



An introduction to Spring

Adrian Colyer, CTO, Interface21
adrian.colyer@interface21.com



Who am I?

- CTO, Interface21
 - the company behind the Spring Framework
 - and many related projects in the Spring portfolio
- In addition to overall responsibility for Spring portfolio...
 - Leader AspectJ project on Eclipse.org
 - Leader Spring OSGi project
 - Founder AJDT project
 - Committer on Core Spring



Agenda

- What *is* the Spring Framework?
 - and what does it do for me?
- Spring concepts
 - dependency injection, aop, and portable service abstractions
 - typical application architecture
- Spring in the data access layer
- Spring in the service layer
- The bigger picture

What is the Spring Framework?

- A lightweight container for application objects
 - lifecycle management
 - dependency management and wiring
 - non-invasive enterprise services
- A set of modules simplifying common enterprise application development tasks
 - persistence
 - messaging
 - ...
- An approach to building enterprise applications



The Spring approach

- Program simply using objects
 - aka "POJOs"
 - non-invasive
 - object-oriented
- Retain architectural choice
 - no environmental assumptions or dependencies in application objects
- Facilitate test-driven development



Examples

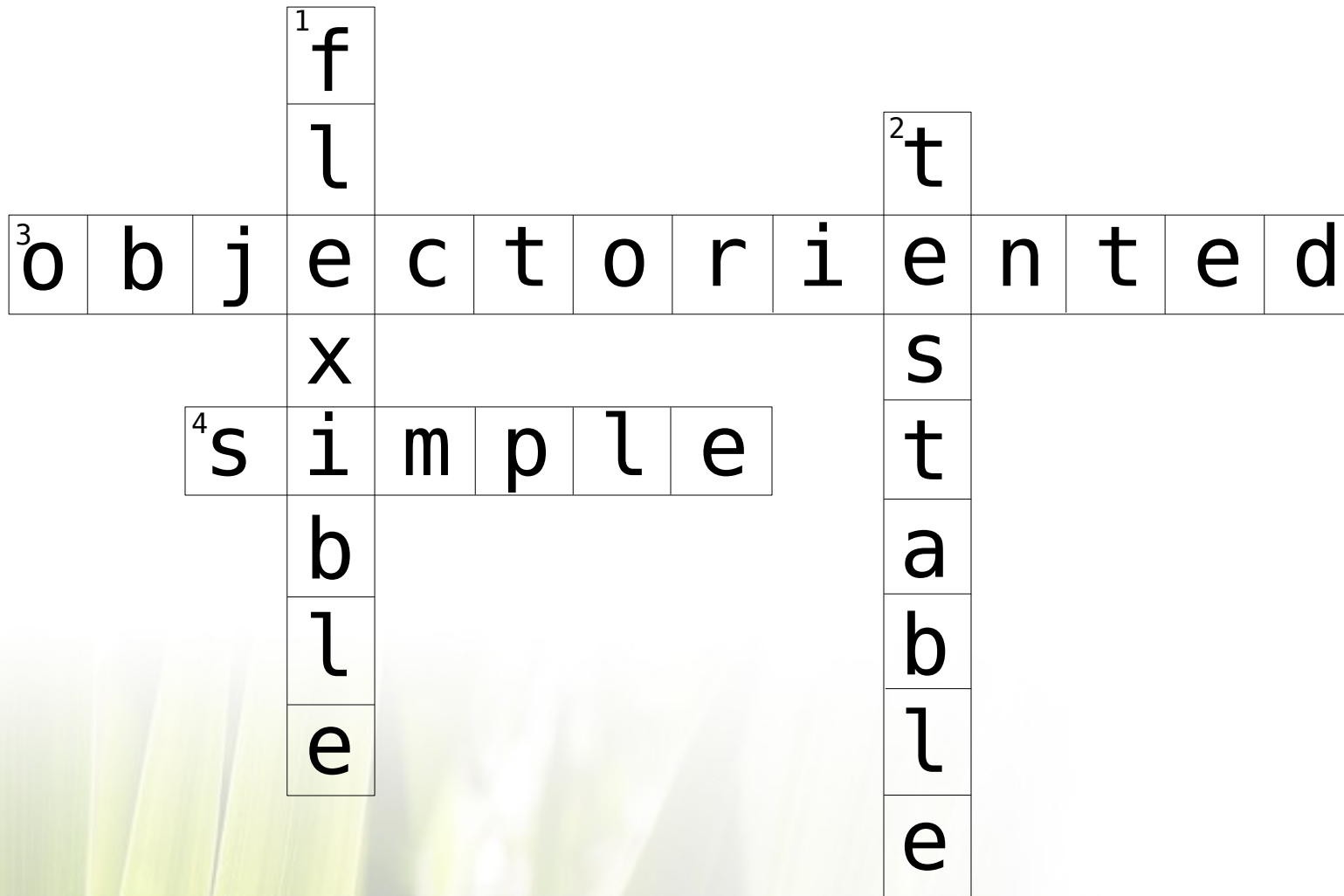
- A Spring "bean" is just an instance of a regular class
 - no need to depend even on Spring APIs
- Can be made transactional
 - without needing to know about any transaction APIs
- Can be exposed for management via JMX
 - without needing to know about any JMX APIs or conventions
- Can be tested in isolation
 - as a "unit" during unit testing
 - in integration testing outside of the app. server



