Spring in Action

THIRD EDITION

제1부

스프링 핵심 개념

스프링이 제공하는 기능은 매우 다양하지만 핵심을 파고들어가 보면 스프링의 주요 기능은 종속객체 주입(DI: Dependency Injection)과 애스펙트 지향 프로그래밍(AOP: Aspect-Oriented Programming)으로 귀결된다. **1장 "스프링 속으로"**에서는 DI와 AOP의 기본적인 개념을 살펴보고, 이 두 기능이 애플리케이션 객체 간의 결합도를 줄이는 데 어떻게 기여하는지 알아본다.

2장 "빈 와이어링"에서는 DI를 이용해 애플리케이션 객체 간의 결합도를 줄이기 위해 스프링의 XML 기반 설정을 사용하는 방법에 대해 좀 더 깊이 있게 살펴본다. 여기서 애플리케이션 객체를 정의하고 의존객체와 연결하는 방법을 배운다.

XML이 스프링을 설정할 수 있는 유일한 방법은 아니다. 2장에서 배운 내용에 이어 **3장 "XML 설정 최소화"**에서는 최소한의 XML(또는 전혀 XML을 사용하지 않아도) 애플리케이션 객체 연결을 가능하게 하는 스프링의 몇 가지 새로운 기능을 살펴본다.

4장 "애스펙트 지향 스프링"에서는 스프링의 AOP 기능을 이용해 시스템 전반에 걸친 서비스(보안이나 감사 등)와 그 서비스를 받는 객체 간의 결합도를 줄이는 방법을 살펴본다. 이 장은 스프링 AOP를 이용해 선언적 트랜잭션과 보안을 제공하는 방법을 배우게 될 6장과 9장의 기반이 되는 장이다.

Spring in Action
THIRD EDITION

1장
스프링 속으로

1장에서 다룰 내용

- 스프링의 코어 모듈 살펴보기
- 애플리케이션 객체 간의 결합도 줄이기
- AOP를 이용해 횡단관심사 관리하기
- 스프링의 빈 컨테이너

모든 것은 빈(bean)에서 시작됐다.

1996년, 자바(Java)라는 프로그래밍 언어는 아직 참신하고, 흥미진진하고, 전도유망한 언어에 불과했다. 애플릿을 이용해 풍부하고 동적인 웹 애플리케이션을 만드는 데 매료되어 자바 진영으로 모여들었던 많은 개발자는 얼마 지나지 않아 자바 언어의 가능성이 저글링하는 만화 캐릭터를 그려내는 정도에 그치지 않는다는 것을 알게 됐다. 이전의 다른 언어와 달리 자바는 개별 컴포넌트로 이뤄진 복잡한 애플리케이션을 만들 수 있게 해 줬다. 애플릿에 이끌려 자바를 찾아왔던 개발자들은 컴포넌트에 매료되어 남게 됐다.

그해 12월, 썬 마이크로시스템즈가 자바를 위한 소프트웨어 컴포넌트 모델을 정의한 자바빈즈(JavaBeans) 1.00-A 명세를 발표했다. 이 명세는 자바 객체를 재사용이 가능하게 하고 단순한 자바 객체를 이용해 복잡한 애플리케이션을 쉽게 구성할 수 있게 해 주는 일련의 코딩 정책을 정의한 것이었다. 자바빈즈는 이처럼 재사용할 수 있는 애플리케이션 컴포넌트를 정의하는 범용 수단으로 사용하려는 의도로 만들어졌지만, 실제로는 주로 사용자 인터페이스 위젯을 만드는 데 사용되고 있었다. '실질적인' 작업을 하기에는 너무 단순해 보였던 것이다. 엔터프라이즈 개발자들은 그 이상의 것을 원했다.

첨단 애플리케이션은 자바빈즈 명세에서 직접적으로 제공하지 않는 트랜잭션 지원이나 보안, 분산 컴퓨팅 등의 서비스를 필요로 하는 경우가 많다. 그래서 1998년 썬(Sun)은 엔터프라이즈 자바빈즈(EJB: Enterprise JavaBeans) 명세 1.0을 발표했다. 이 명세는 자바 컴포넌트 개념을 서버 측으로 확장해 엔터프라이즈 서비스를 제공한 것이었지만, 원래 자바빈즈 명세가 가졌던 간결함을 그대로 유지하지는 못했다. 사실 EJB는 이름 말고는 자바빈즈 명세와 유사한 점이 전혀 없다.

EJB를 기반으로 한 많은 애플리케이션이 성공적으로 구축되어 왔지만, EJB는 엔터프라이즈 애플리케이션 개발을 간소화하겠다는 본래의 취지에 전혀 부합하지 못했다. EJB의 선언적 프로그래밍 모델이 트랜잭션이나 보안 같은 개발의 기반 구조를 상당 부분 간소화해 준 것은 사실이지만, 배포 디스크립터(deployment descriptor)나 홈 인터페이스, 리모트 인터페이스, 로컬 인터페이스 등의 코드 때문에 개발이 복잡하고 어려워진다. 시간이 지나면서 점점 더 많은 개발자가 EJB에 환멸을 느끼게 됐다. 결과적으로 최근에는 EJB의 인기가 시들해지고 개발자는 좀 더 쉬운 방법을 찾아 나섰다.

이렇게 해서 이제 자바 컴포넌트 개발은 다시 처음으로 돌아왔다. 애스펙트 지향 프로그래밍(AOP: Aspect-Oriented Programming)과 종속객체 주입(DI: Dependency Injection)[역주1] 같은 새로운 프로그래밍 기법들 덕분에 이전에는 EJB로만 가능했던 것들을 자바빈으로도 할 수 있게 됐다. 사용하기 복잡한 EJB를 쓰지 않아도 평범한 자바 객체(POJO: Plain-Old Java Object)로도 EJB 같은 선언적 프로그래밍 모델이 가능해진 것이다. 간단한 자바빈으로도 충분하다면 더 이상 불편한 EJB 컴포넌트를 사용할 필요가 없다.

이렇게 되자, 당연한 일이지만 EJB도 POJO 지향 프로그래밍 모델을 사용할 수 있도록 변화했다. DI와 AOP 개념을 채택하면서 EJB는 예전보다 훨씬 간결해졌다. 그러나 대부분의 개발자들이 느끼기에는 변화의 폭이 너무 작고 느렸다. EJB 3 명세가 발표됐을 때 POJO 기반의 다른 개발 프레임워크들은 이미 자바 커뮤니티에서 사실상의 표준으로 자리 잡고 있었다.

우리가 이 책을 통해 살펴볼 스프링 프레임워크는 바로 이러한 가벼운 POJO 기반 개발 진영을

역주1 종속객체 주입, 즉 DI(Dependency Injection)는 스프링에서 가장 기본이 되면서도 매우 중요한 의미를 갖는다. 기존에 가장 많이 쓰이던 표현은 '의존성 주입'이었다. 하지만 이 표현은 '없는 의존성을 만들어 주입한다'는 오해를 일으키고, 이 때문에 DI의 이해가 어려웠다고 생각한다. DI는 없는 의존성을 주입하는 것이 아니라 의존성은 이미 존재하되, 실제 객체가 필요로 하는 종속객체를 주입하는 것을 의미한다. 따라서 '의존성 주입'보다 뜻이 명확하도록 '종속객체 주입'이라는 표현을 사용하겠다. 또 이 책에서는 원어의 억지스러운 번역보다는 쉽고 자연스럽게 받아들일 수 있는 번역을 택했음을 미리 밝혀 둔다.

진두지휘하고 있다. 이 장에서는 우선 스프링이 어떤 것인지 전체적인 감을 잡고 이 책의 나머지 부분을 이해하는 데 필요한 정보를 얻을 수 있도록 스프링 프레임워크를 개략적으로 소개할 것이다. 이 장을 읽고 나면 스프링이 어떤 종류의 문제를 해결해 주는지 파악할 수 있을 것이다. 중요한 것부터 먼저 하는 게 일의 순서가 아닐까! 자, 이제 스프링의 모든 것을 알아보자.

1.1 자바 개발 간소화

스프링(Spring)은 로드 존슨(Rod Johnson)이 그의 책 『Expert One-on-One: J2EE Design and Development』(『Expert One-on-One J2EE 설계와 개발』, 정보문화사)를 통해 소개한 오픈 소스 프레임워크로서 엔터프라이즈 애플리케이션 개발의 복잡함을 겨냥해 만들어졌다. 스프링은 EJB로만 할 수 있었던 작업을 평범한 자바빈을 사용해서 할 수 있게 해 준다. 스프링이 서버 측 개발에만 유용한 것은 아니다. 간소함, 테스트의 용이함, 낮은 결합도라는 견지에서 보면 모든 자바 애플리케이션이 스프링의 덕을 볼 수 있다.

다른 이름의 빈(bean)

스프링에서는 애플리케이션 컴포넌트를 가리켜 '**빈(bean)**'과 '**자바빈(JavaBean)**'이라는 용어를 자유롭게 사용하고 있지만, 이것이 꼭 자바빈 규약(JavaBeans specification) 전체를 문자 그대로 지켜야 한다는 의미는 아니다. POJO(Plain-Old Java Object, 평범한 자바 객체)라면 어떤 것이나 스프링의 컴포넌트가 될 수 있다. 이 책에서는 느슨한 수준의 자바빈 정의를 사용하며, POJO와 동의어라 생각해도 괜찮다.

이 책을 통해 스프링의 다양한 내용을 살펴보겠다. 하지만 스프링이 제공하는 거의 모든 기본 사상은 몇 가지 기초적인 개념으로 귀결되며, 이는 스프링의 기본 임무인 '**자바 개발 간소화**'에 모든 초점을 맞춘다.

이것이 핵심 키워드다! 특정 기능을 간소화하는 프레임워크는 많이 존재하다. 하지만 스프링의 목적은 자바 개발을 폭넓게 간소화하는 데 있다. 이 내용은 더 많은 설명이 필요해 보인다. 그럼 어떻게 자바 개발을 간소화할 수 있을까?

자바 복잡도에 대한 공격을 지원하기 위해 스프링은 네 가지 주요 전략을 채택했다.

- POJO를 이용한 가볍고(lightweight) 비 침투적인(non-invasive) 개발[역주2]
- DI와 인터페이스 지향(interface orientation)을 통한 느슨한 결합도(loose coupling)
- 애스펙트와 공통 규약을 통한 선언적 프로그래밍
- 애스펙트와 템플릿을 통한 상투적인 코드 축소

스프링이 수행하는 거의 모든 작업은 이 네 가지 전략 중 하나 이상에 속한다. 앞으로 스프링이 자바 개발을 간소화하는 방법에 대한 구체적인 예제를 살펴보면서 이와 같은 사상을 확장해 나가겠다. 먼저 스프링이 POJO 지향 개발을 통해 어떻게 침투적인 개발을 최소화할 수 있는지부터 알아보자.

