

7 (or so) Reasons to use Spring

Arjen Poutsma
SpringSource
VMware

About Spring

Aims of Spring

- Reduce the complexity of Java J2EE development
- Simplify without sacrificing power
- Facilitate best practices
- Grew from practical experience

*Simple things should
be simple.
Complex things
should be possible.*

Alan Kay

Technical Aims of Spring

- Write POJOs
- Apply Java EE services to POJOs
 - Make it transactional
 - Expose via JMX
 - ...

POJO development

- Plain Old Java Object
- Not bound to any environment
 - No environment-specific imports
 - Not dependent on lookup mechanism
 - Dependencies are injected
 - Prolongs life
- **Test out of the container**

Modular

- Spring is not a “package deal”
- All features can be used independently
 - Though they strengthen each other

Reason #1 Dependency Injection

The Case for Dependency Injection

- Applications consist of multiple components
- How obtain these dependencies?

Pull Configuration

- JNDI lookup
- Properties file
- Service Locator anti-pattern
- Many issues
 - Most importantly: hard to test

Solution: Dependency Injection

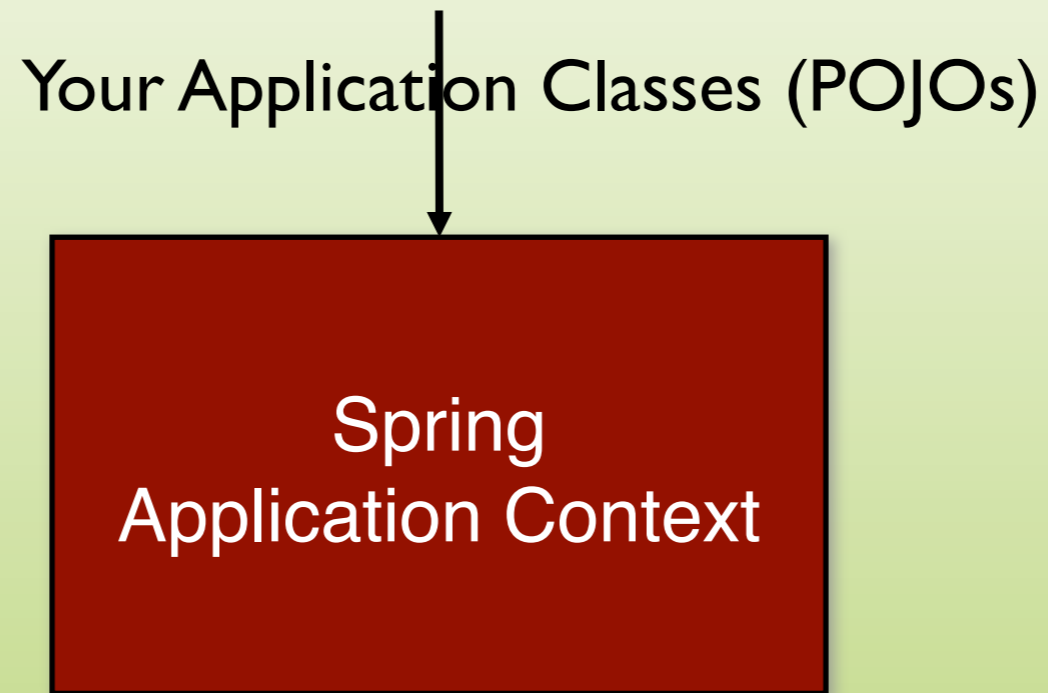
- Rather than lookup dependencies
- Let something give them to me
 - Spring

How Spring Works

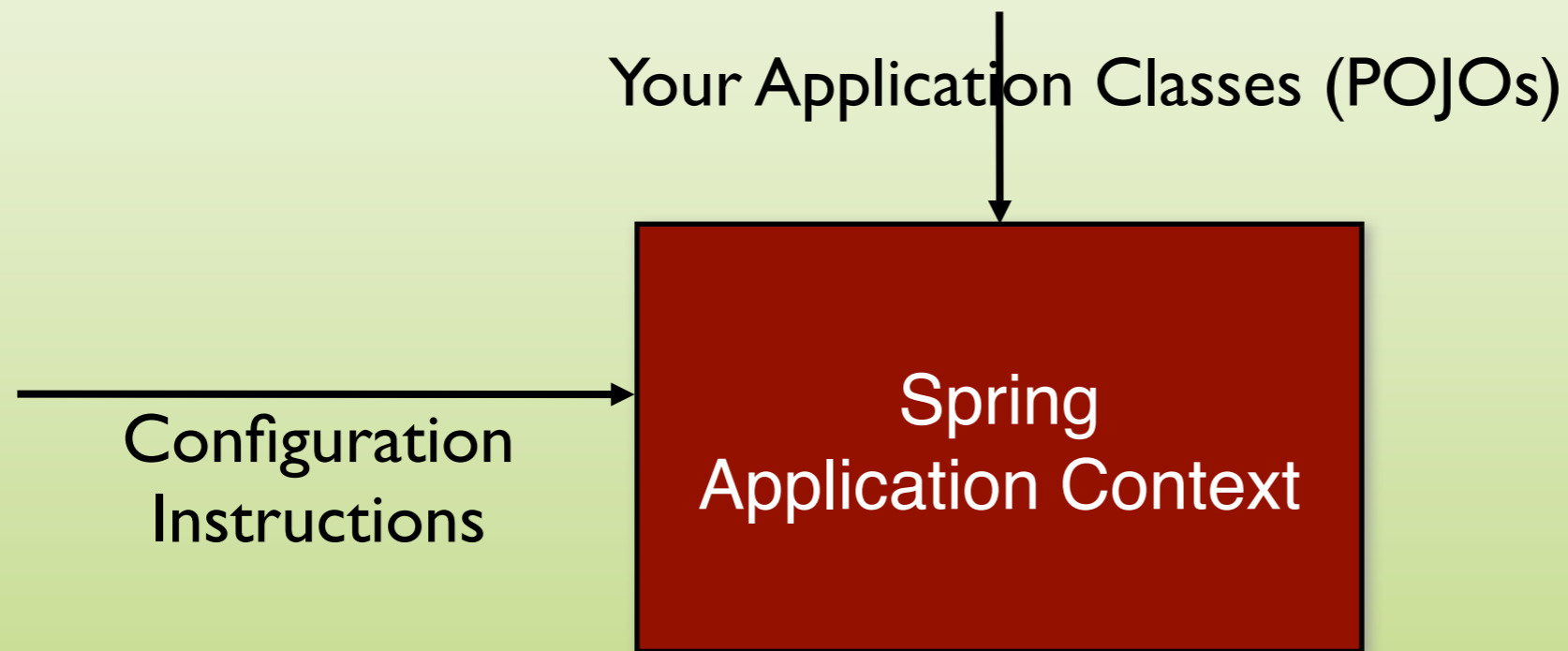
A dark red rectangular box with a thin black border, containing the text "Spring Application Context" in white. The box is centered horizontally and vertically on the slide.

Spring
Application Context

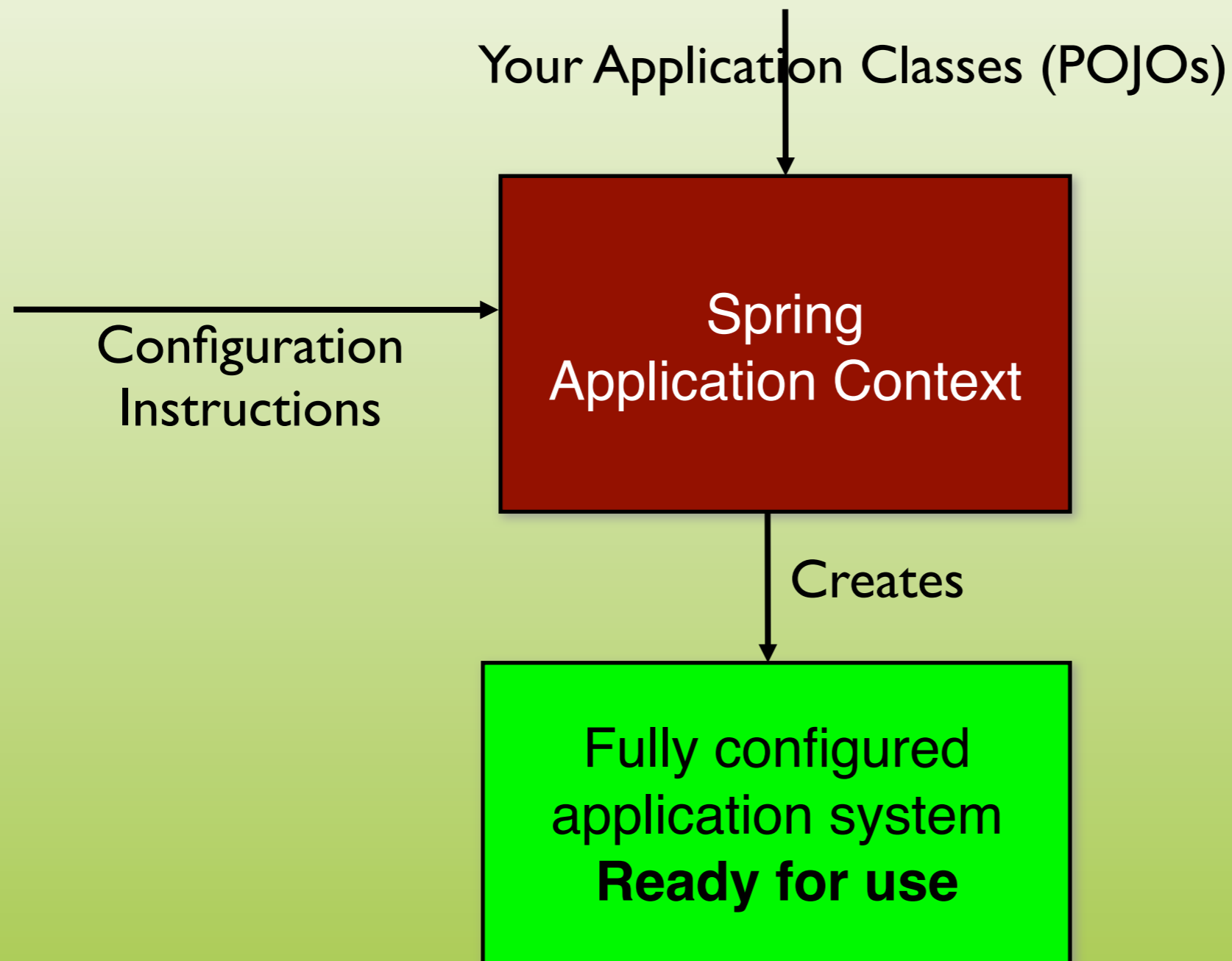
How Spring Works



How Spring Works



How Spring Works



Example

Example

```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```

Example

```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```

Needed to perform money transfers
between accounts

Example

```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```

Needed to perform money transfers
between accounts

