을 줄이기 위함

추상적인 수준(인터페이스 정의 위주)의 도메인 클래스들와 시스템 아키텍쳐 ➔
클래스 내부 설계 및 관계 개선,
UI /저장소 설계,
관련 클래스 추가,
미들웨어 관련 클래스 추가 등

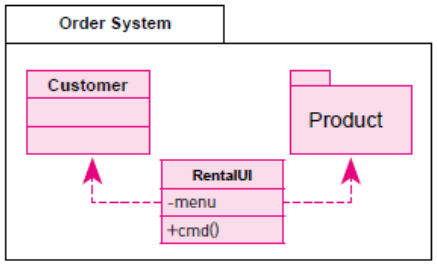
- 패러다임에 따른 작업
  - 구조적 방법
    - 프로시저 안의 로직 (알고리즘) 설계
  - 객체지향 방법
    - 클래스 안의 메소드 설계

상세 설계



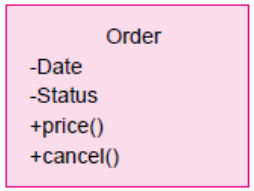
아키텍처 설계

서브시스템이나 패키지 수준
– 아키텍처 스타일이 적용되는 수준



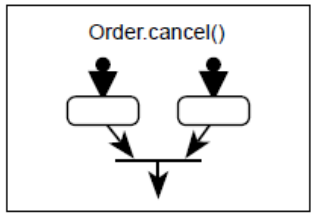
서브시스템 설계

패키지 내부의 클래스
– 설계 패턴이 적용되는 수준



클래스 설계

클래스 내부의 데이터와 메소드 수준



메소드 설계

메소드 내부 알고리즘 설계

7.1 설계 패턴

- 디자인 패턴이란?
  - 소프트웨어 설계에서 자주 발생하는 문제에 대한 일반적이고 반복적인 해결책을 말한다.
  - 여러가지 상황에 적용될 수 있는 일종의 템플릿

- 디자인 패턴 구성 요소
  - 패턴의 이름과 구분
  - 문제 및 배경 – 패턴이 사용되는 분야 또는 배경
  - 솔루션 – 패턴을 이루는 요소들, 관계, 협동과정
  - 사례 – 적용 사례
  - 결과 – 패턴의 이점, 영향
  - 샘플 코드 – 예제 코드

설계 패턴

출처: Design Patterns
저자: 사인방(Gang of Four, 줄여 GoF)으로서, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

- **패턴의 분류**
  - **Gamma의 23개 패턴**

		목적에 의한 분류		
		생성유형	구조적	행위적
범위	클래스	팩토리 메소드	어뎁터(클래스)	인터프리터 템플릿 메소드
	객체	추상 팩토리 싱글톤 프로토타입 빌더	어뎁터(객체) 브리지 컴포지트 데코레이터 퍼싸드 플라이웨이트 프록시	책임 체인 커맨드 반복자 중재자 메멘토 옵서버 상태 전략 비지터
		객체 생성	객체 결합	객체 들간의 메시지 교환