1.1.1 POJO의 힘

오랜 기간 자바 개발 경험이 있다면 인터페이스의 구현이나 클래스의 확장을 강요하는 프레임워크를 본 적이 있으며, 심지어 이런 프레임워크로 작업한 경험도 있으리라 생각된다. 전형적인 예가 바로 EJB 2 시대의 무상태 세션 빈(stateless session bean)이다. 코드 1.1의 간단한 `HelloWorldBean`에서 보듯이 EJB 2 명세의 요구사항은 다소 무겁다.

코드 1.1 EJB 2.1은 침략적인 API로, 일반적으로 필요하지도 않은 메소드를 강제로 구현하게 한다.

```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean {
  public void ejbActivate() { ← 왜 이런 메소드가 필요할까?
  }

  public void ejbPassivate() {
  public void ejbPassivate() {
  }

  public void ejbRemove() {
  }
```

역주2 침투적이란 EJB와 같이 특정 기술을 적용했을 때 그 기술과 관련된 코드나 규약 등이 등장하는 경우를 말한다. 꼭 필요한 기능이 아니지만 특정 기술을 사용한다는 이유로 특정 클래스, 인터페이스, API 등의 코드가 등장할 경우를 말하며, 이는 개발의 복잡성을 가중시킬 수 있다.

```
  public void setSessionContext(SessionContext ctx) {
  }

  public String sayHello() { ← EJB의 핵심 비즈니스 로직
    return "Hello World";
  }

  public void ejbCreate() {
  }
}
```

SessionBean 인터페이스는 여러 생명주기 콜백 메소드(이러한 메소드는 'ejb'로 시작한다) 구현으로 EJB의 생명주기에 연결시킨다. 하지만 필요하지 않음에도 불구하고 SessionBean 인터페이스가 EJB의 생명주기에 **강제로** 연결시킨다는 말이 더 정확한 표현이다. 실제로 HelloWorldBean의 대부분의 코드는 전적으로 프레임워크를 위한 용도다. 묻고 싶다. 누가 누구를 위해 일하는가?

하지만 EJB 2만 침략적으로 다가온 건 아니다. 다른 인기 있는 프레임워크인 스트러츠(Struts), 웹워크(WebWork), 그리고 태피스트리(Tapestry)의 초기 버전 등도 간단한 자바 클래스가 아니라 자신만의 프레임워크를 강요했다. 이런 무거운(heavyweight) 프레임워크는 개발자에게 불필요한 코드가 흩어져 있는 클래스 작성을 강요하고, 프레임워크에 묶어두고, 테스트 수행도 어려웠다.

반면에 스프링은 API를 이용하여 애플리케이션 코드의 분산을 가능한 막는다. 스프링은 스프링에 특화된 인터페이스 구현이나 스프링에 특화된 클래스 확장을 거의 요구하지 않는다. 스프링 기반 애플리케이션의 클래스에는 스프링이 사용한다는 표시도 거의 없다. 최악의 경우, 클래스에 스프링의 애너테이션(annotation)이 붙지만 그렇지 않은 경우엔 POJO다.

설명을 위해 코드 1.1의 HelloWorldBean 클래스를 스프링이 관리하는 빈으로 함수를 다시 작성하면 코드 1.2와 같다.

코드 1.2 스프링은 HelloWorldBean에 불합리한 요구를 하지 않는다.

```
package com.habuma.spring;

public class HelloWorldBean {
  public String sayHello() { ← 이것이 실제 필요한 전부다.
    return "Hello World";
  }
}
```

더 나아지지 않았는가? 정신없던 생명주기 메소드는 모두 사라졌다. 실제로 `HelloWorldBean`은 구현하지도, 확장하지도 않았고, 심지어 스프링 API에서 어떤 것도 임포트하지 않았다. `HelloWorldBean`은 군살이 없고, 평범하며, 모든 구문은 POJO다.

간단한 형태에도 불구하고 POJO는 매우 강력할 수 있다. 스프링이 POJO에 힘을 불어넣는 방법 중 하나는 DI를 활용한 조립이다. 그럼, DI를 통해 애플리케이션 객체 상호 간의 결합도를 낮추는 방법을 알아보자.

1.1.2 종속객체 주입

'**종속객체 주입(DI: Dependency Injection)**'이라는 문구가 다소 위협적으로 들려서 복잡한 프로그래밍 기술의 개념이나 디자인 패턴이 떠오를 수 있다. 하지만 DI는 생각만큼 복잡하지 않다. 사실 프로젝트에 DI를 적용해 보면 코드가 훨씬 더 간단해지고, 이해하기 쉬우며, 테스트하기도 쉬워진다.

HelloWorld 예제보다 복잡한 실제 애플리케이션에서는 두 개 이상의 클래스가 서로 협력하여 비즈니스 로직을 수행한다. 이때 각 객체는 협력하는 객체에 대한 레퍼런스(즉, 종속객체)를 얻을 책임을 진다. 그 결과, 결합도가 높아지고 테스트하기 힘든 코드가 만들어지기 쉽다.

예를 들어, 코드 1.3에 있는 `Knight` 클래스를 생각해 보자.

코드 1.3 DamselRescuingKnight는 RescueDamselQuest만 떠날 수 있다.

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
  private RescueDamselQuest quest;

  public DamselRescuingKnight() {
    quest = new RescueDamselQuest(); ← RescueDamselQuest에 강하게 결합된다.
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

보다시피 `DamselRescuingKnight`는 생성자 안에 자신의 원정(Quest)인 `RescueDamselQuest`를 생성한다. 이것은 `DamselRescuingKnight`가 `RescueDamselQuest`에 강하게 결합되도록 하며, 기사의 원정 출정 목록을 심각하게 제한한다. 도움이 필요한 여성이 있는 곳에 기사도 있다. 하지만 용을 물리쳐야 하거나 원탁[역주3]이 필요하다면, 또한 순회하려면… 이런… 그냥 앉아만 있어야 한다.

게다가 `DamselRescuingKnight`에 대한 단위 테스트를 작성하기도 몹시 어렵다. 단위 테스트에서는 기사의 `embarkOnQuest()`가 호출될 때 quest의 `embark()` 메소드의 호출을 확인하고 싶어한다. 하지만 여기서는 이를 수행하는 명확한 방법이 없다. 안타깝게도 `DamselRescuingKnight`는 테스트하지 못한 채로 남아 있게 된다.

결합도는 머리가 둘 달린 짐승이다. 결합도가 높은 코드는 한편으로 테스트와 재활용이 어렵고, 이해하기도 어려우며, 오류를 하나 수정하면 다른 오류가 발생하는 경향이 있다. 반면에 전혀 결합이 없는 코드는 아무것도 할 수 없다. 무언가 쓸 만한 일을 하려면 클래스들끼리 어떻게든 서로 알고 있어야 한다. 결합은 필요하지만 주의해서 관리해야 한다.

반면에 DI를 이용하면 객체는 시스템에서 각 객체를 조율하는 제3자에 의해 생성 시점에 종속객체(dependency)가 부여된다. 객체는 종속객체를 생성하거나 얻지 않는다. 즉, 종속객체는 종속객체가 필요한 객체에 주입된다.

역주3 원탁(Round Table)은 영국의 전설적 통치자 아서 왕과 그 기사들이 앉았던 둥근 탁자다.

이 의미를 설명하기 위해 코드 1.4에 있는 BraveKnight를 살펴보자. 이 클래스는 용감할 뿐만 아니라 발생할 어떤 종류의 원정도 떠날 수 있다.

코드 1.4 BraveKnight는 부여된 어떤 Quest도 충분히 감당할 정도로 유연하다.

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
  private Quest quest;

  public BraveKnight(Quest quest) {
    this.quest = quest; ← Quest가 주입된다.
  }

  public void embarkOnQuest() throws QuestException {
    quest.embark();
  }
}
```

보다시피 DamselRescuingKnight와 달리 BraveKnight는 자신의 원정(Quest)을 생성하지 않는다. 대신에 생성 시점에 생성자 인자로 원정이 부여된다. 이와 같은 종류의 종속객체 주입을 **생성자 주입**(constructor injection)이라고 한다.

게다가 부여된 원정은 모든 원정이 구현하는 인터페이스인 Quest 타입으로 제공된다. 따라서 BraveKnight는 RescueDamselQuest, SlayDragonQuest, MakeRoundTableRounderQuest, 또는 부여된 다른 Quest 구현을 떠날 수 있다.

여기서 요점은 BraveKnight가 Quest의 특정 구현체에 결합되지 않는다는 사실이다. Quest 인터페이스를 구현하기만 하면 기사에게 어떤 종류의 원정을 떠나도록 요청하든 문제가 되지 않는다. 이것이 바로 DI의 주요 이점인 '느슨한 결합도(loose coupling)'다. 어떤 객체가 자신이 필요로 하는 종속객체를 인터페이스를 통해서만 알고 있다면(구현 클래스나 인스턴스화되는 방법이 아니라) 사용하는 객체 쪽에는 아무런 변경 없이 종속객체를 다른 구현체로 바꿀 수 있다.

실제 종속객체를 바꾸는 가장 일반적인 방법 중 하나는 테스트 동안의 모조 구현체 이용이다. 강한 결합도(tight coupling)로 인해 DamselRescuingKnight를 적절히 테스트할 수 없었지만, 코드 1.1과 같이 Quest의 모조 구현체를 제공하여 BraveKnight를 쉽게 테스트할 수 있다.

코드 1.5 BraveKnight 테스트를 위하여 모조 Quest를 주입한다.

```
package com.springinaction.knights;

import static org.mockito.Mockito.*;

import org.junit.Test;

public class BraveKnightTest {
  @Test
  public void knightShouldEmbarkOnQuest() throws QuestException {
    Quest mockQuest = mock(Quest.class); ← 모조 Quest 생성

    BraveKnight knight = new BraveKnight(mockQuest); ← mockQuest 주입
    knight.embarkOnQuest();

    verify(mockQuest, times(1)).embark();
  }
}
```

여기서는 `Quest` 인터페이스의 모조 구현체를 만들기 위해 Mockito로 알려진 모조 객체 프레임워크를 사용했다. 모조 객체가 생겼으면 `BraveKnight`의 새로운 인스턴스를 생성하고, 생성자를 통해 모조 `Quest`를 주입한다. `embarkOnQuest()` 메소드를 호출한 후에는 Mockito에게 `Quest`의 `embark()` 메소드가 정확히 한 번 호출됐는지 확인하도록 한다.

기사에게 원정 임무 주입

이렇게 해서 `BraveKnight` 클래스는 모든 원정 임무를 부여받을 수 있게 됐다. 그러면 이제 `BraveKnight` 클래스에 어떤 `Quest`를 부여할지를 어떻게 지정할 수 있을까?

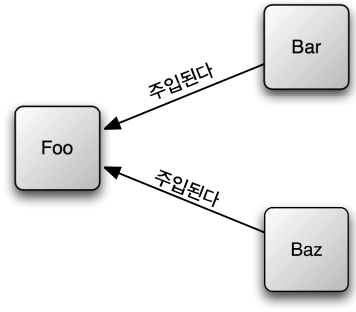

그림 1.1 종속객체 주입은 객체가 스스로 종속객체를 획득하는 것과는 반대로 객체에 종속객체가 부여되는 것을 의미한다.

애플리케이션 컴포넌트 간의 관계를 정하는 것을 **와이어링(wiring)**이라고 한다. 스프링에서 컴포넌트를 와이어링하는 방법은 여러 가지가 있지만 가장 일반적인 방법은 XML을 이용하는 방법이다. 코드 1.6은 `BraveKnight`에게 `Slay DragonQuest`를 부여하는 간단한 스프링 설정 파일인 knights.xml이다.

코드 1.6 스프링의 XML 설정을 이용하여 BraveKnight에게 SlayDragonQuest를 주입하기

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" /> ← 원정 빈 주입
  </bean>

  <bean id="quest"
        class="com.springinaction.knights.SlayDragonQuest" /> ← SlayDragonQuest 생성
</beans>
```

이 예제는 빈과 빈을 와이어링하는 간단한 예를 보여준다. 지금은 자세한 내용에 신경 쓰지 않아도 된다. 자세한 내용은 2장에서 빈을 와이어링하는 다른 방법들과 함께 살펴볼 예정이다.