Example : Voca

- Part of UK's Critical National Infrastructure
 - Process Direct Debits, Direct Credits and Standing Orders to move money between banks
 - Over 5 billion transactions worth €4.5 trillion in 2005
 - 15% of Europe's Direct Debits and Direct Credits are handled by Voca
 - Over 70% of the UK population use Direct Debits to pay household bills; Direct Credits are used to pay over 90% of UK salaries
 - Over 72 million items on a peak day
 - They have never lost a payment
- In production on Spring-based implementation of this infrastructure
 - replaced existing mainframe system
 - better scalability and throughput!

The Spring approach



The Spring approach

Simple does not mean
simplistic



The Spring approach

Simpler can be more
powerful



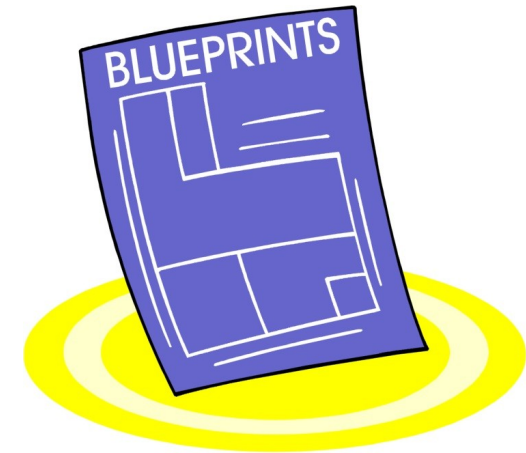
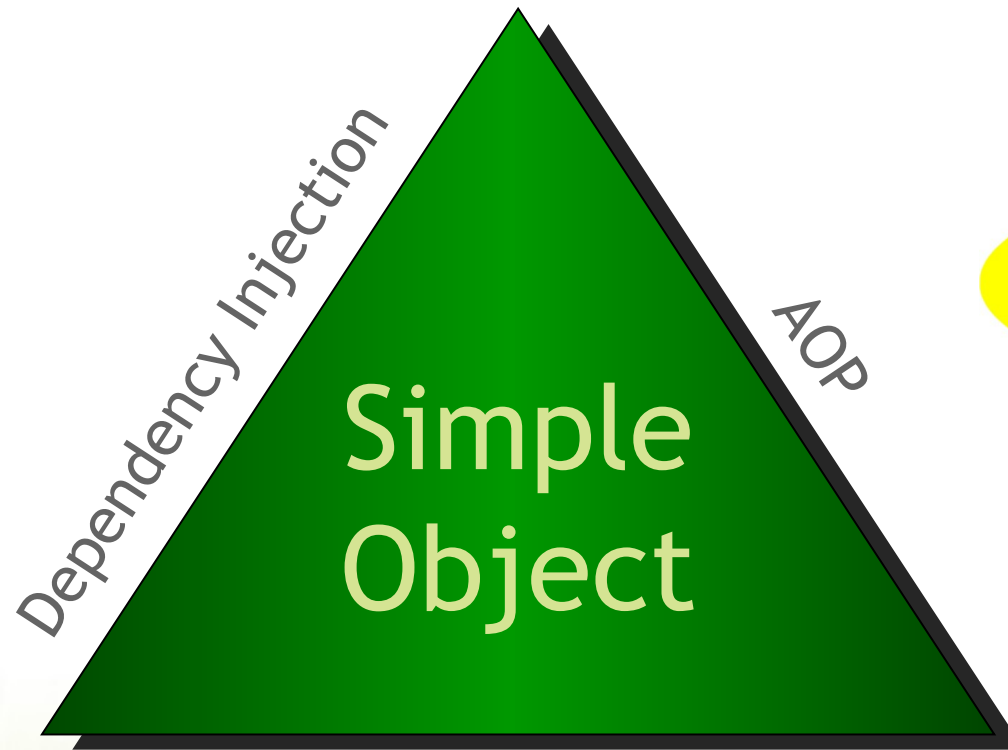
Example

- Components are simple Java objects that know nothing about the execution environment
 - can switch from local to global transactions with only a configuration change
 - architectural decision can be deferred
 - can take advantage of underlying infrastructure without coupling code
 - e.g. `WebLogicPlatformTransactionManager`



Spring Concepts

Enabling Technologies



Portable Service Abstractions



Don't call me, ok?