```
public class JdbcAccountRepository implements AccountRepository {  
    public JdbcAccountRepository(DataSource ds) {  
        this.dataSource = ds;  
    }  
    ...  
}
```

Example

```
public class TransferServiceImpl implements TransferService {  
    public TransferServiceImpl(AccountRepository ar) {  
        this.accountRepository = ar;  
    }  
    ...  
}
```

Needed to perform money transfers
between accounts

```
public class JdbcAccountRepository implements AccountRepository {  
    public JdbcAccountRepository(DataSource ds) {  
        this.dataSource = ds;  
    }  
    ...  
}
```

Needed to load accounts from the database

Example

Example

```
<beans>

  <bean id="transferService" class="app.impl.TransferServiceImpl">
    <constructor-arg ref="accountRepository" />
  </bean>

  <bean id="accountRepository" class="app.impl.JdbcAccountRepository">
    <constructor-arg ref="dataSource" />
  </bean>

  <bean id="dataSource" class="com.oracle.jdbc.pool.OracleDataSource">
    <property name="URL" value="jdbc:oracle:thin:@localhost:1521:BANK" />
    <property name="user" value="moneytransfer-app" />
  </bean>

</beans>
```

“Spring Sucks!”

- Spring is XML
- XML is Evil
- Being Evil sucks

- Therefore, Spring sucks

Spring != XML

- @Component
- @Autowired
- JSR-250
 - @Resource
 - @PostConstruct/@PreDestroy
- @Configuration

@Component

@Autowired

@Component

```
public class TransferServiceImpl implements TransferService
```

@Autowired

```
public TransferServiceImpl(AccountRepository ar) {
```

```
    this.accountRepository = ar;
```

```
}
```

```
...
```

```
}
```

Why not use Java EE Dependency Injection?

- Testing, inside IDE
- Java EE only allows injection of JNDI-managed objects
- Spring injects everything
 - Primitives (configuration)

Reason #2

JdbcTemplate

JDBC

- Object/Relational Mapping is popular
- But JDBC continues to be important
 - Batch Operations
 - Set-based operations
 - Stored procedures

Redundant, Error Prone Code

```
public List findByLastName(String lastName) {
    List personList = new ArrayList();
    Connection conn = null;
    String sql = "select first_name, age from PERSON where last_name=?";
    try {
        DataSource dataSource = DataSourceUtils.getDataSource();
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, lastName);
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            String firstName = rs.getString("first_name");
            int age = rs.getInt("age");
            personList.add(new Person(firstName, lastName, age));
        }
    } catch (Exception e) { /* ??? */ }
    finally {
        try {
            conn.close();
        } catch (SQLException e) { /* ??? */ }
    }
    return personList;
}
```

Redundant, Error Prone Code

```
public List findByLastName(String lastName) {  
    List personList = new ArrayList();  
    Connection conn = null;  
    String sql = "select first_name, age from PERSON where last_name=?";  
    try {  
        DataSource dataSource = DataSourceUtils.getDataSource();  
        conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, lastName);  
        ResultSet rs = ps.executeQuery();  
        while (rs.next()) {  
            String firstName = rs.getString("first_name");  
            int age = rs.getInt("age");  
            personList.add(new Person(firstName, lastName, age));  
        }  
    } catch (Exception e) { /* ??? */ }  
    finally {  
        try {  
            conn.close();  
        } catch (SQLException e) { /* ??? */ }  
    }  
    return personList;  
}
```

Redundant, Error Prone Code

```
public List findByLastName(String lastName) {  
    List personList = new ArrayList();  
    Connection conn = null;  
    String sql = "select first_name, age from PERSON where last_name=?";  
    try {  
        DataSource dataSource = DataSourceUtils.getDataSource();  
        conn = dataSource.getConnection();  
        PreparedStatement ps = conn.prepareStatement(sql);  
        ps.setString(1, lastName);  
        ResultSet rs = ps.executeQuery();  
        while (rs.next()) {  
            String firstName = rs.getString("first_name");  
            int age = rs.getInt("age");  
            personList.add(new Person(firstName, lastName, age));  
        }  
    } catch (Exception e) { /* ??? */ }  
    finally {  
        try {  
            conn.close();  
        } catch (SQLException e) { /* ??? */ }  
    }  
    return personList;  
}
```

The bold matters - the rest
is boilerplate

JdbcTemplate

```
int count = jdbcTemplate.queryForInt(  
    "SELECT COUNT(*) FROM CUSTOMER");
```

JdbcTemplate

```
int count = jdbcTemplate.queryForInt(  
    "SELECT COUNT(*) FROM CUSTOMER");
```

- Acquisition of the connection
- Participation in the transaction
- Execution of the statement
- Processing of the result set
- Handling any exceptions
- Release of the connection

All handled
by Spring

Querying With SimpleJdbcTemplate

```
public int getCountOfPersonsOlderThan(int age) {  
    return jdbcTemplate().queryForInt(  
        "select count(*) from PERSON where age > ?", age);  
}
```

Domain Objects

```
public Person getPerson(int id) {  
    return getSimpleJdbcTemplate().queryForObject(  
        "select first_name, last_name from PERSON where id=?",  
        new PersonMapper(), id);  
}
```

Domain Objects

```
public Person getPerson(int id) {  
    return getSimpleJdbcTemplate().queryForObject(  
        "select first_name, last_name from PERSON where id=?",  
        new PersonMapper(), id);  
}
```

```
class PersonMapper implements ParameterizedRowMapper<Person> {  
    public Person mapRow(ResultSet rs, int i) {  
        return new Person(rs.getString(1), rs.getString(2));  
    }  
}
```

Reason #3

Exception Hierarchy

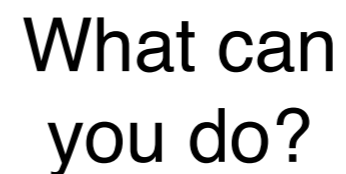
Poor Exception Handling

```
public List findByLastName(String lastName) {
    List personList = new ArrayList();
    Connection conn = null;
    String sql = "select first_name, age from PERSON where last_name=?";
    try {
        DataSource dataSource = DataSourceUtils.getDataSource();
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, lastName);
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            String firstName = rs.getString("first_name");
            int age = rs.getInt("age");
            personList.add(new Person(firstName, lastName, age));
        }
    } catch (Exception e) { /* ??? */ }
    finally {
        try {
            conn.close();
        } catch (SQLException e) { /* ??? */ }
    }
    return personList;
}
```

Poor Exception Handling

```
public List findByLastName(String lastName) {
    List personList = new ArrayList();
    Connection conn = null;
    String sql = "select first_name, age from PERSON where last_name=?";
    try {
        DataSource dataSource = DataSourceUtils.getDataSource();
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, lastName);
        ResultSet rs = ps.executeQuery();
        while (rs.next()) {
            String firstName = rs.getString("first_name");
            int age = rs.getInt("age");
            personList.add(new Person(firstName, lastName, age));
        }
    } catch (Exception e) { /* ??? */ }
    finally {
        try {
            conn.close();
        } catch (SQLException e) { /* ??? */ }
    }
    return personList;
}
```

What can
you do?



What can you do?

- Nothing
- Logging
- Wrapping
- Retry

Dead Lock Loser

```
} catch (SQLException ex) {  
    if (ex.getErrorCode() == 60) {// check for ORA-00060  
        return findByLastName(lastName);  
    } else {  
        throw ex;  
    }  
}
```

Retry

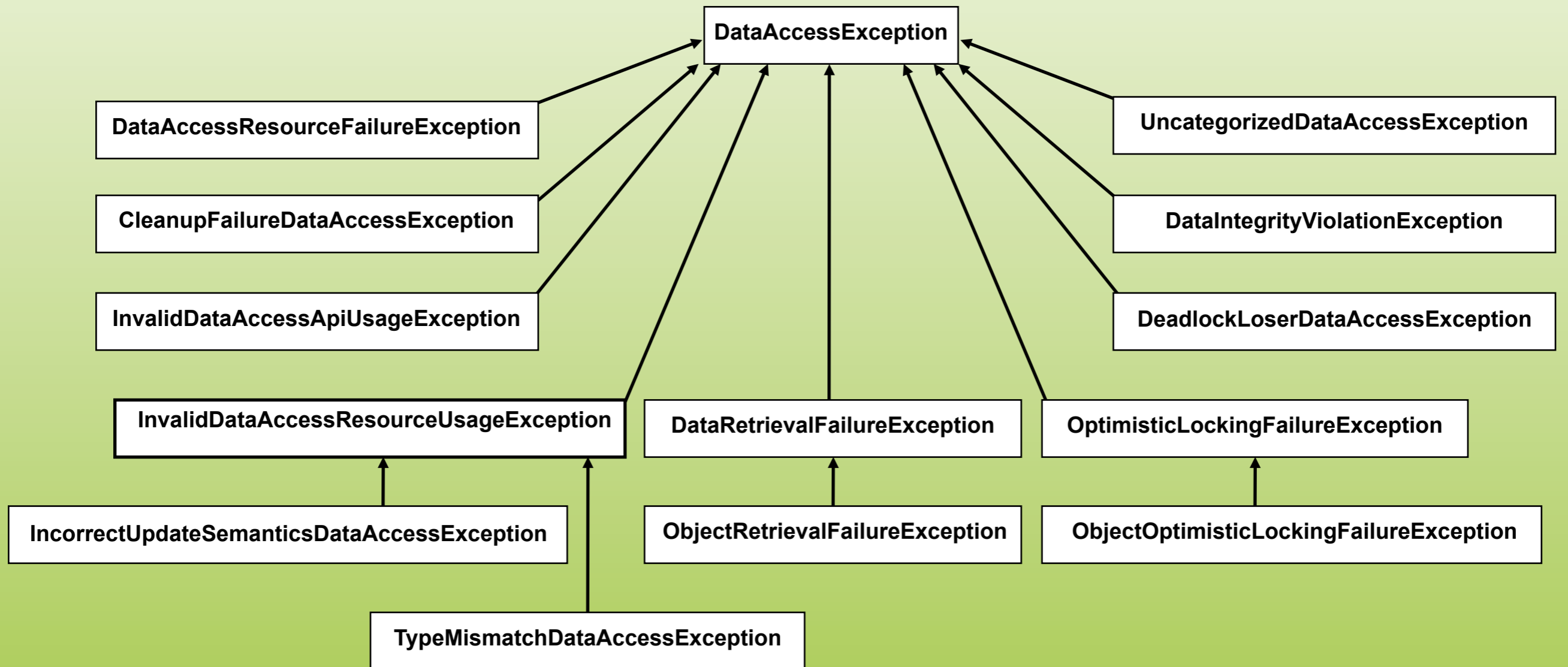
- Ugly code
- Not portable
 - JDBC vs JPA
 - Oracle vs SQL Server

In Spring

- Spring does exception translation
- Into a rich Exception Hierarchy
 - Catch DeadLockLoserException
 - Use AOP!

DataAccessException

Hierarchy (subset)



Reason #4

Aspect-Oriented Programming

What Problem Does AOP Solve?

Aspect-Oriented Programming (AOP)
enables modularization of cross-cutting
concerns

What are Cross-Cutting Concerns?

Generic functionality that is needed in many places in your application

- Logging and Tracing
- Transaction Management
- Security
- Caching
- Error Handling
- Performance Monitoring
- Custom Business Rules

An Example Requirement

- Perform a role-based security check before every application method

An Example Requirement

- Perform a role-based security check before every application method

A sign this requirement is a cross-cutting concern

Without AOP

- Failing to modularize cross-cutting concerns leads to two things
 - Code tangling
 - A coupling of concerns
 - Code scattering
 - The same concern spread across modules

Tangling