`BraveKnight`와 `Quest` 사이의 관계를 정의했으니 이제 이 XML 설정 파일을 로드하여 애플리케이션을 구동해 볼 차례다.

실행해 보기

스프링 애플리케이션에서 **애플리케이션 컨텍스트**(application context)는 빈에 관한 정의들을 바탕으로 빈들을 엮어 준다. 스프링 애플리케이션 컨텍스트는 애플리케이션을 구성하는 객체의 생성과 와이어링을 전적으로 책임진다. 스프링에는 애플리케이션의 여러 구현체가 존재하며, 각각의 주요 차이점은 설정을 로드하는 방법에 있을 뿐이다.

knights.xml에서는 빈들이 XML 파일에 선언되어 있으므로 애플리케이션 컨텍스트로 `ClassPathXmlApplicationContext`를 사용하면 좋다. 이 스프링 컨텍스트 구현체는 애플리케이션의 클래스 패스에 있는 하나 이상의 XML 파일에서 스프링 컨텍스트를 로드한다. 코드 1.7에 있는 `main()` 메소드는 `ClassPathXmlApplicationContext`를 사용해 knights.xml을 로드하고 `Knight`에 대한 레퍼런스를 얻는다.

코드 1.7 KnightMain.java는 기사를 포함하는 스프링 컨텍스트를 로드한다.

```
package com.springinaction.knights;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class KnightMain {
  public static void main(String[] args) {
    ApplicationContext context =
        new ClassPathXmlApplicationContext("knights.xml"); ← 스프링 컨텍스트 로드

    Knight knight = (Knight) context.getBean("knight"); ← 기사 빈 얻기
    knight.embarkOnQuest(); ← 기사 사용
  }
}
```

여기서 `main()` 메소드는 knights.xml 파일을 기반으로 스프링 애플리케이션 컨텍스트를 생성한다. 그런 다음 ID가 `knight`인 빈을 조회하기 위해 팩토리로 애플리케이션 컨텍스트를 사용한다. `Knight` 객체에 대한 레퍼런스를 얻은 후에 간단히 `embarkOnQuest()` 메소드를 호출해 기사를 주어진 원정의 길로 떠나보낸다. 이 클래스는 기사가 어떤 유형의 `Quest`를 떠나는지에 대해서는 아무것도 알지 못한다. `BraveKnight`를 다룬다는 사실도 알지 못한다. knights.xml 파일만이 어떤 구현체인지 확실히 알고 있다.

지금까지 DI를 간단히 살펴봤다. 앞으로도 이 책을 통해 DI에 대한 더 많은 내용을 알아보겠다. 하지만 DI를 더 자세히 알아보고 싶다면 DI를 상세히 다루는 단지 R. 프라사나(Dhanji R. Prasanna)의 『Dependency Injection』을 읽어보기 바란다.

하지만 지금은 또 다른 스프링의 자바 간소화 전략인 '애스펙트(Aspect)를 통한 선언적 프로그래밍'을 알아보자.

1.1.3 애스펙트 적용

DI가 소프트웨어 컴포넌트의 결합도를 낮춰 준다면, 애스펙트 지향 프로그래밍은 애플리케이션 전체에 걸쳐 사용되는 기능을 재사용할 수 있는 컴포넌트에 담을 수 있게 해 준다.

애스펙트 지향 프로그래밍은 소프트웨어 시스템 내부의 관심사들을 서로 분리(separation of

concerns)하는 기술이라고 설명할 수 있다. 시스템은 보통 특정한 기능을 책임지는 여러 개의 컴포넌트로 구성된다. 그러나 각 컴포넌트는 대체로 본연의 특정한 기능 외에 로깅이나 트랜잭션 관리, 보안 등의 시스템 서비스도 수행해야 하는 경우가 많다. 이러한 시스템 서비스는 시스템의 여러 컴포넌트에 관련되는 경향이 있기 때문에 **횡단관심사(cross-cutting concerns)**라고 한다.

이러한 관심사가 여러 컴포넌트에 퍼지게 되면 코드는 다음 두 가지 차원에서 복잡해진다.

- 시스템 전반에 걸친 관심사를 구현하는 코드가 여러 컴포넌트에서 중복되어 나타난다. 이때 문제는 이 관심사의 구현을 변경해야 하는 경우 여러 컴포넌트를 모두 변경해야 한다는 것이다. 이 관심사를 별도의 모듈로 추상화해서 각 컴포넌트에서 하나의 메소드만 호출할 수 있도록 만든다고 하더라도, 여전히 이 메소드가 여러 컴포넌트에서 중복되어 나타나는 문제는 동일하다.
- 컴포넌트의 코드가 본연의 기능과 관련 없는 코드로 인해 지저분해진다. 주소록에 주소를 등록하는 메소드는 보안 상태가 유지됐는지 아닌지에는 신경 쓸 필요 없이 주소를 등록하는 방법에만 관여하는 것이 좋다.

그림 1.2는 위에 설명한 복잡성을 그림으로 나타낸 것이다. 왼쪽에 있는 비즈니스 객체들은 시스템 서비스와 매우 긴밀하게 관련을 맺고 있다. 각 객체가 본연의 책임을 수행할 뿐만 아니라 로깅과 보안, 트랜잭션 컨텍스트에 대해서도 알아야 한다.

AOP는 시스템 서비스를 모듈화해서 컴포넌트에 선언적으로 적용할 수 있게 해 준다. AOP를 이용하면 시스템 서비스에 대해서는 전혀 알지 못하면서 응집도가 높고 본연의 관심사에 집중하는 컴포넌트를 만들 수 있다. 다시 말해 애스펙트는 POJO를 말 그대로 평범하게 해 준다.

그림 1.3에서 묘사하고 있는 것처럼 애스펙트를 애플리케이션의 여러 컴포넌트를 덮는 담요처럼 생각하면 개념을 잡는 데 도움이 될 것이다. 이 그림에서 애플리케이션의 핵심은 비즈니스 기능을 구현하는 모듈들로 구성되어 있고, AOP를 이용해서 이 핵심 애플리케이션 위에 추가적인 기능을 여러 겹으로 덮고 있다. 이때 핵심 기능을 구현하는 모듈에는 아무런 변화도 가하지 않고 추가적인 기능을 선언적으로 적용할 수 있다. 이 개념은 매우 강력한 개념으로서, 보안, 트랜잭션, 로깅 등을 처리하느라고 애플리케이션의 핵심 비즈니스 로직이 지저분해지는 것을 막아 준다.

기사 예제로 다시 돌아가, 기본적인 스프링 애스펙트를 추가해 보면서 스프링에서 애스펙트를 적용하는 방법을 구체적으로 살펴보자.

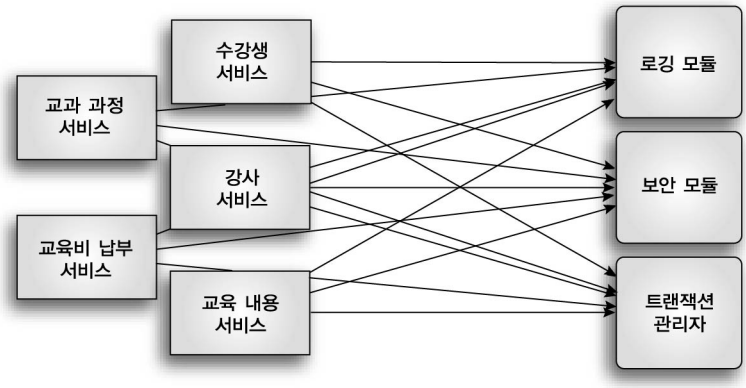

그림 1.2 시스템 관련 관심사가 주 관심사가 아닌 모듈들에 로깅이나 보안 등 시스템 전반에 걸친 관심사에 대한 호출들이 여기저기 흩어져 있다.

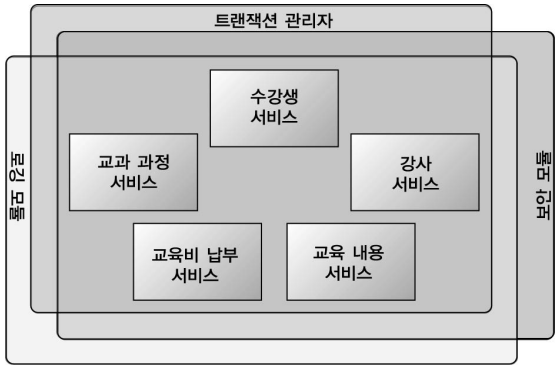

그림 1.3 AOP를 이용하면 시스템 전반에 걸친 관심사가 관련 컴포넌트를 감싼다. 이렇게 함으로써 애플리케이션의 컴포넌트들이 본연의 비즈니스 기능에 집중할 수 있다.

AOP 실습

음유시인으로 알려진 음악을 좋아하는 이야기꾼에 의해 기사의 업적이 노래로 기록되기 때문에 음유시인의 서비스를 이용하여 `BraveKnight`의 출정과 복귀를 기록하고 싶다고 가정해 보자. 사용할 `Minstrel` 클래스는 코드 1.8과 같다.

코드 1.8 Minstrel은 중세 시대의 음악을 좋아하는 로깅 시스템이다.

```
package com.springinaction.knights;

public class Minstrel {
  public void singBeforeQuest() { ← 원정 전에 호출됨
    System.out.println("Fa la la; The knight is so brave!");
  }

  public void singAfterQuest() { ← 원정 후에 호출됨
    System.out.println(
            "Tee hee he; The brave knight did embark on a quest!");
  }
}
```

보다시피 `Minstrel`은 두 개의 메소드가 있는 간단한 클래스다. `singBeforeQuest()` 메소드는 기사가 원정을 떠나기 전에 호출되며, `singAfterQuest()` 메소드는 기사가 원정을 마친 후에 호출된다. 코드에서 작업하기 위해서는 간단해야 한다. 따라서 `Minstrel`을 사용하기 위해서 `BraveKnight`를 적절히 수정하자. 첫 번째 시도는 다음과 같다.