- Goal: components that are simple units, testable in isolation
 - can't have environmental assumptions in code
 - can't have global application knowledge
 - beyond "there must exist a service like this and I need access to it"
 - need to eliminate boilerplate, error-prone code
- Fundamentally achieved by "Inversion of Control"
 - bean is given collaborators (no lookups)
 - "template" classes handle common interactions

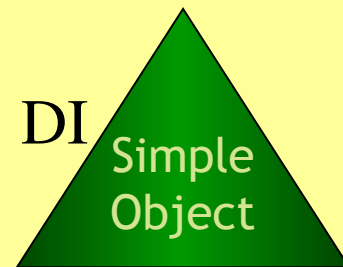
Dependency Injection

- Following JavaBeans conventions, write a simple Java class with constructor and/or setter methods

```
public class AccountServiceImpl implements
    AccountService {
    private AccountDao accountDao;

    public void setAccountDao(AccountDao dao) {
        this.accountDao = dao;
    }

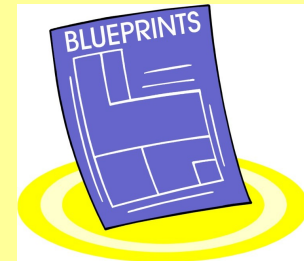
    ...
}
```



Dependency Injection

- Define a component (a "bean") in the blueprint
- Configure its properties

```
<bean id="accountService"  
    class="...AccountServiceImpl">  
    <property name="accountDao"  
        ref="accountDao" />  
</bean>  
  
<bean id="accountDao"  
    class="...HibernateAccountDao">  
    ...  
</bean>
```





Spring is responsible for...

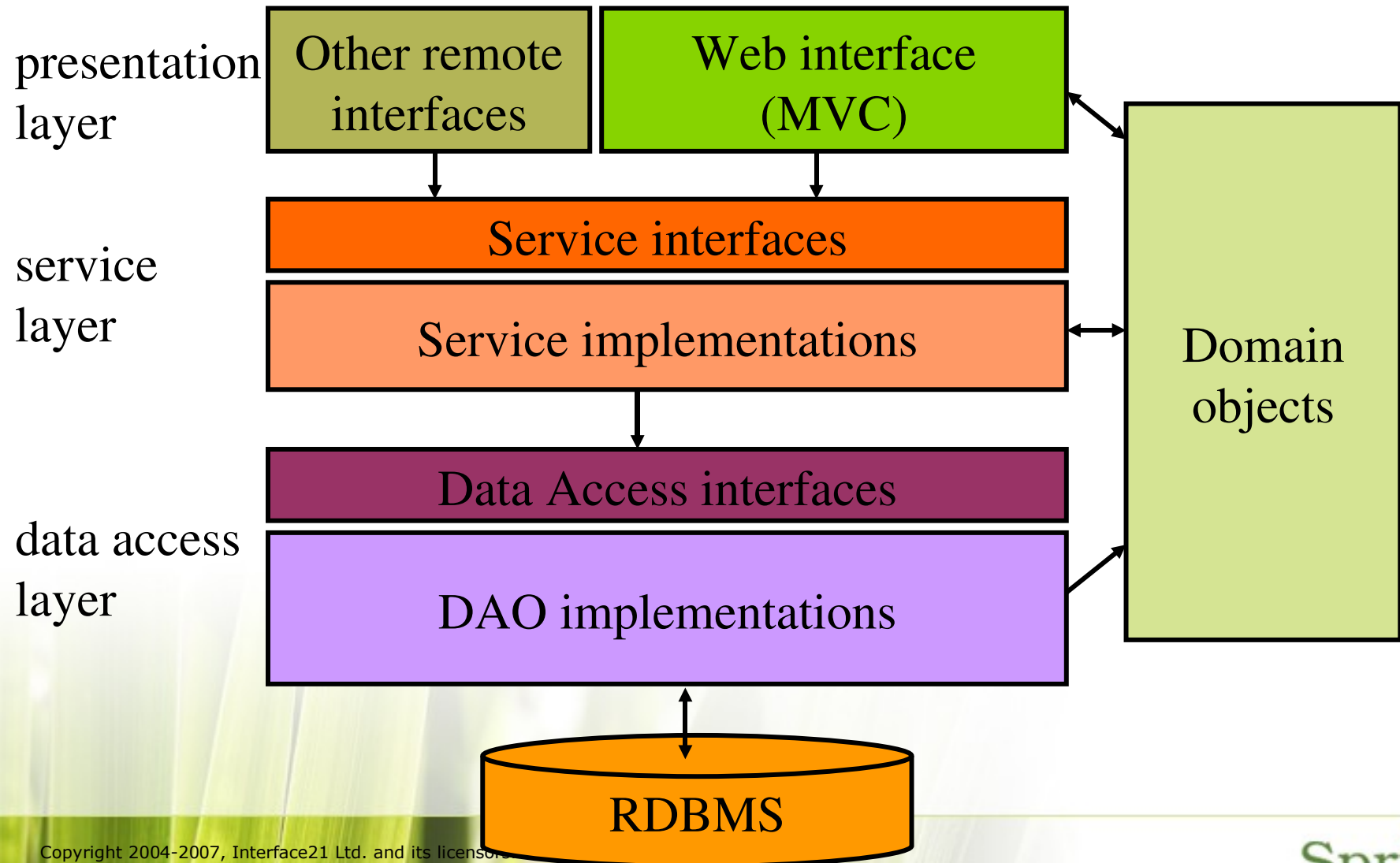
- **Instantiating** component instances
 - account service, account dao, session factory, data source, etc.
- **Configuring** component instances
 - setting simple properties
- **Decorating** components
 - ensuring enterprise services such as transaction management are in place
- **Assembling** components into a fully functioning application
 - wiring components together so that they can do their jobs