```
public class RewardNetworkImpl implements RewardNetwork {
    public RewardConfirmation rewardAccountFor(Dining dining) {
        if (!hasPermission(SecurityContext.getPrincipal())) {
            throw new AccessDeniedException();
        }
        Account a = accountRepository.findByCreditCard(...)
        Restaurant r = restaurantRepository.findByMerchantNumber(...)
        MonetaryAmount amt = r.calculateBenefitFor(account, dining);
        ...
    }
}
```

Tangling

```
public class RewardNetworkImpl implements RewardNetwork {  
    public RewardConfirmation rewardAccountFor(Dining dining) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        Account a = accountRepository.findByCreditCard(...  
        Restaurant r = restaurantRepository.findByMerchantNumber(...  
        MonetaryAmount amt = r.calculateBenefitFor(account, dining);  
        ...  
    }  
}
```

Mixing of concerns

Scattering

```
public class HibernateAccountManager implements AccountManager {  
    public Account getAccountForEditing(Long id) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }  
}
```

```
public class HibernateMerchantReportingService implements  
MerchantReportingService {  
    public List<DiningSummary> findDinings(String merchantNumber,  
                                           DateInterval interval) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
        ...  
    }  
}
```

Scattering

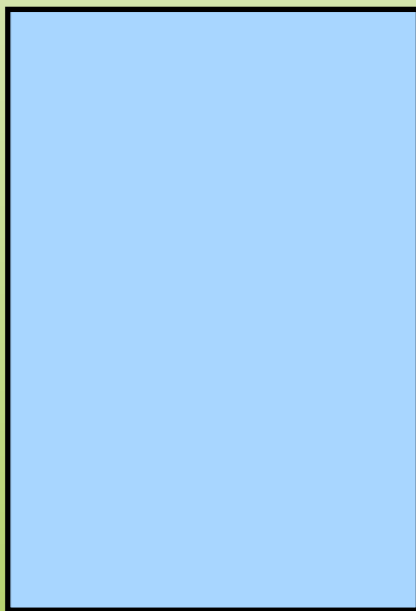
```
public class HibernateAccountManager implements AccountManager {  
    public Account getAccountForEditing(Long id) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
    }  
    ...  
}
```

Duplication