코드 1.9 Minstrel 메소드를 호출해야 하는 BraveKnight

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
  private Quest quest;
  private Minstrel minstrel;

  public BraveKnight(Quest quest, Minstrel minstrel) {
    this.quest = quest;
    this.minstrel = minstrel;
  }

  public void embarkOnQuest() throws QuestException {
    minstrel.singBeforeQuest(); ← 기사가 자체적인 음유 시인을 관리해야 할까?
    quest.embark();
    quest.embark();
    minstrel.singAfterQuest(
  }
```

기교를 부려봤다. 하지만 일부는 여기서 적절해 보이지 않는다. 정말로 기사의 관심 범위 내에서 음유 시인을 관리해야 할까? 음유 시인은 기사의 요청 없이도 그의 일을 수행한다. 궁극적으로 음유 시인의 일은 기사의 업적에 대해 노래하는 것이다. 왜 기사가 음유 시인에게 그의 작업을 수행하도록 상기시켜 줘야 하는가?

게다가 기사는 음유 시인에 대해 알아야 하므로, 강제로 `Minstrel`을 `BraveKnight`에 주입(inject)한다. 이는 `BraveKnight`의 코드를 복잡하게 만들 뿐만 아니라 음유 시인이 없는 기사를 원하는 경우 당황스러워진다. 만일 `Minstrel`이 `null`이라면? 이와 같은 상황을 다루기 위해 몇 가지 `null` 체크 로직을 도입해야 할까?

간단한 `BraveKnight` 클래스가 점차 복잡해지기 시작하고 `nullMinstrel` 시나리오까지 처리해야 한다면 그 복잡함은 더해진다. 하지만 AOP를 사용하면 기사의 원정에 대해 노래할 음유 시인을 선언할 수 있으며, 기사는 `Minstrel` 메소드를 직접 처리하는 일에서 해방된다.

`Minstrel`을 애스펙트로 바꾸려면 스프링 설정 파일의 하나로 선언하기만 하면 된다. 여기서는 knights.xml 파일을 수정하여 `Minstrel`을 애스펙트로 선언했다.

코드 1.10 Minstrel을 애스펙트로 선언하기

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <bean id="knight" class="com.springinaction.knights.BraveKnight">
    <constructor-arg ref="quest" />
</bean>
  <bean id="quest"
        class="com.springinaction.knights.SlayDragonQuest" />

  <bean id="minstrel"
     class="com.springinaction.knights.Minstrel" /> ← Minstrel 빈 선언

  <aop:config>
    <aop:aspect ref="minstrel">
```

```
      <aop:pointcut id="embark"
          expression="execution(* *.embarkOnQuest(..))" /> ← 포인트컷 정의

      <aop:before pointcut-ref="embark"
                  method="singBeforeQuest"/> ← 비포 어드바이스 선언

      <aop:after pointcut-ref="embark"
                 method="singAfterQuest"/> ← 애프터 어드바이스 선언

    </aop:aspect>
  </aop:config>
</beans>
```

여기서 스프링의 aop 설정 네임스페이스를 사용하여 Minstrel 빈이 애스펙트라고 선언한다. 먼저 Minstrel을 빈으로 선언해야 한다. 그런 다음 <aop:aspect> 엘리먼트에서 빈을 참조한다. 애스펙트를 더 정의해 보자면, embarkOnQuest() 메소드가 실행되기 전에 Minstrel의 singBeforeQuest() 메소드가 호출되어야 한다고 선언한다(<aop:before>를 이용하여). 이것을 **비포 어드바이스**(before advice)라 부른다. 그리고 embarkOnQuest() 메소드가 실행된 후에 singAfterQuest() 메소드가 호출되어야 한다고 선언한다(<aop:after>를 이용하여). 이것을 **애프터 어드바이스**(after advice)라 부른다.

양쪽 경우 모두 pointcut-ref 애트리뷰트는 embark라는 이름의 포인트컷(pointcut)을 참조한다. 이 포인트컷은 앞에 있는 <pointcut> 엘리먼트에 어드바이스가 적용될 위치를 선택하는 expression 애트리뷰트와 함께 정의돼 있다. expression 구문은 AspectJ의 포인트컷 표현식 언어다.

AspectJ나 AspectJ 포인트컷 표현식의 작성 방법을 자세히 모른다고 해도 걱정할 필요는 없다. 4장에서 스프링의 AOP에 대해 더 자세히 논의할 것이다. 지금은 스프링이 BraveKnight가 원정을 떠나기 전과 후에 Minstrel의 singBeforeQuest()와 singAfterQuest() 메소드를 호출한다는 정도만 알아도 충분하다.

이렇게 해서 몇 줄 안 되는 XML 설정으로 Minstrel을 스프링 애스펙트로 바꿨다. 지금 완전히 이해가 되지 않더라도 걱정할 필요는 없다. 4장에서 다양한 스프링 AOP 예제를 통해 충분히 이해할 수 있는 시간을 갖게 된다. 지금은 이 예제에서 다음에 설명하는 두 가지 점만 분명하게 알고 넘어가면 된다.

첫 번째는 Minstrel이 여전히 POJO라는 점이다. Minstrel이 애스펙트로 사용될 것임을 나타내는 내용은 Minstrel에 전혀 포함되어 있지 않다. 스프링 컨텍스트에서 선언적으로 애스펙트가 된다.

두 번째이면서 가장 중요한 점은 BraveKnight가 Minstrel을 명시적으로 호출하지 않아도 Minstrel이 BraveKnight에 적용될 수 있다는 사실이다. 실제로 BraveKnight는 Minstrel의 존재를 전혀 인식하지 못한다.

또한 중요한 것은 Minstrel을 애스펙트로 바꾸기 위해 몇 가지 스프링 마법을 사용했지만, 먼저 <bean>으로 선언돼야 한다는 사실이다. 여기서의 핵심은 종속객체 주입 같은 다른 스프링 빈과 함께 수행할 수 있는 스프링 애스펙트를 이용하면 무엇이든 가능하다는 점이다.

기사에 대해 노래하기 위해 애스펙트를 사용하는 일도 즐거울 수 있지만, 스프링의 AOP는 더 실용적인 목적으로 사용된다. 나중에 보겠지만 스프링 AOP는 선언적 트랜잭션(6장)과 보안(9장) 등의 서비스를 제공하기 위해 도입한다.

하지만 지금은 스프링이 자바 개발을 간소화시켜 주는 또 하나의 방법을 알아보자.

1.1.4 템플릿을 이용한 상투적인 코드 제거

코드를 작성하다가 이전에 이미 작성했던 코드 같다고 느껴본 적이 있는가? 이것은 데자뷰(déjà vu)가 아니다. 이것이 바로 상투적인 코드다. 즉, 공통 작업이나 간단한 작업을 위해 반복적으로 작성해야 하는 코드다.

안타깝게도 자바 API에는 상투적인 코드가 많이 포함돼 있다. 상투적 코드의 대표적인 예는 데이터베이스에서 데이터를 조회하는 JDBC 작업이다. 예를 들어, 이전에 JDBC로 작업한 적이 있어도 다음에도 유사한 코드를 작성해야 하는 경우가 있다.

코드 1.11 JDBC 등 상당수의 자바 API에는 상투적인 코드 작성이 많이 포함돼 있다.

```
public Employee getEmployeeById(long id) {
  Connection conn = null;
  PreparedStatement stmt = null;
  ResultSet rs = null;
  try {
    conn = dataSource.getConnection();
```

```
    stmt = conn.prepareStatement(
          "select id, firstname, lastname, salary from " +
          "employee where id=?"); ← 직원 조회
    stmt.setLong(1, id);
    rs = stmt.executeQuery();
    Employee employee = null;
    if (rs.next()) {
      employee = new Employee();
      employee.setId(rs.getLong("id")); ← 데이터로부터 객체 생성
      employee.setFirstName(rs.getString("firstname"));
      employee.setLastName(rs.getString("lastname"));
      employee.setSalary(rs.getBigDecimal("salary"));
    }
    return employee;
  } catch (SQLException e) { ← 여기서 무엇을 수행해야 하지?
  } finally {
      if(rs != null) { ← 정리 작업
        try {
          rs.close();
        } catch(SQLException e) {}
      }
      if(stmt != null) {
        try {
        stmt.close();
        } catch(SQLException e) {}
      }

      if(conn != null) {
        try {
          conn.close();
        } catch(SQLException e) {}
      }
  }

  return null;
}
```

보다시피 JDBC 코드는 직원의 이름과 급여를 데이터베이스에서 조회한다. 하지만 조회하는 부분을 찾으려면 열심히 살펴봐야 한다. 그 이유는 몇 줄 안 되는 직원 조회 코드가 JDBC 형식의 더미에 묻혀버렸기 때문이다. 먼저 커넥션(connection)을 생성하고, 그런 다음에 질의객체(statement)를 생성하며, 마지막으로 결과를 조회한다. 그리고 예외를 던지더라도(throw) 수행할 수 있는 작업이 많지 않음에도 불구하고, JDBC의 분노를 달래기 위해 검사형 예외(checked exception)인 `SQLException`을 잡아야(catch) 한다.

마지막으로 모든 작업을 완료한 후에는 커넥션과 질의객체, 그리고 결과 집합을 닫는 정리 작업을 수행해야 한다. 이 또한 분노를 일으킬 수 있다. 따라서 여기서도 별도로 `SQLException`을 잡아야 한다.

코드 1.11에서 가장 주목할 부분은 거의 모든 JDBC 작업을 위해 작성했던 코드와 완벽히 동일하다는 사실이다. 직원 조회를 위해 수행하는 작업은 극히 일부분이다. 대부분은 JDBC의 상투적인 코드다.

상투적인 코드 작업에 JDBC만 있는 것은 아니다. 많은 기능은 종종 유사한 상투적인 코드를 요구한다. JMS, JNDI, 그리고 REST 서비스의 소비에는 수많은 공통적인 반복 코드가 포함돼 있다.

스프링은 템플릿에 상투적인 코드를 캡슐화하여 반복적인 코드를 제거하는 방법을 찾는다. 스프링의 `JdbcTemplate`은 전통적인 JDBC에서 필요한 모든 형식 없이도 데이터베이스 작업을 수행할 수 있게 한다.

예를 들어, 스프링의 `SimpleJdbcTemplate`(`JdbcTemplate`의 서브클래스로 자바 5 기능을 활용한다)을 사용하면 JDBC API가 요구하는 작업이 아니라 직원 데이터를 조회하는 작업에 초점을 맞추도록 `getEmployeeById()` 메소드를 재작성할 수 있다. 업데이트된 `getEmployeeById()` 메소드는 코드 1.12와 같다.