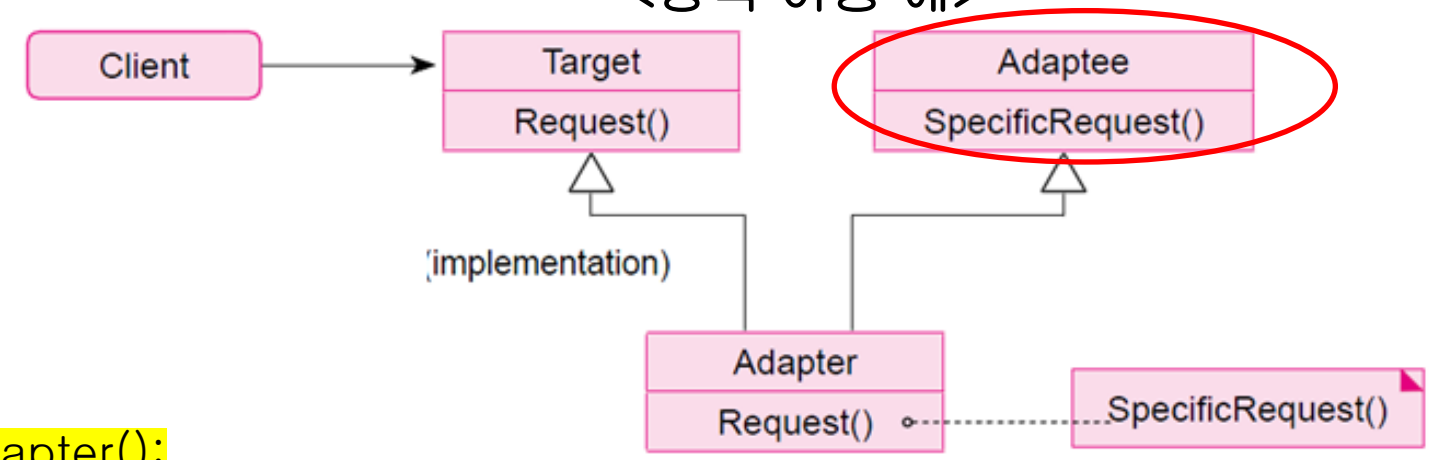
어댑터 패턴

- 이미 개발된 클래스, 즉 레거시 시스템의 인터페이스를 다른 클래스의 요구에 맞게 인터페이스를 변환해주는 것

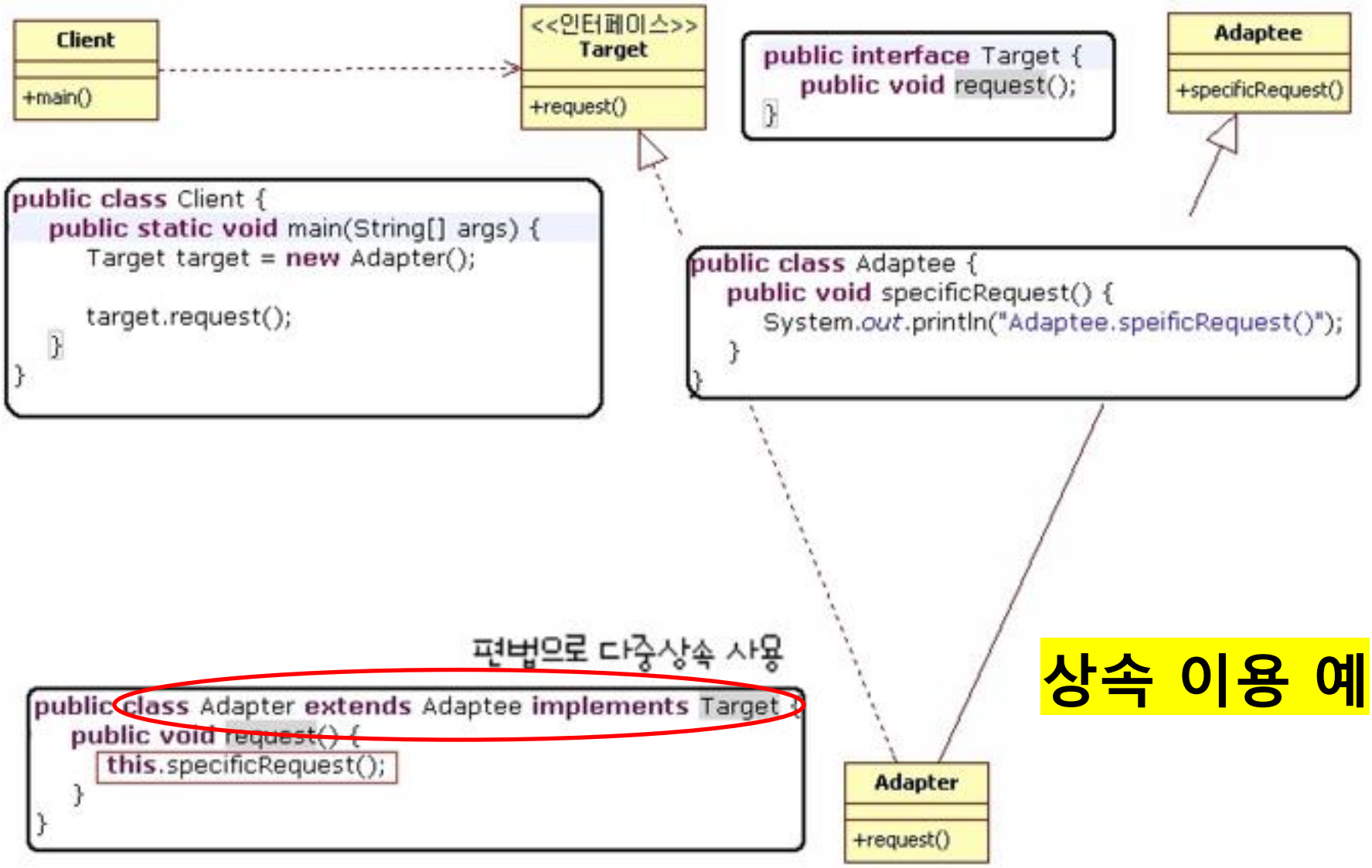
- 이미 만들어져 있는 클래스를 사용하고 싶지만 인터페이스가 원하는 방식과 일치하지 않을 때 사용