```
public class HibernateMerchantReportingService implements  
MerchantReportingService {  
    public List<DiningSummary> findDinings(String merchantNumber,  
                                           DateInterval interval) {  
        if (!hasPermission(SecurityContext.getPrincipal())) {  
            throw new AccessDeniedException();  
        }  
    }  
    ...  
}
```

Without AOP



BankService




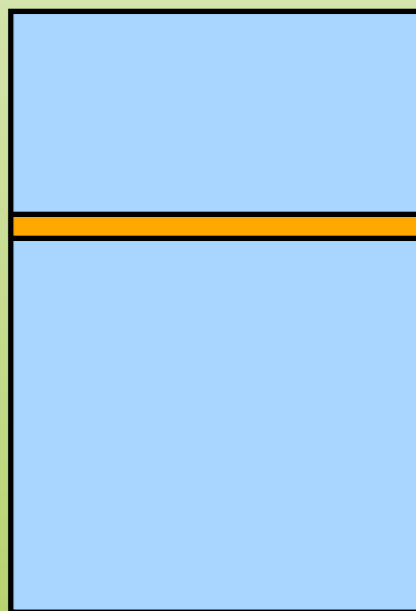
CustomerService



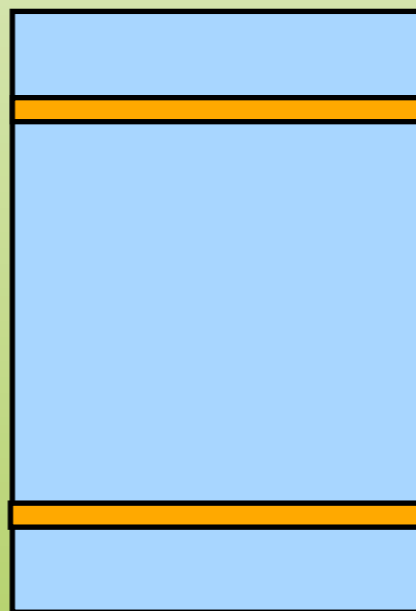
ReportingService

Without AOP

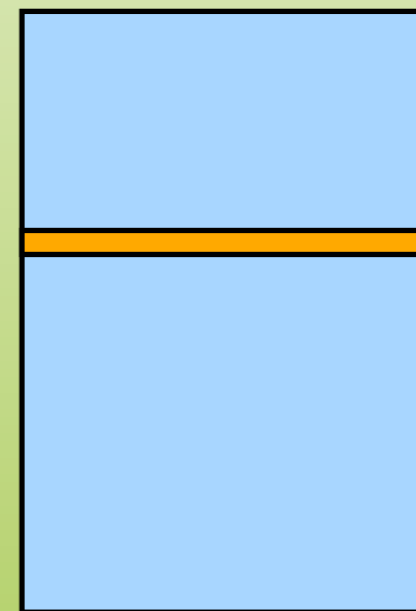
 Security



BankService

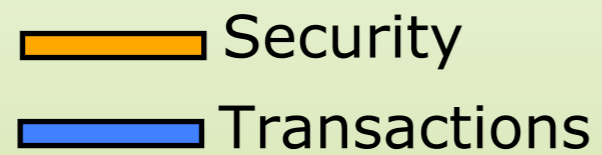


CustomerService



ReportingService

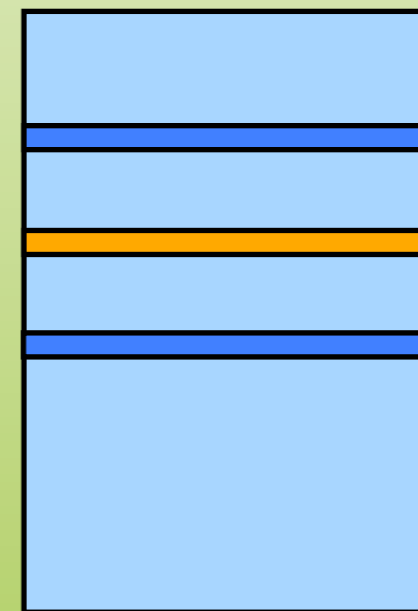
Without AOP



BankService

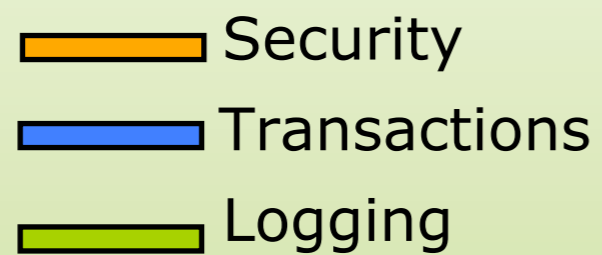


CustomerService



ReportingService

Without AOP



BankService

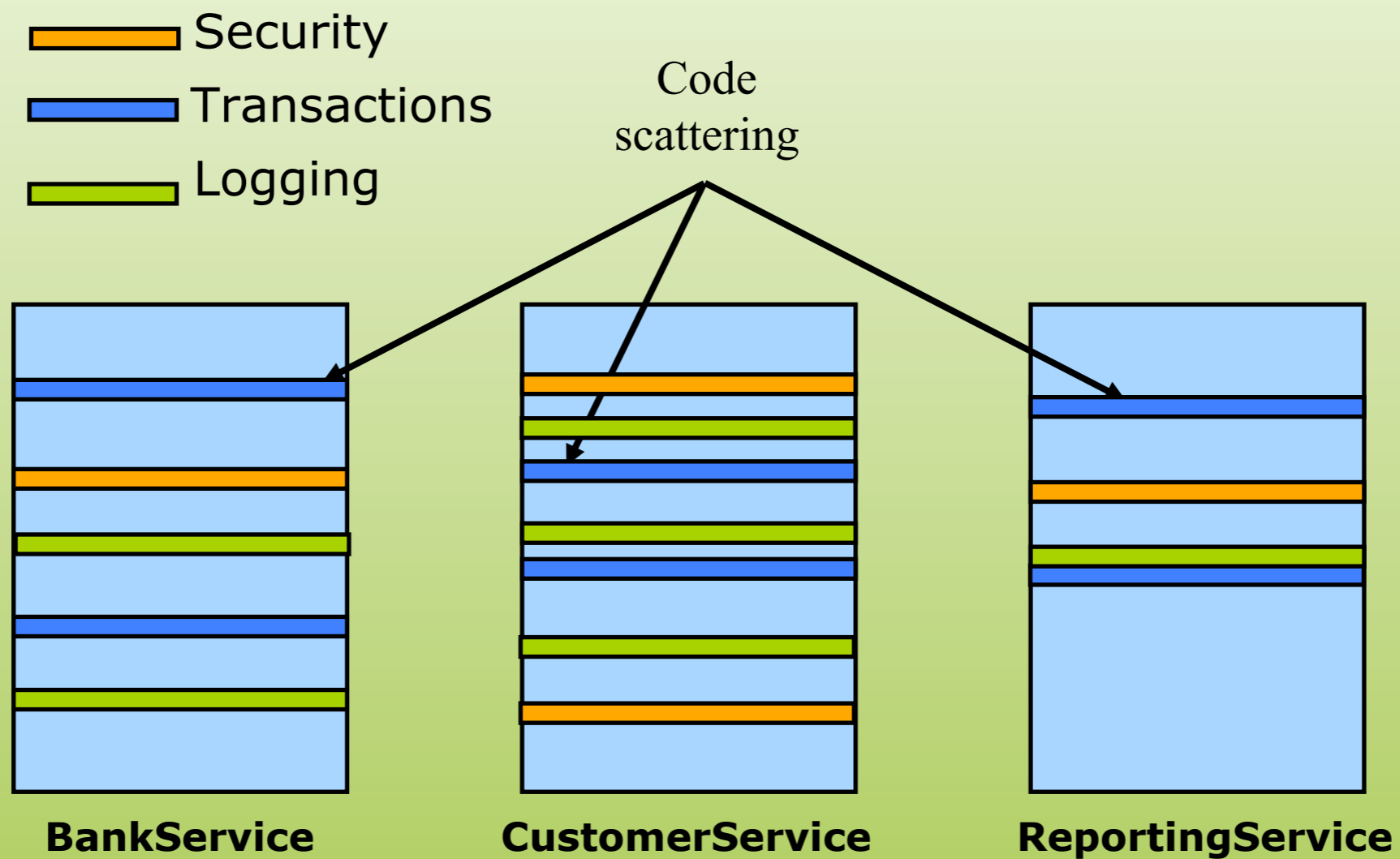


CustomerService

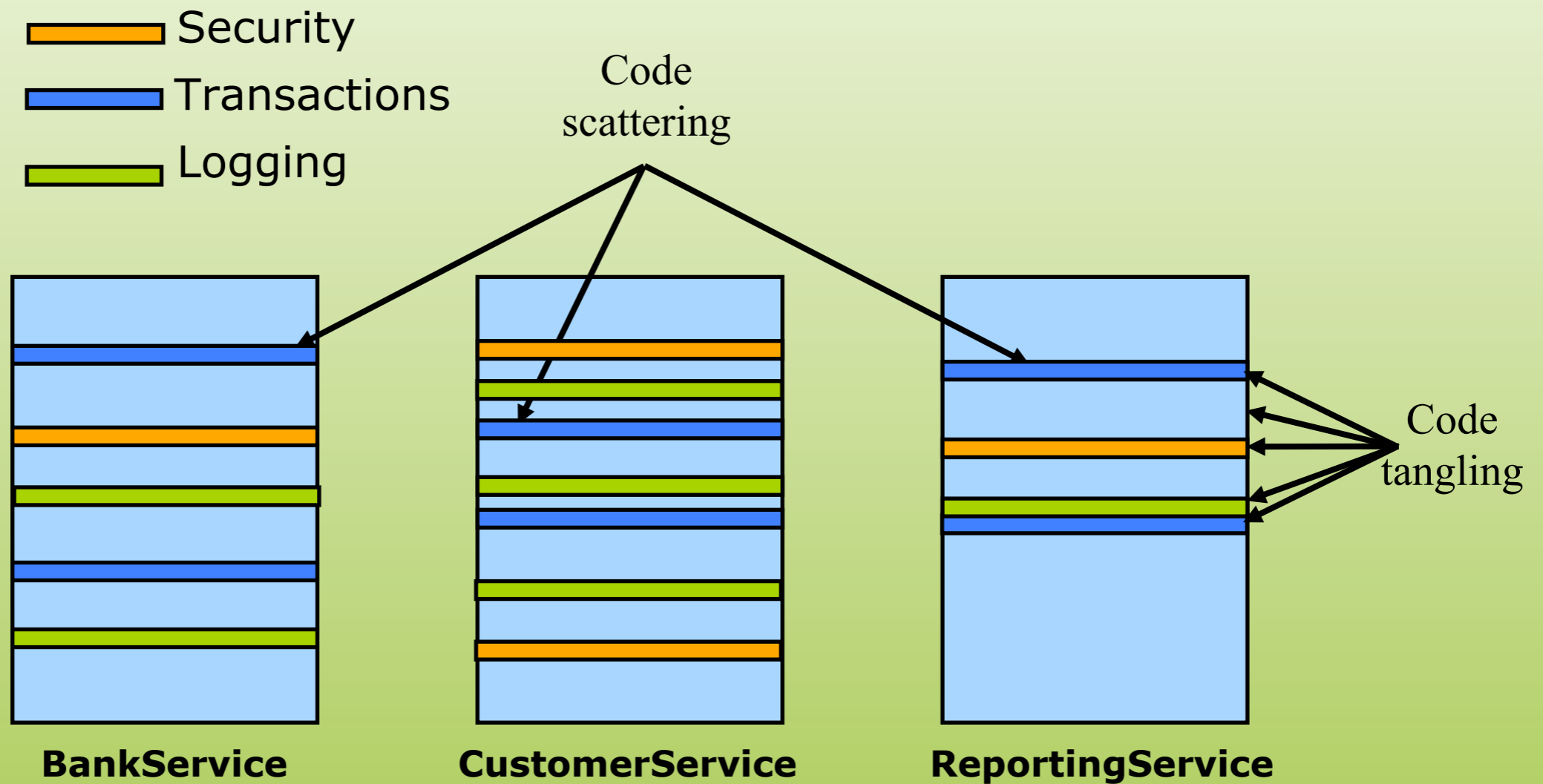


ReportingService

Without AOP