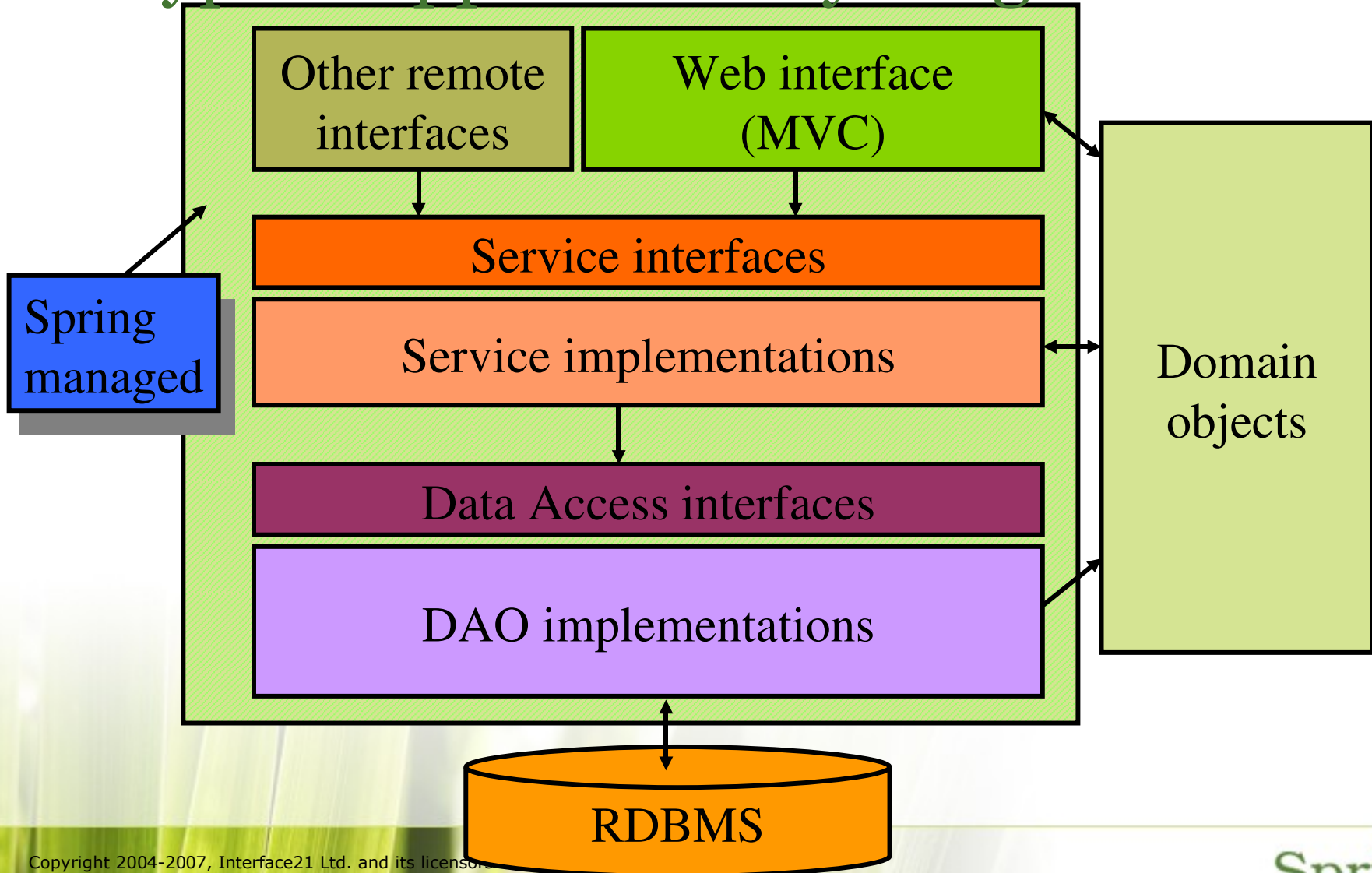
Core Container

- Facilities provided for configuration and assembly are extensive
 - no need to "code for Spring"
 - can configure Spring to work with existing classes
- Lifecycle management
 - singleton
 - prototype
 - request
 - session
 - global session