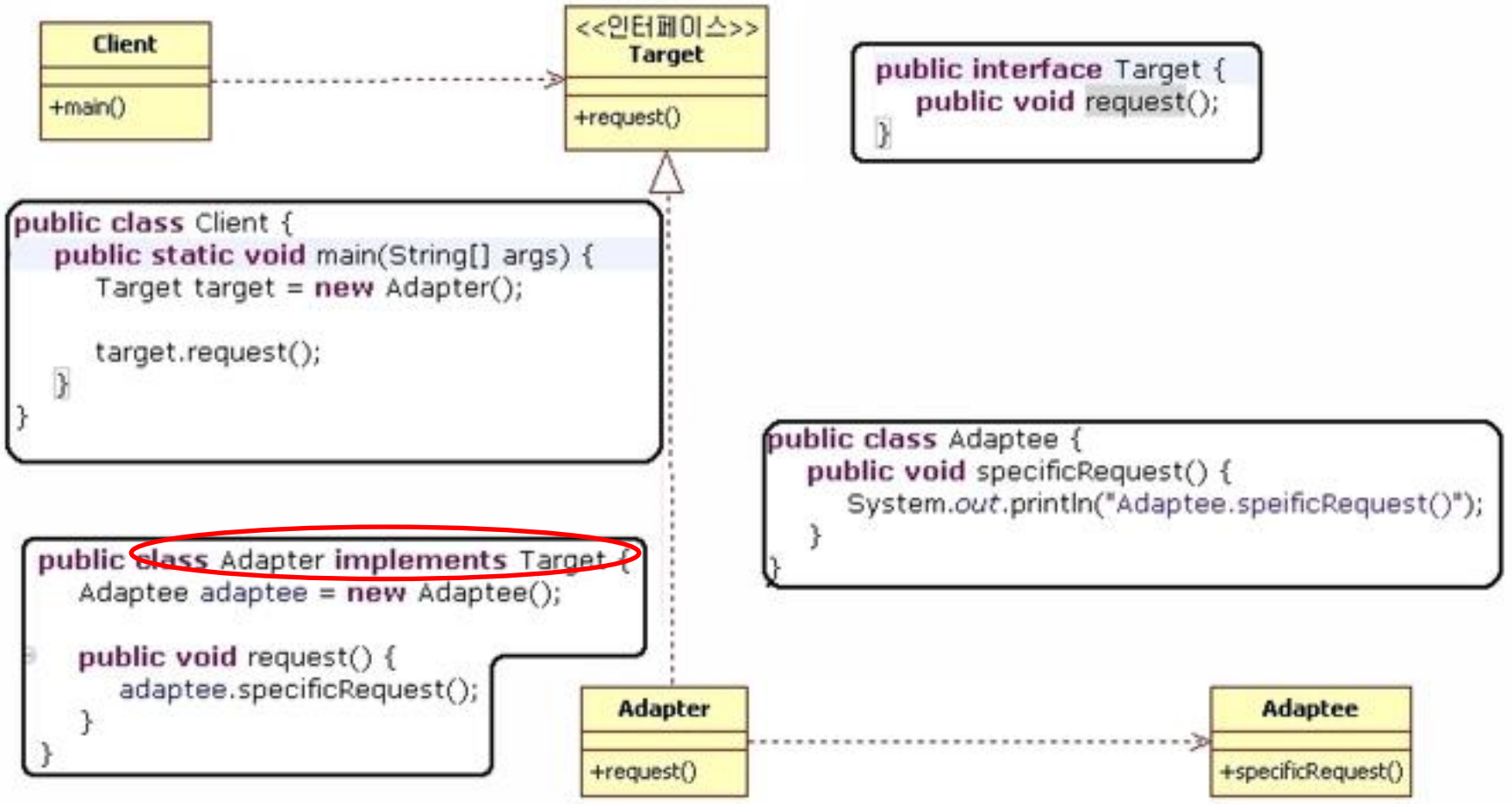
- 패턴 적용 방법
  - 위임을 이용
  - 상속을 이용

<상속 이용 예>




```
//Client
Target a = new Adapter();
a.Request();
///
```


Client
+main()
<<인터페이스>>
Target
+request()
public interface Target {
public void request();
}
Adaptee
+specificRequest()
public class Client {
public static void main(String[] args) {
Target target = new Adapter();
target.request();
}
}
public class Adaptee {
public void specificRequest() {
System.out.println("Adaptee.speificRequest()");
}
}
편법으로 다중상속 사용
public class Adapter extends Adaptee implements Target {
public void request() {
this.specificRequest();
}
}
Adapter
+request()
상속 이용 예


Client
+main()
<<인터페이스>>
Target
+request()
public interface Target {
public void request();
}
public class Client {
public static void main(String[] args) {
Target target = new Adapter();
target.request();
}
}
public class Adaptee {
public void specificRequest() {
System.out.println("Adaptee.speificRequest()");
}
}
public class Adapter implements Target {
Adaptee adaptee = new Adaptee();
public void request() {
adaptee.specificRequest();
}
}
Adapter
+request()
Adaptee
+specificRequest()

위임 이용 예

7.2 클래스 설계

- 분석 단계에서 아직 확정되지 않은 클래스 내부 부분 중 구현에 필요한 중요한 사항을 결정하는 작업
  - 클래스 추출 및 클래스 간 관계 분석
- 상태와 오퍼레이션의 관계를 상세히 설계해야 함
- 클래스가 가지는 속성 값에 따라 오퍼레이션 구현이 달라짐
  - 객체의 상태 변화 모델링 필수

클래스 인터페이스의 정의

- 왜 클래스의 인터페이스 정의가 중요한가
  - 클래스 구현 할 때 ➔ 내부 자료구조, 오퍼레이션의 프로토타입에 관심
  - 클래스 사용 할 때: public 멤버에 관심, 어떤 동작을 하고, 어떻게 호출하고, 호출시 지켜야하는 약속에 관심.
