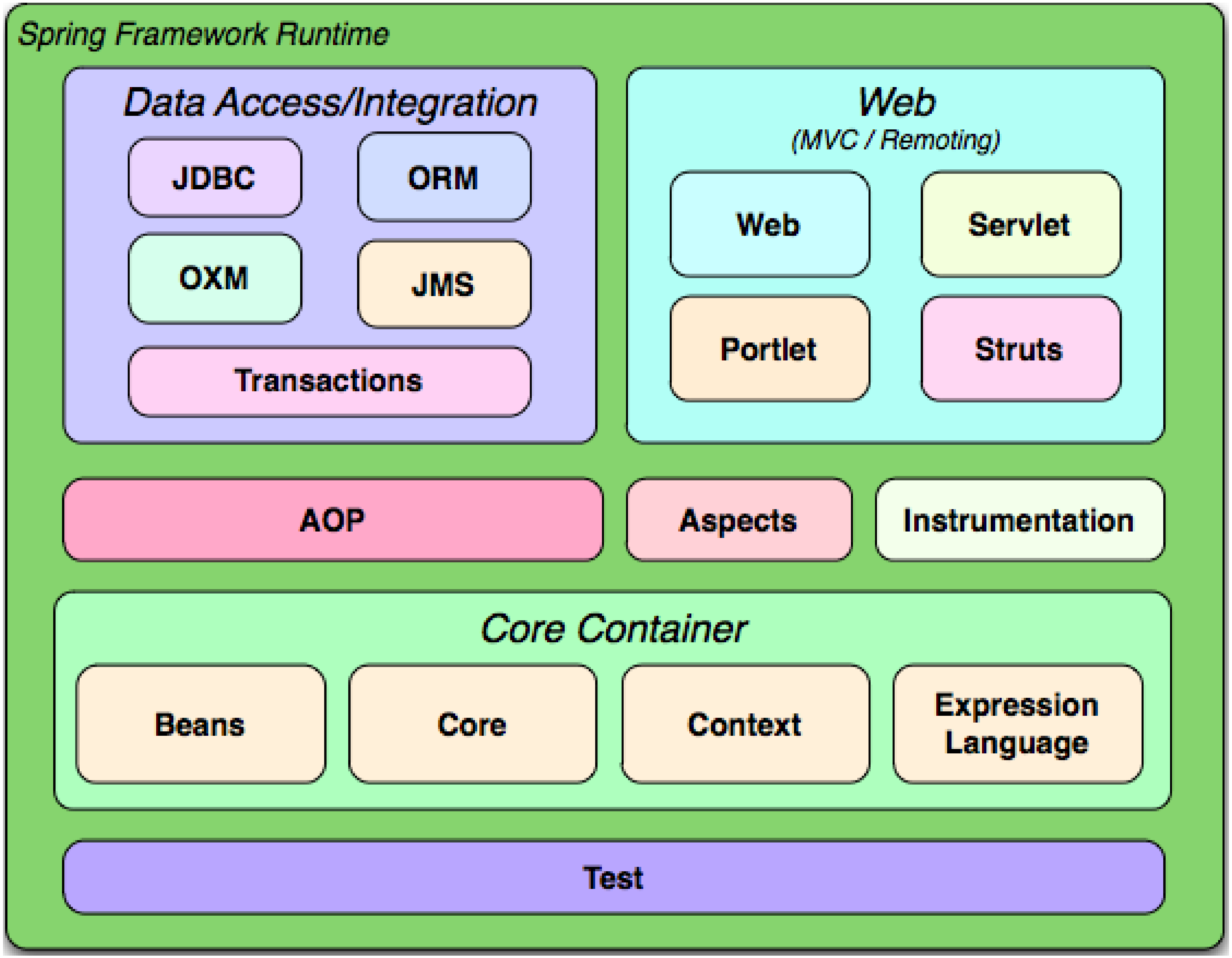
Spring Framework 4.x

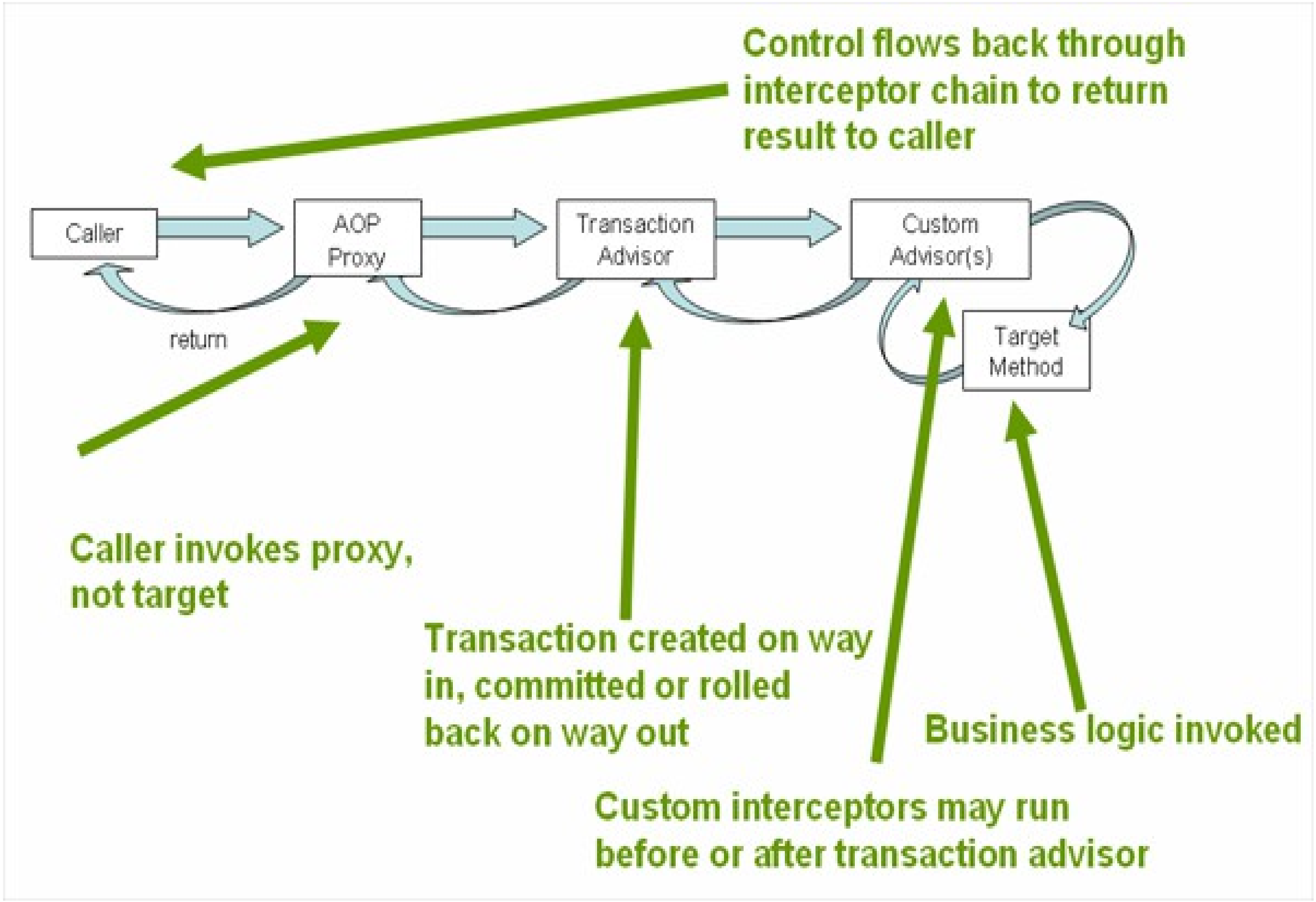
JTA: Transactions

David Lucas
Lucas Software Engineering, Inc.
www.lse.com
ddlucas@lse.com


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
(MVC / Remoting)
Web
Servlet
Portlet
Struts
AOP
Aspects
Instrumentation
Core Container
Beans
Core
Context
Expression Language
Test