코드 1.12 템플릿은 코드가 작업에 바로 집중할 수 있게 한다.

```
public Employee getEmployeeById(long id) {
  return jdbcTemplate.queryForObject(
          "select id, firstname, lastname, salary " + ← SQL쿼리
          "from employee where id=?",
          new RowMapper<Employee>() {
            public Employee mapRow(ResultSet rs,
                    int rowNum) throws SQLException { ← 결과를 객체에 매핑
              Employee employee = new Employee();
              employee.setId(rs.getLong("id"));
              employee.setFirstName(rs.getString("firstname"));
              employee.setLastName(rs.getString("lastname"));
              employee.setSalary(rs.getBigDecimal("salary"));
              return employee;
            }
          },
          id); ← 쿼리 파라미터 지정
}
```

보다시피 새롭게 수정된 `getEmployeeById()`는 매우 간단하면서도 실제로 데이터베이스에서 직원을 조회하는 작업에 초점을 맞춘다. 템플릿의 `queryForObject()` 메소드에는 SQL 쿼리, `RowMapper`(결과 집합 데이터를 도메인 객체에 매핑), 그리고 쿼리 파라미터가 부여될 수 있다. `getEmployeeById()`에서는 이전에 보였던 JDBC의 상투적인 코드가 보이지 않는다. 이것은 모두 템플릿 내부에서 처리된다.

지금까지 POJO 지향 개발, DI, AOP, 그리고 템플릿을 이용하여 자바 개발의 복잡성을 공격하는 방법을 살펴봤다. 또한 XML 기반 설정 파일에서 빈과 애스펙트를 설정하는 방법도 살펴봤다. 그런데 이러한 파일을 어떻게 로드할 수 있을까? 그리고 어디에 로드될까? 이제부터 애플리케이션의 빈이 위치하는 스프링 컨테이너를 살펴보자.

1.2 빈을 담는 그릇, 컨테이너

스프링 애플리케이션에서는 스프링 컨테이너(container)에서 객체가 태어나고, 자라고, 소멸한다. 그림 1.4와 같이 스프링 컨테이너는 객체를 생성하고, 서로 엮어 주고, 이들의 전체 생명주기(생성에서 소멸까지)를 관리한다.

스프링 컨테이너가 어떤 객체를 생성하고 구성해서 서로 엮어 줘야 하는지 알 수 있게 설정하는 방법은 다음 장에서 설명하겠다. 우선은 객체들의 삶의 터전인 스프링 컨테이너 자체부터 이해하는 것이 더 중요하다. 스프링 컨테이너에 대한 이해를 토대로 객체가 어떻게 관리되는지를 이해해야 한다.

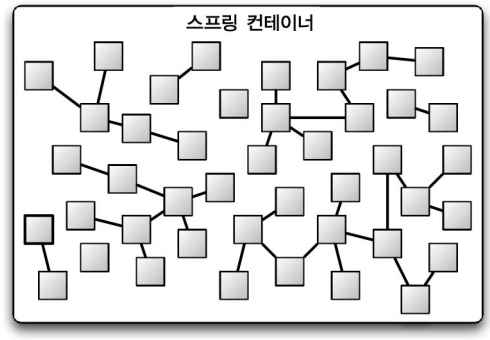

그림 1.4 스프링 애플리케이션에서 객체는 스프링 컨테이너 내에서 태어나고, 더불어 살아간다.

스프링 컨테이너는 스프링 프레임워크의 핵심부에 위치한다. 스프링 컨테이너는 종속객체 주입을 이용해서 애플리케이션을 구성하는 컴포넌트를 관리하며, 협력 컴포넌트 간 연관관계의 형성도 여기에서 이뤄진다. 이러한 짐을 컨테이너에 덜어버린 객체들은 더 명확하고 이해하기 쉬우며, 재사용을 촉진하고, 단위 테스트가 용이해진다.

스프링 컨테이너는 여러 가지가 있다. 스프링에는 여러 컨테이너 구현체가 존재하며, 이들은 크게 두 가지로 분류된다. 첫 번째 부류는 **빈팩토리**(`org.springframework.beans.factory.BeanFactory` 인터페이스에 의해 정의된다)로, 이는 DI에 대한 기본적인 지원을 제공하는 가장 단순한 컨테이너다. 두 번째 부류는 **애플리케이션 컨텍스트**(`org.springframework.context.ApplicationContext` 인터페이스에 의해 정의된다)로, 빈팩토리를 확장해 프로퍼티 파일에 텍스트 메시지를 읽고 해당 이벤트 리스너에 대한 애플리케이션 이벤트 발행 같은 애플리케이션 프레임워크 서비스를 제공하는 컨테이너다.

스프링으로 작업할 때 빈팩토리나 애플리케이션 컨텍스트 중에 아무거나 사용해도 상관없지만, 빈팩토리는 대부분의 애플리케이션에 대하여 지나치게 저수준의 기능을 제공한다. 따라서 빈팩토리보다는 애플리케이션 컨텍스트를 더 선호한다. 따라서 앞으로는 애플리케이션 컨텍스트에 초점을 맞춰 작업할 예정이며, 빈팩토리에 대해서는 많은 시간을 할애하지 않겠다.

1.2.1 또 하나의 컨테이너, 애플리케이션 컨텍스트

스프링에는 다양한 종류의 애플리케이션 텍스트가 있다. 그 중에서 가장 많이 접하게 될 세 가지는 다음과 같다.

- `ClassPathXmlApplicationContext` – 클래스패스에 위치한 XML 파일에서 컨텍스트 정의 내용을 로드한다.
- `FileSystemXmlApplicationContext` – 파일 시스템에서, 즉 파일 경로로 지정된 XML 파일에서 컨텍스트 정의 내용을 로드한다.
- `XmlWebApplicationContext` – 웹 애플리케이션에 포함된 XML 파일에서 컨텍스트 정의 내용을 로드한다.

`XmlWebApplicationContext`는 7장에서 웹 기반 스프링 애플리케이션을 설명할 때 살펴보기로 한다. 지금은 간단히 `FileSystemXmlApplicationContext`를 이용해서 파일 시스템에서 애플리케이션 컨텍스트를 로드하거나, `ClassPathXmlApplicationContext`를 이용해서 클래스패스에서 애플리케이션 컨텍스트를 로드하자.

파일 시스템이나 클래스패스에서 애플리케이션 컨텍스트를 로드하는 방법은 빈팩토리 때와 유사하다. 다음은 `FileSystemXmlApplicationContext`를 로드하는 코드다.

```
ApplicationContext context = new
        FileSystemXmlApplicationContext("c:/foo.xml");
```

클래스패스에서 로드하는 `ClassPathXmlApplicationContext`도 다를 바 없다.

```
ApplicationContext context = new
        ClassPathXmlApplicationContext("foo.xml");
```

둘 사이에 차이가 있다면 `FileSystemXmlApplicationContext`의 경우 foo.xml을 파일 시스템에서 지정된 경로(c:/fool.xml)로 찾고, `ClassPathXmlApplicationContext`는 클래스패스에 포함된 모든 경로(JAR 파일 포함)에서 찾으려 한다는 점이다.

애플리케이션 컨텍스트를 얻은 다음 켄텍스트의 `getBean()` 메소드를 호출하여 스프링 컨테이너에서 빈을 조회할 수 있다.

이제 스프링 컨테이너를 생성하는 기초 방법을 익혔으니 컨테이너 내부 빈의 생명주기를 좀 더 살펴보자.

1.2.2 빈의 일생

보통의 자바 애플리케이션에서 빈의 생명주기는 매우 단순하다. 자바의 `new` 키워드를 이용하거나 역직렬화(deserialization)를 통해 빈을 인스턴스화하고 이를 바로 사용한다. 빈이 더 이상

사용되지 않으면 가비지 컬렉션(garbage collection) 후보가 되어 언젠가는 메모리 덩어리가 됐다가 허공 속으로 사라질 것이다.

반면에 스프링 컨테이너 내에서 빈의 생명주기는 좀 더 정교하다. 빈이 생성될 때 스프링이 제공하는 커스터마이징 기회를 이용하려면 스프링의 빈 생명주기를 이해해 두는 것이 좋다. 그림 1.5는 BeanFactory 컨테이너 내에서 빈이 갖는 구동(startup) 생명주기를 보여준다.

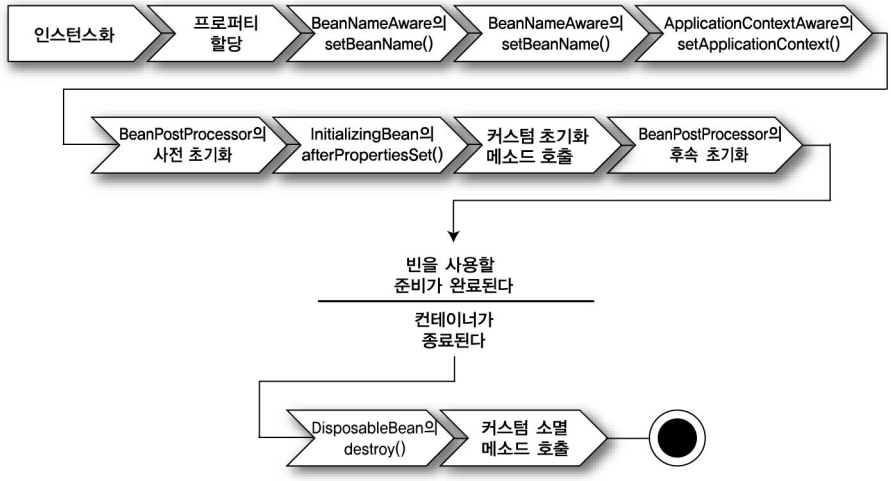

그림 1.5 빈은 스프링 컨테이너에서 탄생과 소멸 사이에 몇 단계의 과정을 거치게 된다. 각 단계는 스프링이 빈을 관리하는 방법을 제어할 수 있는 기회이기도 하다.

보다시피 빈팩토리는 빈이 사용 가능한 상태가 되기 전에 몇 가지 준비(setup) 과정을 수행한다. 그림 1.5의 각 과정을 상세히 설명하면 다음과 같다.

1. 스프링이 빈을 인스턴스화한다.
2. 스프링이 값과 빈의 레퍼런스를 빈의 프로퍼티에 주입한다.
3. 빈이 `BeanNameAware`를 구현하면, 스프링이 빈의 ID를 `setBeanName()` 메소드에 넘긴다.
4. 빈이 `BeanFactoryAware`를 구현하면, `setBeanFactory()` 메소드를 호출하여 빈팩토리 자체를 넘긴다.
5. 빈이 `ApplicationContextAware`를 구현하면, 스프링이 `setApplicationContext()` 메소드를 호출하고 본 애플리케이션 컨텍스트(enclosing application context)에 대한 참조를 넘긴다.

6. 빈이 `BeanPostProcessor` 인터페이스를 구현하면, 스프링은 `postProcessBeforeInitialization()` 메소드를 호출한다.
7. 빈이 `InitializingBean` 인터페이스를 구현하면, 스프링은 `afterPropertiesSet()` 메소드를 호출한다. 마찬가지로 빈이 `init-method`와 함께 선언됐으면, 지정한 초기화 메소드가 호출된다.
8. 빈이 `BeanPostProcessor`를 구현하면, 스프링은 `postProcessAfterInitialization()` 메소드를 호출한다.
9. 이 상태가 되면 빈은 애플리케이션에서 사용할 준비가 된 것이며, 애플리케이션 컨텍스트가 소멸될 때까지 애플리케이션 컨텍스트에 남아 있다.
10. 빈이 `DisposableBean` 인터페이스를 구현하면, 스프링은 `destroy()` 메소드를 호출한다. 마찬가지로 빈이 `destroy-method`와 함께 선언됐으면, 지정된 메소드가 호출된다.