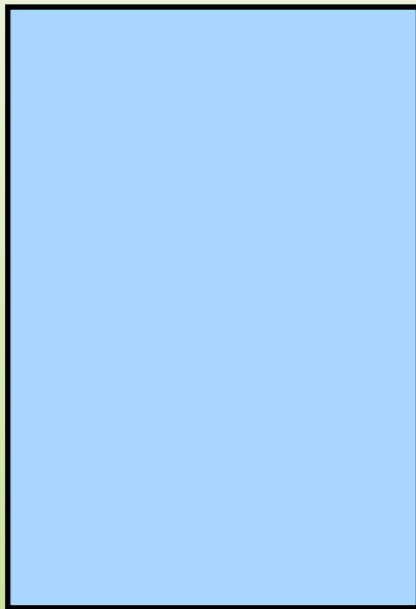
Without AOP



How AOP Works

1. Implement your mainline application logic
2. Write aspects to implement your cross-cutting concerns
3. Weave the aspects into your application

AOP based



BankService

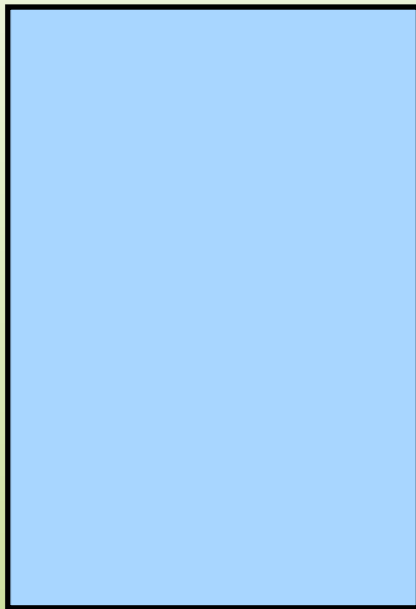


CustomerService



ReportingService

AOP based



BankService



CustomerService

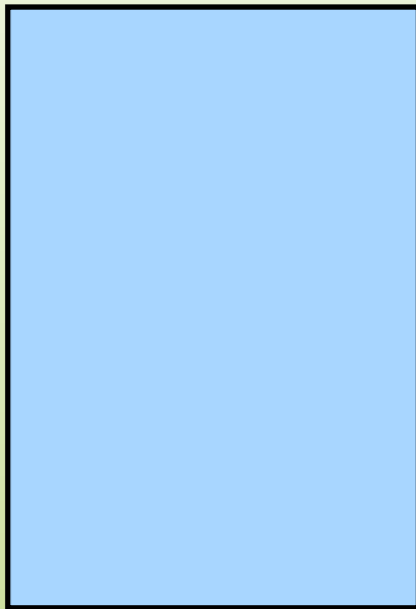


ReportingService



Security
Aspect

AOP based



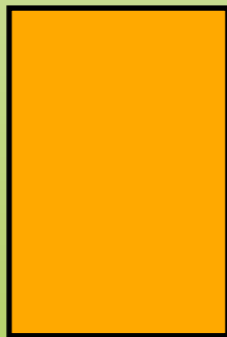
BankService



CustomerService



ReportingService

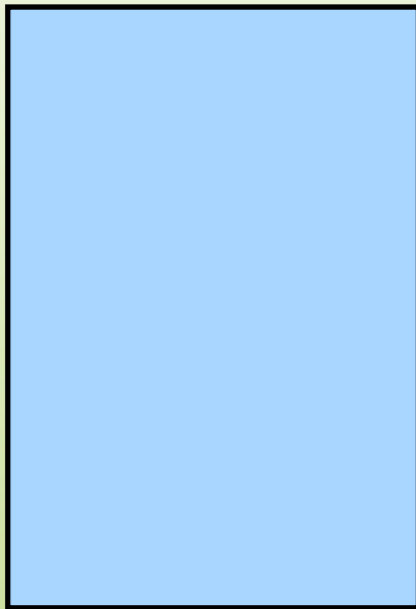


Security
Aspect



Transaction
Aspect

AOP based



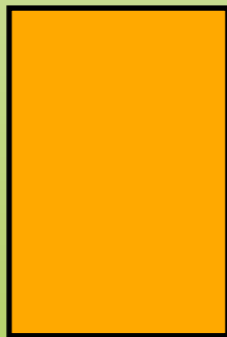
BankService



CustomerService



ReportingService



Security
Aspect

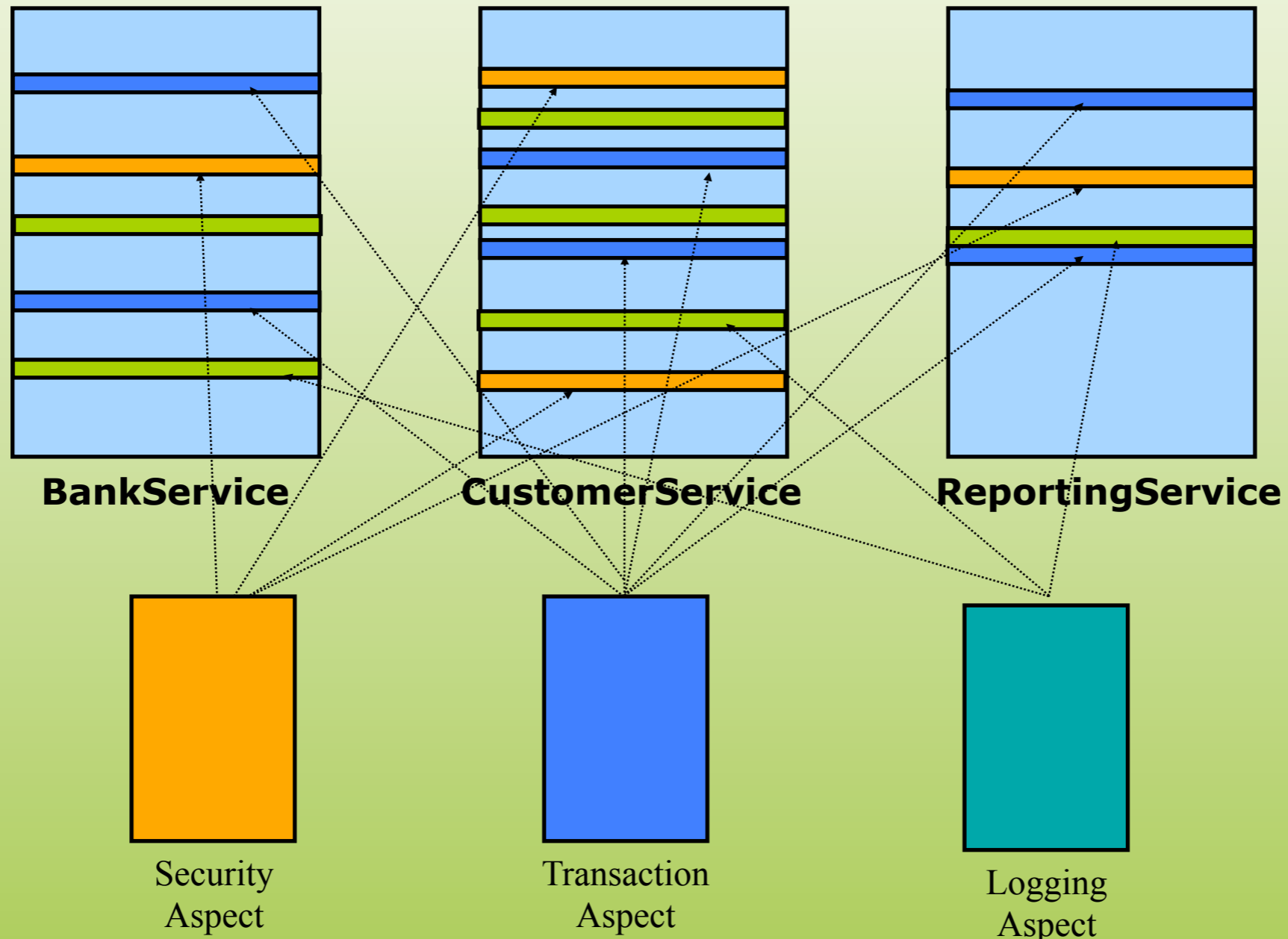


Transaction
Aspect



Logging
Aspect

AOP based



AOP Quick Start

- Consider this basic requirement

Log a message every time a property is about to change

- How can you use AOP to meet it?

Target Object