Spring Framework: JTA

- JTA: Java Transaction API
- Spring provides consistent transaction management for multiple resources
  - database
  - messaging
  - RMI / IIOP
- Configured using XML file


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
Caller invokes proxy, not target
Transaction created on way in, committed or rolled back on way out
Business logic invoked
Custom interceptors may run before or after transaction advisor

Spring Framework: JTA

- Simple DataSource TransactionManager
- Not Two Phase Commit
- Only Supports JDBC transactions
- Local Transaction Support (not Global)

```
<!-- a PlatformTransactionManager is still required -->

<bean id="transactionManager"
   class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <!-- (this dependency is defined somewhere else) -->
   <property name="dataSource" ref="dataSource"/>

</bean>
```

Spring Framework: JTA

- JTA Transaction Manager wraps a real manager
- Global Transaction Support (Two Phase Commit)
- Automatically finds the real manager in most containers
  - WebLogic
  - WebSphere
  - JBoss
- JTA supports transactions in JDBC and JMS.

```
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />
```

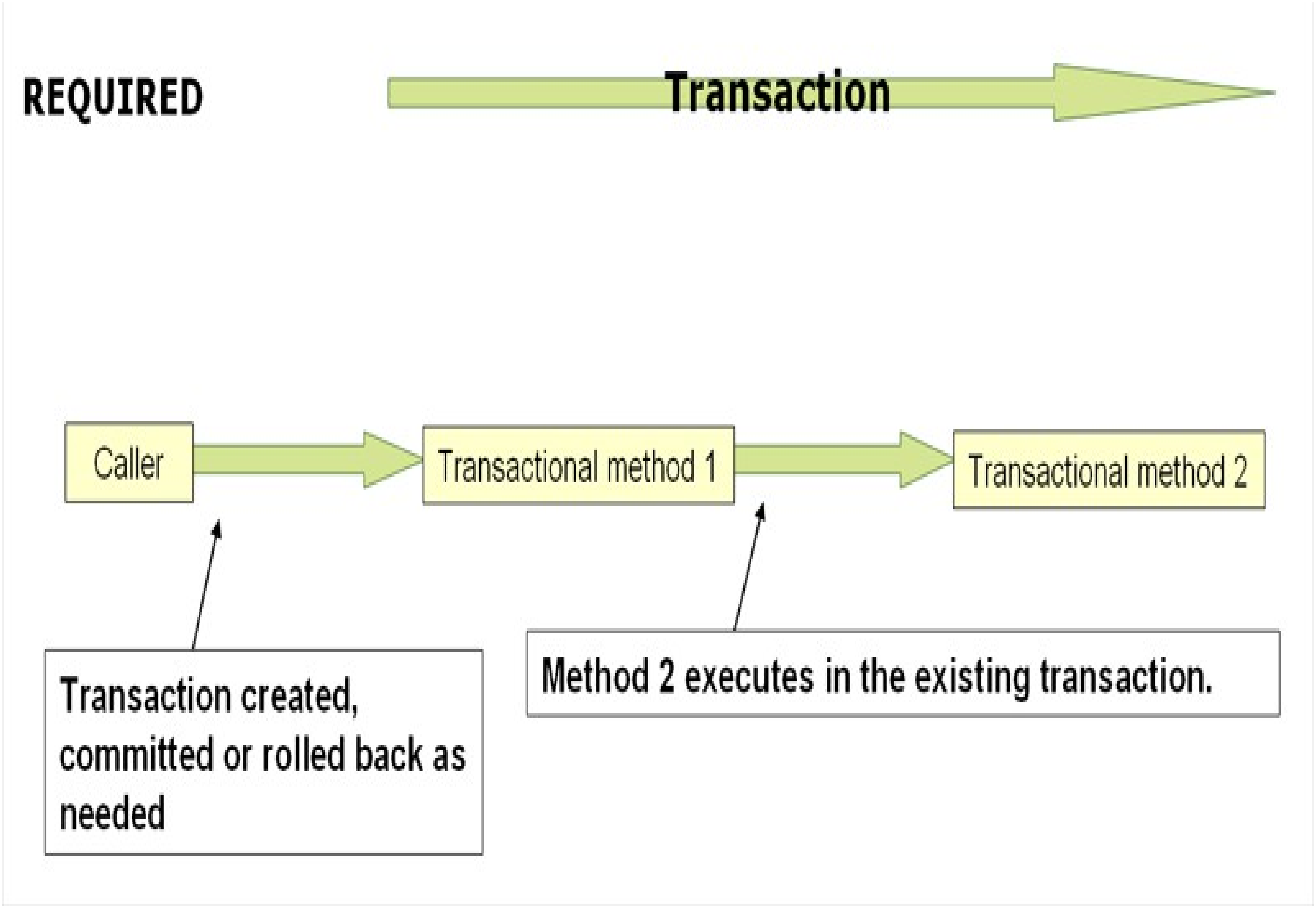
Spring Framework: JTA

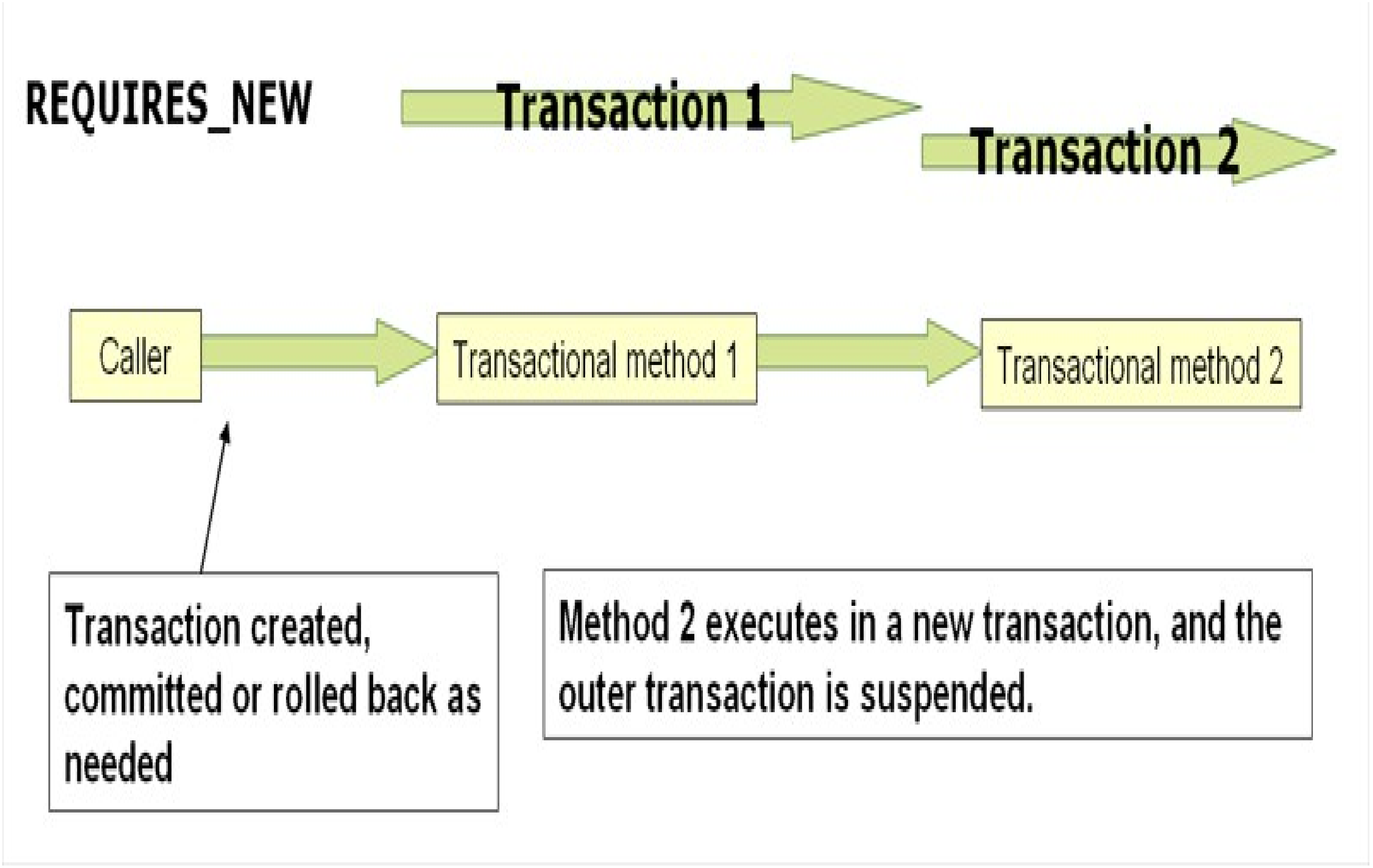
- @Transactional annotation wraps methods with transaction control using transaction manager calls to begin / commit / rollback
  - isolation and propagation level config
  - read-only optimization option
  - timeout (max time before rollback)
  - applied at class or method level

```
@Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
        public void updateFoo(Foo foo) {
            // do something
        }
```