Typical application layering



Typical application layering





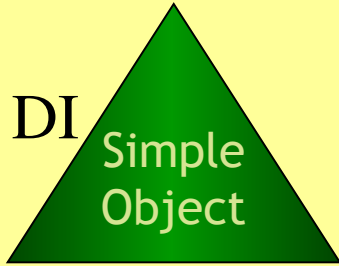
Isolated Units

- Web views
 - don't know which controller requested them
- Web controllers
 - don't know how view objects are found
 - don't know where services come from
- Business services
 - don't know anything about web layer
 - don't know where repository objects come from
 - don't know about tx, security etc.
- Data Access objects
 - don't know anything about upper layers
 - don't know about tx, session management etc.

Example: data source

```
public class MyJdbcDao implements MyDao {  
    private DataSource dataSource;  
  
    public void setDataSource(DataSource ds) {  
        this.dataSource = ds;  
    }  
  
    ...  
}
```

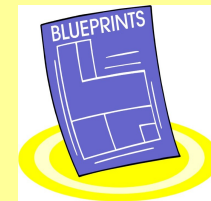
DI
Simple
Object



Example: data source

```
<jee:jndi-lookup id="dataSource"
  jndi-name="java:comp/env/jdbc/myds" />

<bean id="mydao" class="MyJdbcDao">
  <property name="dataSource"
    ref="dataSource" />
</bean>
```

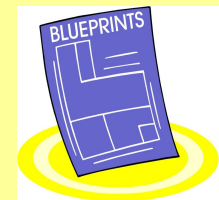


- Inside application server

Example: data source

```
<bean id="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName" value="..." />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>

<bean id="mydao" class="MyJdbcDao">
  <property name="dataSource"
    ref="dataSource" />
</bean>
```



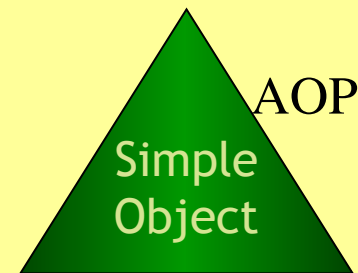


AOP

- Dependency injection ensures beans don't know where collaborators and configuration comes from
- AOP (aspect-oriented programming) eliminates dependencies on enterprise service APIs
 - (and a lot more besides ;))
- Spring AOP is a simple runtime proxying implementation
 - no special build steps or compilers etc.
- No need to become an AOP expert to take advantage of the built-in facilities

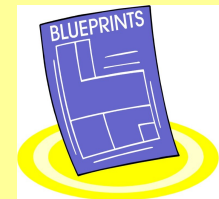
Example: Transactions

```
@Transactional
public class AccountServiceImpl implements
    AccountService {
    private AccountDao accountDao;
    ...
    @Transactional(readOnly=true)
    public void Money getBalance(
        Long accountId) { ... }
}
```



Transaction blueprint

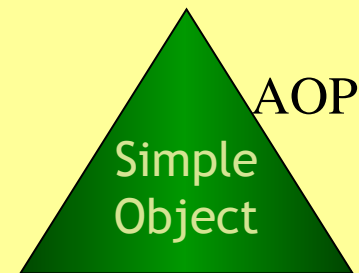
```
<!--  
    tell Spring to honour Transactional  
    annotations  
-->  
<tx:annotation-driven/>  
  
<bean id="transactionManager"  
    class="org.springframework.jta.transaction.JtaTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```



Example: Transactions

```
public class AccountServiceImpl implements
    AccountService {
    private AccountDao accountDao;
    ...

    public void Money getBalance(
        Long accountId) { ... }
}
```

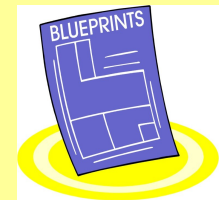


Transaction blueprint

```
<aop:config>
  <aop:advisor
    pointcut="...SystemArchitecture.businessService()"
    advice-ref="txDemarcation"/>
</aop:config>

<tx:advice id="txDemarcation">
  <tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>

<bean id="transactionManager" ... />
```





Enterprise application vocabulary

the **vocabulary** of enterprise
applications business service
service layer
repository
controller dao web layer
data access layer



Requirements

- Many requirements are expressed *in terms of this vocabulary*
 - the **service layer** should be transactional
 - when a Hibernate **dao operation** fails the exception should be translated
 - **service layer** objects should not call the **web layer**
 - a **business service** that fails with a concurrency related failure can be retried

Meaningful abstractions

It would be **simpler...**

and more *powerful*

Meaningful abstractions

if we could use these
~~abstractions~~
directly in the
implementation



Scenario...

- You have your own **data access layer** written using Hibernate 3
 - not using the Spring HibernateTemplate
- In the **service layer**, you want to insulate yourself from Hibernate exceptions, and take advantage of Spring's fine-grained `DataAccessException` hierarchy
- After throwing a hibernate exception from a **data access operation**, convert it into a `DataAccessException`...



Step 1: Define the abstraction

```
@Aspect
public class SystemArchitecture {

    ...