```
public class SimpleCache implements Cache {  
    private int cacheSize;  
    private DataSource dataSource;  
  
    public void setCacheSize(int size) {  
        cacheSize = size;  
    }  
  
    public void setDataSource(DataSource ds) {  
        dataSource = ds;  
    }  
    ...  
}
```

The Aspect

The Aspect

@Aspect

The Aspect

```
@Aspect
```

```
public class PropertyChangeTracker {
```

The Aspect

@Aspect

```
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());
```

The Aspect

@Aspect

```
public class PropertyChangeTracker {  
    private Logger logger = Logger.getLogger(getClass());
```

The Aspect

```
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
```

The Aspect

```
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange() {
```

The Aspect

```
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange() {
        logger.info("Property about to change...");
    }
}
```

The Aspect

```
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange() {
        logger.info("Property about to change...");
    }
}
```

The Aspect

```
@Aspect
public class PropertyChangeTracker {
    private Logger logger = Logger.getLogger(getClass());

    @Before("execution(void set*(*))")
    public void trackChange() {
        logger.info("Property about to change...");
    }
}
```

Tell Spring about the Aspect

```
<beans>  
  
  <aop:aspectj-autoproxy/>  
  
  <bean id="propertyChangeTracker" class="example.PropertyChangeTracker" />  
  
  <bean name="cache-A" class="example.SimpleCache" ../>  
  <bean name="cache-B" class="example.SimpleCache" ../>  
  <bean name="cache-C" class="example.SimpleCache" ../>  
  
</beans>
```

Run...

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("application-config.xml");  
Cache cache = (Cache) context.getBean("cache-A");  
cache.setCacheSize(2500);
```

Run...

```
ApplicationContext context =  
    new ClassPathXmlApplicationContext("application-config.xml");  
Cache cache = (Cache) context.getBean("cache-A");  
cache.setCacheSize(2500);
```

INFO: Property about to change...

Reason #5 @Transactional

Why use Transactions?

- Atomic
 - Each unit of work is an all-or-nothing operation
- Consistent
 - Database integrity constraints are never violated
- Isolated
 - Uncommitted changes are not visible to other transactions
- Durable
 - Committed changes are permanent

Local Transaction Management

- Transactions can be managed at the level of a local resource
 - Such as the database
- Requires programmatic management of transactional behavior on the Connection

Example

```
public void updateBeneficiaries(Account account) {  
    ...  
    try {  
        conn = dataSource.getConnection();  
        conn.setAutoCommit(false);  
        ps = conn.prepareStatement(sql);  
        for (Beneficiary b : account.getBeneficiaries()) {  
            ps.setBigDecimal(1, b.getSavings().asBigDecimal());  
            ps.setLong(2, account.getEntityId());  
            ps.setString(3, b.getName());  
            ps.executeUpdate();  
        }  
        conn.commit();  
    } catch (Exception e) {  
        conn.rollback();  
        throw new RuntimeException("Error updating!", e);  
    }  
}
```

Problems with Local Transactions

- Connection management code is error-prone
- Transaction demarcation belongs at the service layer
 - Multiple data access methods may be called within a transaction
 - Connection must be managed at a higher level

Passing Connections

```
public RewardConfirmation rewardAccountFor(Dining dining) {  
    Connection conn = DataSourceUtils.getConnection();  
    conn.setAutoCommit(false);  
    try {  
        ...  
        accountRepository.updateBeneficiaries(account, conn);  
        rc = rewardRepository.confirmReward(contrib, dining, conn);  
        conn.commit();  
    }  
    catch (Exception e) {  
        conn.rollback();  
        throw new RuntimeException("reward failed", e);  
    }  
}
```

Programmatic JTA

- Application Servers enable use of the Java Transaction API (JTA)
- The UserTransaction object is bound to JNDI
- Transactions can be managed from the service layer
- May call multiple data access methods

JTA

```
public RewardConfirmation rewardAccountFor(Dining dining) {
    Context ctx = new InitialContext();
    UserTransaction transaction = (UserTransaction)
        ctx.lookup("java:comp/UserTransaction");

    transaction.begin();
    try {
        ...
        accountRepository.updateBeneficiaries(account);
        confirmation = rewardRepository.confirmReward(contribution, dining);
        transaction.commit();
    }
    catch (Exception e) {
        transaction.rollback();
        throw new RuntimeException("failed to reward", e);
    }
}
```

Programmatic JTA Problems

- Depends on an Application Server environment
- Still requires code to manage transactions
- The code is error-prone

@Transactional

```
public class RewardNetworkImpl implements RewardNetwork {  
    @Transactional  
    public RewardConfirmation rewardAccountFor(Dining d) {  
        // atomic unit-of-work  
    }  
}
```

@Transactional

- Works though AOP
- Consistent approach
 - JDBC
 - JPA
 - Hibernate
 - JMS
 - JTA
 - ...

Reason #6

Scripting Languages

Scripting Languages

- More and more popular
- Especially when running on the JVM
- Mix-and-match approach
 - Front-end in JRuby
 - Back-end in Java

Dynamic Language Support in Spring

- Spring container supports
 - Groovy
 - JRuby
 - BeanShell

JRuby

```
package org.springframework.scripting;  
  
public interface Messenger {  
  
    String getMessage();  
  
}
```

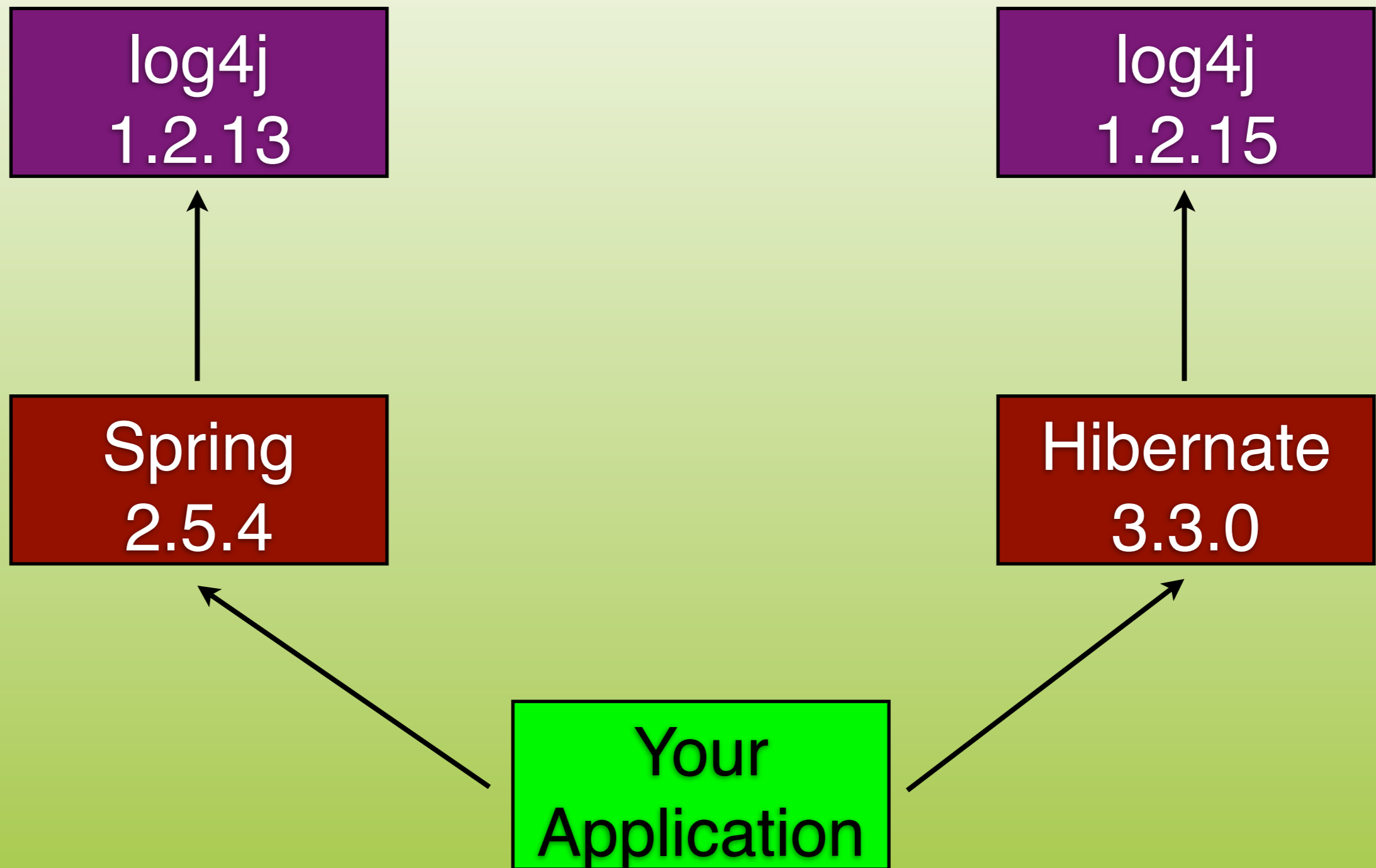
```
require 'java'  
  
class RubyMessenger  
    include org.springframework.scripting.Messenger  
  
    def setMessage(message)  
        @@message = message  
    end  
  
    def getMessage  
        @@message  
    end  
  
end
```

```
<lang:jruby id="messageService"  
    script-interfaces="org.springframework.scripting.Messenger"  
    script-source="classpath:RubyMessenger.rb">  
  
    <lang:property name="message" value="Hello World!" />  
  
</lang:jruby>
```