Spring Framework: JTA

- Propagation provides JTA information about the type of transaction and how it processes with other transaction
  - PROPAGATION_REQUIRED: DEFAULT, will create a transaction if one is not already created and will participate if one exists.
  - PROPAGATION_REQUIRES_NEW: will create a new transaction and not participate in existing one, if other exists, will suspend it
  - PROPAGATION_SUPPORTS: will use transaction if exists, otherwise will not create one (read only operations)
  - PROPAGATION_NESTED: will leverage save point or sub transactions if supported by resources.
  - Other options that allow for error handling if exception exists or does not


REQUIRED
Transaction
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in the existing transaction.


REQUIRES_NEW
Transaction 1
Transaction 2
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in a new transaction, and the outer transaction is suspended.

Spring Framework: JTA

- Unit Testing can utilize a third party Transaction Manager
- Atomikos is open source and very stable
  - Transaction Essentials (Apache License 2)
  - DataSource wrappers for XA and non-XA connections
  - Support for JMS

Spring Framework: JTA

- Maven Setup
  - add dependency for Atomikos Transaction Essentials AI
  - add dependency for Spring TX
  - add geronimo-jta
  - add geronimo
- Properties Setup (application.properties)

```
my.account.checking.number=1234567890
```

- `my.account.savings.number=2341567891`

- `# transaction timeout, 5 minutes in debug`
- `transaction.timeout=300`

Spring Framework: JTA

```
<bean id="dataSource" name="dataSource,datasource"
        class="com.atomikos.jdbc.nonxa.AtomikosNonXADataSourceBean"
        lazy-init="true" init-method="init" destroy-method="close" >
        <property name="uniqueResourceName" value="NONXADBMS" />
        <property name="driverClassName" value="${jdbc.driverClassName}" />
        <property name="url" value="${jdbc.url}" />
        <property name="user" value="${jdbc.username}" />
        <property name="password" value="${jdbc.password}" />
        <property name="readOnly" value="false" />
        <property name="poolSize" value="1" />
        <property name="maxPoolSize" value="4" />
        <property name="minPoolSize" value="0" />
        <property name="testQuery" value="select 1 from dual" />
   </bean>

   <bean id="atomikosTransactionManager"
        class="com.atomikos.icatch.jta.UserTransactionManager"
        init-method="init" destroy-method="close">
        <property name="forceShutdown" value="true"/>
        <property name="transactionTimeout" value="${transaction.timeout}" />
   </bean>
```

Spring Framework: JTA

```
<bean id="atomikosUserTransaction"
       class="com.atomikos.icatch.jta.UserTransactionImp">
      <property name="transactionTimeout" value="$
{transaction.timeout}" />
</bean>

<!-- Configure the Spring framework to use JTA transactions from Atomikos
   -->
   <bean id="transactionManager"

   class="org.springframework.transaction.jta.JtaTransactionManager">
         <property name="transactionManager">
             <ref bean="atomikosTransactionManager" />
         </property>
         <property name="userTransaction">
             <ref bean="atomikosUserTransaction" />
         </property>
   </bean>


   <!-- enable the configuration of transactional behavior based on
   annotations -->
   <tx:annotation-driven transaction-manager="transactionManager"/>
```