이제 스프링 컨테이너가 어떻게 생성되고 로드되는지 알게 됐지만, 우리는 아직 컨테이너에 아무것도 넣지 않았기 때문에 별 쓸모가 없다. 스프링 DI를 활용하려면 컨테이너에 애플리케이션 객체를 넣고 서로 연결해 줘야 한다. 빈 와이어링(wiring)은 2장에서 좀 더 상세히 살펴보겠다.

그에 앞서, 스프링 프레임워크가 무엇으로 구성되어 있고 최신 버전의 스프링이 제공하는 기능이 무엇인지 현대 스프링 현황을 조사해 보자.

1.3 스프링 현황

앞서 살펴봤듯이, 스프링 프레임워크는 DI, AOP, 그리고 상투적인 코드 축소를 통해 자바 개발을 간소화하는 데 초점을 맞춘다. 이 모두 스프링이 수행하는 작업이며, 유용하다. 하지만 스프링에는 눈에 보이는 것보다 더 많은 기능이 존재한다.

스프링 프레임워크 안에서 스프링이 자바 개발을 쉽게 할 수 있는 여러 가지 방법을 찾을 수 있다. 하지만 스프링 프레임워크 자체를 넘어서 웹 서비스, OSGi, 플래시(Flash), 그리고 심지어 .NET 등의 영역으로 스프링을 확장하여 코어 프레임워크를 구축하려는 프로젝트의 더 큰 생태계가 있다.

먼저 코어 스프링 프레임워크가 어떤 도움을 주는지 자세히 분석해 보자. 그런 다음, 안목을 넓혀서 더 큰 스프링 포트폴리오의 다른 구성요소를 검토하도록 하자.

1.3.1 스프링 모듈

스프링 프레임워크는 여러 개의 개별 모듈로 구성되어 있다. 스프링 프레임워크를 다운로드하여 압축을 풀면 그림 1.6과 같이 dist 디렉터리에서 20개의 다른 JAR 파일을 볼 수 있다.

스프링을 구성하는 20개의 JAR 파일은 그림 1.7에 보이는 여섯 개의 기능 카테고리 중 하나에 속한다.

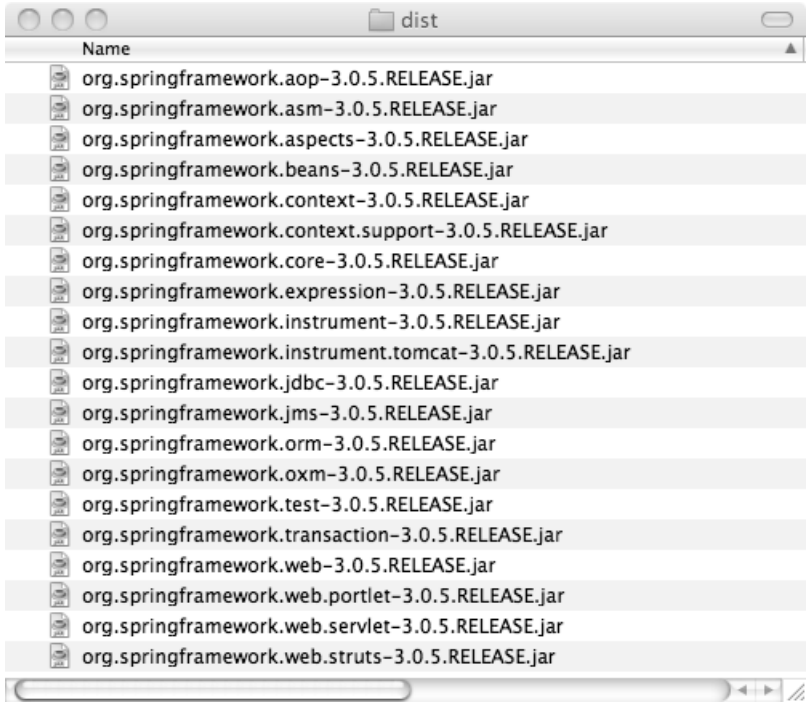

그림 1.6 JAR 파일에는 스프링 프레임워크 배포파일이 존재한다.

이 모듈들은 전체적으로 보면 엔터프라이즈 애플리케이션 개발에 필요한 모든 것을 제공한다. 그러나 애플리케이션을 만들 때 항상 이 스프링 프레임워크 모듈 전체를 이용해야 하는 것은 아니다. 스프링 프레임워크는 다른 프레임워크나 라이브러리와 쉽게 통합될 수 있으므로 원하는 기능이 스프링에 없어도 직접 개발할 필요 없이 다른 프레임워크나 라이브러리를 사용할 수 있다.

스프링 모듈들이 전체적인 구조 하에서 각각 어떤 기능을 담당하는지 하나씩 살펴보기로 하자.

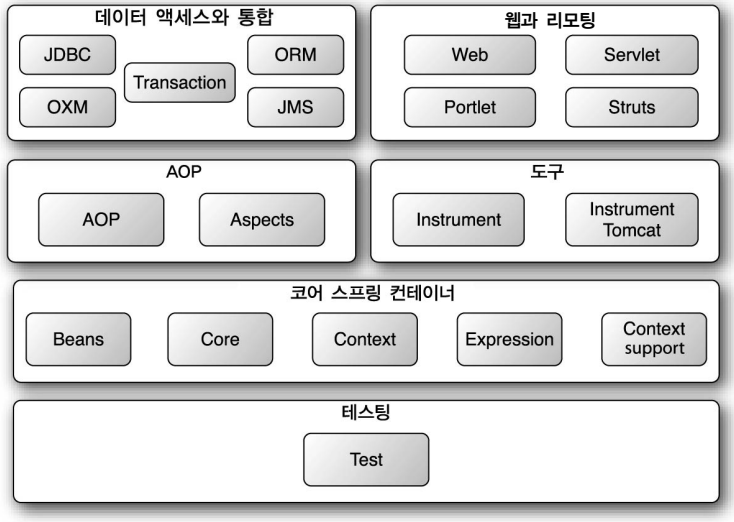

그림 1.7 스프링 프레임워크는 여섯 개의 잘 정의된 모듈로 구성돼 있다.

코어 스프링 컨테이너

스프링 프레임워크의 핵심은 스프링 애플리케이션에서 빈의 생성, 설정, 그리고 처리 방법을 관리하는 컨테이너다. 이 모듈 내에서 DI를 제공하는 스프링 빈팩토리를 확인할 수 있다. 빈팩토리를 구성하다 보면 여러 가지 스프링의 애플리케이션 컨텍스트 구현체가 보이는데, 각각은 스프링을 구성하는 다른 방법을 제공한다.

빈팩토리와 애플리케이션 컨텍스트 외에도 이 모듈은 이메일, JNDI 액세스, EJB 통합, 그리고 스케줄링 등의 다양한 엔터프라이즈 서비스도 제공한다.

보다시피 모든 스프링의 모듈은 코어 컨테이너 위에 구축된다. 애플리케이션 구성 시 암묵적으로 코어 컨테이너 클래스를 사용한다. 코어 모듈은 스프링의 DI를 살펴보는 2장을 시작으로 이 책의 전반에 걸쳐 다룬다.

스프링의 AOP 모듈

스프링은 AOP 모듈을 통해 애스펙트 지향 프로그래밍을 풍부하게 지원한다. AOP 모듈은 스프링 애플리케이션에서 애스펙트를 개발할 수 있는 기반이 되는 것으로서, DI처럼 애플리케이션 객체 간의 결합도를 낮추는 데 기여한다. AOP는 주로 애플리케이션 전체에 걸친 관심사(트랜잭션이나 보안 등)와 각 객체 간의 결합도를 낮추는 데 이용된다.

스프링의 AOP 지원에 대해서는 4장에서 자세히 살펴본다.

데이터 액세스와 통합

JDBC를 이용해 작업하다 보면 커넥션을 얻어오고, 질의객체를 생성하고, 결과 집합을 처리하고, 커넥션을 닫는 코드를 수없이 반복하게 된다. 스프링의 JDBC와 **DAO(Data Access Objects, 데이터 액세스 객체)** 모듈은 이렇게 반복되는 코드를 추상화하므로 이 모듈을 이용하면 데이터베이스 관련 코드를 깔끔하고 간단하게 만들 수 있고, 데이터베이스 리소스를 닫지 않아서 발생할 수 있는 문제를 예방할 수 있다. 또한 이 모듈에는 여러 종류의 데이터베이스 서버가 제공하는 오류 메시지 위에 의미 있는 예외 계층이 추가되어 있다. 이 모듈을 이용하면 더 이상 암호문 같은 SQL 오류 메시지를 이해하려고 머리를 싸매지 않아도 된다!

스프링의 ORM 모듈은 JDBC보다 **객체 관계 매핑(ORM: Object-Relational Mapping)** 도구를 선호하는 사람들을 위한 것이다. 스프링의 ORM 모듈은 DAO 모듈 위에 올라가서 ORM 솔루션용 DAO를 만드는 편리한 방법을 제공한다. 스프링은 고유한 ORM 솔루션을 구현하지 않고, 하이버네이트, 자바 퍼시스턴스 API, 자바 데이터 객체(JDO), iBATIS SQL Maps 등 널리 사용되는 ORM 프레임워크와의 연결고리를 제공한다. 스프링의 트랜잭션 관리 기능은 JDBC뿐만 아니라 이 ORM 프레임워크들도 지원한다.

5장에서 스프링 데이터 액세스를 살펴볼 때 스프링의 템플릿 기반 JDBC 추상화가 어떻게 JDBC 코드를 크게 간소화하는지 알아본다.

이 모듈은 또한 메시징을 통해 다른 애플리케이션과의 비동기식 통합을 위하여 JMS(Java Message Service)를 이용한 스프링 추상화를 포함한다. 그리고 스프링 3.0 현재, 이 모듈은 원래 스프링 웹 서비스(Spring Web Services) 프로젝트의 일부분이었던 객체-XML 매핑(OXM: Object-XML Mapping)을 포함한다.

또한 이 모듈은 스프링의 AOP 모듈을 이용하여 스프링 애플리케이션에서 객체들의 트랜잭션 관리 서비스를 제공한다. 스프링의 트랜잭션 지원 기능은 6장에서 자세히 살펴본다.

웹과 리모팅

MVC(Model-View-Controller) 패러다임은 사용자 인터페이스가 애플리케이션 로직과 분리되는 웹 애플리케이션을 만드는 경우에 일반적으로 사용되는 접근 방식이다. 자바의 유명한 MVC 프레임워크 가운데에는 아파치 스트럿츠(Apache Struts), JSF, 웹워크(WebWork), 태피

스트리(Tapestry) 등이 있다.