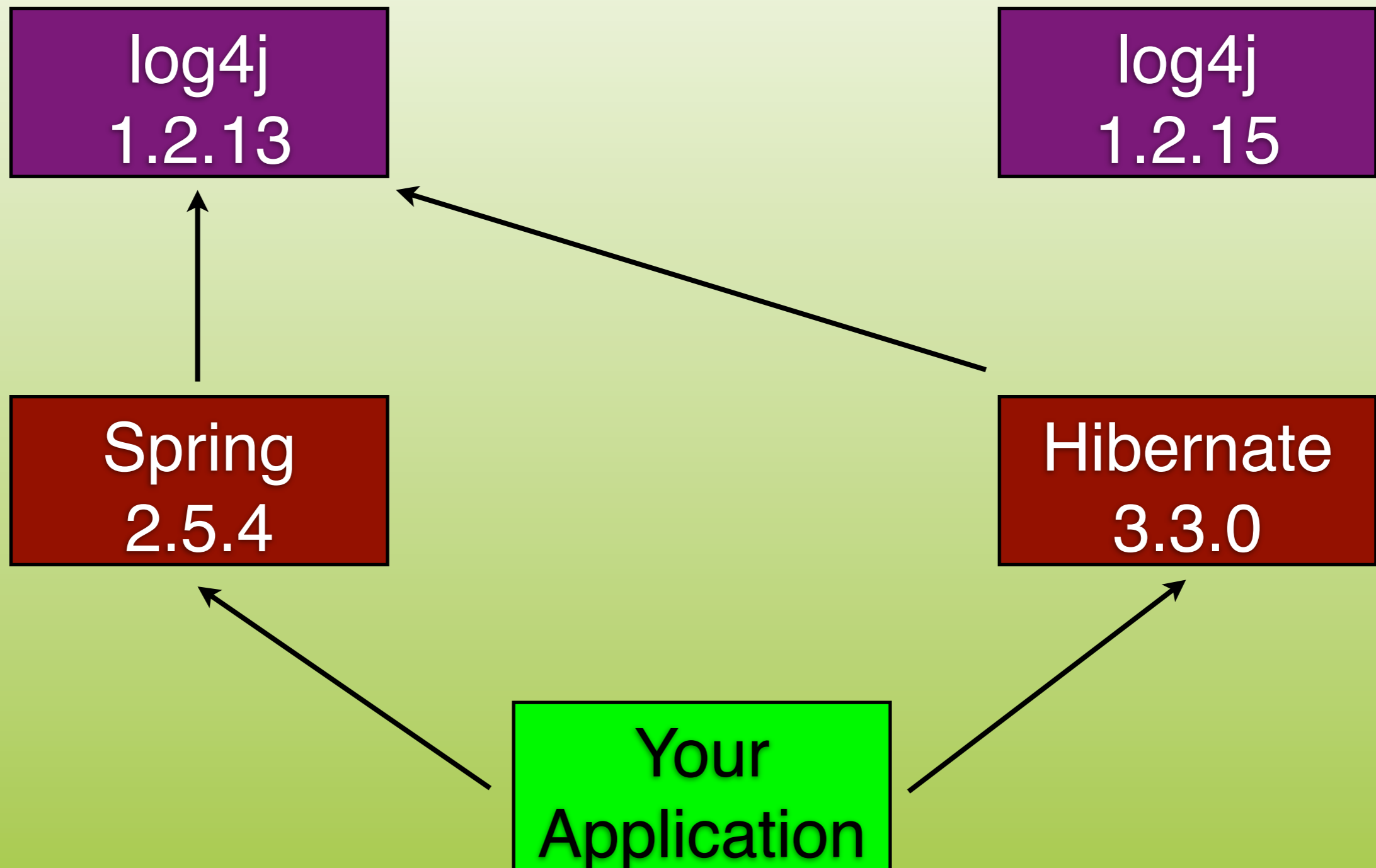
Reason #7

OSGi

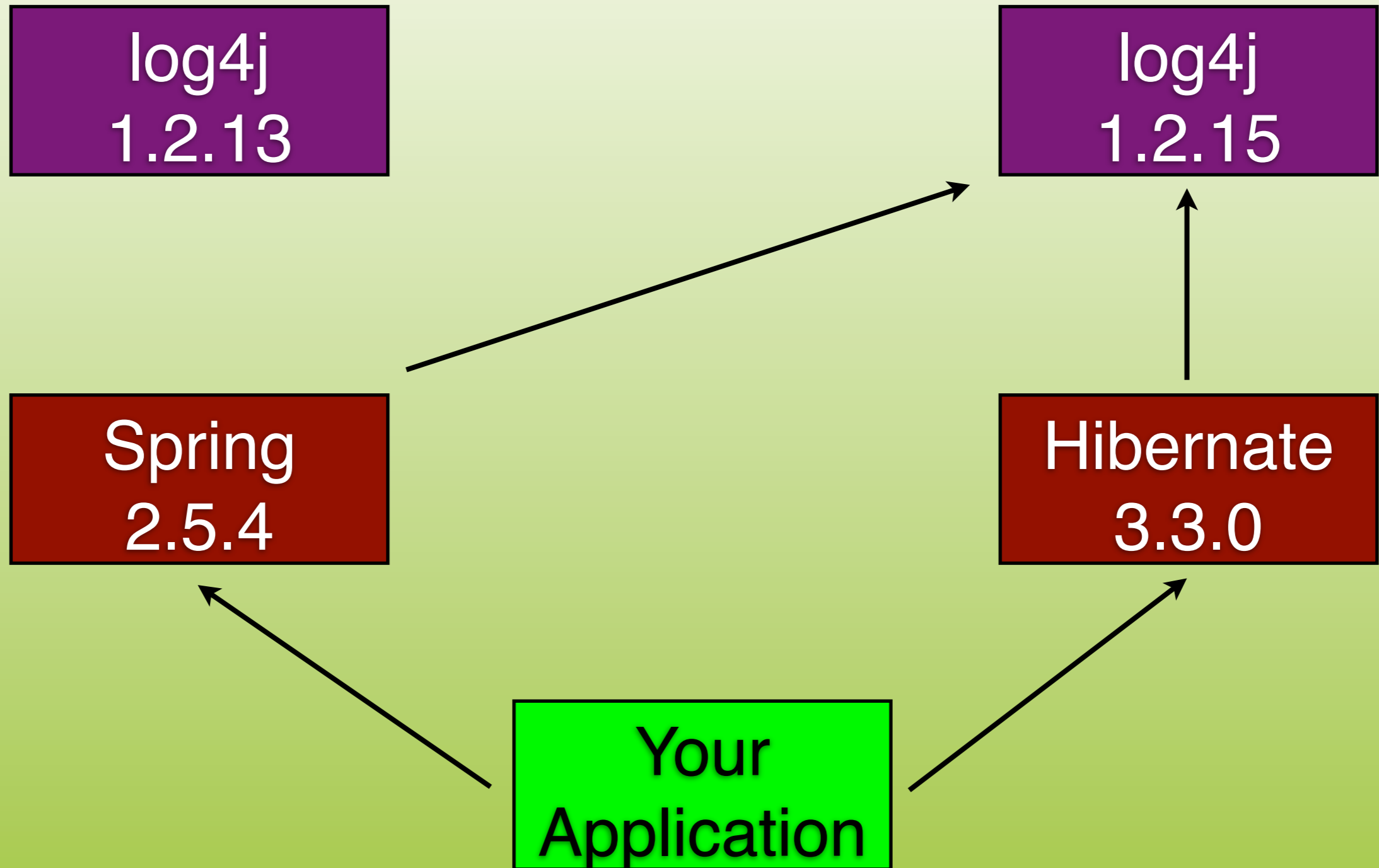
JAR Hell



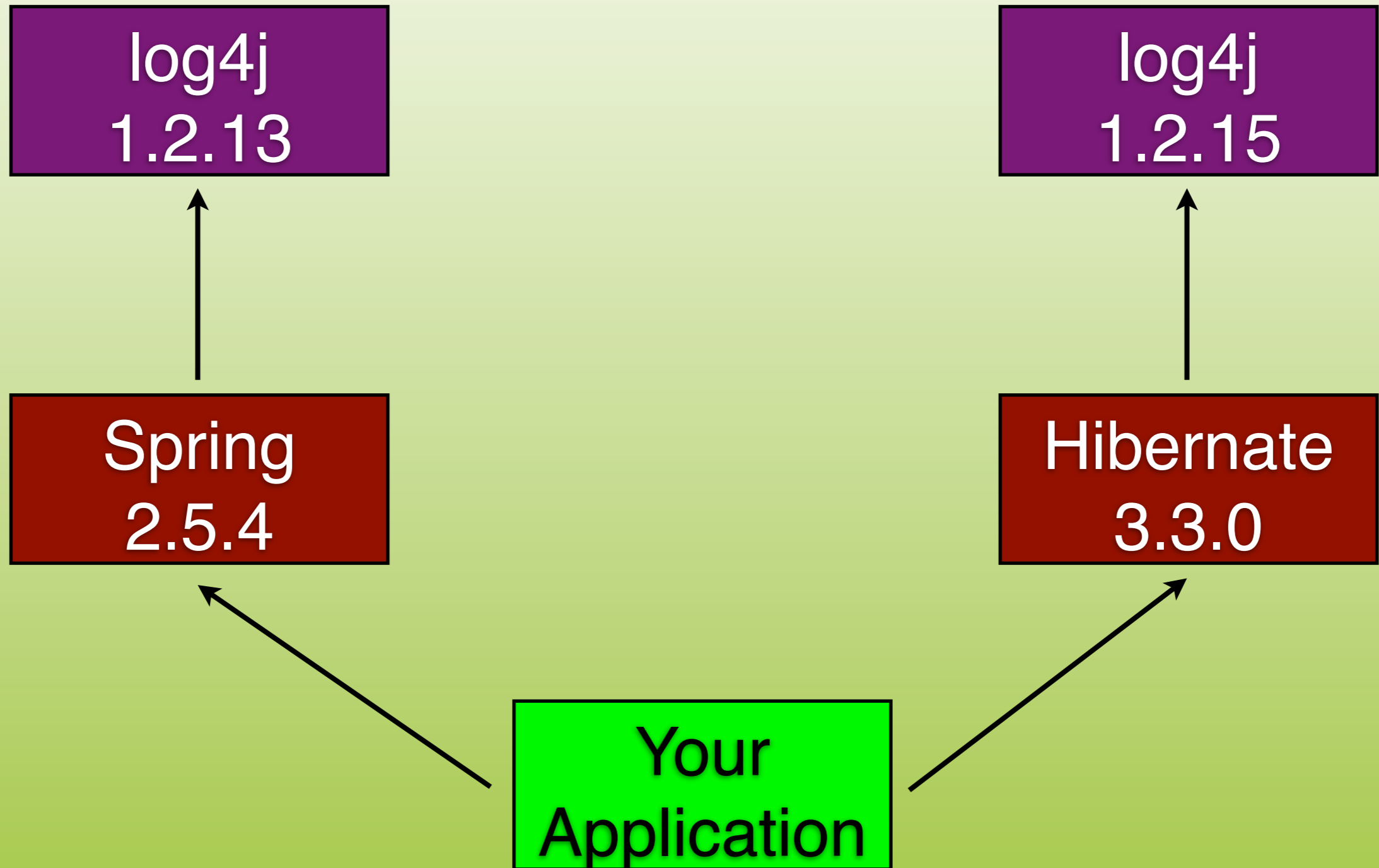
JAR Hell



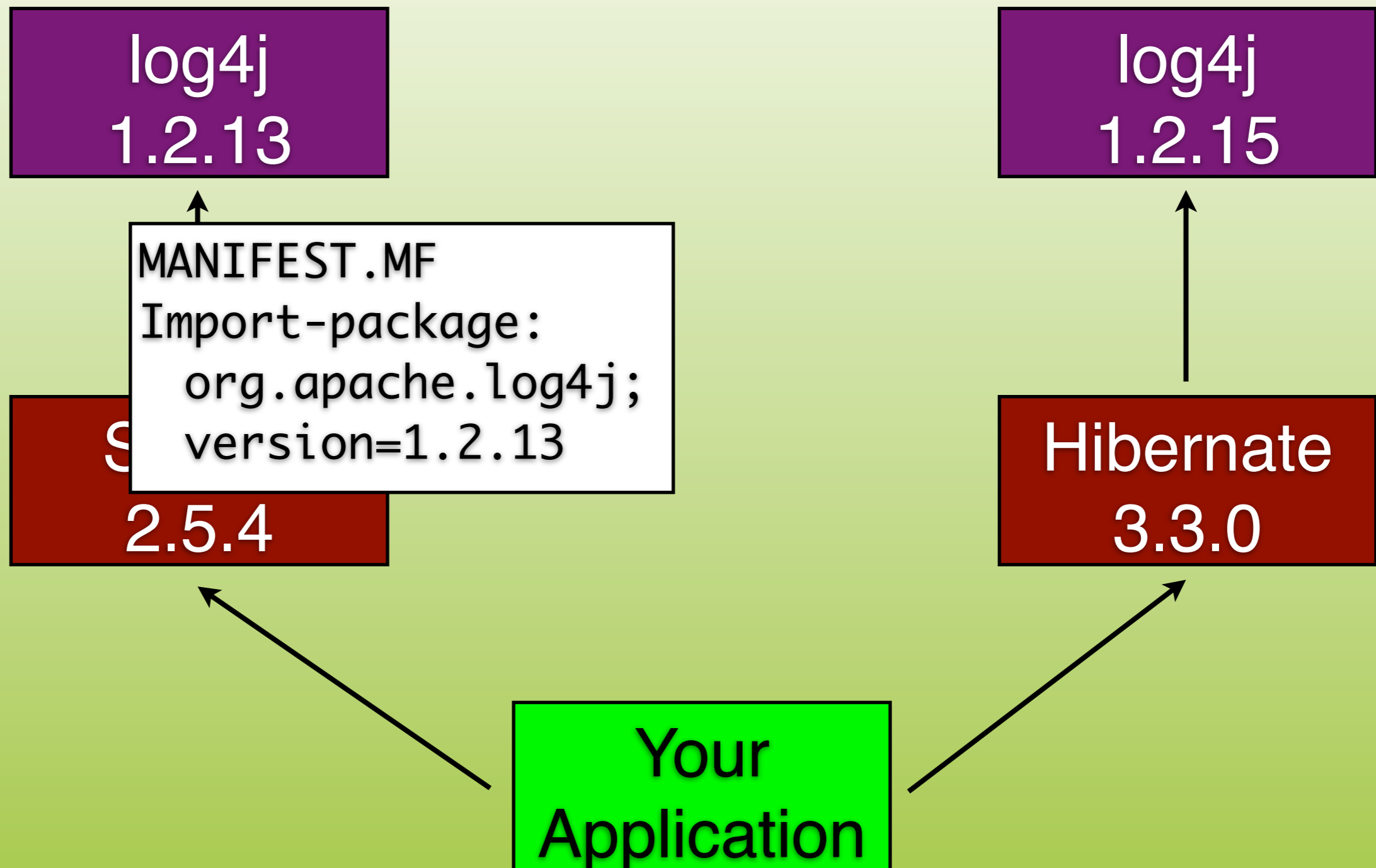
JAR Hell



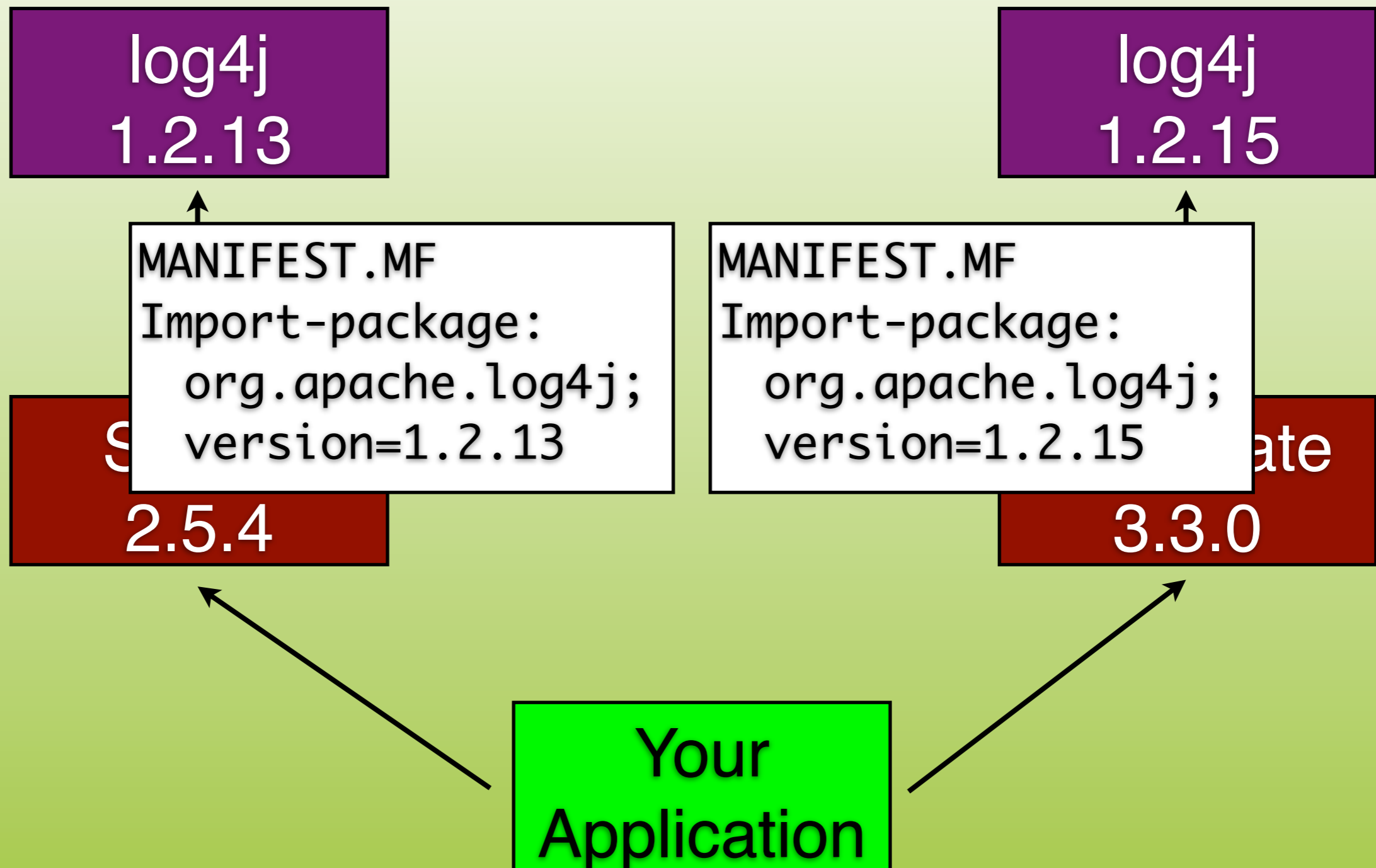
OSGi



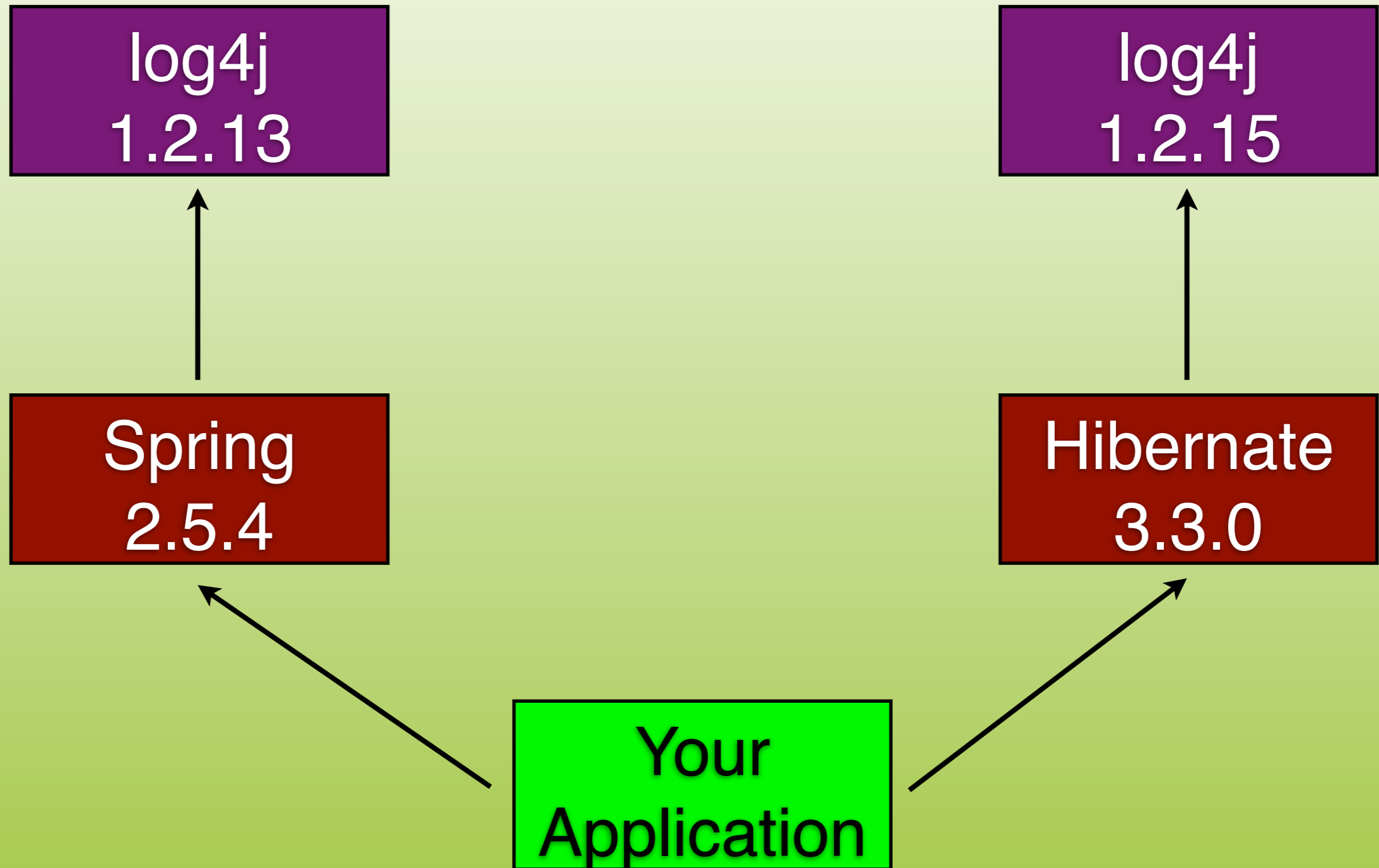
OSGi



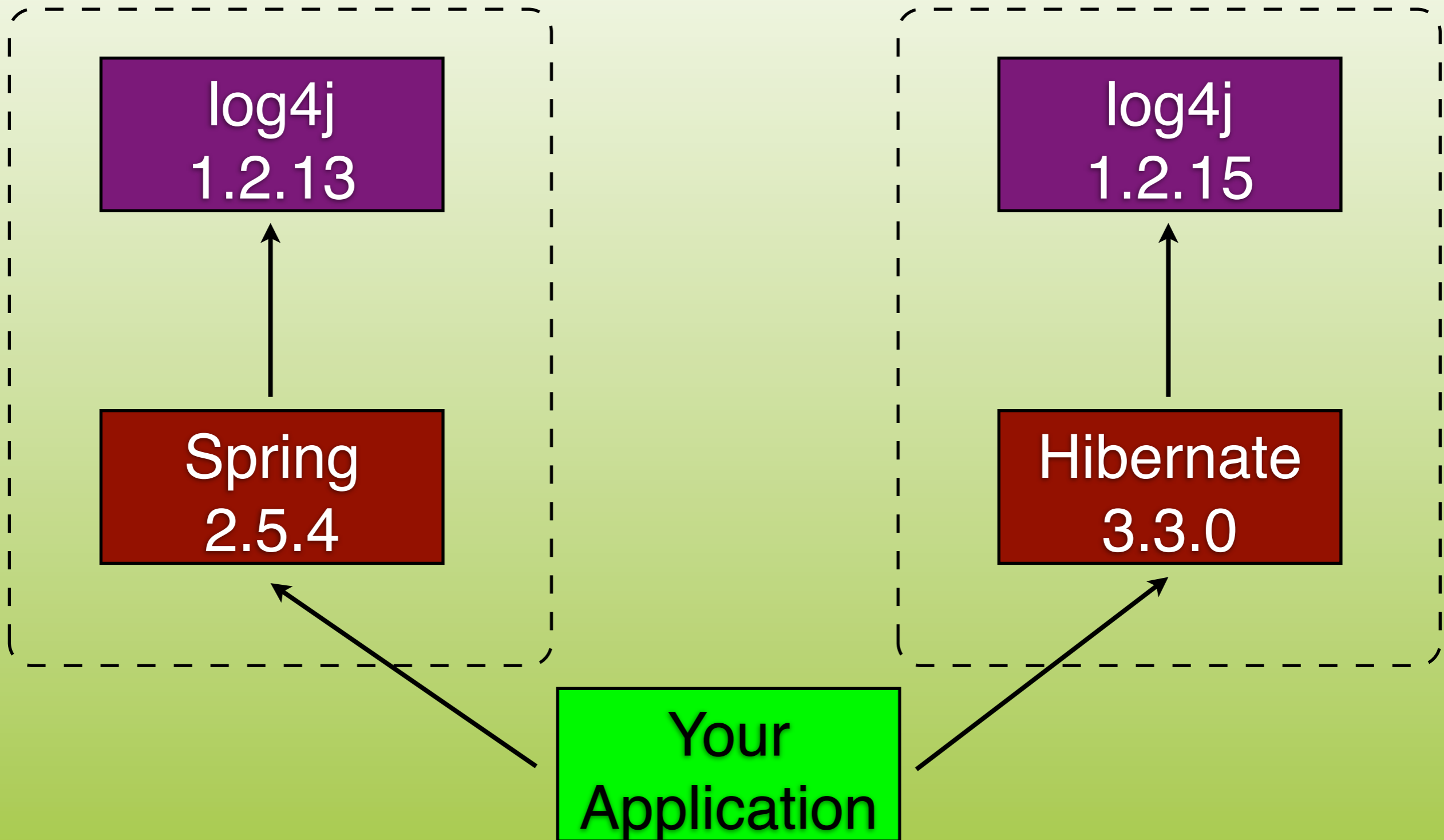
OSGi



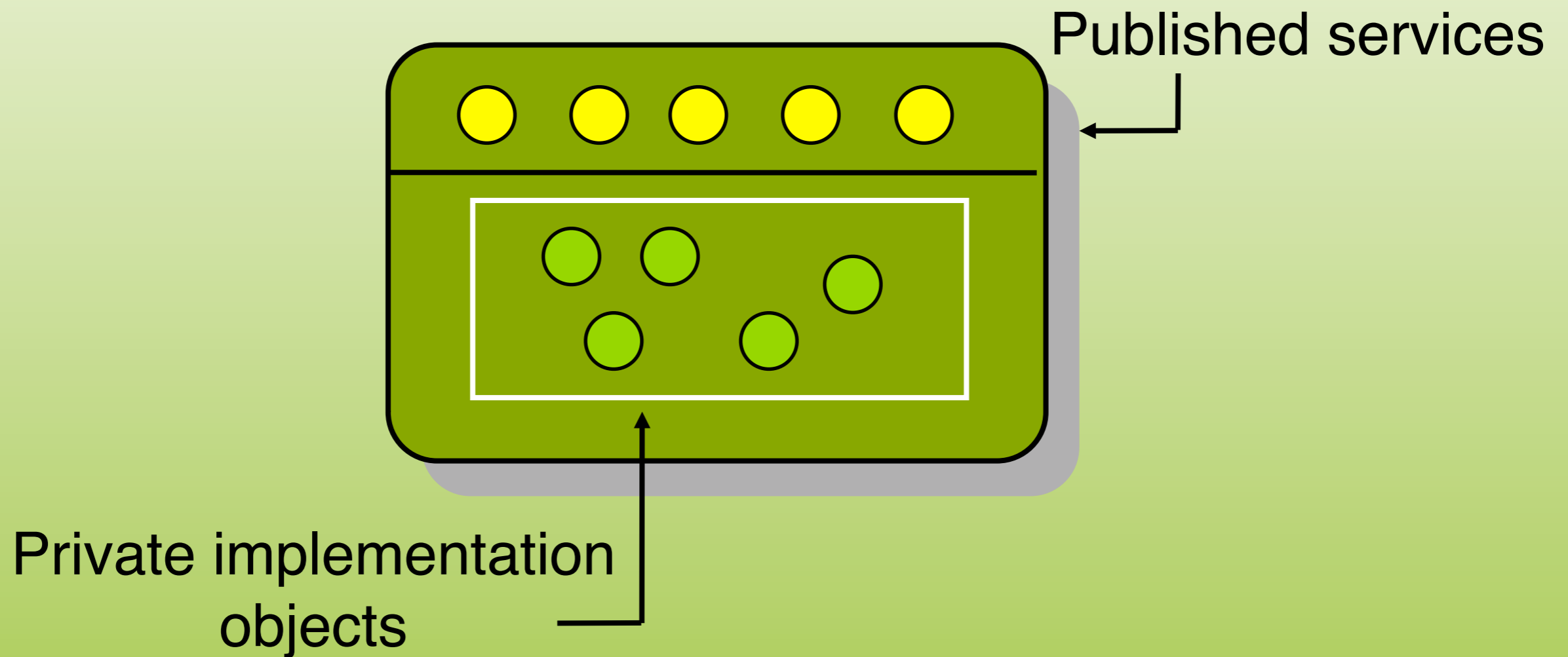
OSGi



OSGi



OSGi Service contribution



OSGi™ – Dynamic System for Java

- Partition an app into modules
 - Module = Bundle = Jar = a set of classes
- Each module has its own:
 - class space
 - lifecycle
- Strict visibility rules
- Understands versioning
- Everything is *Dynamic!*

OSGi

- Import specific versions of dependencies
- Export as services

Spring Dynamic Modules

- Decorate objects as OSGi services
- Dependency injection for OSGi services
- Remove OSGi dependencies
- Easier to test

Exporting OSGi Service

```
<bean id="myPojo" class="someObject" />
```

```
<osgi:service
```

```
  ref="myPojo"
```

```
  interface="com.springsource.MyService" />
```

Importing OSGi service

```
<osgi:reference  
    id="osgiService"  
    interface="com.springframework.DynamicService"/>  
  
<bean id="consumer" class..>  
    <property name="service" ref="osgiService">  
</bean>
```

But wait, there is more!

- There is more!
- JMS, Web, Security, Web Services, Batch, Integration, ...
- Check out www.springframework.org

Q & A