Spring Framework: LocalTransaction

```
@Bean
public DataSourceTransactionManager transactionManager()
{
    DataSourceTransactionManager ds = new
DataSourceTransactionManager();
    ds.setDataSource(dataSource());
    return ds;
}
```

Spring Framework: JpaTransaction

```
@Bean
public JpaTransactionManager transactionManager(EntityManagerFactory emf) {
    JpaTransactionManager jpa = new JpaTransactionManager(emf);
    jpa.setDataSource(dataSource());
    return jpa;
}

@Bean
public JpaVendorAdapter jpaVendorAdapter() {
    HibernateJpaVendorAdapter jpaVendorAdapter = new
HibernateJpaVendorAdapter();
    jpaVendorAdapter.setDatabase(Database.H2);
    jpaVendorAdapter.setGenerateDdl(true);
    return jpaVendorAdapter;
}

@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
    LocalContainerEntityManagerFactoryBean lemfb = new
LocalContainerEntityManagerFactoryBean();
    lemfb.setDataSource(dataSource());
    lemfb.setJpaVendorAdapter(jpaVendorAdapter());
    lemfb.setPackagesToScan(getClass().getPackage().getName());
    Properties props = new Properties();
//...
    lemfb.setJpaProperties(props);
    return lemfb;
}
```

LAB #1: Transactions

Lab Assignment #1:

We need to add a transfer method to our various objects to move money from one account to another. Modify the Bank implementation to support Transactions using the @Transactional method.

```
public interface Bank {
    String getBankName();
    double getAccountBalance(String accountNumber);
    double creditAccount(String accountNumber, double amount);
    double debitAccount(String accountNumber, double amount);
    void transfer(String fromAccountNumber,
          String toAccountNumber, double amount);
}
```

LAB #1: Transactions

Lab Assignment #1:

We need to add a transfer method to our various objects to move money from one account to another. Modify the Bank implementation to support Transactions using the @Transactional method.

```
public interface Bank {
    String getBankName();
    double getAccountBalance(String accountNumber);
    double creditAccount(String accountNumber, double amount);
    double debitAccount(String accountNumber, double amount);
    void transfer(String fromAccountNumber,
          String toAccountNumber, double amount);
}
```

LAB #2: Transactions

Lab Assignment #1 Continued:

1) Step through debugger, do you see the aspect interceptors?
2) Set log4j.properties file to turn on Spring debugging:

```
#
#  Spring
#
log4j.logger.org.springframework=INFO
log4j.logger.org.springframework.tx=DEBUG
log4j.logger.org.springframework.transaction=DEBUG
log4j.logger.org.springframework.transaction.support=DEBUG
log4j.logger.org.springframework.transaction.annotation=DEBUG
log4j.logger.org.springframework.context=INFO
log4j.logger.org.springframework.dao=INFO
log4j.logger.org.springframework.orm=INFO
log4j.logger.org.springframework.jpa=INFO
log4j.logger.org.springframework.jta=DEBUG
log4j.logger.org.springframework.jms=INFO
log4j.logger.org.springframework.jms.connection=INFO
log4j.logger.org.springframework.jms.core=INFO
log4j.logger.org.springframework.jdbc=INFO
log4j.logger.org.springframework.web=INFO
log4j.logger.org.springframework.web.servlet=INFO
log4j.logger.org.springframework.web.servlet.view=INFO
log4j.logger.org.springframework.test.context.junit4=INFO
```

Spring Framework: Resources

- http://www.javaworld.com/javaworld/jw-01-2009/jw-01-spring-transactions.html
- http://static.springsource.org/spring/docs/4.3.x/spring-framework-reference/pdf/spring-framework-reference.pdf
- http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework
- http://www.atomikos.com
- <u>Spring In Action</u>, Walls
- <u>Pro Spring</u>, Harrop and Machacek