    @Pointcut("execution(* a.b.c.dao.*.*(..))")
    public void dataAccessOperation() {}

    ...
}
```



Step 2: Use the abstraction

- How do we make use of this `dataAccessOperation` abstraction?
- Advice!
- Advice is associated with a pointcut expression
- Executes every time a join point matched by the expression occurs



Which advice kind?

- "After throwing a hibernate exception from a **data access operation**, convert it into a `DataAccessException`..."

After throwing

```
@AfterThrowing(  
    throwing="hibernateEx",  
    pointcut="SystemArchitecture.dataAccessOperation()"  
)  
public void rethrowAsDataAccessException(  
    HibernateException hibernateEx) {  
    // convert exception and rethrow...  
}
```



Where does advice live?

- Advice is declared in an **aspect**
- Aspects are like classes
 - instances
 - state (fields)
 - behaviour (methods)
- Aspects can also have
 - pointcuts
 - advice
 - and a few other things...



Aspect

```
@Aspect
public class HibernateExceptionTranslator {
    // ...
    @AfterThrowing(
        throwing="hibernateEx",
        pointcut="SystemArchitecture.dataAccessOperation()"
    )
    public void rethrowAsDataAccessException(
        HibernateException hibernateEx) {
        // convert exception and rethrow...
    }
}
```

Step 3: Configuration

```
<aop:aspectj-autoproxy/>  
  
<bean id="hibernateExceptionTranslator"  
      class="HibernateExceptionTranslator">  
    <property name="hibernateTemplate"  
              ref="hibernateTemplate"/>  
</bean>
```

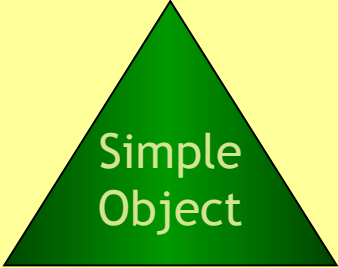


Portable Service Abstractions

- For when only an API will do
- Template classes based around principle of inversion of control
- Typically handle
 - resource acquisition and release
 - exception translation
 - iteration (if applicable)
- Leaving you to focus on the business logic

Example: JdbcTemplate

```
jdbcTemplate.queryForInt(  
    "SELECT COUNT(0) FROM T_CLIENT " +  
    "WHERE TYPE=? AND CURRENCY=?",  
    clientType, currency);
```

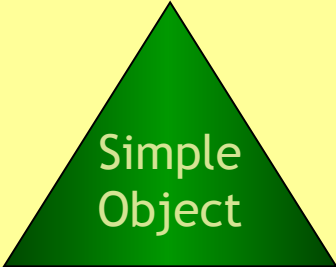
A green triangle pointing upwards, containing the text 'Simple Object'.

Simple
Object

PSA

Example: JdbcTemplate