  - 클래스 확장 할 때: public 멤버에 관심. 어떤 동작하는 가에 관심

- API(Application Program Interface) 정의
  - 서브시스템의 서비스 인터페이스

클래스 인터페이스의 정의

- 협약에 의한 설계(Design by Contract)
  - 소프트웨어 컴포넌트에 대한 정확한 인터페이스 명세를 위하여 선행조건, 결과조건, 불변조건을 나타내는 설계 방법
  - 선행조건(precondition)
    - 오퍼레이션이 호출되기 전에 참이 되어야 할 조건
  - 결과조건(postcondition)
    - 오퍼레이션이 수행된 후 만족하여야 하는 조건
  - 불변조건(invariant)
    - 클래스 내부가 실행되는 동안 항상 만족하여야 하는 조건
    - ex) 리스트에 있는 노드가 항상 오름차순으로 되어야 함

Design by Contract Example

```
//Example (using JavaDoc tags)
 /*
 @invariant forall Node n in element () :
      n.prev() != null
        implies
             n.value().compareTo(n.prev().value() > 0)
 */
Public class OrderedList {
  /*
     @pre contains(aNode) == false
     @post contains(aNode) == true
  */
  public void insertNode(final Node aNode) {
    ...
  }
```

The "doc comments" format used by Javadoc is the de facto industry standard for documenting Java classes.

- In many programming languages, contracts are implemented with assert. Asserts are by default compiled away in release mode in C/C++, and similarly deactivated in C# and Java.

- Design by contract does not replace regular testing strategies, such as unit testing, integration testing and system testing. Rather, it complements external testing with internal self-tests.

클래스 재사용 – (1) 프레임워크

- 실정에 맞도록 커스터마이즈 할 수 있게 고안된 재사용 가능한 부분적인 응용 프로그램.

- 객체지향에서 재사용 빈도가 높은 것 ➔(전문가의 지식을 토대로) 프레임워크로 만들어 재사용

- 자료 처리나 이동 통신과 같은 특수한 기술을 목표로 함
  - 라이브러리와 달리 자료처리, 통신, UI등 응용 도메인을 목표로 함.