스프링이 다양한 유명 MVC 프레임워크와 잘 통합되기는 하지만, 웹과 리모팅 모듈에는 애플리케이션의 웹 계층에서 결합도를 낮추는 스프링의 기술을 잘 반영하는 MVC 프레임워크가 별도로 만들어져 있다. 이 프레임워크는 두 가지 형태로 구성되어 있다. 하나는 기존 웹 애플리케이션을 위한 서블릿 기반 프레임워크이고, 다른 하나는 자바 포틀릿(portlet) API에 대한 개발을 위한 포틀릿 기반 애플리케이션이다.

사용자 접촉(user-facing) 웹 애플리케이션뿐만 아니라 이 모듈은 다른 애플리케이션과 상호작용하는 애플리케이션을 개발하기 위한 다양한 리모팅 옵션도 제공한다. 스프링의 리모팅 기능에는 RMI(Remote Method Invocation), Hessian, Burlap, JAX-WS, 그리고 스프링의 HTTP 호출자 등이 있다.

7장에서 스프링의 MVC 프레임워크를 살펴본 다음, 10장에서 스프링의 리모팅 기능을 확인해 보겠다.

테스팅

개발자가 작성하는 테스트의 중요성을 인식하여 스프링은 스프링 애플리케이션 테스트에 전념하는 모듈을 제공한다.

이 모듈 안에서 JNDI, 서블릿, 그리고 포틀릿과 동작하는 코드의 단위 테스트 작성을 위한 모조 객체 구현을 확인할 수 있다. 통합 테스트의 경우, 이 모듈은 스프링 애플리케이션 컨텍스트에서 빈을 로드하고 이 컨텍스트에 있는 빈과의 작업을 지원한다.

스프링의 테스팅 모듈은 4장에서 처음 다룬다. 그런 다음 5장과 6장에서 스프링 데이터 액세스와 트랜잭션을 테스트하는 방법을 살펴보면서 배운 내용을 넓혀 가겠다.

1.3.2 스프링 포트폴리오

스프링은 눈에 보이는 게 다가 아니다. 실제로 스프링 프레임워크 다운로드에 있는 것보다 더 많은 기능이 존재한다. 만일 코어 스프링 프레임워크에서 멈춘다면, 다양한 스프링 포트폴리오가 제공하는 풍부한 잠재력을 놓치게 된다. 전체 스프링 포트폴리오에는 코어 스프링 프레임워크와 서로 연관되어 구축된 다양한 프레임워크와 라이브러리가 있다. 전체 스프링 포트폴리오는 자바 개발의 모든 측면에서 스프링 프로그래밍 모델을 제공해 준다.

스프링 포트폴리오가 제공하는 모든 기능을 설명하려면 책 몇 권의 분량이 필요하다. 따라서 여기서는 코어 스프링 프레임워크의 범위를 넘어선 일부 내용을 포함하여 스프링 포트폴리오의 몇 가지 주요 부분 위주로 살펴보겠다.

스프링 웹 플로

스프링 웹 플로(Spring Web Flow)는 스프링의 코어 MVC 프레임워크를 기반으로 목표를 향해 사용자를 안내하는(마법사나 장바구니를 떠올리면 된다) 대화형, 흐름기반 웹 애플리케이션 구축을 지원한다. 스프링 웹 플로는 8장에서 자세히 논의하겠으며, 스프링 웹 플로에 대한 더 많은 정보는 홈페이지 http://www.springsource.org/webflow에서 확인할 수 있다.

스프링 웹 서비스

코어 스프링 프레임워크는 웹 서비스로 스프링 빈을 선언적으로 배포할 수 있지만, 이러한 서비스는 틀림없이 구조적으로 열악한 구현우선(contract-last) 모델[역주4]을 기반으로 한다. 서비스에 대한 규약은 빈의 인터페이스에서 결정된다. 스프링 웹 서비스(Spring Web Services)는 규약우선(contract-first) 웹 서비스 모델을 제공하는데, 여기서 서비스 규약을 충족하기 위해서 서비스 구현체가 작성된다.

이 책에서 스프링 웹 서비스에 대해 논의하지는 않겠다. 하지만 더 많은 정보는 스프링 웹 서비스의 홈페이지인 http://static.springsource.org/spring-ws/sites/2.0에서 확인할 수 있다.

스프링 시큐리티

보안은 많은 애플리케이션의 핵심 요소다. 스프링 AOP를 이용하여 구현된 스프링 시큐리티(Spring Security)는 스프링 기반 애플리케이션에 선언적 보안 메커니즘을 제공한다. 애플리케이션에 스프링 시큐리티를 추가하는 방법은 9장에서 살펴보겠다. 더 많은 정보는 스프링 시큐리티의 홈페이지 http://static.springsource.org/spring-security/site에서 확인할 수 있다.

스프링 인티그레이션

많은 엔터프라이즈 애플리케이션은 다른 엔터프라이즈 애플리케이션과 상호작용해야 한다. 스

역주4 구현우선(contract-last) 모델은 자바 소스 코드를 먼저 작성한 후에 WSDL을 자동 생성하여 구현하는 방식(code-first라고도 함)이며, 규약우선(contract-first) 모델은 WSDL 파일을 먼저 작성한 후 자바 소스 코드를 작성하는 방식이다.

프링 인티그레이션(Spring Integration)은 몇 가지 공통적인 통합 패턴의 구현체를 스프링의 선언적 방식으로 제공한다.

이 책에서는 스프링 인티그레이션은 다루지 않는다. 하지만 스프링 인티그레이션에 대한 더 많은 정보는 마크 피셔(Mark Fisher), 요나스 파트너(Jonas Partner), 마리우스 보고예비치(Marius Bogoevici), 그리고 이바인 펄드(Iwein Fuld)가 쓴 『Spring Integration in Action』을 읽어보거나 스프링 인티그레이션 홈페이지인 http://www.springsource.org/spring-integration을 살펴보기 바란다.

스프링 배치

데이터의 일괄 작업이 필요하다면 배치 처리만큼 좋은 방법은 없다. 배치 애플리케이션을 개발한다면 스프링 배치(Spring Batch)를 이용해 스프링의 강건한 POJO 지향 개발 모델을 활용할 수 있다.

스프링 배치는 이 책의 범위 밖이다. 하지만 티에리 템플라(Thierry Templier)와 아르노 꼬골르느(Arnaud Cogoluègnes)가 쓴 『Spring Batch in Action』에서 많은 정보를 얻을 수 있다. 또한 스프링 배치 홈페이지 http://static.springsource.org/spring-batch에서 스프링 배치에 대해 배울 수 있다.

스프링 소셜

소셜 네트워크는 인터넷에서 떠오르는 트렌드로, 페이스북이나 트위터 등의 소셜 네트워크 사이트와의 통합 기능을 갖춘 애플리케이션도 계속 증가하는 추세다. 관심 있는 분야라면 스프링의 소셜 네트워크 확장 기능인 스프링 소셜(Spring Social)을 살펴보기 바란다.

스프링 소셜은 비교적 최신 분야이며 이 책에서 별도로 다루지는 않지만, 보다 많은 정보는 http://www.springsource.org/spring-social에서 확인 가능하다.

스프링 모바일

모바일 애플리케이션은 소프트웨어 개발의 또 하나의 중요한 분야다. 많은 사용자들이 선호하는 단말이 스마트폰과 태블릿 기기로 이동하고 있다. 스프링 모바일(Spring Mobile)은 모바일 웹 애플리케이션 개발을 지원하는 스프링의 새로운 확장 기능이다.

스프링 모바일과 관련된 프로젝트로는 스프링 안드로이드(Spring Android) 프로젝트가 있다.

이 새로운 프로젝트는 스프링 프레임워크가 제공하는 단순함을 안드로이드용 네이티브 애플리케이션 개발에 적용하는 데 그 목적이 있다. 처음부터 이 프로젝트는 안드로이드 애플리케이션 내에서 사용할 수 있는 스프링의 `RestTemplate` 버전을 제공한다(`RestTemplate`은 11장에서 다룬다).

이 프로젝트 역시 새로운 분야이고 이 책의 범위를 벗어나지만, 더 다양한 내용은 http://www.springsource.org/spring-mobile과 http://www.springsource.org/spring-android에서 배울 수 있다.

스프링 DM

스프링 DM(Spring Dynamic Module)은 OSGi의 동적 모듈화 방식과 스프링의 선언적 DI가 조화를 이룬다. 스프링 DM을 이용하면, OSGi 프레임워크 내에서 서비스를 선언적으로 발행하고 소비하는 별개의 높은 응집도와 낮은 결합도의 모듈로 구성된 애플리케이션을 구축할 수 있다.

주목할 점은 OSGi 세계에서의 엄청난 충격으로 인해 선언적 OSGi 서비스에 대한 스프링 DM 모델은 OSGi Blueprint Container로 OSGi 명세 자체가 공식화됐다. 또한 SpringSource는 스프링 DM을 Gemini 패밀리의 일부로 이클립스 프로젝트로 전환했으며, 지금은 Gemini Blueprint로 알려져 있다.

스프링 LDAP

DI와 AOP 외에도 스프링 프레임워크를 통해 적용되는 또 다른 공통 기법은 JDBC 쿼리 또는 JMS 메시징같이 불필요하게 복잡한 작업에 대해 템플릿 기반의 추상화를 만드는 것이다. 스프링 LDAP은 LDAP에 스프링 스타일의 템플릿 기반 액세스를 제공하며, 일반적으로 LDAP 작업에 포함된 단순 반복적인 코드를 제거한다.

스프링 LDAP에 관한 더 많은 정보는 http://www.springsource.org/ldap에서 확인 가능하다.

스프링 리치 클라이언트

웹 기반 애플리케이션이 전통적인 데스크톱 애플리케이션에서 주목을 빼앗아간 듯 보인다. 하지만 아직 스윙(Swing) 애플리케이션을 개발한다면 스윙에 스프링의 힘을 제공하는 풍부한 애플리케이션 툴킷인 스프링 리치 클라이언트(Spring Rich Client)를 확인해 보기 바란다.

스프링.NET

.NET 프로젝트에서도 DI와 AOP를 포기할 필요가 없다. 스프링.NET(Spring.NET)은 .NET 플랫폼에 대해서도 스프링의 느슨한 결합과 애스펙트 지향 기능을 동일하게 제공한다.

핵심적인 DI와 AOP 기능 외에도 스프링.NET은 ADO.NET, NHibernate, ASP.NET, 그리고 MSMQ 작업에 대한 모듈을 포함해 .NET 개발을 단순화하기 위한 몇 가지 모듈이 있다.

스프링.NET에 관한 더 많은 정보는 http://www.springframework.net에서 확인할 수 있다.

스프링 – 플렉스

어도비(Adobe)의 플렉스(Flex)와 에어(AIR)는 리치 인터넷 애플리케이션 개발을 위한 강력한 기능을 제공한다. 이러한 리치 UI가 서버 측 자바 코드와 상호작용해야 할 경우 BlazeDS로 알려진 리모팅과 메시징 기법을 이용할 수 있다. 스프링-플렉스(Spring-Flex) 통합 패키지는 플렉스와 에어 애플리케이션이 BlazeDS를 이용하여 서버 측 스프링 빈과 통신할 수 있게 한다. 또한 스프링 루(Spring Roo)를 위한 부가기능이 있으며, 플렉스 애플리케이션의 신속한(rapid) 애플리케이션 개발을 가능하게 한다.