```
List<Person> people = template.query(  
    "SELECT id,name FROM person",  
    new ParameterizedRowMapper<Person>() {  
        public Person mapRow(ResultSet rs,int rowNum)  
            throws SQLException {  
            return new Person(rs.getLong("id"),  
                               rs.getString("name"));  
        }  
    }  
);
```

A green triangle pointing upwards, containing the text 'Simple Object' and 'PSA' below it.

Simple
Object

PSA



JDBC coding with Spring

With Spring, only the highlighted lines still need to be coded

1. Define connection parameters
2. Open the connection
3. Specify the statement
4. Prepare and execute the statement
5. Specify loop to iterate through the results
6. Do the work for each iteration
7. Process exceptions
8. Handle transactions
9. Close the connection/return to pool



JdbcTemplate: portability

- Sophisticated exception translation
 - to fine-grained `DataAccessException` hierarchy
 - used for all persistence modules
 - no need to code for specific database
- Consistent approach to
 - stored procedures
 - stored functions
 - LOB handling



Spring in the Data Access Layer



Data Access Layer

- Template classes for wide variety of popular data access technologies
 - JDBC
 - iBatis
 - Hibernate
 - Jdo
 - Toplink
 - JPA
 - ...



Common concepts and approach

- Template class manages resource acquisition and release, iteration, exception mapping
- Automatically participant in transactions
- Optional DaoSupport superclasses provide easy access to templates
 - HibernateDaoSupport
 - JdbcDaoSupport
 - JpaDaoSupport
 - ...
- Fully configured in blueprint



Transaction management

- Daos automatically participate in transactions
- Simply select the appropriate transaction manager implementation
 - DataSourceTransactionManager -> JDBC
 - HibernateTransactionManager -> Hibernate (plus JDBC)
 - JtaTransactionManager -> global transactions
- Spring drives the resource manager APIs as needed under the covers
 - e.g. Hibernate SessionFactory and Session



Exception translation

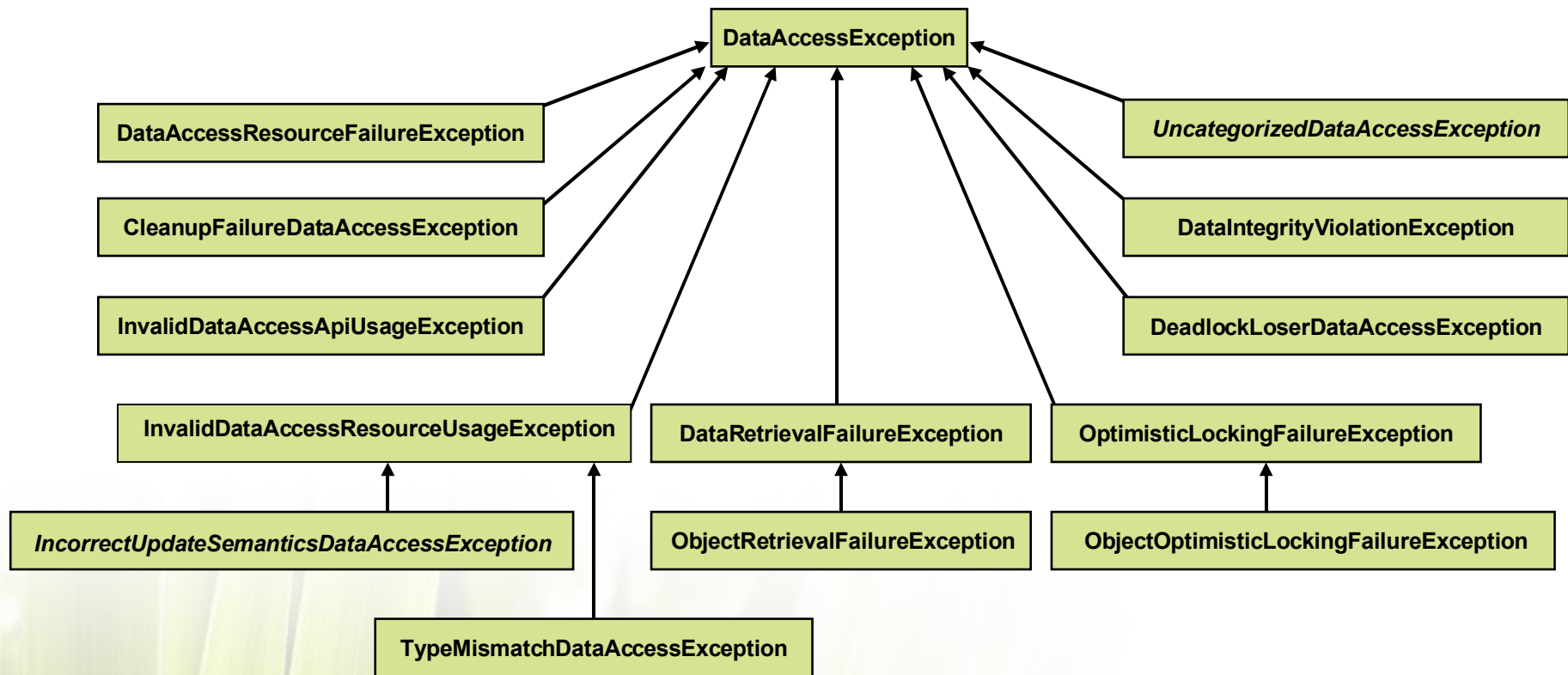
- SQLException codes tied to particular database
- Don't want service layer tied to particular persistence technology (HibernateException vs. SQLException vs. JpaException ...)
- Spring maps exceptions thrown by persistence APIs into portable service abstraction: `DataAccessException`
 - fine-grained
 - independent of database and driver
 - programs simpler and clearer



Example

- Catch "DataIntegrityViolationException"
 - works in all databases
 - works through ORM tools
 - code is self-documenting
 - cf. reliance on SQL error codes
 - interpreting codes will fail silently if move database
- Catch "DeadlockLoserDataAccessException"
 - etc.

Spring DAO: Consistent Exception Hierarchy (subset)





Integration testing

- Spring places high-value on testing
 - make it easy and fast to unit test
 - make it easy and fast to integration test persistence layer
 - with live database underneath
- AbstractTransactionalSpringContextTests
 - JUnit superclass
 - configured with a list of blueprint files
 - creates application context
 - tests run in transaction
 - automatically rolled back
 - can mix ORM code with JDBC to verify results



Spring in the Service Layer

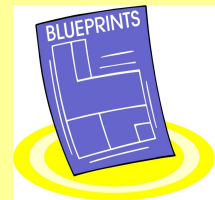


Service layer

- Easy to define core business services
- Separate interface from implementation
- All needed service and data access objects are provided through dependency injection
- Transparent support for
 - transaction management
 - security (in conjunction with Spring Security)
 - management via JMX
 - exposing services via remote interfaces

JMX example