- 응용 프레임워크
  - 재정의 할 수 있는 후크 메소드(hook method)가 있어 확장성을 향상
  - 후크 메소드 (일반적으로 extension point, hot spot)
    - 응용 도메인의 인터페이스와 동작이 특수한 경우 약간의 차이를 체계적으로 분리할 수 있음

클래스 재사용 – (1) 프레임워크

- **프레임워크의 위치에 따라 구분**
  - **인프라 구조 프레임워크: SW 개발 간소화 목적. OS, 디버거, 통신시스템 등**
  - **미들웨어 프레임워크: 분산된 응용들간의 정보교환 및 통합**
  - **엔터프라이즈 응용 프레임워크: 통신, 항공기, 환경, 제조, 금융 등과 같은 비즈니스 액티비티와 같은 도메인 전용**

    (Security, Logging, Auditing, Utilities, Configuration, Metadata, User accounts management, Common Business Objects, Generic Workflow, Common data access layer 등 지원)

클래스 재사용 – (1) 프레임워크

- **프레임워크 확장하는데 사용되는 기술**
  - **화이트박스 프레임워크**
    - 프레임워크 기초 클래스(base class)를 상속하고, Template Method와 같은 디자인 패턴을 사용하여 미리 정의된 후크 메서드(hook method)를 재정의(overriding)하는 방식
    - **프레임워크의 내부 구조에 대한 사전 지식이 필요.**
    - 프레임워크 클래스 계층도의 세부사항과 밀접하게 결합되어 유연성이 결여된 시스템을 구축할 가능성이 많아 짐
  - **블랙박스 프레임워크**
    - 객체 합성(object composition)이나 위임(delegation)을 사용하여 구현
    - 프레임워크에서 정의한 인터페이스를 실현하는 컴포넌트를 구현하고, Strategy와 같은 디자인 패턴을 사용하여 이 컴포넌트를 프레임워크 안에 위임하여 통합함.
    - 화이트박스 프레임워크 보다, 사용하거나 확장하기는 쉽지만 설계하거나 구현하기는 더 어려움

Whitebox Framework Example

```
package ar.cpfw.whitebox;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

/* This framework allows you to show text on a dialog box.
   To use it, you have to extends from the PrintOnScreen class
   and implement their abstract method.
   The framework provides the PrintOnScreen#print method which
   makes the magic. */
public abstract class PrintOnScreen {
  public void print() { //template method
    JFrame frame = new JFrame();
    JOptionPane.showMessageDialog(frame, textToShow());
    frame.dispose();
  }

  protected abstract String textToShow(); //hook method
}
```

```
package ar.cpfw.utilization;
import ar.cpfw.whitebox.PrintOnScreen;
//Client
public class MyApplication extends PrintOnScreen {
  @Override
  protected String textToShow() {
    return "printing this text on "
        + "screen using PrintOnScreen "
        + "white Box Framework";
  }
}
```

```
//Client
package ar.cpfw.utilization;

public class Main {
  public static void main(String args[]) {
    MyApplication m = new MyApplication();
    m.print();
  } }
```

Blackbox Framework Example

```
package ar.cpfw.blackbox;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
/* This framework allows you to show text on a dialog box.
   To use it, you have to implement the interface TextToShow.
   Then, create an instance of the class PrintOnScreen passing
   your implementation of TextToShow as a constructor param. */
public final class PrintOnScreen {
  TextToShow textToShow;
  public PrintOnScreen(TextToShow text) {
    this.textToShow = text;
  }

  public void print() {
    JFrame frame = new JFrame();
    JOptionPane.showMessageDialog(frame, textToShow.text());
    frame.dispose();
  } }
```

```
package ar.cpfw.blackbox;

public interface TextToShow {
   String text(); //hook method
}
```

```
//Client
package ar.cpfw.utilization;
import ar.cpfw.blackbox.TextToShow;
public class MyTextToShow implements TextToShow {
  @Override
  public String text() {
    return "printing this text on "
         + "screen using PrintOnScreen "
         + "black Box Framework";
  }
}
```

```
//Client
package ar.cpfw.utilization;

import ar.cpfw.blackbox.PrintOnScreen;
public class Main {
  public static void main(String args[]) {
    // MyTextToShow 컴포넌트를 만들어, PrintOnScreen 프레임워크에
    // 위임 방법으로 통합한다.
    PrintOnScreen m = new PrintOnScreen (new MyTextToShow());
    m.print();
  }
}
```