스프링-플렉스의 탐구는 http://www.springsource.org/spring-flex에서 시작할 수 있으며, 스프링 액션스크립트(Spring ActionScript)는 http://www.springactionscript.org에서 살펴보기 바란다. 액션스크립트에서 스프링 프레임워크의 다양한 혜택을 제공한다.

스프링 루

점점 더 많은 개발자들이 스프링을 이용하여 작업함에 따라 스프링과 관련 프레임워크에 관한 공통 용어들과 모범 사례(best practice)가 등장하고 있다. 동시에 애플리케이션 개발 작업을 단순하게 하는 스크립트 기반 개발 모델과 함께 루비 온 레일스(Ruby on Rails)와 그레일스(Grails) 등의 프레임워크가 떠오르고 있다.

스프링 루(Spring Roo)는 상호작용적인 도구 환경을 제공하여 스프링 애플리케이션의 신속한 개발을 가능하게 하며, 지난 몇 년간 확인된 모범 사례를 통합했다.

루와 다른 신속한 애플리케이션 개발 프레임워크와의 차이점은 스프링 프레임워크를 사용하여 자바 코드를 만든다는 점이다. 결과물은 많은 기업 개발 조직에 낯선 언어로 코딩된 별도의 프레임워크가 아니라 하늘에 한 점 부끄럼 없는 스프링 애플리케이션이다.

스프링 루에 관한 더 많은 정보는 http://www.springsource.org/roo에서 확인할 수 있다.

스프링 확장

지금까지 설명한 모든 프로젝트 외에도 http://www.springsource.org/extensions에 커뮤니티 주도의 스프링 확장 모음이 있다. 여기에 포함된 유용한 기능은 다음과 같다.

- 파이썬(Python) 언어를 위한 스프링 구현체
- 블롭(Blob) 저장
- db4o와 CouchDB 퍼시스턴스
- 스프링 기반 워크플로(workflow) 관리 라이브러리
- 스프링 시큐리티를 위한 커버로스(Kerberos)와 SAML 확장

1.4 스프링의 새로운 기능

이 책의 2판을 작성한 후 거의 3년이 흘렀고, 그 기간 동안 많은 일들이 벌어졌다. 스프링 프레임워크에는 두 번의 중요한 릴리즈가 있었는데, 각각은 애플리케이션 개발을 쉽게 하는 새로운 기능과 개선을 가져왔다. 그리고 스프링 포트폴리오의 몇 가지 다른 멤버도 큰 변화를 겪었다.

이 책은 이와 같은 변화를 다룬다. 하지만 지금은 스프링에서 새로워진 기능이 무엇인지부터 간단히 살펴보자.

1.4.1 스프링 2.5에서 새로워진 기능

2007년 11월, 스프링 프레임워크의 2.5 버전이 릴리즈됐다. 스프링 2.5의 의미는 애너테이션 기반 개발이 채택됐다는 데 있다. 스프링 2.5 이전에는 XML 기반 설정이 표준이었다. 하지만 스프링 2.5에서 도입된 다양한 방식의 애너테이션 활용은 스프링 설정에 필요한 XML의 양을 급격히 줄였다.

- `@Autowired` 애너테이션을 통한 애너테이션 기반 DI와 `@Qualifier`를 이용한 세분화된 오토와이어링(auto-wiring) 제어
- 생명주기 메소드를 위한 `@PostConstruct`와 `@PreDestroy`뿐만 아니라 명명된 리소스의 DI를 위한 `@Resource`를 포함한 JSR-250 애너테이션 지원
- `@Component` 애너테이션(또는 다양한 스테레오타입 애너테이션 중 하나)을 적용한 스프링 컴포넌트 자동 탐지(auto-detection)

- 스프링 웹 개발을 크게 간소화시키는 완전히 새로운 애너테이션 기반 스프링 MVC 프로그래밍 모델
- JUnit 4와 애너테이션 기반의 새로운 통합 테스트 프레임워크

스프링 2.5의 애너테이션은 중요한 얘기지만 다음과 같은 내용도 더 있다.

- JDBC 4.0, JTA 1.1, JavaMail 1.4, 그리고 JAX-WS 2.0를 포함한 전체 자바 6와 자바 EE 5 지원
- 이름으로 스프링 빈에 애스펙트를 위빙(weaving)하는 새로운 빈-이름 포인트컷 표현식
- AspectJ 로딩 시점 위빙을 위한 지원 기능 내장
- 애플리케이션 컨텍스트 세부정보 설정을 위한 `context` 네임스페이스와 메시지 드리븐(message-driven) 빈 설정을 위한 `jms` 네임스페이스를 포함한 새로운 XML 설정 네임스페이스
- `SqlJdbcTemplate`에서의 명명된 파라미터 지원

이 책을 통해 이와 같은 새로운 스프링 기능을 살펴본다.

1.4.2 스프링 3.0에서 새로워진 기능

스프링 2.5의 훌륭한 기능들로 인해 뒤이은 스프링 3.0에서 가능한 기능이 무엇인지 상상하기 쉽지 않았다. 하지만 3.0 릴리즈와 함께 스프링은 애너테이션 기반 사상의 유지와 다양한 새로운 기술로 한 단계 도약했다.

- XML, JSON, RSS, 또는 다른 적절한 응답을 이용하여 REST 스타일 URL에 응답하는 스프링 MVC 컨트롤러를 포함한 스프링 MVC의 전체 규모 REST 지원. 스프링 3.0의 새로운 REST 지원은 11장에서 살펴본다.
- 다른 빈과 시스템 속성을 포함하여 다양한 소스로부터 값의 주입을 가능하게 하여 스프링 DI를 새로운 수준으로 제공하는 새로운 표현식 언어(expression language). 스프링의 표현식 언어는 2장에서 자세히 살펴본다.
- 쿠키와 요청 헤더에서 각각 값을 가져오는 `@CookieValue`와 `@RequestHeader`를 포함한 스프링 MVC를 위한 새로운 애너테이션. 7장에서 스프링 MVC를 살펴보면서 이러한 애너테이션의 사용법을 알아본다.
- 스프링 MVC의 쉬운 설정을 위한 새로운 XML 네임스페이스
- JSR-303(Bean Validation) 애너테이션을 이용한 선언적 유효성 검증 지원

- 새로운 JSR-330 DI 명세 지원
- 비동기식과 스케줄링된 메소드의 애너테이션 지향 선언
- XML이 없이 스프링 설정이 가능한 새로운 애너테이션 기반의 설정 모델. 이 새로운 설정 방식은 2장에서 살펴본다.
- 객체-XML(OXM) 매핑 기능이 스프링 웹 서비스(Spring Web Services) 프로젝트에서 코어 스프링 프레임워크로 이동

스프링 3.0의 새로운 기능만큼이나 스프링 3.0에 없는 기능이 무엇인지도 파악하는것도 중요하다. 특히, 스프링 3.0으로 시작하려면 이제는 자바 5 이상이 필요하다. 자바 1.4는 더 이상 스프링에서 지원하지 않는다.

1.4.3 스프링 포트폴리오에서 새로워진 기능

코어 스프링 프레임워크 외에도 스프링을 기반으로 하는 프로젝트에는 흥미로운 새로운 움직임이 있다. 모든 세부적인 변경사항을 다루기에 지면이 부족하지만 중요하다고 생각되는 몇 가지 항목을 살펴보면 다음과 같다.

- **스프링 웹 플로 2.0(Spring Web Flow 2.0)**은 대화형 웹 애플리케이션 생성을 더 쉽게 만드는 간편해진 플로 정의 스카마와 함께 릴리즈됐다.
- 스프링 웹 플로 2.0과 함께 **스프링 자바스크립트(Spring JavaScript)**와 **스프링 페이스(Spring Faces)**가 생겼다. 스프링 자바스크립트는 동적인 동작을 이용하여 웹 페이지의 점진적 향상을 가능하게 하는 자바스크립트 라이브러리다. 스프링 페이스는 스프링 MVC와 스프링 웹 플로 내에서 뷰 기술로 JSF의 사용을 가능하게 한다.
- 예전의 아씨지 시큐리티(Acegi Security) 프레임워크는 완벽히 정비되어 **스프링 시큐리티 2.0(Spring Security 2.0)**으로 릴리즈됐다. 이러한 새로운 구현에서 스프링 시큐리티는 애플리케이션 보안을 설정하는 데 필요한 XML의 양을 극적으로 줄이는 새로운 설정 스키마를 제공한다.

이 책을 쓰는 시점에도 스프링 시큐리티는 계속 발전하고 있다. 스프링 시큐리티 3.0이 최근에 릴리즈됐는데, 보안 제약조건을 선언하는 스프링의 새로운 표현식 언어를 활용하여 선언적 방식의 보안 적용을 더 간소화한다.

보다시피 스프링은 활동적이며, 계속해서 진화하는 프로젝트다. 엔터프라이즈 자바 애플리케이션 개발을 더 용이하게 만드는 것을 목표로 새로운 무엇이 항상 존재한다.

1.5 요약

지금까지 스프링을 이용해 어떤 일을 할 수 있는지 알아봤다. 스프링은 엔터프라이즈 자바 개발자의 작업을 쉽게 하고, 결합도가 낮은 코드를 만드는 것을 목적으로 한다. 여기서 핵심은 DI와 AOP다.

1장에서는 스프링의 DI에 대해 간단히 살펴봤다. DI는 객체들이 상호 간의 종속관계나 구체적인 구현 방법을 알 필요가 없도록 애플리케이션 객체들을 연결하는 방법이다. 종속객체를 필요로 하는 객체가 직접 종속객체를 얻어오는 것이 아니라, 종속객체를 필요로 하는 객체에 종속객체가 부여되는 방식이다. 종속객체를 필요로 하는 객체는 인터페이스를 통해서 주입된 객체만 알기 때문에 결합도가 낮아진다.

DI에 이어 스프링의 AOP 지원에 대해서도 간단히 살펴봤다. AOP를 이용하면 일반적으로 애플리케이션의 여러 곳에 흩어져 있는 로직을 한 곳으로, 즉 애스펙트로 모을 수 있다. 각 애스펙트는 스프링이 빈들을 와이어링할 때 런타임에 연결되어 효과적으로 빈에 새로운 기능을 부여한다.

DI와 AOP는 스프링의 중추에 해당한다. 스프링 프레임워크를 잘 사용하려면 이 두 가지 주요 기능을 잘 이해하고 사용해야 한다. 이 장에서는 스프링의 DI와 AOP 기능의 기본적인 사항들을 알아봤지만, 뒤에 나오는 여러 장을 통해 DI와 AOP를 자세히 알아보겠다. 그럼 이제 잔소리는 그만 접어 두고 2장으로 이동하여 스프링에서 DI를 이용해 객체들을 연결하는 방법을 알아보자.