```
<bean id="mbeanExporter"  
  class="org.sfw...MBeanExporter">  
  <property name="beans">  
    <map>  
      <entry key="i21:service=messageService">  
        <ref local="messageService"/>  
      </entry>  
      <!-- other objects to be exported here -->  
    </map>  
  </property>  
</bean>
```



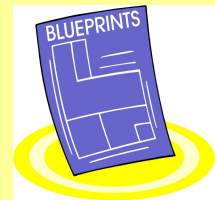


Remoting support

- Expose existing bean for remote access over a variety of protocols
 - coarse grained remoting interface recommended
- Access any remote service over a variety of protocols
- Protocols:
 - RMI
 - IIOP
 - HttpInvoker
 - Hessian, Burlap
 - SLSB
 - Xfire
- See also... Spring Web Services

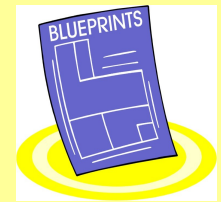
Remoting example: exporter

```
<bean name="/httpInvoker/gameRenter"  
  class="o.sfw...HttpInvokerServiceExporter">  
  <property name="service" ref="gameRenter"/>  
  <property name="serviceInterface">  
    <value>  
      c.s.training.gamecast.service.GameRenter  
    </value>  
  </property>  
</bean>
```



Remoting example: client

```
<bean id="gameRenterService"  
  class="o.sfw..HttpInvokerProxyFactoryBean">  
  <property name="serviceUrl">  
    <value>  
      http://localhost:8080/httpInvoker/gameRenter  
    </value>  
  </property>  
  <property name="serviceInterface">  
    <value>  
      c.s.training.gamecast.service.GameRenter  
    </value>  
  </property>  
</bean>
```





Other key services provided

- JMS integration
 - message driven POJOs
 - with transactional receive
 - JmsTemplate
- Scheduling
 - Quartz, JDK Timers
- Asynchronous task execution
 - portable abstraction
 - several out of the box implementations
 - thread pool
 - commonj work manager



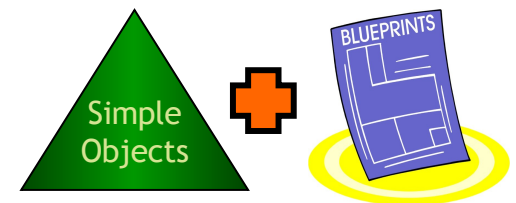
The Bigger Picture

Web Layer

- Spring MVC
 - request-response web framework
 - strong separation of model, view and controller
 - test units in isolation
 - Many view technologies
 - JSP/JSTL
 - XML & XSLT
 - Pdf
 - Excel
 - Velocity
 - Freemarker
 - ...

Web Layer

- Binding and validation
- Internationalization
- Themes
- Multipart uploads etc.
- POST-redirect-GET
- Spring WebFlow
 - separate project
 - for controlled navigation, *conversations*
 - multi-page forms etc.
 - back button, history, bookmarking, ...





Spring Security (Acegi)

- Authentication and authorization
 - exceedingly capable, wide range of authentication mechanisms integrated
 - e.g. enterprise single sign-on
- Security at web layer, service layer, repositories, and even domain objects
- Role-based & ACLs
- Domain-instance protection and filtering
- Remember-me, anonymous, ...



Other Projects

- Spring Web Services
 - contract-first web services
 - WS-Security integration through Spring Security
 - OXM framework (c.f. ORM support)
- Spring LDAP
 - LdapTemplate for accessing directories
- Spring Rich Client
 - Assemble Swing applications from simple objects + blueprints



Other projects

- Spring OSGi
 - the ultimate in modularity and runtime control
 - versioning, concurrent deployment
- Spring SCA
 - support for `implementation.spring` in SCA
- AspectJ
 - full AOP support, tightly integrated with Spring



Summary



Summary

- Spring is...
 - a lightweight container
 - a set of modules addressing common enterprise application development tasks
- Spring promotes
 - simple objects, with no knowledge of environment
 - easy to test (unit + integration)
- Core concepts
 - DI, AOP, Portable service abstractions
- Delivers
 - productivity, predictability, quality, scalability